Testen

Prof. Sven Apel

Universität des Saarlandes

Basierend auf Material von Andreas Zeller und Norbert Siegmund

Teil II

Testebenen

Testebenen

1. Unit-Tests
2. Integrationstests
3. Systemtests
4. Akzeptanztests

Unit-Tests

Ziel: Individuelle Einheiten werden getestet, um deren korrekte Funktionsweise sicherzustellen

Typischerweise *automatisiert*

Oft durch Entwickler spezifiziert

Fokus auf *eine Einheit* (Funktion/Methode/Modul)

Verwende Mocks (Stubs), falls andere Einheiten aufgerufen werden

JUnit – Unit Testing Framework

JUnit-Test ist eine Methode in einer Klasse, die nur für das Testen verwendet wird

- Annotationen markieren Methoden, die einen Test spezifizieren (`@org.junit.Test`)

Innerhalb der Methode wird das *erwartete Ergebnis* gegen das des *ausgeführten Codes verglichen*

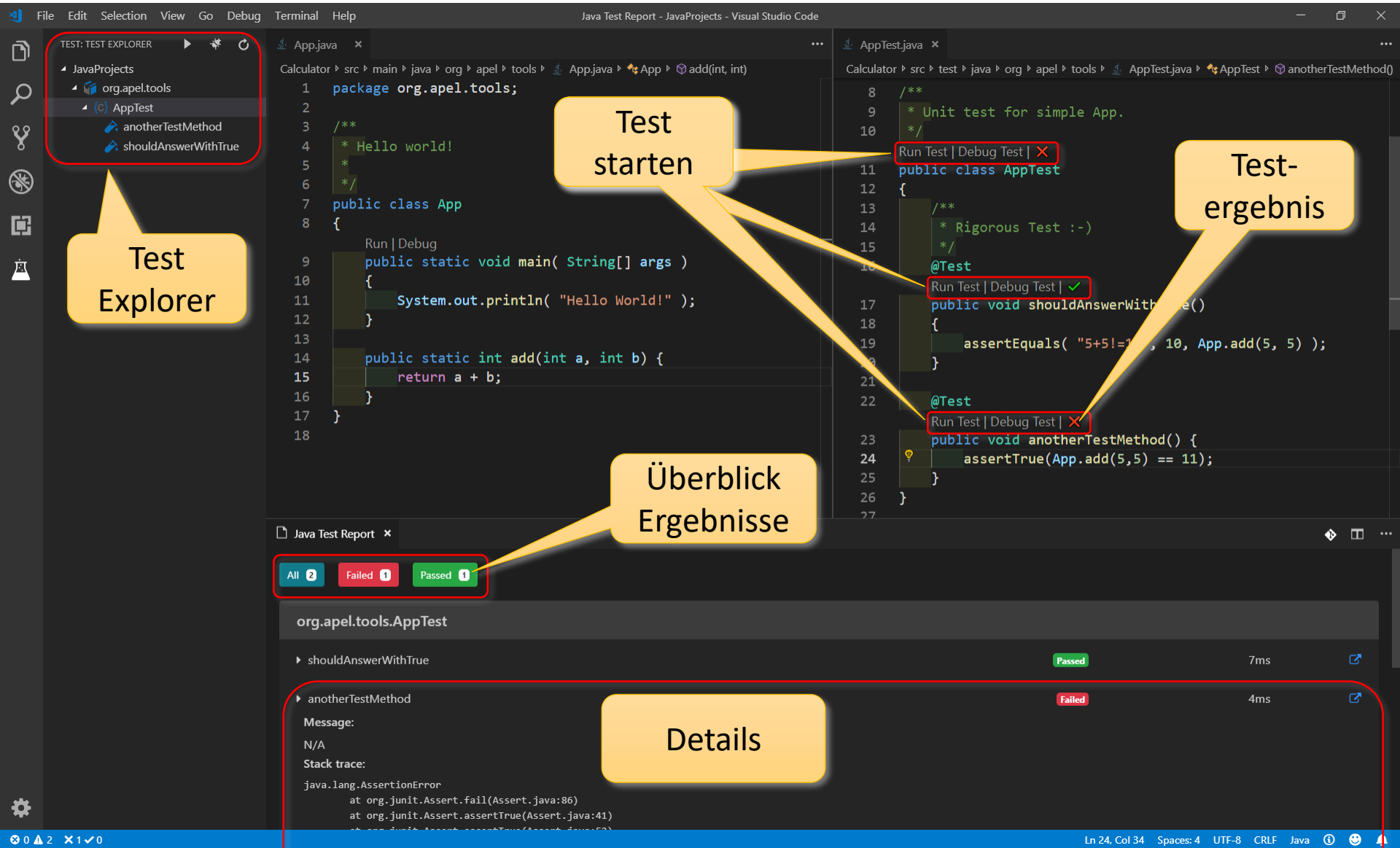
```
@Test public void multiplicationOfZeroIntegersShouldReturnZero() {
  // MyClass is tested
  MyClass tester = new MyClass();
  // Tests
  assertEquals("10 x 0 must be 0", 0, tester.multiply(10, 0));
  assertEquals("0 x 10 must be 0", 0, tester.multiply(0, 10));
  assertEquals("0 x 0 must be 0", 0, tester.multiply(0, 0));
}
```

JUnit – Methoden und Annotationen

Statement	Description
fail(message)	Let the method fail. Might be used to check that a certain part of the code is not reached or to have a failing test before the test code is implemented. The message parameter is optional.
assertTrue([message,] boolean condition)	Checks that the boolean condition is true.
assertFalse([message,] boolean condition)	Checks that the boolean condition is false.
assertEquals([message,] expected, actual)	Tests that two values are the same. Note: for arrays the reference is checked not the content of the arrays.
assertEquals([message,] expected, actual, tolerance)	Test that float or double values match. The tolerance is the number of decimals which must be the same.
assertNull([message,] object)	Checks that the object is null.
assertNotNull([message,] object)	Checks that the object is not null.
assertSame([message,] expected, actual)	Checks that both variables refer to the same object.
assertNotSame([message,] expected, actual)	Checks that both variables refer to different objects.

Annotation	Description
@Test public void method()	The `@Test` annotation identifies a method as a test method.
@Test (expected = Exception.class)	Fails if the method does not throw the named exception.
@Test(timeout=100)	Fails if the method takes longer than 100 milliseconds.
@Before public void method()	This method is executed before each test. It is used to prepare the test environment (e.g., read input data, initialize the class).
@After public void method()	This method is executed after each test. It is used to cleanup the test environment (e.g., delete temporary data, restore defaults). It can also save memory by cleaning up expensive memory structures.
@BeforeClass public static void method()	This method is executed once, before the start of all tests. It is used to perform time intensive activities, for example, to connect to a database. Methods marked with this annotation need to be defined as `static` to work with JUnit.
@AfterClass public static void method()	This method is executed once, after all tests have been finished. It is used to perform clean-up activities, for example, to disconnect from a database. Methods annotated with this annotation need to be defined as `static` to work with JUnit.
@Ignore or @Ignore("Why disabled")	Ignores the test method. This is useful when the underlying code has been changed and the test case has not yet been adapted. Or if the execution time of this test is too long to be included. It is best practice to provide the optional description, why the test is disabled.

JUnit in VS Code



Integrationstests

Module werden *kombiniert* und *als Gruppe* getestet

Nach Unit-Tests, vor Systemtests

Zweck: Erfüllen Module im *Zusammenspiel*
funktionale und nicht-funktionale Anforderungen?

Top-down oder *Bottom-up*

Systemtests

Test des *kompletten* Systems

Konzentration auf:

- Fehler, die aus Interaktionen zwischen Teilen herrühren
- Validierung, dass das gesamte System funktioniert und *nicht-funktionale Anforderungen* erfüllt

Orientierung oft an *Anwendungsfällen*

Meistens durch separates Test-Team

Viele verschiedene Arten von Systemtests

- GUI, Usability, Performance, Barrierefreiheit, Stresstests, …

Regressionstest

Idee: Nach *jeder* Änderung werden *alle* Tests ausgeführt

Sicherstellung, dass alles, was *vor* der *Änderung* funktioniert hat, auch *nach* der Änderung weiterhin funktioniert

Tests müssen *deterministisch* und *wiederholbar* sein

Tests sollten gesamte Funktionalität umfassen

- Jedes Interface
- Alle Grenzsituationen /-fälle
- Jedes Feature
- Jede Zeile Code
- Alles was irgendwie falsch gehen kann

Akzeptanztests

Erfüllt das System alle spezifizierten Anforderungen?

Echte statt simulierte Daten

Alpha-Tests: durch Entwickler und andere Teams

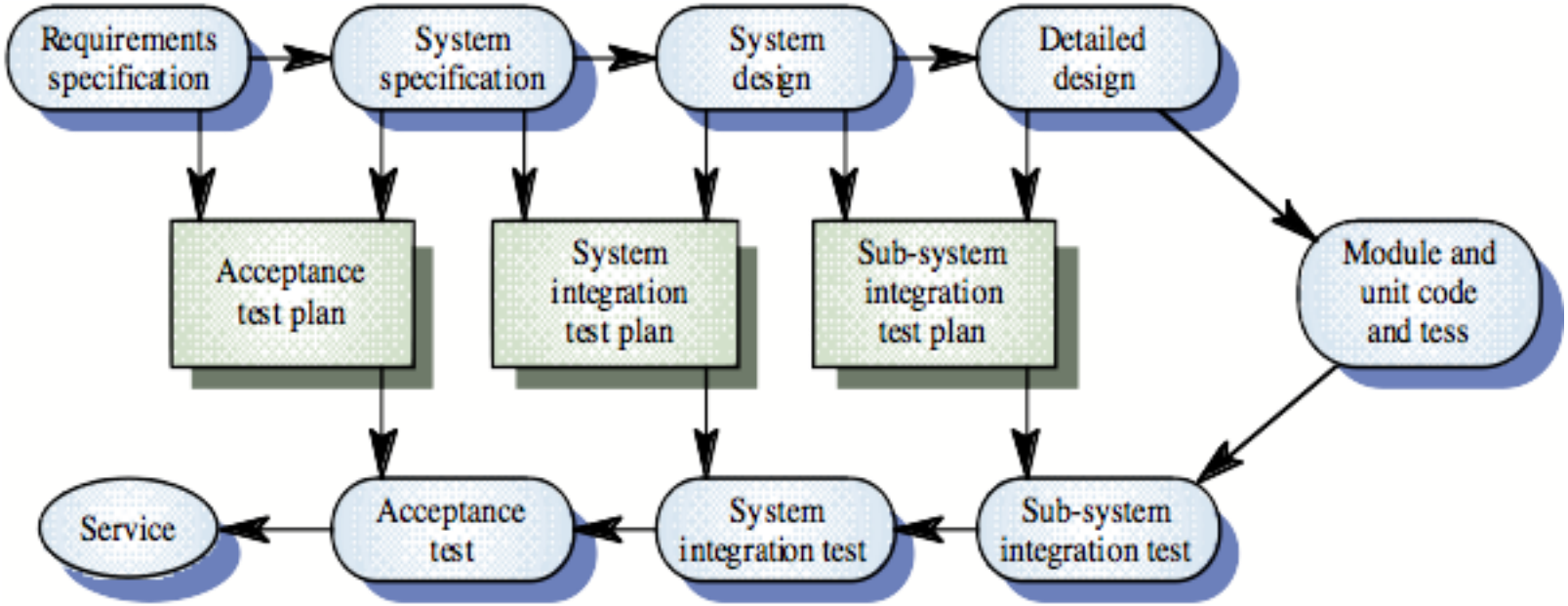
Beta-Tests: durch Endnutzer

Wann findet man die meisten Fehler?

Zeitpunkt	Rate
Unit-Test	30%
Integrationstest	35%
Systemtest	40%
Akzeptanztest	bis zu 75%

Wie spielt alles zusammen?

Testplan muss erstellt werden, *sobald die Anforderungen formuliert* sind und *ständig verfeinert* werden



Plan sollte *regulär* überarbeitet und Tests *wiederholt* und *erweitert* werden