Programmierung 2

DHBW Mannheim - WWI17SCB - May 2018

Slides

https://goo.gl/U45wY3

5. Testing

Testing - Lernziele

* Verstehen, warum Testing wichtig ist
* Verschiedene Arten und Typen von Testing kennen
* Eigene Tests von Java-Programmen mit JUnit schreiben
* Die Methodik "Test Driven Development" kennen und anwenden können

Warum Testing?

Eine double -> int Konvertierung...

```
P_M_DERIVE(T_ALG.E_BH) := UC_16S_EN_16NS (TDB.T_ENTIER_16S
                                          ((1.0/C_M_LSB_BH) *
                                          G_M_INFO_DERIVE(T_ALG.E_BH)))
```

https://de.wikipedia.org/wiki/Ariane_V88

Metric vs Imperial Units...

* Im September 1999 verglüht der Mars Climate Orbiter" in der Atmosphäre des Roten Planeten.
* Grund: Die Berechnung der Kraft der kosmischen Strahlung wurde durch die Software in "imperial pounds" ausgeführt - erwartet wurden metrische Newton

https://en.wikipedia.org/wiki/Mars_Climate_Orbiter#Cause_of_failure

Deployment issues...

* Am 1.8.2012 vernichtet die Investmentfirma Knight Capital ca. 460 Millionen USD in 45min
* Grund: Eine Software namens "Power Peg", die als Teil einer Simulationsumgebung gedacht war und Aktien teuer kauft und billig verkauft
* Aufgrund eines falsch durchgeführten Updates wurde diese Software auf die Live-Umgebung losgelassen...

http://www.theregister.co.uk/2012/08/03/bad_algorithm_lost_440_million_dollars/

Race Conditions...

* 3 Menschen werden in den 80er-Jahren bei Bestrahlungen mit dem Therapiegerät Therac-25 getötet und zahlreiche weitere schwer verstrahlt
* Grund: Aufgrund einer Race Condition wurde bei sehr schneller Dateneingabe im Display des Geräts der falsche Betriebsmodus angezeigt.

https://en.wikipedia.org/wiki/Therac-25

Warum Testing?

Testing is the process of evaluating a system or its component(s) with the intent to find whether it satisfies the specified requirements.

---> Sicherstellung des Funktionsumfangs

Warum Testing?

A process of analyzing a software item to detect the differences between existing and required conditions (that is defects / errors / bugs)

---> Aufspüren von existierenden Bugs

Warum Testing?

Continuous testing is about fast and continuous feedback. Specifically, it is the practice in which tests are run as part the build pipeline so that every check-in and deployment is validated.

---> Vermeidung der Einführung von neuen Bugs

Arten von Tests

Wie testet man Software?

* Manuell vs Automatisch
* Funktionale vs. nicht-funktionale Tests
* Wann sollte ich anfangen zu testen?

Nicht-funktionale Tests

* Performance Testing
* Load Testing
* Security / Penetration Testing
* Failover / Stability Testing
* Usability Testing

...

Funktionale Tests

* Unit Testing
* Integration Testing
* Behavioral Testing

...

Unit Tests

* Testfälle auf einzelnen "Units" von Source-Code
* In OOP üblicherweise einzelne Klassen
* zB: Student.java --> StudentTest.java
* Ziel: Aufspüren von Bugs in Logik einzelner Methoden und Algorithmen
* Oftmals Testing mit Edge Cases, div/0-Errors, etc.
* Testen der gesamten offenen Schnittstelle einer Klasse
  (--> Kapselung!)

Unit Tests

```
1 public class CalculatorTest {
2   @Test
3   public void testAdd() {
4     MyCalculator tester = new MyCalculator();
5     assertEquals("1 + 2 must be 3", 3, tester.add(1, 2));
6   }
7 }
```

Integration Tests

* Manchmal (fälschlicherweise) auch als "Functional Tests" bezeichnet
* Testen des Zusammenspiels größerer Code-Einheiten
* Von >1 Klasse bis beliebig vielen
* Testen einzelner Prozesse der Software, z.B. OrderProcessingTest
* Testen der Schnittstellen kompletter Module
* Häufig Verwendung von Mocks zur Simulation der Umgebung

Integration Tests

```
1 public class OrderReportingTest {
2   @Test
3   public void testOrderValueIsAggregatedCorrectly() {
4     Order customerOrder1 = new Order(...);
5     customerOrder1.setTotal(250);
6     Order customerOrder2 = new Order(...);
7     customerOrder2.setTotal(750);
8
9     assertEquals("All orders are added up", 1000, OrderReporting.getSumOfAllOrders());
10  }
11 }
```

Behavioral Tests

* Testen des gesamten Systemverhaltens durch möglichst realitätsnaher Nutzung
* Nutzung der gleichen Umgebung wie die Endnutzer
* Bei Webanwendungen z.B. Verwendung von Selenium oder PhantomJS, um echte Browser-Umgebungen zu simulieren
* Programmatisches Auslösen von Klicks und Formulareingaben
* Testen von gesamten Features

Behavioral Tests

```
Feature: Explaining Cucumber
  In order to gain an understanding of the Cucumber testing system
  As a non-programmer
  I want to have an overview of Cucumber that is understandable by non-geeks

  Scenario: A worker seeks an overview of Cucumber
    Given I have a coworker who knows a lot about Cucumber
    When I ask my coworker to give an overview of how Cucumber works
    And I listen to their explanation
    Then I should have a basic understanding of Cucumber
```

What to test?

Test everything that could possibly break.

* Immer 80/20 denken
* Einfache Getter oder Setter testen --> evtl. nicht sinnvoll

http://wiki.c2.com/?TestEverythingThatCouldPossiblyBreak

What to test?

I prefer the phrasing "test anything you could have written incorrectly." If you tend to make mistakes writing simple accessor methods, then test them (of course, you only have accessor methods because some other code that you wrote test-first needs them, right?) -- Keith Ray

* Oder doch?

http://wiki.c2.com/?TestEverythingThatCouldPossiblyBreak

What to test?

* **Es gibt keine feste Regel. Aber Empfehlungen:**
* Testet die häufigsten Anwendungsfälle von so vielen Methoden wie möglich - das vermeidet Regressionen
* Testet Edge Cases - vor allem bei komplexer Logik.
* Sobald ihr einen Bug findet, schreibt einen Test der ihn sichtbar macht. Dann erst fixt den Bug.
* Fangt mit komplexer Businesslogik an - vor allem dort, wo Fehler evtl. nicht sofort sichtbar sind.

Eigene Tests mit JUnit

JUnit

* https://junit.org
* De-Facto-Standard für automatisierte Tests in Java
* Testing Framework --> Test Runner & Assertion Engine
* Ein JUnit-Test ist eine Methode in einer Klasse (Testklasse)
* Ausführung durch den JUnit Test Runner
* Kenntlich gemacht durch @Test-Annotation
* Test-Methoden führen zu testenden Code aus und prüfen mittels **Assertions**, ob der Code sich korrekt verhält

JUnit

```
1 import static org.junit.jupiter.api.Assertions.assertEquals;
2 import org.junit.jupiter.api.Test;
3
4 public class TestSomething {
5     @Test
6     public void multiplicationOfZeroIntegersShouldReturnZero() {
7         MyCalculator tester = new MyCalculator();
8
9         assertEquals("10 x 0 must be 0", 0, tester.multiply(10, 0));
10        assertEquals("0 x 10 must be 0", 0, tester.multiply(0, 10));
11        assertEquals("0 x 0 must be 0", 0, tester.multiply(0, 0));
12    }
13 }
```

Namenskonventionen in Tests

* Test-Klassen enden üblicherweise mit "Test"
* Test-Methoden sind lesbare Sätze - idealerweise sagt der Methodenname genau, was getestet wird
* Oft wird das Schema xShouldY() verwendet

Installation von JUnit - manueller Weg

* JUnit (v4) wird als JAR-File verteilt und kann manuell heruntergeladen und dem Classpath hinzugefügt werden

1. Ladet euch JUnit Core und HamCrest herunter

   http://bit.ly/My9IXz

   http://bit.ly/1gbl25b

Installation von JUnit - manueller Weg

The Central Repository

SEARCH | ADVANCED SEARCH | BROWSE | QUICK STATS

g:"junit" AND a:"junit"

SEARCH

Search Beta Advanced Search | API Guide | Help

We're building something bigger together: Open Jobs

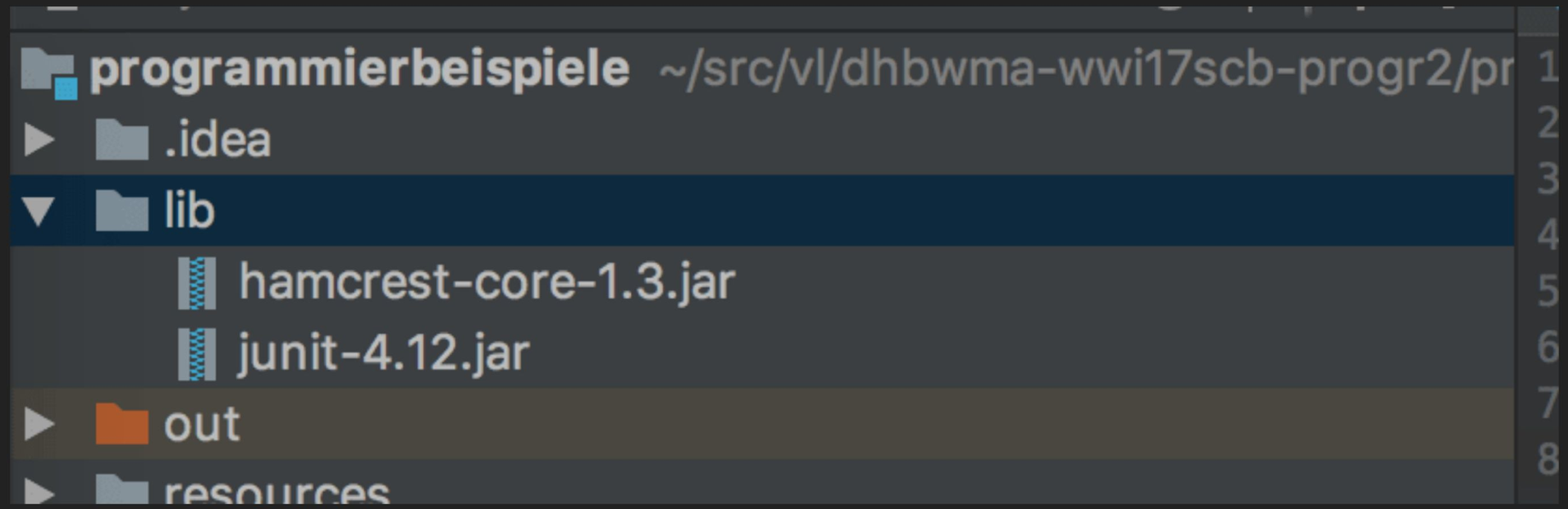
Search Results

< 1 2 > displaying 1 to 20 of 24

GroupId	ArtifactId	Version	Updated	Download
junit	junit	4.12	04-Dec-2014	pom jar javadoc.jar sources.jar
junit	junit	4.12-beta-3	09-Nov-2014	pom jar javadoc.jar sources.jar
junit	junit	4.12-beta-2	25-Sep-2014	pom jar javadoc.jar sources.jar

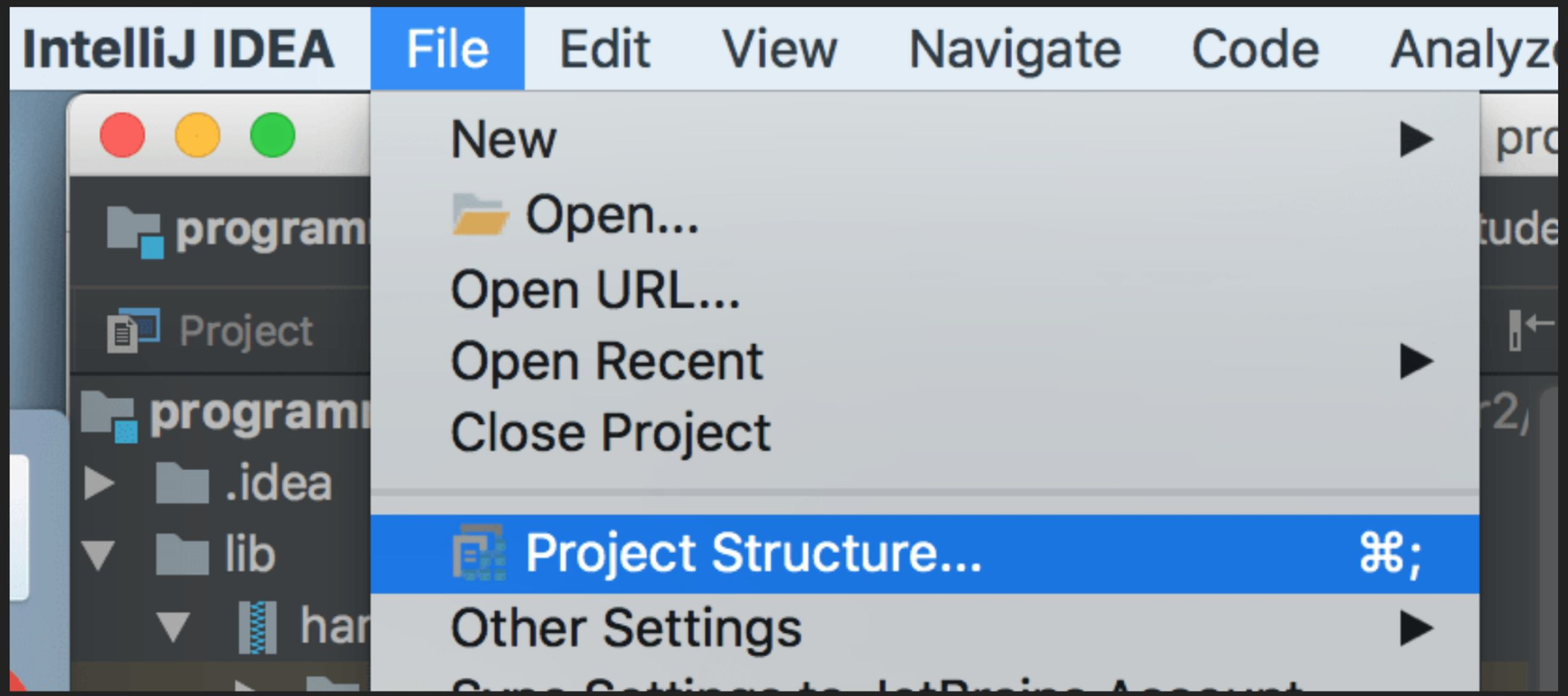
Installation von JUnit - manueller Weg

2. Packt beide Dateien in ein Verzeichnis innerhalb eures Projekts



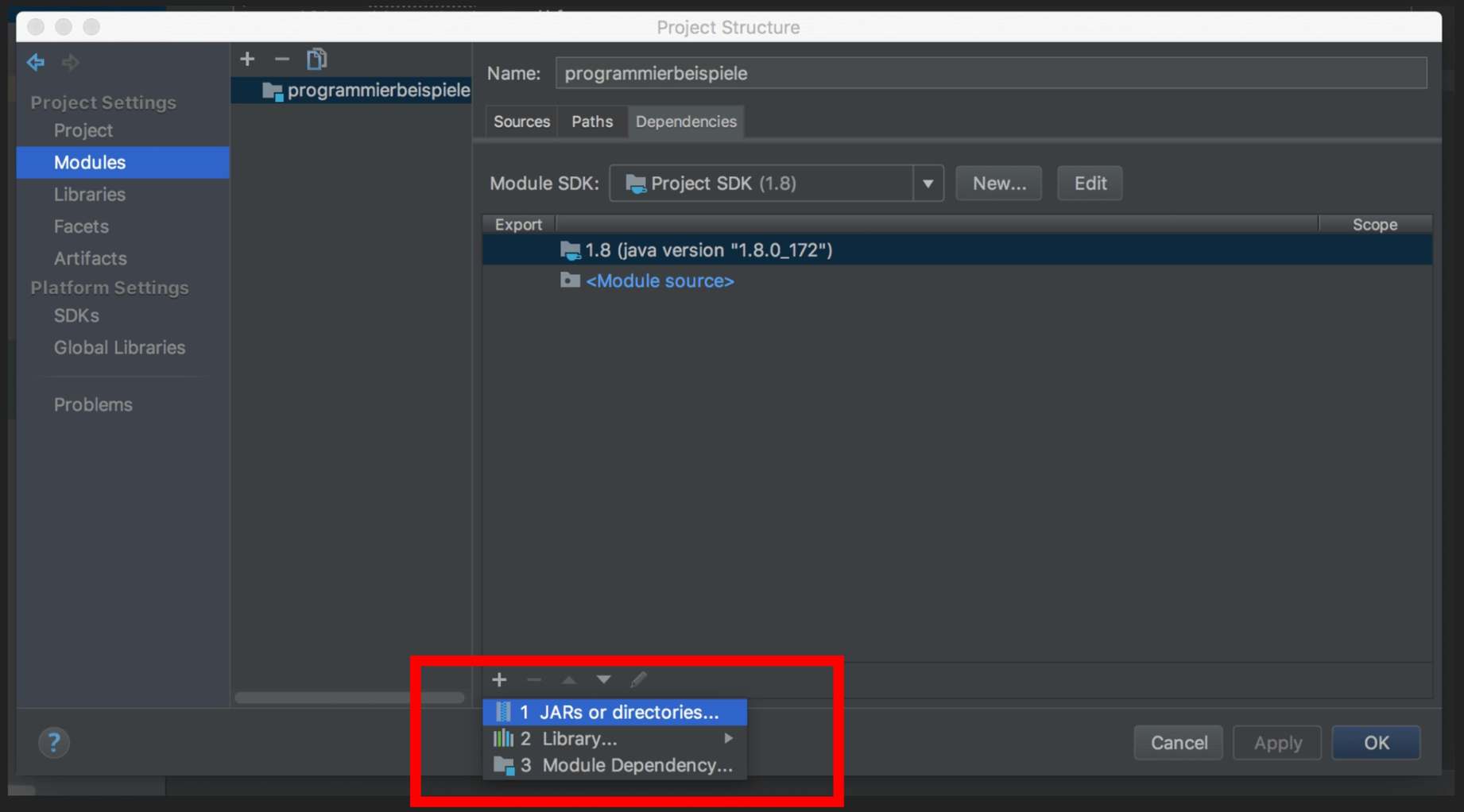
Installation von JUnit - manueller Weg

3. Fügt sie als Dependency hinzu (Beispiel: IntelliJ)



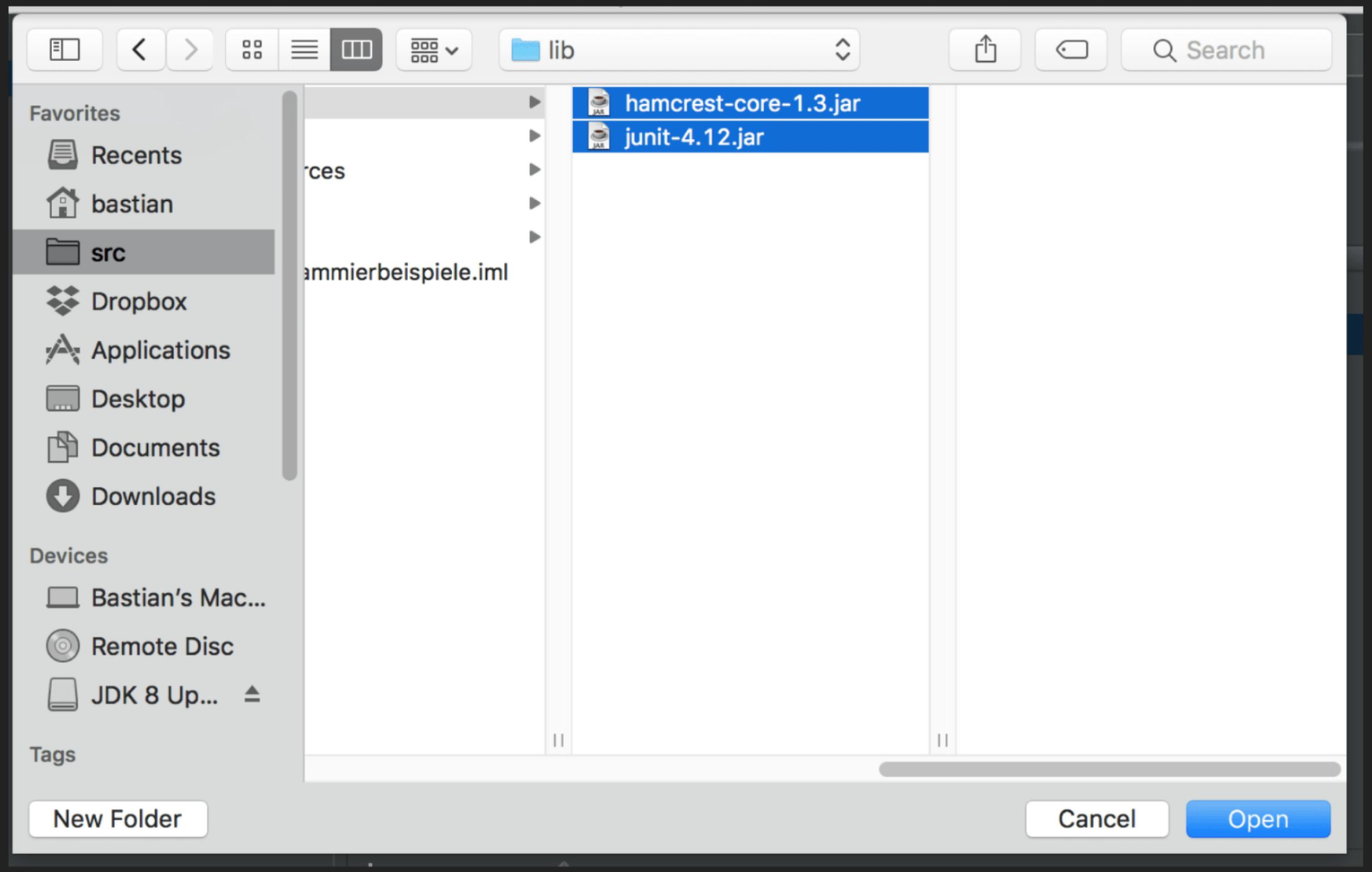
Installation von JUnit - manueller Weg

3. Fügt sie als Dependency hinzu (Beispiel: IntelliJ)



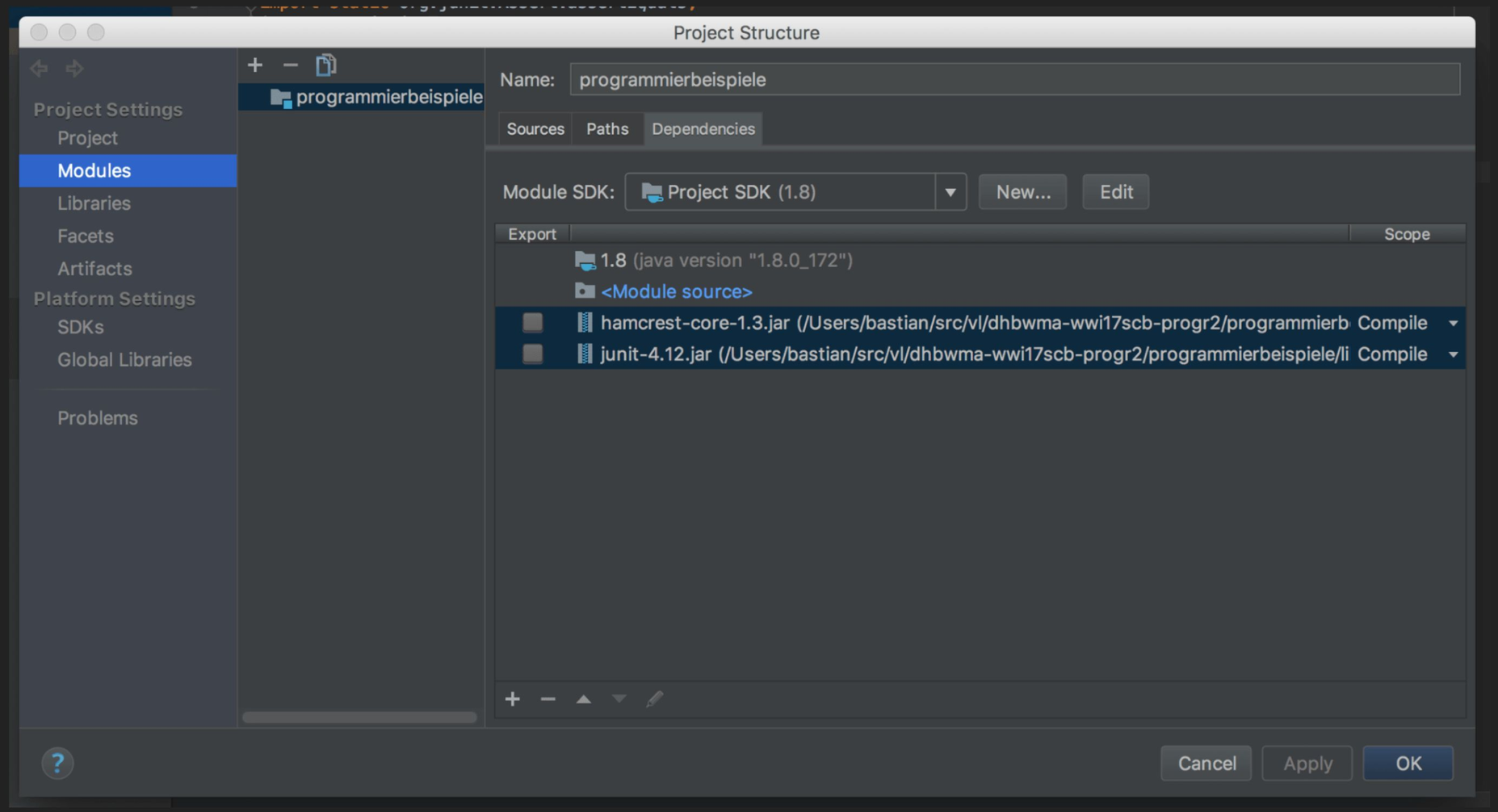
Installation von JUnit - manueller Weg

3. Fügt sie als Dependency hinzu (Beispiel: IntelliJ)



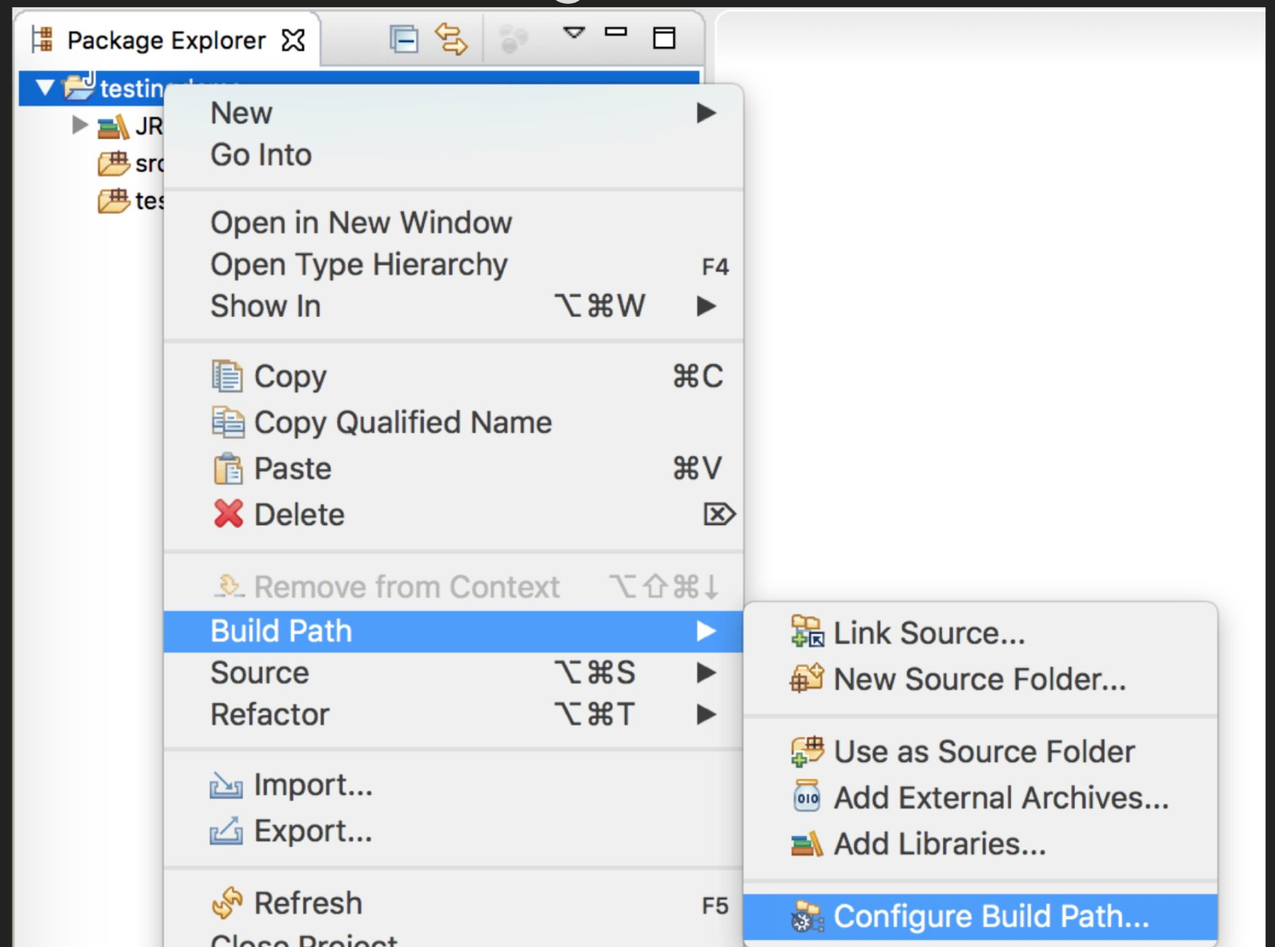
Installation von JUnit - manueller Weg

3. Fügt sie als Dependency hinzu (Beispiel: IntelliJ)



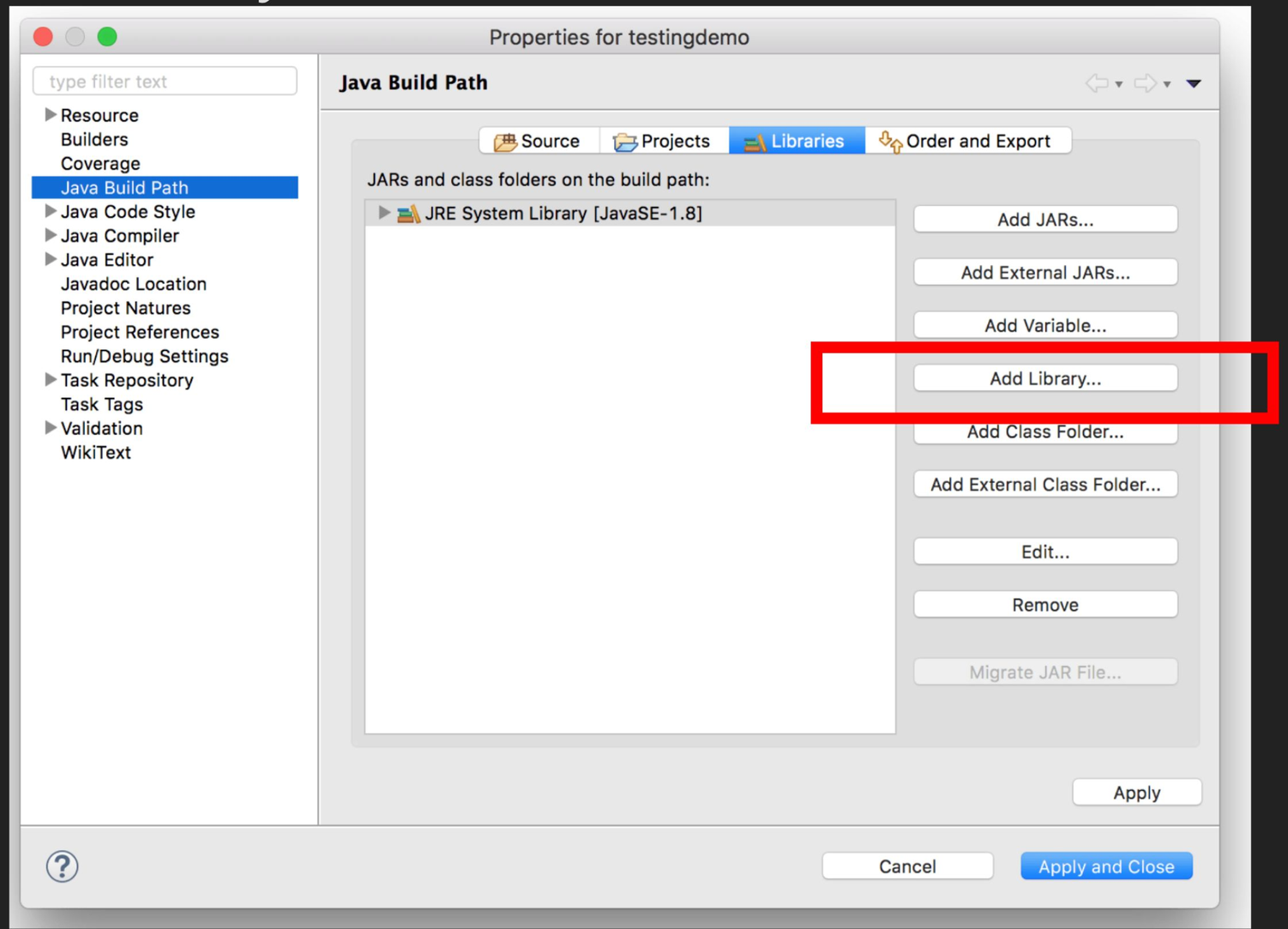
Installation von JUnit - in Eclipse

* Eclipse bringt JUnit bereits mit, ihr braucht es nicht separat herunterladen. Öffnet den Dialog "Build Path":



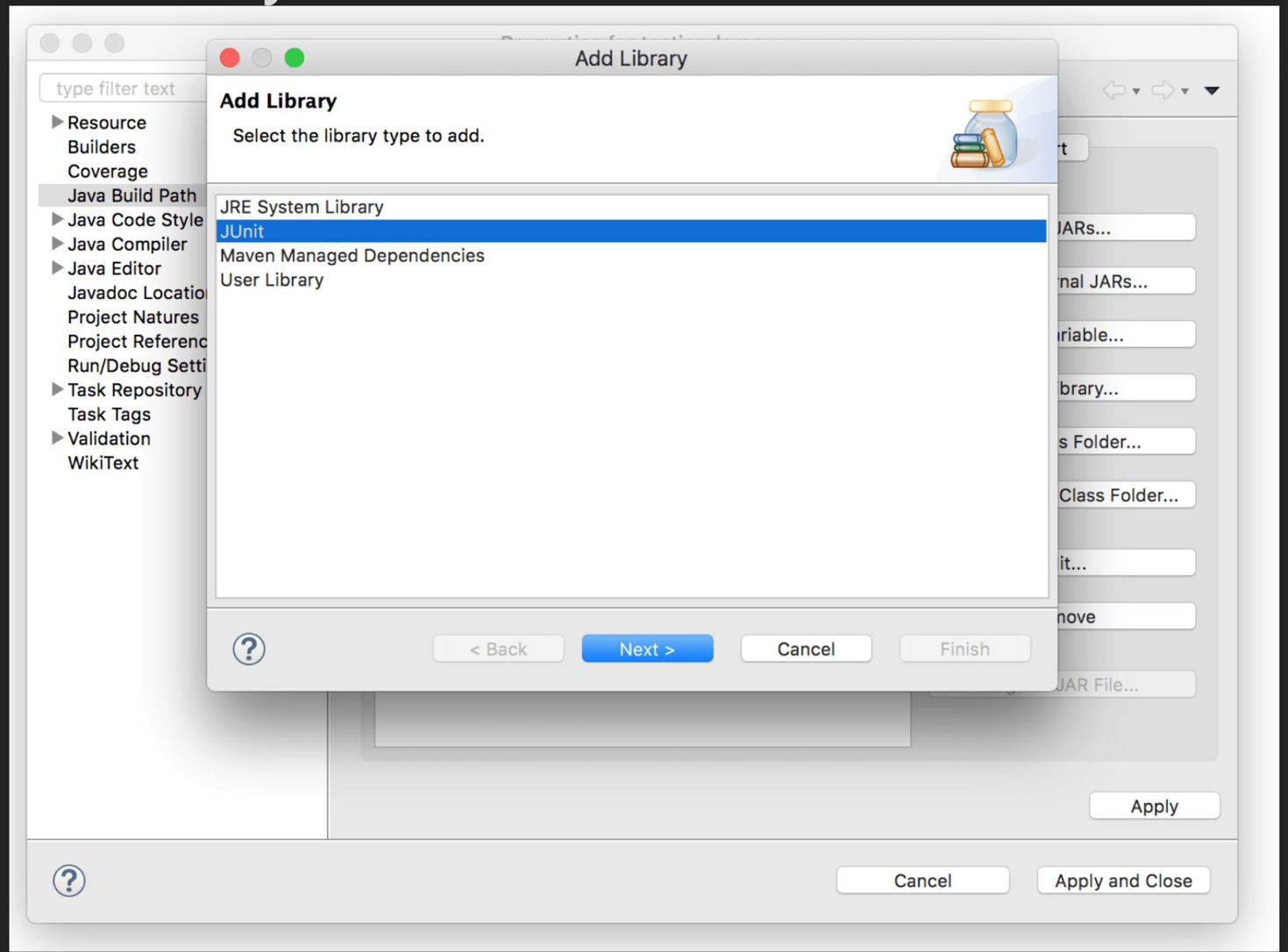
Installation von JUnit - in Eclipse

* Fügt JUnit als Library hinzu:



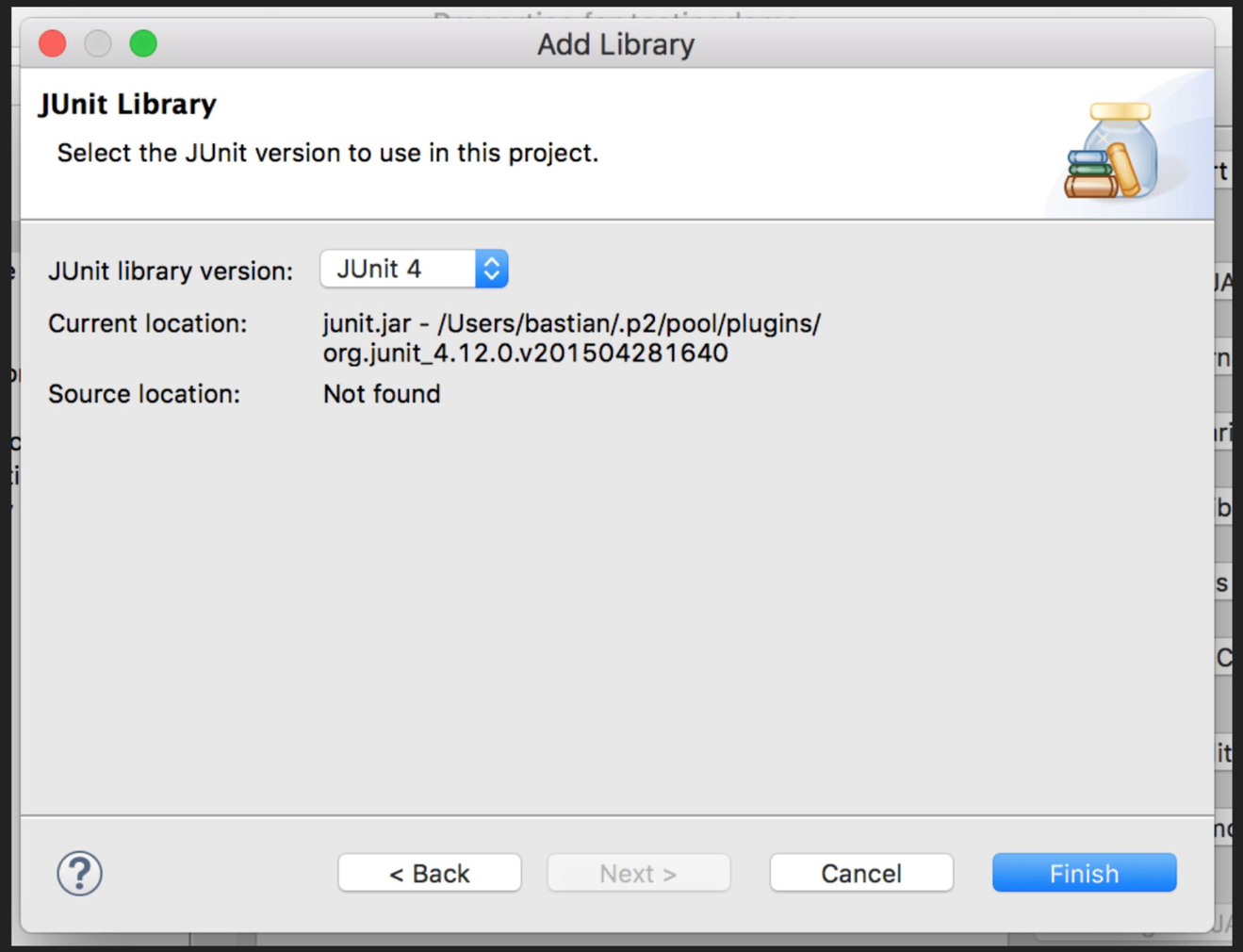
Installation von JUnit - in Eclipse

* Fügt JUnit als Library hinzu:



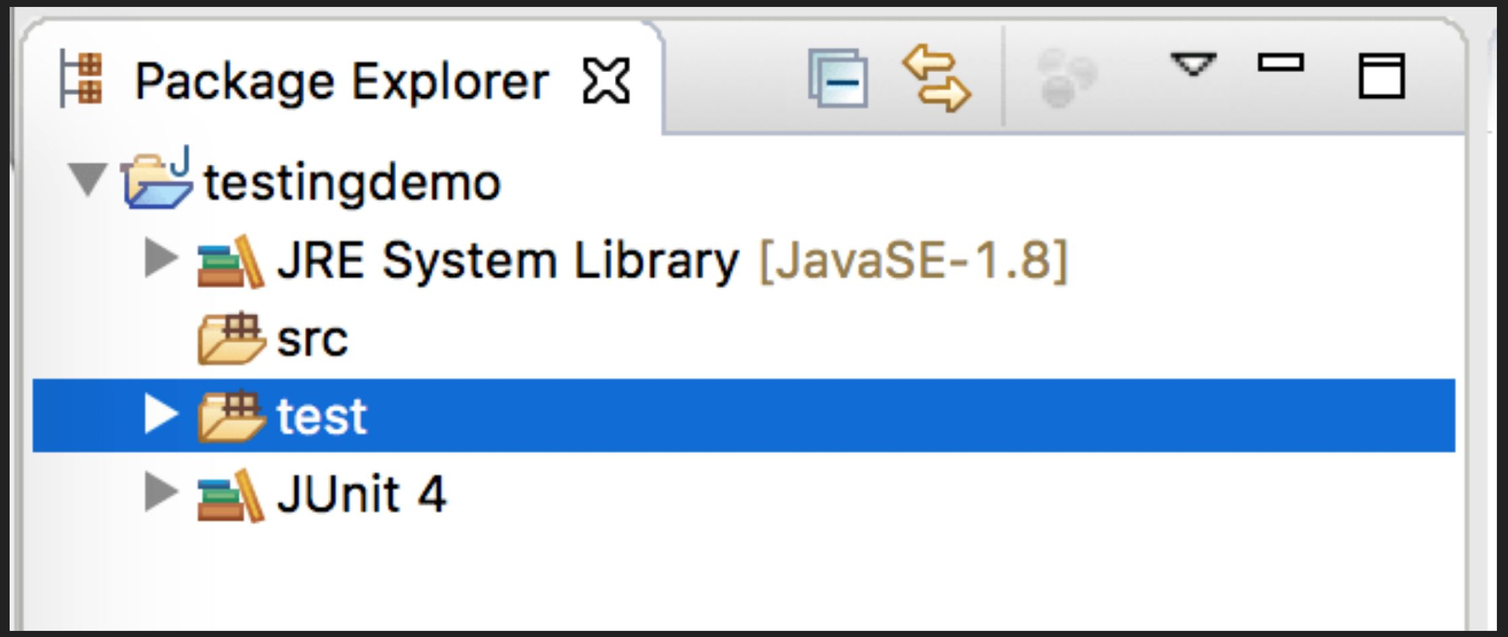
Installation von JUnit - in Eclipse

* Nehmt hier JUnit 4, damit alle auf dem gleichen Stand arbeiten



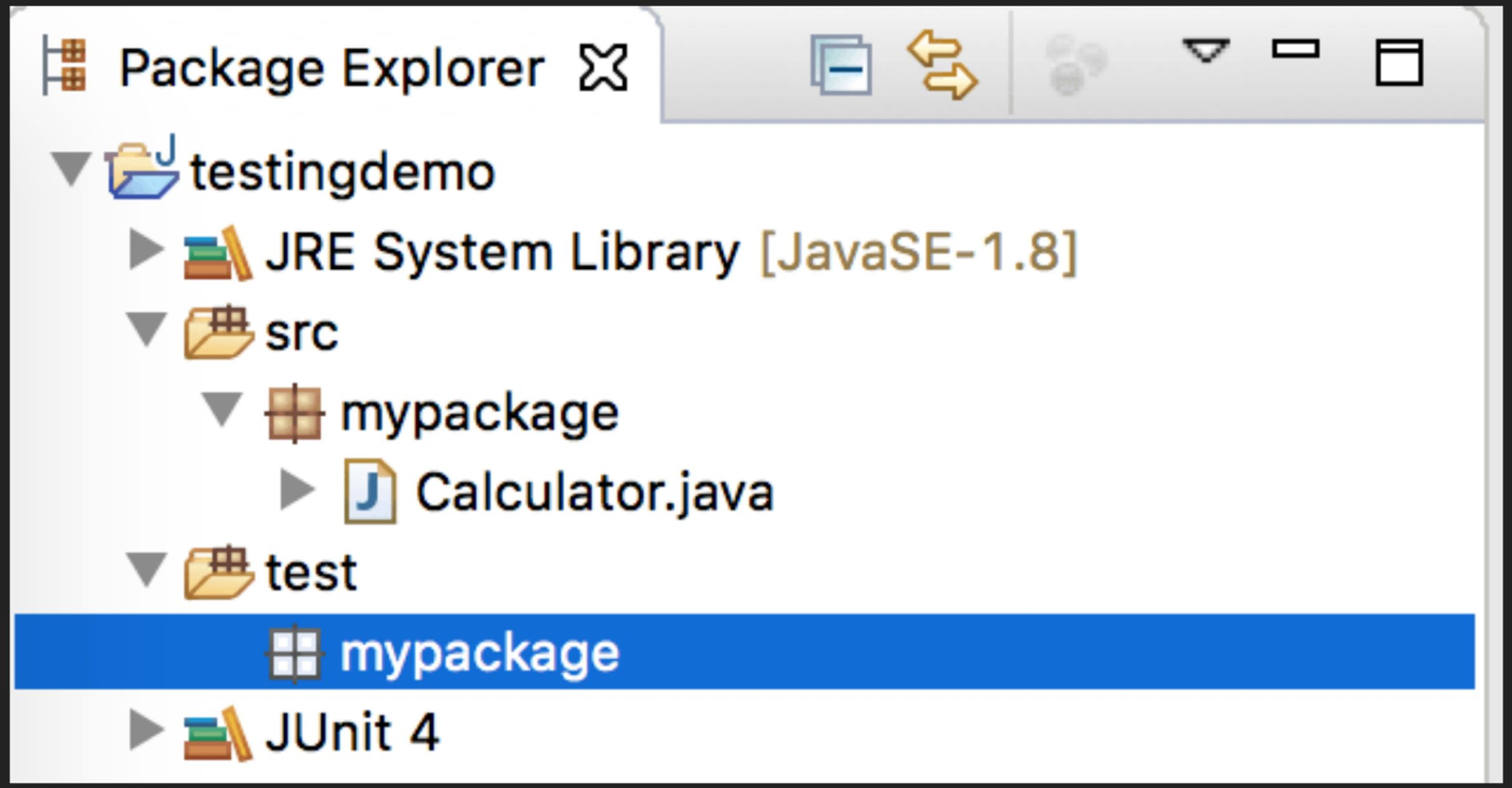
Euer erster Test

* Legt parallel zu "src" ein neues Verzeichnis "test" an und markiert es als Source Directory



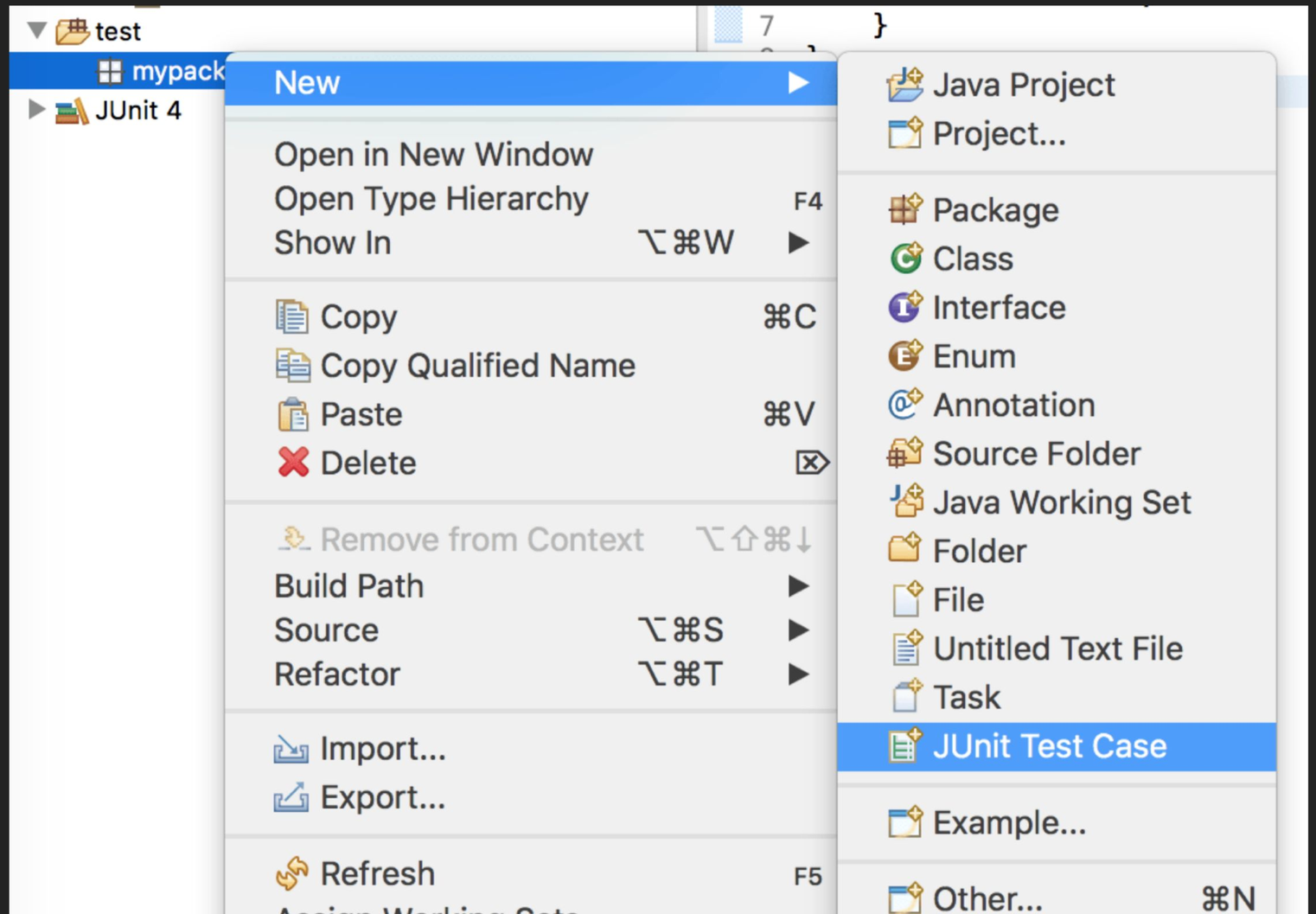
Euer erster Test

* Darin könnt ihr ebenfalls Packages anlegen.
* Empfehlung: Bildet die Paketstruktur eures Codes nach.



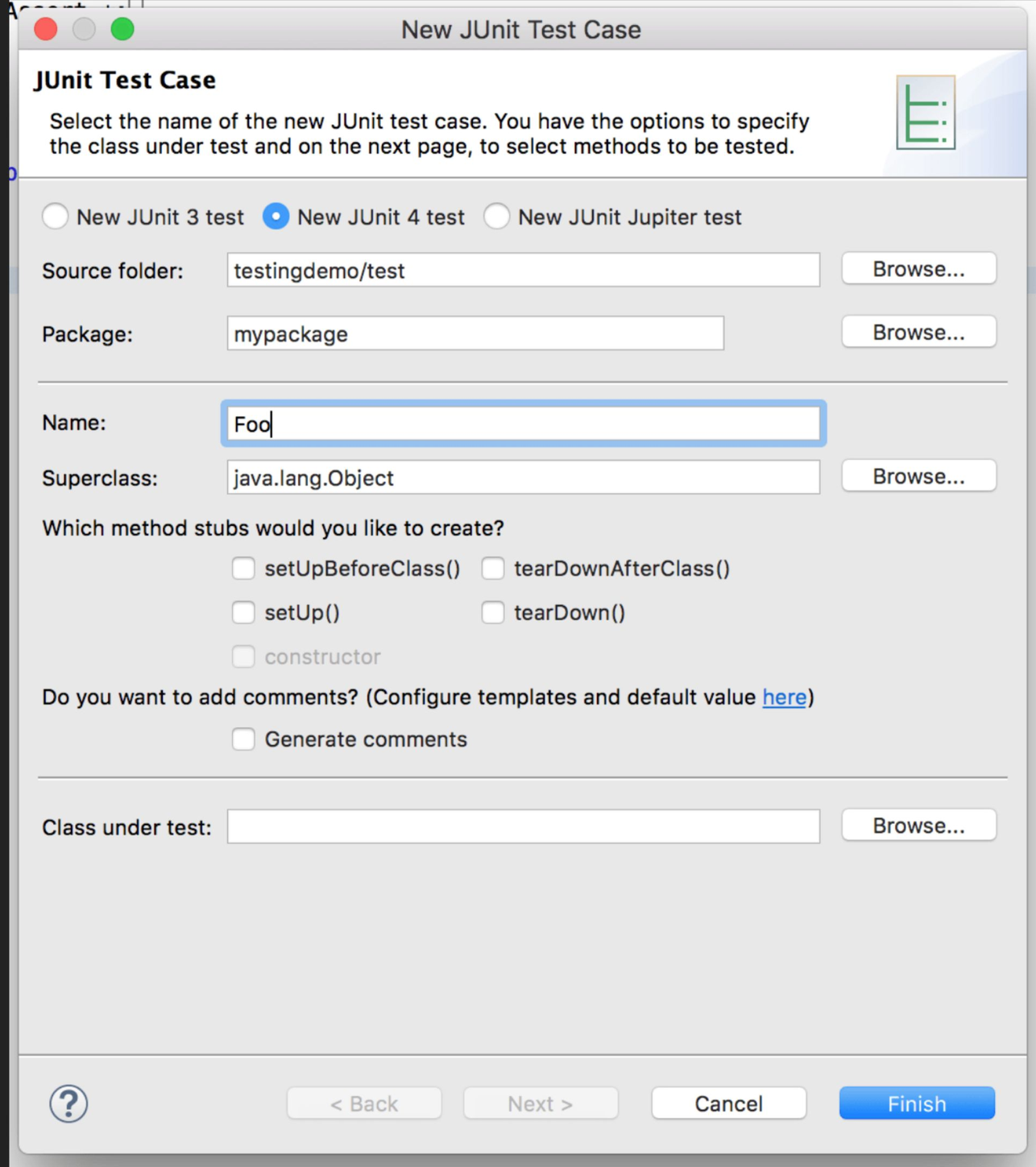
Euer erster Test

* Über "Rechtsklick->New->JUnit Test Case" bekommt ihr Hilfe



Euer erster Test

* Durch die Auswahl einer "Class under Test" bekommt ihr Test-Stubs generiert

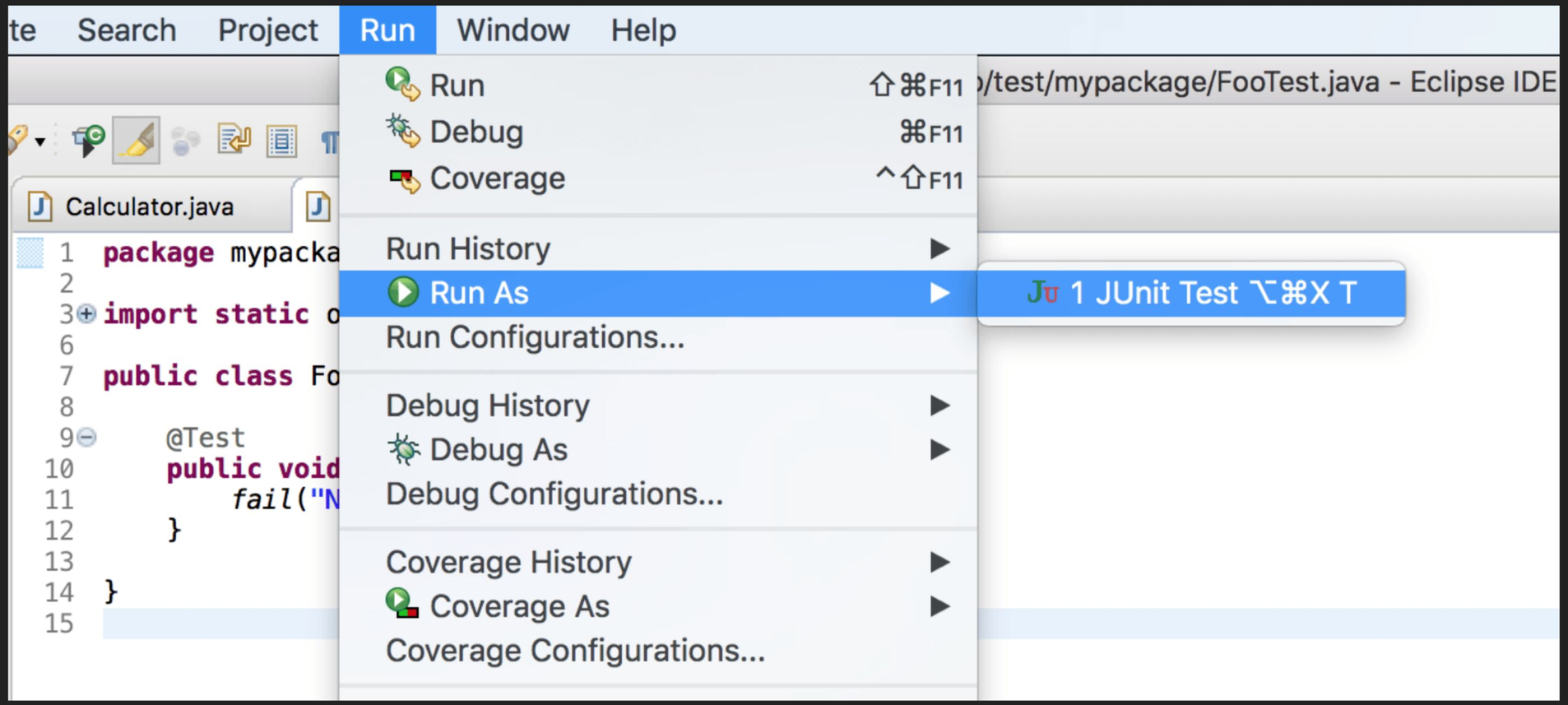


Euer erster Test

```
1  package mypackage;
2
3  import static org.junit.Assert.*;
4  import org.junit.Test;
5
6  public class FooTest {
7
8      @Test
9      public void test() {
10         fail("Not yet implemented");
11     }
12
13 }
```

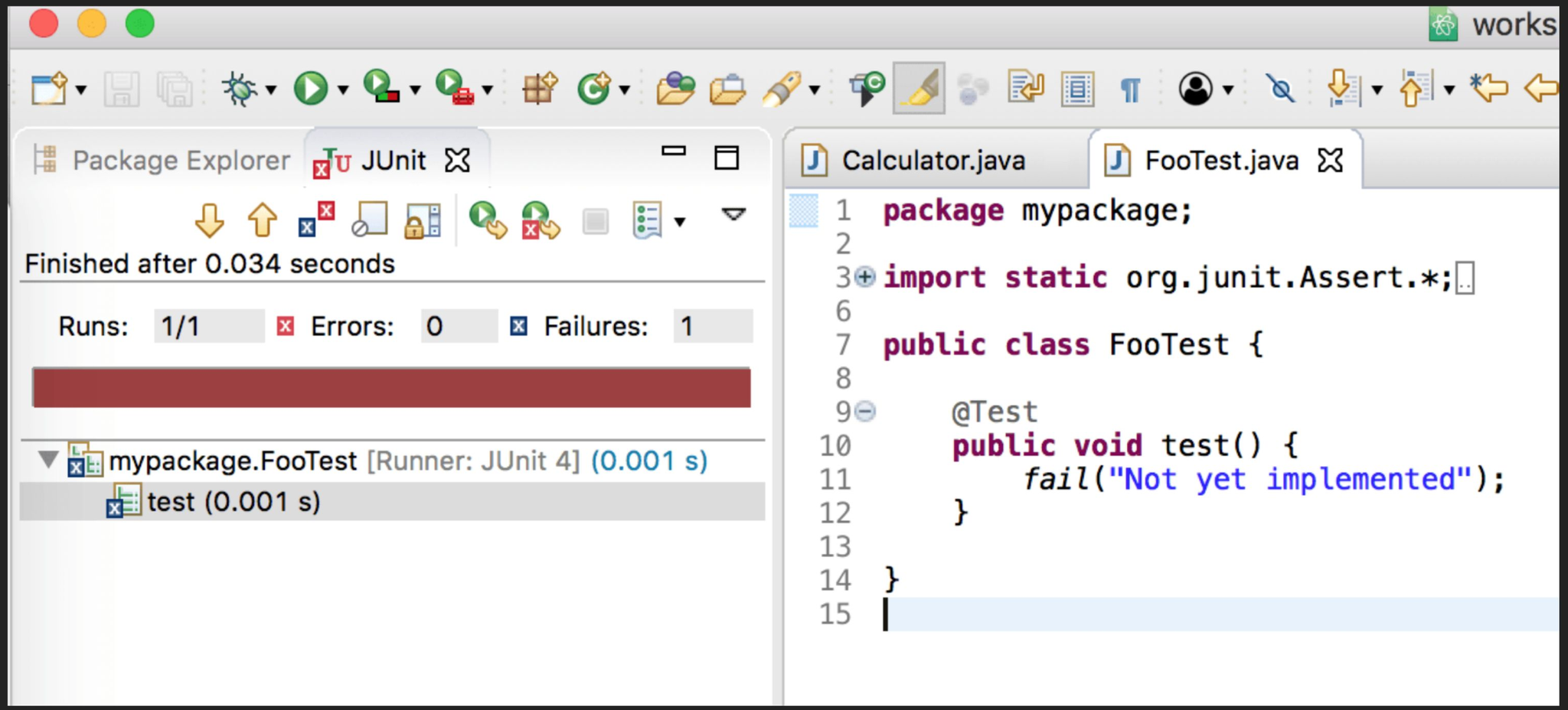
Euer erster Test

* Ausführen des Tests:



Euer erster Test

* Und das Ergebnis:



Assertions

* Häufigste Assertion:

```
assertEquals("Assertion Hint", expected, actual);
```

* Weitere:

```
assertTrue(boolean cond) / assertFalse(boolean cond)
assertNotNull(Object object) / assertNull(Object object)
assertSame(o1, o2) / assertNotSame(o1, o2)
assertArrayEquals(expectedArray, resultArray)
assertThat(value, matcher)
fail(message)
```

Coding Time!

Testing - Codingaufgabe 1

45m

* Im Repository findet ihr unter "Programmierbeispiele" im Ordner src/testing/ex1 vorbereitete Klassen - ihr kennt sie schon
* Kopiert euch diese Klassen in ein neues Projekt
* Installiert JUnit (entweder via IDE oder manuell per Download)
* Die Beispielklassen enthalten einige Bugs (s. nächste Folie)
    1. Schreibt Tests, die diese Fehler abprüfen und fehlschlagen.
    2. Behebt die Bugs.
    3. Führt die Tests erneut aus, um zu sehen ob sie nun grün sind.

Testing - Codingaufgabe 1

45m

* Bugs
  * Der Name der Universität wird nicht korrekt gesetzt.
  * Vorlesungen werden der Universität nicht hinzugefügt.
  * Studenten können eine Vorlesung mehrfach besuchen
  * Die Liste der Vorlesungen, die ein Student besucht,
    enthält ein überflüssiges Komma und Leerzeichen (", ") am Ende

Test Driven Development

Test Driven Development

* TDD ist eine Praxis des Extreme Programming (XP)
* Prämisse: Kein ungetesteter Code im System
* Aus Requirements werden Tests abgeleitet und geschrieben
* Software wird implementiert, um Tests zu erfüllen
* Damit ist automatisch aller Code getestet

Test Driven Development

Traditionell

1. Write code
2. Maybe write tests
3. git push

TDD

1. Write tests
2. Write code to fulfill tests
3. git push

Test Driven Development

* Vorgehen für jeden Entwickler:

  RED

  GREEN

  REFACTOR

  REPEAT

Test Driven Development

* Vorgehen für jeden Entwickler:

  RED

  * Schreibt einen sehr einfachen Testfall für das zu implementierende Verhalten
  * Sollte sich anfühlen wie Beispielcode
  * Sollte compilieren und fehlschlagen --> Test ist rot

Test Driven Development

* Vorgehen für jeden Entwickler:

  GREEN

  * Schreibt möglichst simplen und dummen Code, der den Testfall grün werden lässt.
  * Nicht elegant, schön oder sauber - nur der Test soll grün werden.

Test Driven Development

* Vorgehen für jeden Entwickler:

  REFACTOR

  * Macht den Code sauber - Formatierung, Coding-Standards, Methodennamen, DRY, ...
  * Führt den Test nochmal aus - er sollte weiterhin grün sein.

Test Driven Development

* Vorgehen für jeden Entwickler:

  REPEAT

  * Beginnt von vorne. Habt ihr zuvor Edge Cases nicht betrachtet?

    Schreibt Tests für Edge Cases, lasst sie failen, fixt sie.

  * usw...

Coding Time!

Testing - Codingaufgabe 2

15m

* Erstellt ein neues Projekt und installiert JUnit.
* Entwickelt eine Klasse "Taschenrechner" mit folgenden statischen Methoden:

```
add(int, int) : int
subtract(int, int) : int
multiply(int, int) : int
divide(int, int) : double
```

* Geht "test driven" vor - schreibt also jeweils zuerst die Tests.