

Programmieren

JUnit-Tests



Heusch --
Ratz --

Institut für Automation und angewandte Informatik

```
final List<String> allResults = new ArrayList<String>();  
final Map<String, Integer> typeWordResultCount = new HashMap<String, Integer>();  
final Map<String, Integer> typePoints = new HashMap<String, Integer>();  
evaluation.put(type, typePoints);  
  
for (final Sheet sheet : this.sheets) {  
    final String sheetResult = sheet.getPlayerInput(type);  
    if (sheetResult.startsWith(start) && this.isValidWord(sheetResult, type)) {  
        validWordCountForType++;  
        allResults.add(sheetResult);  
    }  
}
```

Testgetriebene Entwicklung

- ... (auch *testgesteuerte Programmierung*, engl. *test first development* oder *test-driven development* (TDD))
ist eine Methode, die häufig bei der **agilen Entwicklung** von Computerprogrammen eingesetzt wird.
- Bei der testgetriebenen Entwicklung erstellt der Programmierer **Software-Tests konsequent vor den zu testenden Komponenten**.
- Die Testfälle werden auch als Grey-Box-Tests bezeichnet.

Warum Testgetriebene Entwicklung?

- Verbesserung der Code-Qualität
- Steigerung der Produktivität
- Überprüfbarkeit des Erreichens von Zielen
- Leichtere Wartbarkeit

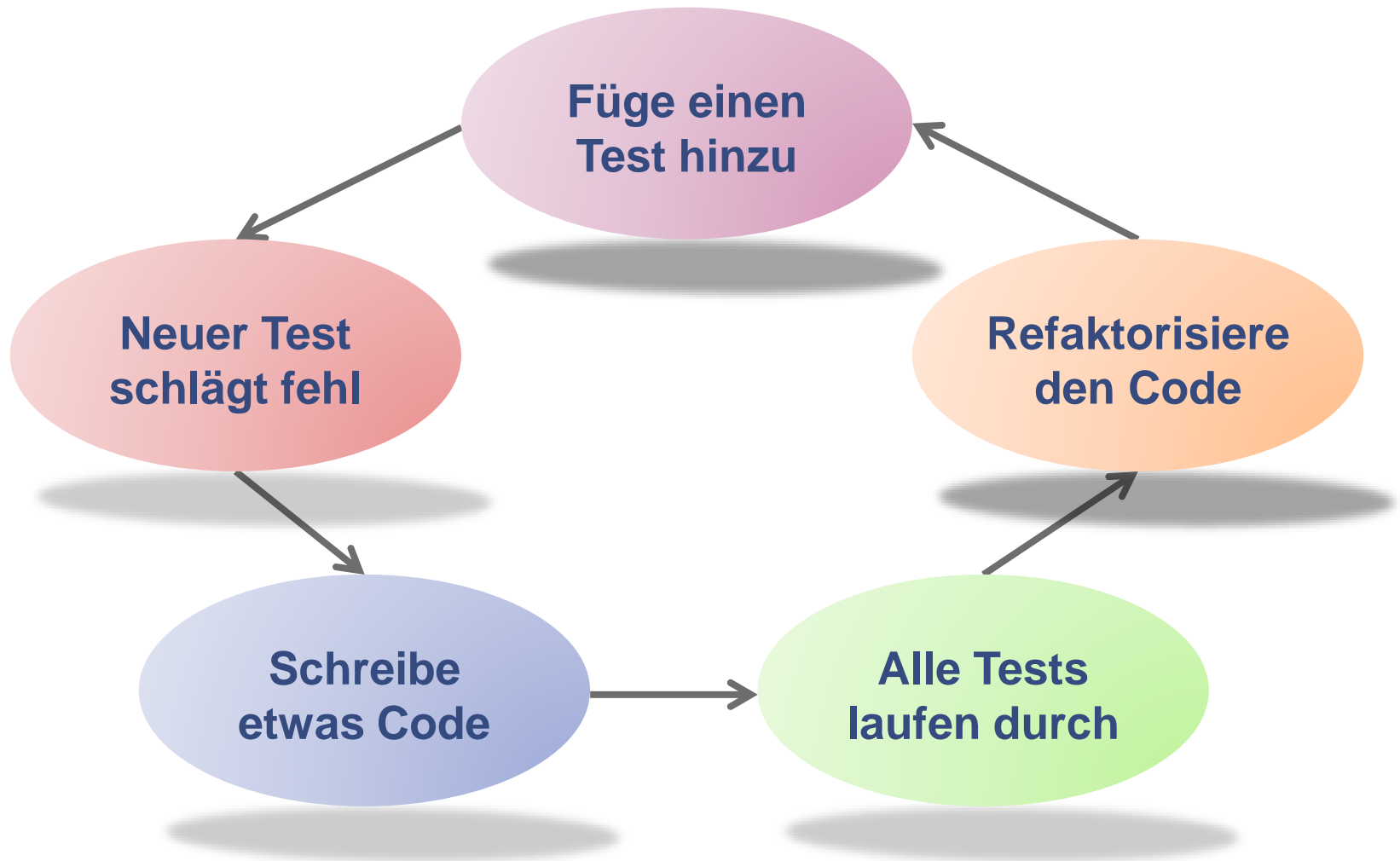


Quelle: <http://dilbert.com/strips/comic/2007-11-26/>

Iteratives Testen und Programmieren

- Der Zyklus des Testens und Programmierens:
 1. Entwirf zunächst (nur) einen Test.
(Es ist noch kein zugehöriger Code vorhanden.)
 2. Schreibe gerade soviel Code, dass der Test zwar abläuft, jedoch noch fehlschlägt.
 3. Schreibe gerade soviel Code, dass dieser Test durchläuft.
- Dieser Prozess wird so lange wiederholt, bis dem Programmierer keine weiteren Tests mehr einfallen, die unter Umständen fehlschlagen könnten, bis der Code also seine durch die Tests spezifizierten Anforderungen erfüllt.
- Anschließend wird der Code in die einfachstmögliche Form refaktorisiert.

Zyklus des Testens und Programmierens

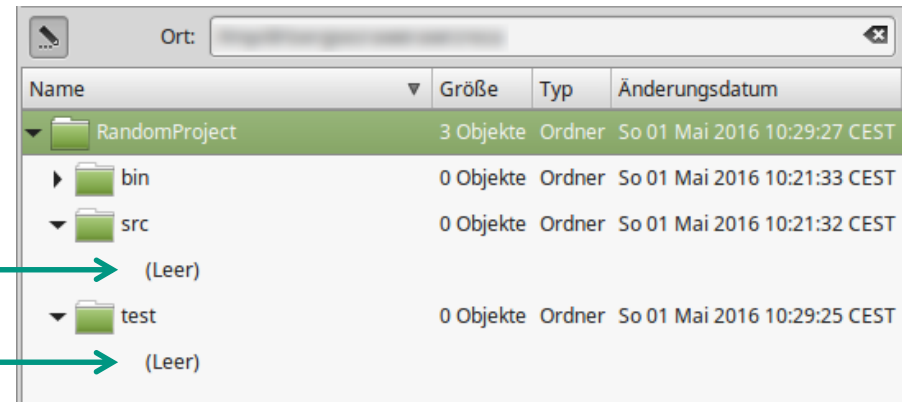


Projektarchitektur

- Bisher zumeist ein Verzeichnis für Quellcode („src“)
 - Test-Klassen sollten jedoch nicht im gleichen Quell-Verzeichnis wie „normaler“ Code abgelegt werden
 - Software muss bei der Entwicklung getestet werden
 - Auslieferung kompilierter Versionen aber ohne Test-Klassen
- Java-Projekt mit separaten Quellcode-Baum für Test-Klassen
- Ordner wird üblicherweise „test“ genannt

Hier die „normalen“ Klassen

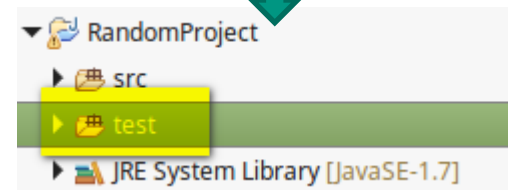
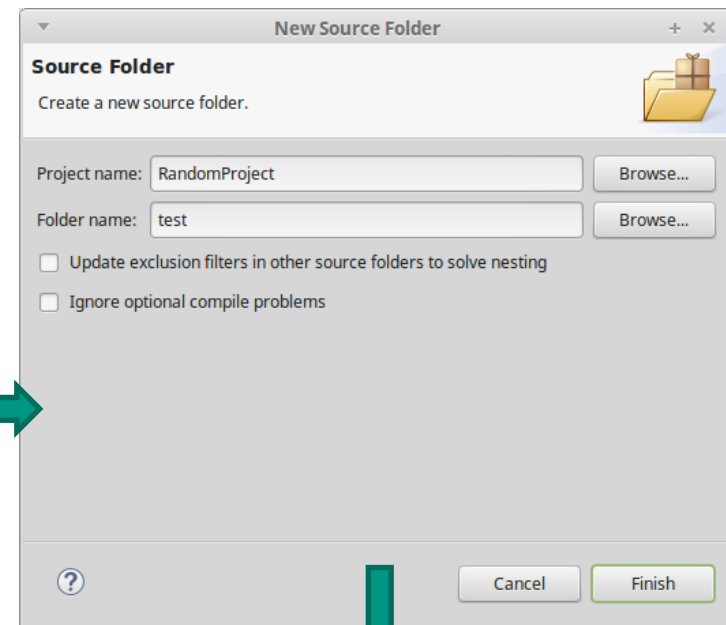
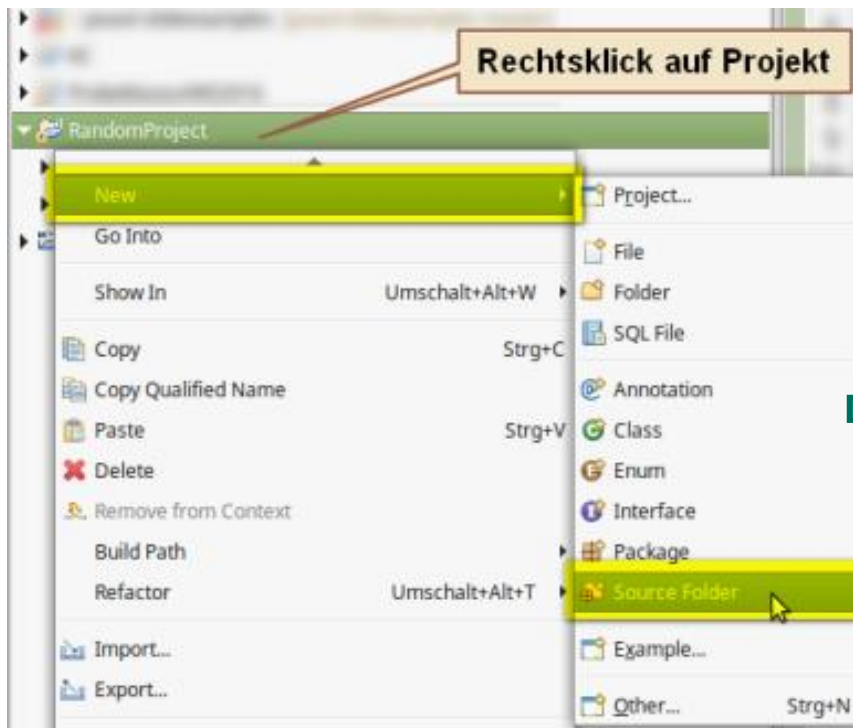
Hier alle „Test“-Klassen



Name	Größe	Typ	Änderungsdatum
RandomProject	3 Objekte	Ordner	So 01 Mai 2016 10:29:27 CEST
bin	0 Objekte	Ordner	So 01 Mai 2016 10:21:33 CEST
src	0 Objekte	Ordner	So 01 Mai 2016 10:21:32 CEST
test	0 Objekte	Ordner	So 01 Mai 2016 10:29:25 CEST

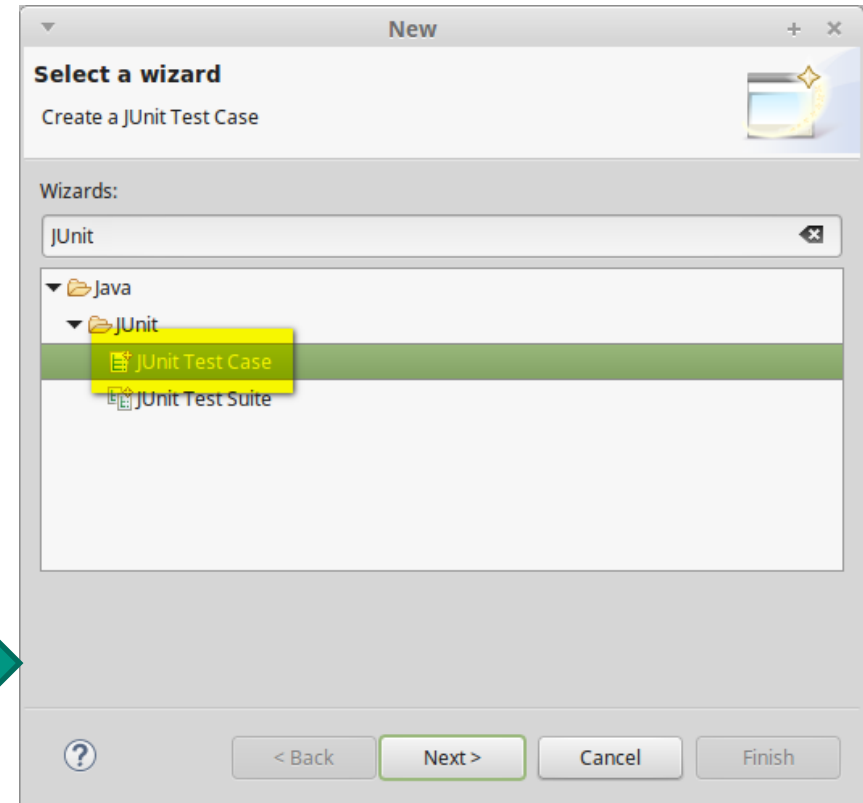
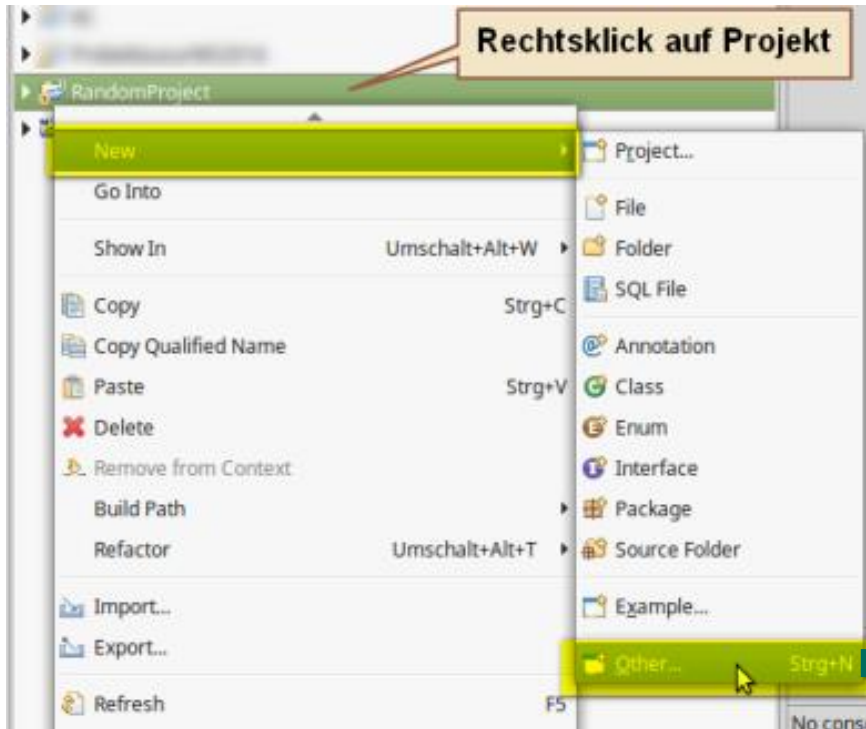
JUnit – Erster Test in Eclipse (1)

- Projekt anlegen
- Quellcode-Baum für Test-Klassen erzeugen



JUnit – Erster Test in Eclipse (2)

- In das Projekt einen „JUnit Test Case“ einfügen



JUnit – Erster Test in Eclipse (3)

Neuen Test anlegen

here)' with an unchecked option 'Generate comments'. At the bottom, there is a 'Class under test:' field with a 'Browse...' button. At the very bottom, there are four buttons: '?', '< Back', 'Next >', 'Cancel', and 'Finish'." data-bbox="228 186 630 880"/>

JUnit-Version wählen
(wir beziehen uns auf JUnit 4)

Quellcode-Baum
(„test“ auswählen!)

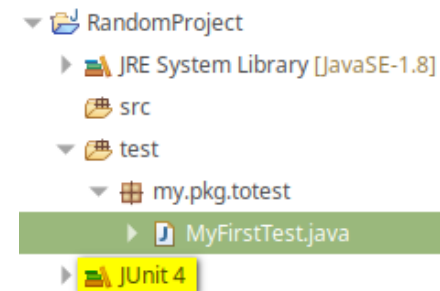
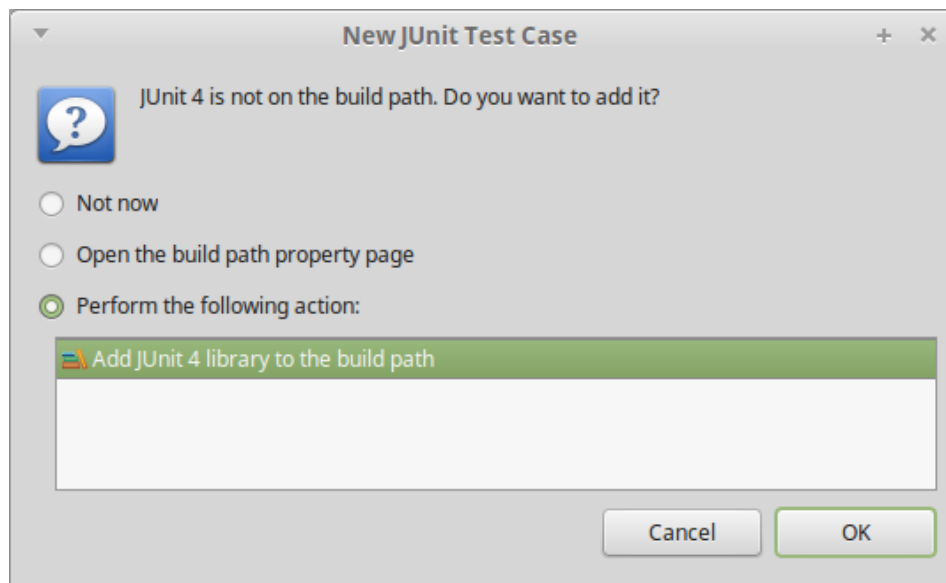
Package für Test-Klasse

Name der Test-Klasse

Zu erstellende
Hilfsmethoden

JUnit – Erster Test in Eclipse (4)

- JUnit benötigt die entsprechenden Bibliotheken im Projekt-Classpath
- Eclipse fragt beim Erstellen der Test-Klasse, ob diese dem Projekt hinzugefügt werden sollen:



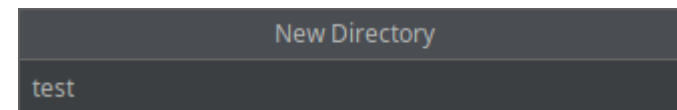
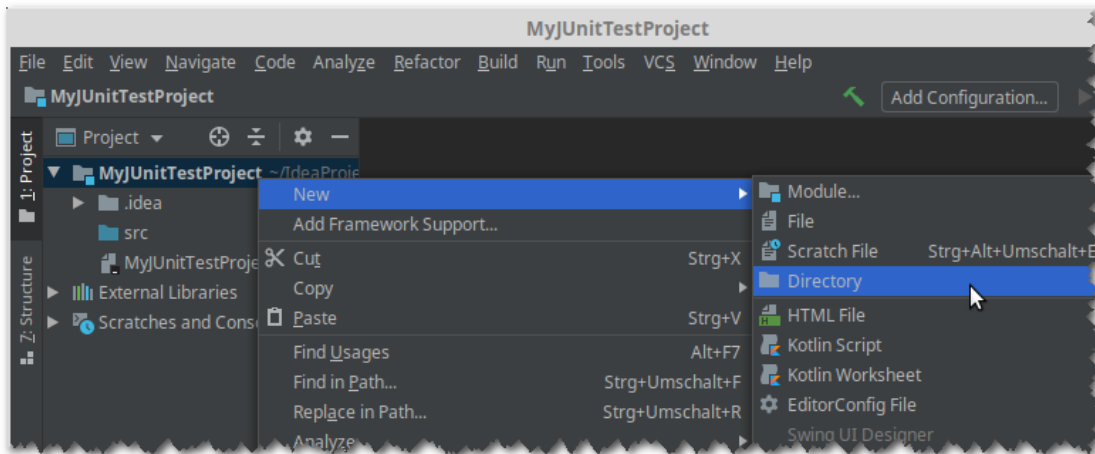
JUnit – Erster Test in Eclipse (5)

- Durch den Assistent erzeugtes Grundgerüst



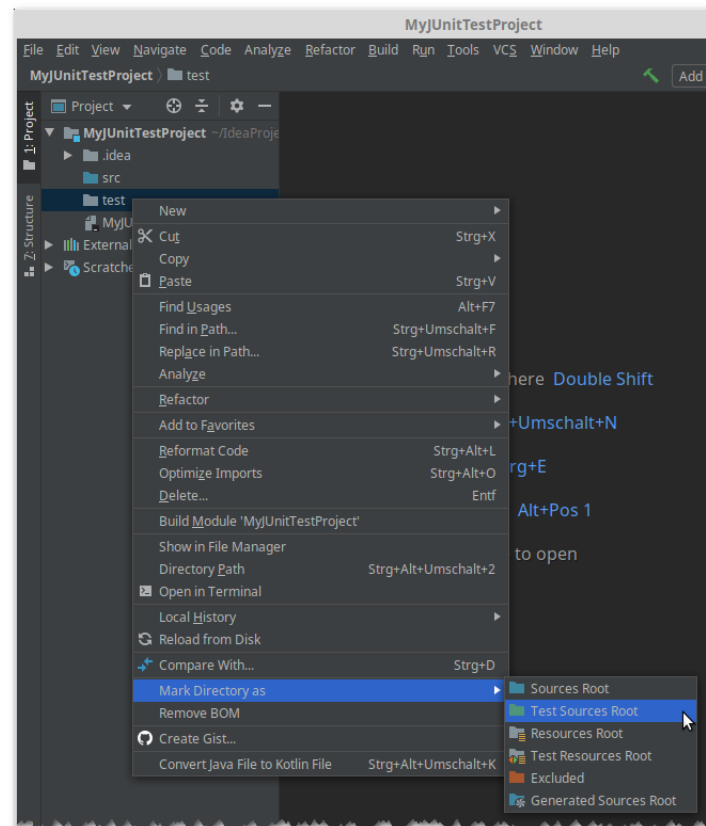
JUnit – Erster Test in IntelliJ (1)

- Projekt anlegen
- Neues Verzeichnis für den Test-Quellcode anlegen



JUnit – Erster Test in IntelliJ (2)

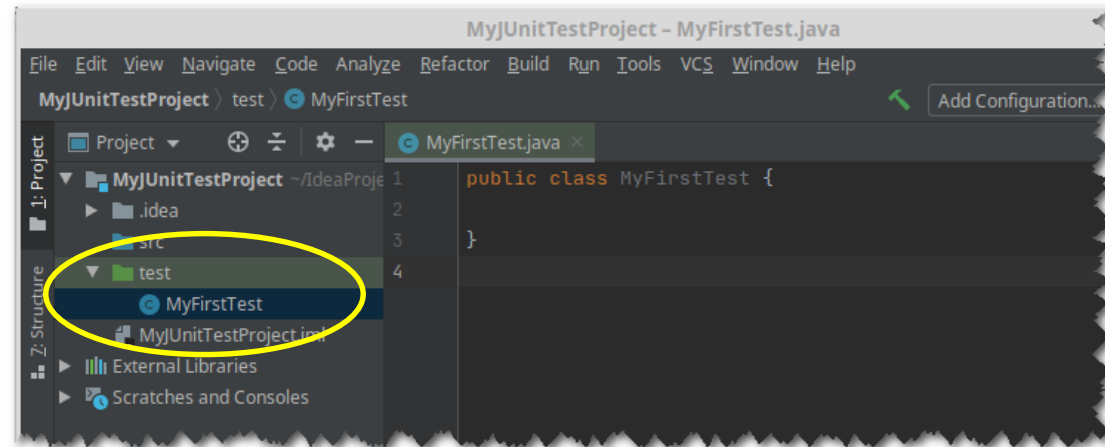
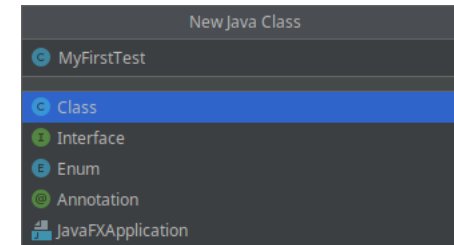
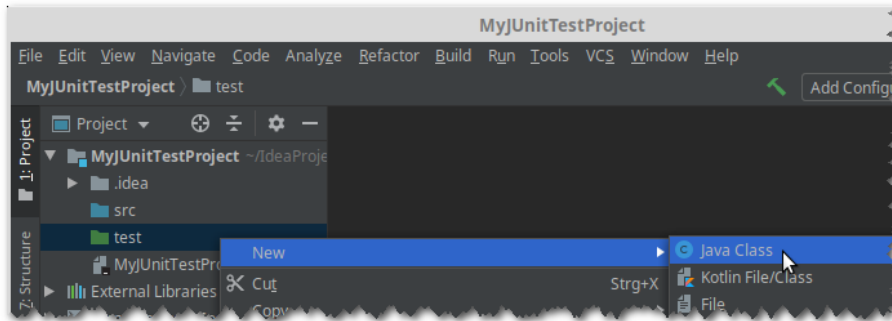
- Verzeichnis als Test-Quellcode-Verzeichnis registrieren



IntelliJ IDEA

JUnit – Erster Test in IntelliJ (3)

■ Klasse für Unit-Tests erstellen



IntelliJ IDEA

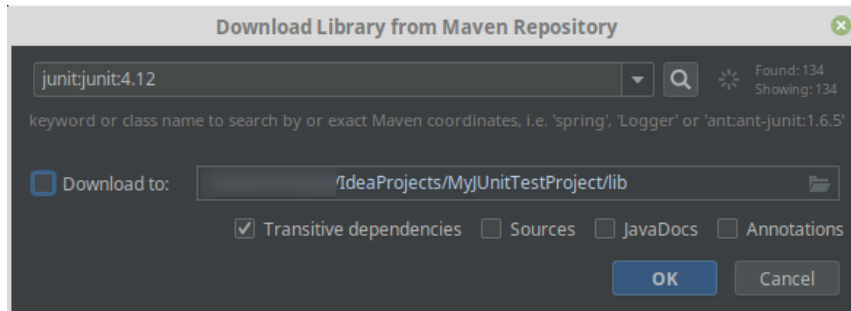
JUnit – Erster Test in IntelliJ (3)

- Beim ersten Verwenden von Test-Code IntelliJ das Test-Framework einbinden lassen



JUnit – Erster Test in IntelliJ (4)

■ Einbindung bestätigen



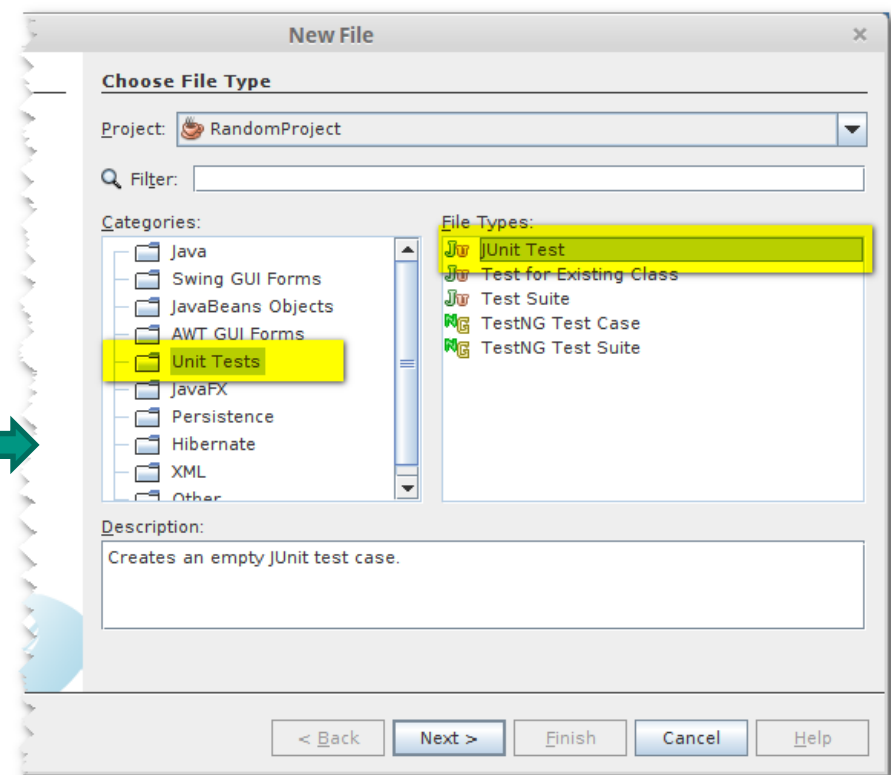
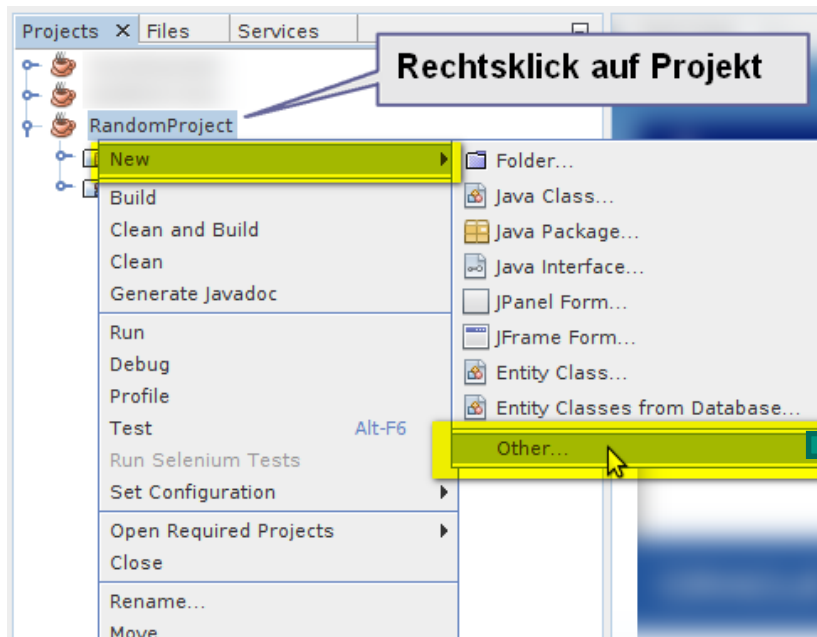
■ Alternative Vorgehensweise(n) siehe: <https://www.jetbrains.com/help/idea/tdd-with-intellij-idea.html>



IntelliJ IDEA

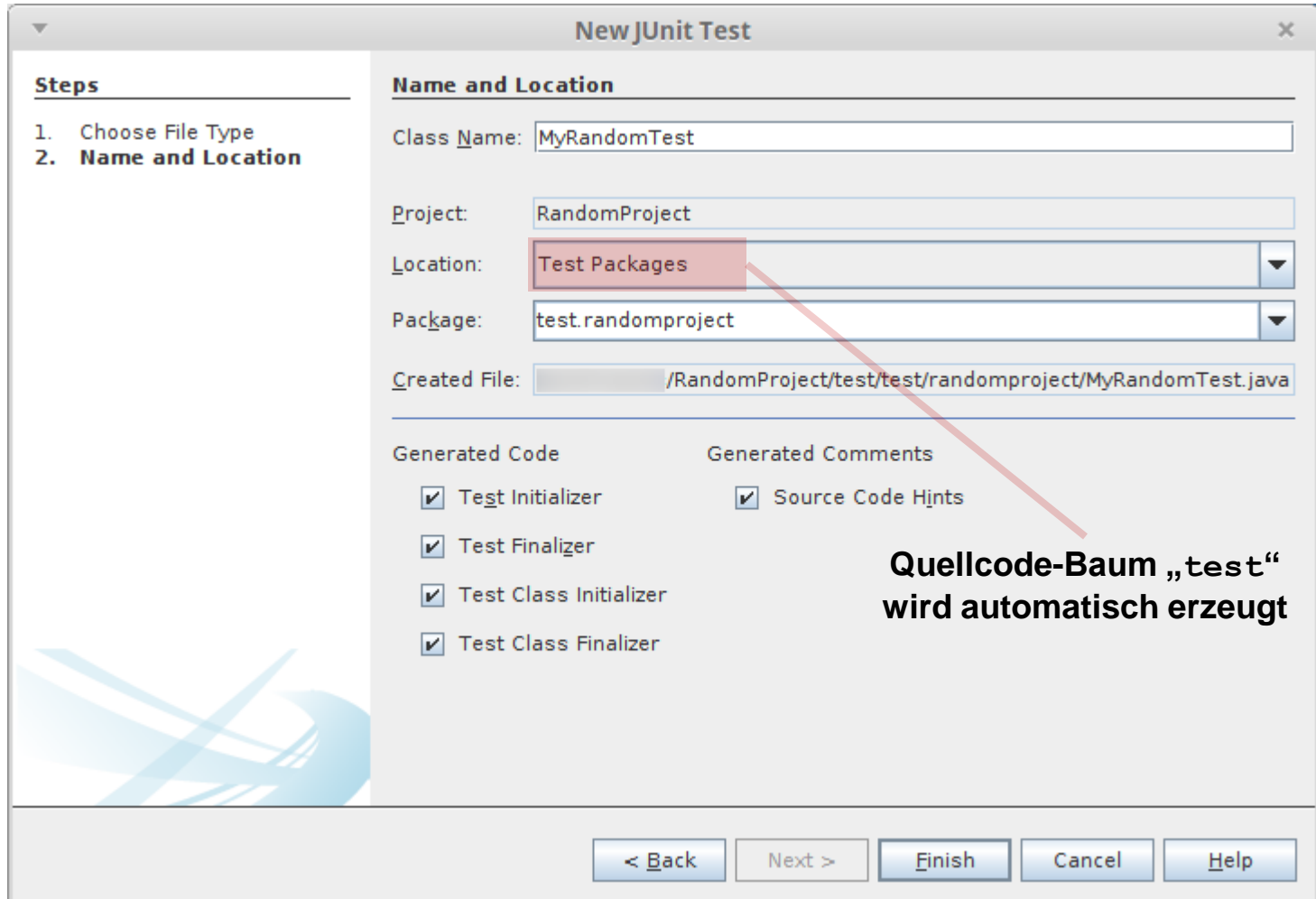
JUnit – Erster Test in NetBeans (1)

- Projekt anlegen
- In das Projekt einen „JUnit Test“ einfügen



JUnit – Erster Test in NetBeans (2)

Neuen Test anlegen



Steps

1. Choose File Type
2. **Name and Location**

Name and Location

Class Name:

Project:

Location:

Package:

Created File:

Generated Code

- ☒ Test Initializer
- ☒ Test Finalizer
- ☒ Test Class Initializer
- ☒ Test Class Finalizer

Generated Comments

- ☒ Source Code Hints

Quellcode-Baum „test“ wird automatisch erzeugt

< Back Next > Finish Cancel Help

JUnit – Erster Test in NetBeans (3)

■ Durch den Assistent erzeugtes Grundgerüst

JUnit-Imports

„Einfache Klasse“,
keine Vererbung

Annotations vor
Methodennamen

Die Test-
Methoden

```

6   package test.randomproject;
7
8   import org.junit.After;
9   import org.junit.AfterClass;
10  import org.junit.Before;
11  import org.junit.BeforeClass;
12  import org.junit.Test;
13  import static org.junit.Assert.*;
14
15  /**
16   *
17   * @author
18   */
19  public class MyRandomTest {
20
21      public MyRandomTest() {
22      }
23
24      @BeforeClass
25      public static void setUpClass() {
26      }
27
28      @AfterClass
29      public static void tearDownClass() {
30      }
31
32      @Before
33      public void setUp() {
34      }
35
36      @After
37      public void tearDown() {
38      }
39
40      // TODO add test methods here.
41      // The methods must be annotated with annotation @Test. For example:
42      //
43      // @Test
44      // public void hello() {}
45  }

```

☒ Test Class Initializer

☒ Test Class Finalizer

☒ Test Initializer

☒ Test Finalizer



Die Annotationen und ihre Bedeutung

Annotation	Beschreibung
@Test <code>public void method()</code>	Identifiziert eine Methode als Test-Methode.
@Test (expected = Exception.class)	Schlägt fehl, wenn die Methode nicht die genannte Exception wirft.
@Test(timeout=100)	Schlägt fehl, wenn die Methode länger als 100ms Laufzeit benötigt.
@Before <code>public void method()</code>	Diese Methode wird vor jedem (einzelnen) Test ausgeführt. Sie wird verwendet, um die Testumgebung vorzubereiten, z.B. Eingabewerte zu lesen oder Klassen zu initialisieren.
@After <code>public void method()</code>	Diese Methode wird nach jedem (einzelnen) Test ausgeführt. Sie wird zum „Aufräumen“ verwendet, z.B. temporäre Daten zu löschen oder Defaultwerte wiederherzustellen. Sie kann auch zum expliziten Freigeben von reserviertem Speicher verwendet werden.
@BeforeClass <code>public static void method()</code>	Diese Methode wird (einmalig) vor dem Start der Tests ausgeführt. Sie wird insb. für zeitaufwändige Aktivitäten verwendet, z.B. das Herstellen einer Datenbankverbindung und das Lesen großer Datenmengen. Die mit <code>@BeforeClass</code> annotierten Methoden müssen <code>static</code> sein!
@AfterClass <code>public static void method()</code>	Diese Methode wird (einmalig) nach Beendigung aller Tests ausgeführt. Sie wird zum „allgemeinen Aufräumen“ verwendet, z.B. das Schließen einer Datenbankverbindung. Die mit <code>@AfterClass</code> annotierten Methoden müssen <code>static</code> sein!
@Ignore <code>public void method()</code>	Die mit <code>@Ignore</code> annotierten (Test-)Methoden werden bei Tests ignoriert. Dies könnte nötig sein, wenn nach einer Codeänderung dieser Test noch nicht adaptiert wurde, oder wenn ein Test aufgrund seiner langen Ausführungszeit (temporär) ausgeschlossen werden soll.

Beispiel: Bankkonto

- Es soll ein einfaches Bankkonto (Klasse `Account`) erstellt werden.
- Die Methoden der Klasse sollen mit Hilfe von JUnit-Test überprüft werden.
- Klasse zu Beginn der Tests:

```
public class Account {  
    private long balance; // Kontostand in Eurocent  
  
    public Account(long initialBalance) {  
        this.balance = initialBalance;  
    }  
  
    public void setBalance(long balance) { // zum Testen  
        this.balance = balance;  
    }  
}
```

Beispiel Bankkonto: Methoden testen

- Es soll eine Klasse `Account` mit den folgende Methoden implementiert und getestet werden, ungültige Eingaben sollen jeweils zu einer `AccountException` führen.

```
public long getBalance()
```

```
public void inpayment(long amount) throws AccountException
```

```
// diese Methode wird Gegenstand einer Übung  
public long outpayment(long amount) throws AccountException
```

```
// diese Methode wird Gegenstand einer Übung  
public boolean isBalancePositive()
```

Test vorbereiten

■ Grundgerüst:

```
import org.junit.runner.RunWith;
import org.junit.runners.JUnit4;

import <package-name-of-account>.Account;
import <package-name-of-account>.AccountException; // vorsorglich

@RunWith(JUnit4.class) // Markiert eine Klasse als JUnit4-Test
public class AccountTest {

    private static Account account; // soll getestet werden
    private static final long INIT_ACCOUNT_BALANCE = 4711;

    public AccountTest(){

    }

}
```

@BeforeClass, @AfterClass

- Für diesen einfachen Fall ist das nicht unbedingt nötig. Hier wird nur ein neues Konto-Objekt erzeugt und eine Info auf der Konsole ausgegeben.

```
/**
 * (Einmalige) Initialisierung, vor allen Tests.
 * Ein Konto-Objekt wird erzeugt.
 */
@BeforeClass
public static void setUpAccount() {
    System.out.println("Create account.");
    AccountTest.account = new Account(INIT_ACCOUNT_BALANCE);
}

// @AfterClass wird hier nicht verwendet
```

Methodennamen
frei wählbar

@Before, @After

- Hier wird der „Standard-Kontostand“ gesetzt.

```
/**
 * Initialisieren vor jedem einzelnen Test. Der Kontostand wird auf
 * einen definierten Betrag gesetzt.
 *
 * Man könnte hier auch (statt in @BeforeClass) ein ganz
 * neues Konto anlegen.
 */
@Before
public void setUp() {
    AccountTest.account.setBalance(INIT_ACCOUNT_BALANCE);
}

// @After wird hier nicht verwendet
```

Methodennamen
frei wählbar

Die Tests (Beispiel `getBalance()`)

- Testen, ob die Abfrage des Kontostands den richtigen Betrag liefert

```
@Test
public void testAccountBalance() throws AccountException {
    assertEquals(INIT_ACCOUNT_BALANCE, account.getBalance());
}
```

Die Tests (Beispiel `inpayment()`)

- Testen, ob zwei sequentielle Einzahlung als Summe gebucht werden

```
@Test
public void inpaymentTest() throws AccountException {
    account.inpayment(500);
    account.inpayment(200);
    assertEquals(INIT_ACCOUNT_BALANCE+500+200, account.getBalance());
}
```

- Testen, ob eine negative Einzahlung zu einer Exception führt

```
@Test (expected=AccountException.class)
public void inpaymentNegativeTest() throws AccountException {
    account.inpayment(-200);
}
```

- Testen, ob eine Einzahlung von 0 den Kontostand nicht ändert

```
@Test
public void inpaymentZeroTest() throws AccountException {
    long balanceBefore = account.getBalance();
    account.inpayment(0);
    assertEquals(balanceBefore, account.getBalance());
}
```

assert-Methoden („Behauptungen“)

- Arrayvergleich:
`assertArrayEquals`
- Wertvergleich:
`assertEquals`, `assertThat`
- Referenzvergleich:
`assertSame`, `assertNotSame`
- Bedingungen:
`assertFalse`, `assertTrue`
- Null oder nicht:
`assertNull`, `assertNotNull`
- Auf jeden Fall fehlschlagen:
`fail`, z.B. `fail("Not implemented yet.");`

Wir erinnern uns: Die Tests für `inpayment()`

- Testen, ob zwei sequentielle Einzahlung als Summe gebucht werden

```
@Test
public void inpaymentTest() throws AccountException {
    account.inpayment(500);
    account.inpayment(200);
    assertEquals(INIT_ACCOUNT_BALANCE+500+200, account.getBalance());
}
```

- Testen, ob eine negative Einzahlung zu einer Exception führt

```
@Test (expected=AccountException.class)
public void inpaymentNegativeTest() throws AccountException {
    account.inpayment(-200);
}
```

- Testen, ob eine Einzahlung von 0 den Kontostand nicht ändert

```
@Test
public void inpaymentZeroTest() throws AccountException {
    long balanceBefore = account.getBalance();
    account.inpayment(0);
    assertEquals(balanceBefore, account.getBalance());
}
```

Beispiel: Implementierung von `inpayment()`

- Mit der „leeren“ Implementierung schlagen zwei Tests fehl:

```
public void inpayment(long amount) throws AccountException {  
}
```

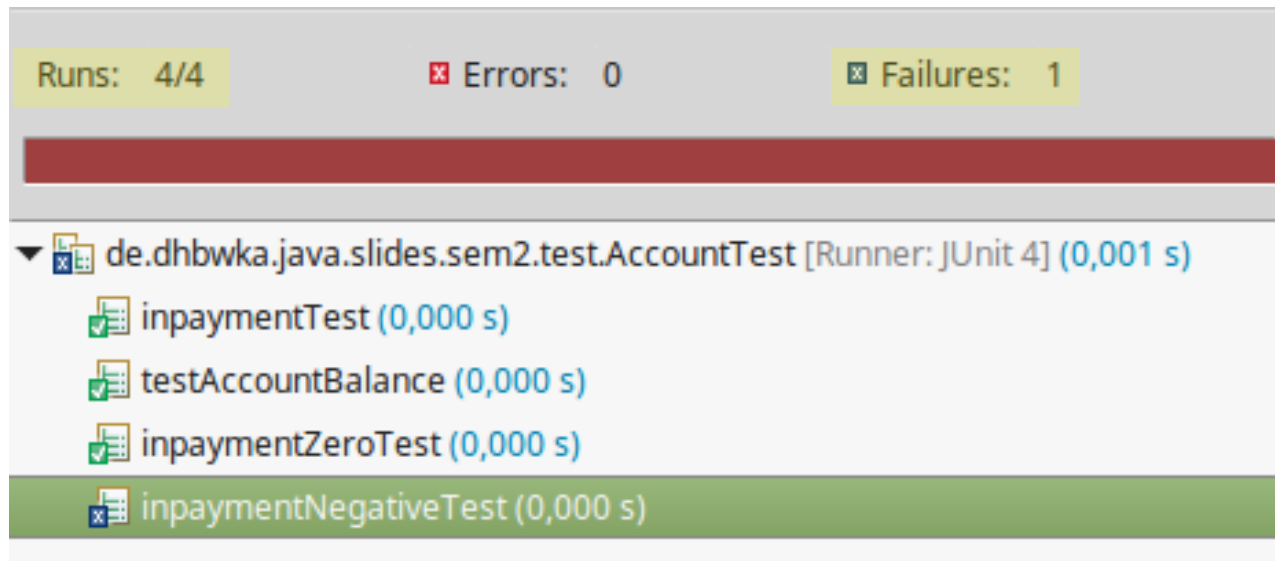
Die Methode `getBalance()` sei hier schon korrekt implementiert



Beispiel: Implementierung von `inpayment()`

- Mit dieser einfachen Implementierung laufen nur drei der vier Tests durch:

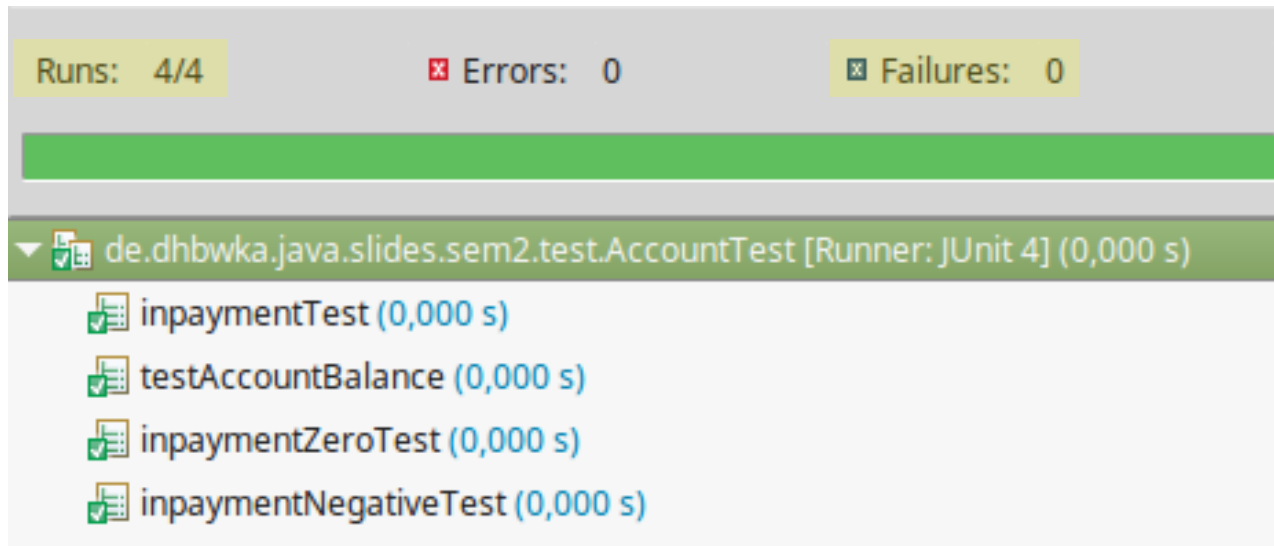
```
public void inpayment(long amount) throws AccountException {  
    this.balance += amount;  
}
```



Beispiel: Implementierung von `inpayment()`

- Eine Erweiterung lässt auch den letzten Test bestehen:

```
public void inpayment(long amount) throws AccountException {  
    if ( amount >= 0 ){  
        this.balance += amount;  
    } else {  
        throw new AccountException("No negative inpayments!");  
    }  
}
```



Aufgabe

- Implementieren Sie die Klasse `Account` und die Tests aus diesen Folien nach!
- Erweitern Sie die Klasse `Account` um die Methodenköpfe:

```
public long outpayment(long amount) throws AccountException
```



```
public boolean isBalancePositive()
```
- Schreiben Sie (jeweils mehrere) Tests für diese Methoden!
- Implementieren Sie die Methoden!

Literatur

- NetBeans-Tutorial: „Writing JUnit Tests in NetBeans IDE”
<https://netbeans.org/kb/docs/java/junit-intro.html>
- Informationen in NetBeans: Help > Javadoc References > JUnit API > Packages: `org.junit` und Unterpakete
- Eclipse-Tutorial: „Java Unit testing with JUnit 4.x in Eclipse”
<http://www.vogella.com/tutorials/JUnit/article.html>