Testen von Softwaresystemen

13. Januar 2015

Überblick

- Was umfasst das Testen von Software?
- Warum sollte man Software testen?
  - *Motivation für Software-Tests*
- Wie sollte man Software testen?
  - *Grundlegende Teststrategien und –prinzipien*
  - *Wie lässt sich das Testen in den Softwareentwicklungsprozess integrieren?*
- Wie kann man Software mit einem Test-Framework testen?
  - *Wann ist das Erstellen von automatischen Testfällen sinnvoll?*
  - *Einführung in das Test-Framework JUnit*

Testen von Software: Definition

Das **Testen** umfasst einen Prozess, in dem Unterschiede zwischen dem erwarteten Verhalten, das durch die Anforderungsspezifikation festgelegt ist, und dem beobachteten Verhalten eines implementierten Software-Moduls gefunden werden sollen.

Tests müssen sein:

- ***anforderungsbezogen***
- ***reproduzierbar***
- ***nachvollziehbar***
- ***überprüfbar***
- ***mit der Fehlersuche koppelbar***

Motivation – warum Testen?

- Besser eine Testgruppe sucht aktiv nach Fehlern als Kunden, die bei der Softwarebenutzung Fehler finden.
- Tests sind einfach durchzuführen und mindestens so wichtig wie das Programmieren selbst.
- Tests werden oftmals vernachlässigt:
  - *meist wegen Zeit- und Spaßmangel*
- Es gibt verschiedene Möglichkeiten zur Durchführung von Tests:
  - *Testausgaben: direkt in Code und Ausgabe*
  - *Debugging: schrittweises Ausführen eines Programms*
  - *Profiling: Testen von Effizienzeigenschaften*
  - *Automatisiertes Testen: Definition von Testfällen in separaten Testklassen, Zusicherung einer Testeigenschaft*

Psychologie des Testens

- „Testen ist der Prozeß, ein Programm mit der Absicht auszuführen, Fehler zu finden.“ (Myers)
- Beurteilung von Testergebnissen:
  - *erfolgreicher Testlauf: Fehler gefunden*
  - *nicht erfolgreicher Testlauf: kein Fehler gefunden*
  - *Bei einem nicht erfolgreichen Test wurden nur Zeit und Geld verschwendet?*
  - *Was ist ein erfolgreicher Testlauf, wenn es um Effizienz geht?*
- Wie kann man durch Tests zeigen, dass ein Programm das tut, was es tun soll?

Grundlegende Teststrategien

- Blackbox-Test: Für diesen Test ist das interne Verhalten und die interne Struktur des Programms nicht bekannt.
  - *Testen des Ein- / Ausgabeverhaltens*
  - *Ein vollständiges Austesten ist meist nicht möglich. (Warum?)*
  - *Wie kann man mit einer endlichen Anzahl von Testfällen maximal viele Fehler finden?*
  - *Wer sollte Blackbox-Tests durchführen?*

Grundlegende Teststrategien

- Whitebox-Test: Definition von Testfällen unter Kenntnis des internen Verhaltens und der internen Struktur des Programms
  - *Vollständiges Austesten eines Moduls nur bei vollständiger Code- und Datenüberdeckung*
  - *Ein Programm kann auch wegen fehlendem Code fehlerhaft sein.*
  - *Codeüberdeckung sagt nichts über datensensible Fehler aus.*
  - *Wieviel Zeit verbringt eine Ausführung in einer Methode?*
  - *Wieviele Objekte werden von welchem Typ angelegt?*
  - *Wer sollte Whitebox-Tests durchführen?*

Testprinzipien

(nach Myers: The Art of Software Testing)

- Ein notwendiger Bestandteil eines Testfalls ist die Definition des zu erwartenden Resultats.
- Überprüfen Sie die Ergebnisse eines jeden Tests gründlich.
- Testfälle müssen für ungültige und unerwartete ebenso wie für gültige und erwartete Ergebnisse definiert werden.
- Welche Tests sollten die Entwickler und welche unabhängige Testpersonen durchführen?

Testprinzipien

(nach Myers)

- Vermeiden Sie Wegwerftestfälle!
- Planen Sie kein Testverfahren unter der stillschweigenden Annahme, dass keine Fehler gefunden werden.
  - *Die Wahrscheinlichkeit für die Existenz weiterer Fehler ist proportional zur Zahl der bereits gefundenen Fehler.*
- Testen ist eine kreative und intellektuell herausfordernde Aufgabe.
  - *Ein guter Testfall ist dadurch gekennzeichnet, dass er mit hoher Wahrscheinlichkeit einen bisher unbekannten Fehler zu entdecken imstande ist.*
  - *Ein erfolgreicher Testfall ist dadurch gekennzeichnet, dass er einen bisher unbekannten Fehler entdeckt.*

Testen im Rahmen des Softwareentwicklungsprozesses

- Beschreibung von Testfällen bereits während der Systemanalyse
  - *Testfälle können aus den Anwendungsfalldiagramm abgeleitet werden.*
  - *Für jeden Anwendungsfall sind bereits Testszenarien zu beschreiben.*
- Testen kleinerer und größerer Einheiten
  - *Testen und Programmieren gehen Hand in Hand*
- wesentliche Testarten:
  - *Klassentest: Eine Klasse wird separat getestet. Darin befinden sich Methodentests.*
  - *Systemtest: Das Gesamtsystem wird getestet.*
- Fazit: Das Testen begleitet den gesamten Softwareentwicklungsprozess.

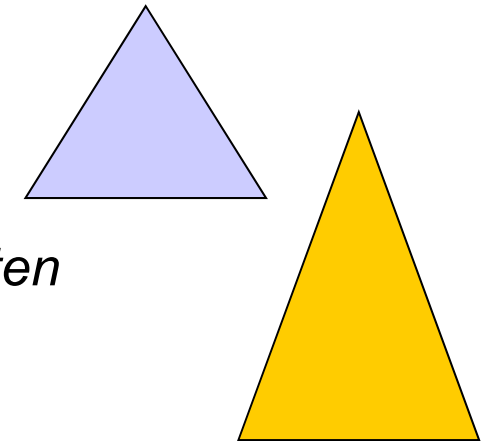
Ein erster Test

Wie kann man das folgende Programm testen?

Das Programm liest drei Werte ein. Diese werden als Längen von Dreiecksseiten interpretiert. Das Programm druckt eine Meldung mit der Feststellung aus, ob das Dreieck ungleichseitig, gleichschenklig oder gleichseitig ist.

(aus „Myers: Methodisches Testen von Programmen“)

- *gleichseitig: drei gleich lange Seiten*
- *gleichschenklig: mindestens zwei gleich lange Seiten*



Mögliche Testfälle

Kategorien von Testfällen:

- 3 ganze Zahlen für ein zulässiges ungleichseitiges Dreieck
- 3 ganze Zahlen für ein zulässiges gleichseitiges Dreieck
- 3 ganze Zahlen für ein zulässiges gleichschenkliges Dreieck
- 3 ganze Zahlen: eine Seite gleich Null, alle drei Seiten gleich Null
- 3 ganze Zahlen: mind. eine Seite hat einen negativen Wert
- 3 ganze Zahlen: die Summe zweier Zahlen ist gleich oder kleiner der dritten
- auch nicht ganzzahlige Werte
- nicht typgerechte Eingabewerte

Beispiel für einen konkreten Testfall:

- Eingabe: 4,4,2
- Testbedingung: Die Ausgabe ist: „Das Dreieck ist gleichschenklig.“

Warum automatisiertes Testen?

- Wenn sich ein Softwaresystem häufig ändert, muss es bei jeder Änderung getestet werden.
  - *Immer wieder dieselben Tests durchführen ist langweilig.*
  - *Deshalb: das Testen automatisieren*
- Extreme Programming (XP):
  - *schnell lauffähige Software*
  - *viele Änderungen der Software in kurzen Zyklen*
- Eine XP-Methode: der Test-First-Ansatz
  - *„Erst Denken, dann Programmieren.“*
  - *„Erst Testen, dann Programmieren.“*
  - *Testen und Programmieren gehen Hand in Hand:*

  „Test a little, write a little, test a little, write a little.“

Automatisiertes Testen mit JUnit

- Was ist JUnit?
- Features von JUnit
- Wie testet man mit JUnit?
- Ein Beispieltest
- JUnit und Eclipse

Was ist JUnit?

- JUnit
  - *ist ein Test-Framework.*
  - *bietet Klassen an, um geschriebenen Quelltext leicht zu prüfen.*
  - *unterstützt automatisiertes Testen, d.h. es verlangt während der Tests keine Benutzerinteraktion.*
  - *verlangt ein wenig Disziplin.*
  - *ist einfach anzuwenden.*
- Was ist JUnit nicht?
  - *Ein Wundermittel – die Tests schreiben sich nicht von selbst.*
- Mehr Informationen: **`www.junit.org`**

JUnit - Prinzipien

- JUnit unterstützt
  - *das automatisches Testen: Zeitersparnis bei häufigen Änderungen*
  - *im Wesentlichen Klassen- und Modultests*
  - *Greybox-Tests: grobe Programmstrukturen müssen bekannt sein.*
- Grundlagen:
  - *Zu jeder verfassten Klasse eine Testklasse entwerfen. Diese enthält Testfälle zu dieser Klasse.*
- Ein Testfall besteht aus
  - *Aufbau einer Testumgebung*
  - *Aufruf der zu testenden Methode*
  - *ein oder mehrere Testbedingungen*

Welche Unterstützung bietet Junit?

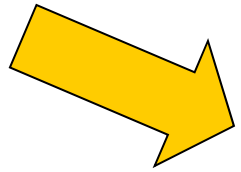
- Installation:
  - *Archiv* ***junit.jar*** *dem* ***CLASSPATH*** *hinzufügen.*
  - *z.B.:* ***classpath=%classpath%;INSTALL_DIR\junit-4.5.jar;INSTALL_DIR***
  - *Installationstest:* ***java org.junit.runner.JUnitCore org.junit.tests.AllTests***
- Zum Schreiben von Tests werden benötigt:
  - Zur Definition von Tests: ***org.junit.Test***
  - Zur Definition von Testbedingungen: ***org.junit.Assert***
  - (Zur Definition von Testsuiten: ***org.junit.TestSuite***)
- Ausführen von Tests:
  - ***java org.junit.runner.JUnitCore TestClass1.class [...weitere Testklassen...]***

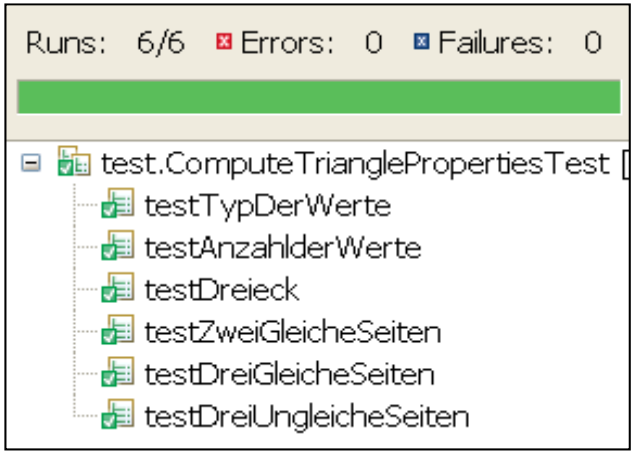
Beispiel: Gleichseitige Dreiecke

```
public String computeProperties(String[] args) {
    if (args.length != 3)
        returnString = "Die Anzahl der eingegebene Werte ist nicht gleich 3.";
    else {
        arg0 = args[0];
        arg1 = args[1];
        arg2 = args[2];

        try {
            a = Integer.parseInt(arg0);
            b = Integer.parseInt(arg1);
            c = Integer.parseInt(arg2);

            if (a > 0 && b > 0 && c > 0) {
                System.out.println("Eingebene Werte: " + a + " " + b + " "
                        + c);
                if ((a + b < c) || (a + c < b) || (b + c < a))
                    returnString = "Die eingegebenen Zahlen ergeben kein gültiges Dreick.";
                else if ((a == b) || (a == c) || (b == c)) {
                    returnString = "Das Dreieck ist gleichschenklig.";
                    if (a == b && b == c)
                        returnString = "Das Dreieck ist gleichseitig.";
                } else
                    returnString = "Das Dreieck ist nicht gleichschenklig.";
            } else
                returnString = "Mindestens ein Wert ist kleiner oder gleich Null.";
        } catch (NumberFormatException e) {
            returnString = "Mindestens eine Eingabe ist keine ganze Zahl.";
        }
    }
    return returnString;

}
```

Beispiel: Testfallklasse

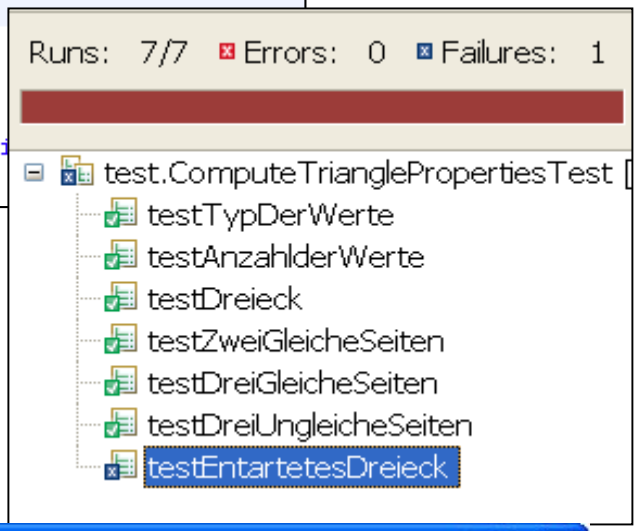
```
package test;

import static org.junit.Assert.*;

public class ComputeTrianglePropertiesTest {
    ComputeTriangleProperties computation = null;
    String result;

    @Before
    public void setUp() throws Exception {
        computation = new ComputeTriangleProperties();
    }

    @After
    public void tearDown() throws Exception {
    }

    @Test
    public void testTypDerWerte() {
        String[] args = {"a", "b", "c"};
        result = computation.computeProperties(args);
        assertEquals(result,"Mindestens eine Eingabe ist keine ganze Zahl." );
    }

    @Test
    public void testAnzahlderWerte() {
        String[] args = {"1"};
        result = computation.computeProperties(args);
        assertEquals(result,"Die Anzahl der eingegebene Werte ist nicht gleich 3." );
    }
```

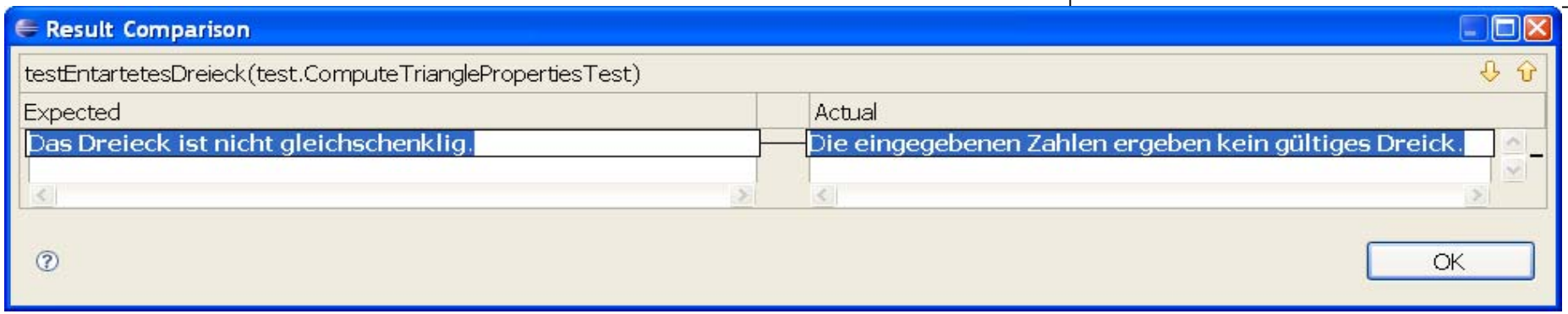




Beispiel: Fehlgeschlagener Testfall

```
@Test
public void testEntartetesDreieck() {
    String[] args = {"3", "2", "1"};
    result = computation.computeProperties(args);
    assertEquals(result, "Die eingegebenen Zahlen ergeben kein gülti
}
```





Testausgabe (ohne Eclipse)

```
JUnit version 4.5
...Eingebene Werte: 1 2 4
.Eingebene Werte: 1 2 2
.Eingebene Werte: 2 2 2
.Eingebene Werte: 3 2 4
.Eingebene Werte: 3 2 1
E.
Time: 0,031
There was 1 failure:
1) testEntartetesDreieck(test.ComputeTrianglePropertiesTest)
org.junit.ComparisonFailure: expected:<D[ie eingegebenen Zahlen
   ergeben kein gültiges Dreick].> but was:<D[as Dreieck ist nicht
   gleichschenklig].>
        at org.junit.Assert.assertEquals(Assert.java:123)
        at org.junit.Assert.assertEquals(Assert.java:145)
        at
   test.ComputeTrianglePropertiesTest.testEntartetesDreieck(Compute
   TrianglePropertiesTest.java:70)
....
```

Grundsätzlicher Testablauf

- Jeder Test wird grundsätzlich gekapselt:
  - *Vor jedem Test können die Werte in separaten Methoden initialisiert und nach dem Test aufgeräumt werden. Dazu werden die Methoden mit denTags @Before und @After annotiert.*
  - *Mit diesen Methoden wird ein grundsätzliches Testszenario aufgebaut und nach dem Test wieder abgebaut.*
  - *Beispiel: Öffnen und Schliessen einer Datei*
- Testablauf:
  - *optional: @Before setUp()*
  - *Aufruf der zu testenden Methode, eingeleitet mit @Test*
  - *In der Test-Methode: Überprüfung, ob das Ergebnis eines Tests mit einer Behauptung übereinstimmt. Dazu wird eine Assert-Methode aufgerufen.*
  - *optional: @After tearDown()*

Assert und Fail

- Assertion: Testbedingung, die erfüllt sein muss.
- Beispiele für Arten von Testbedingungen:
  - ***`True`***: *Bedingung ist wahr.*
  - ***`False`***: *Bedingung ist falsch.*
  - ***`Null`***: *Objekt gleich Null.*
  - ***`NotNull`***: *Objekt nicht gleich Null.*
  - ***`Same`***: *Objekte stimmen überein.*
  - ***`NotSame`***: *Objekte stimmen nicht überein.*
  - ***`Equals`***: *Ruft* ***`Object.equals`*** *auf.*
- Beispiele für Testbedingungen:
  - ***`assertTrue(expected.equals(result));`***
  - ***`assertEquals(a,b);`***
  - ***`assertEquals("a=b", a, b);`*** ***(mit Kommentar)***
- **`AssertionFailedError`**, wenn Test fehlschlägt.
- **`Fail:`** Test schlägt fehl

Fehler oder Fehler?

- JUnit unterscheidet zwei Arten von Fehlern:
  - failures*: Fehler, die durch die negative Auswertung einer zuvor gestellten Behauptung entstanden sind.*
  - errors*: Fehler, die unerwartet entstanden sind, wie z.B. eine* **`ArrayIndexOutOfBoundException`**
- Die Klasse **`TestFailure`** dient nur zur Speicherung der Fehler im Vector.

Testsuiten

- Testsuiten dienen dazu, verschiedene Tests in einer bestimmten Reihenfolge aufzurufen.
- Suiten können dazu verwendet werden, verschiedene Klassen eines Paketes bzw. Projektes auf einmal zu testen.
  - *Pakettests, Klassentests, Methodentests*
- Mit den Tags @RunWith und @SuiteClasses werden die beteiligten Testklassen angegeben.

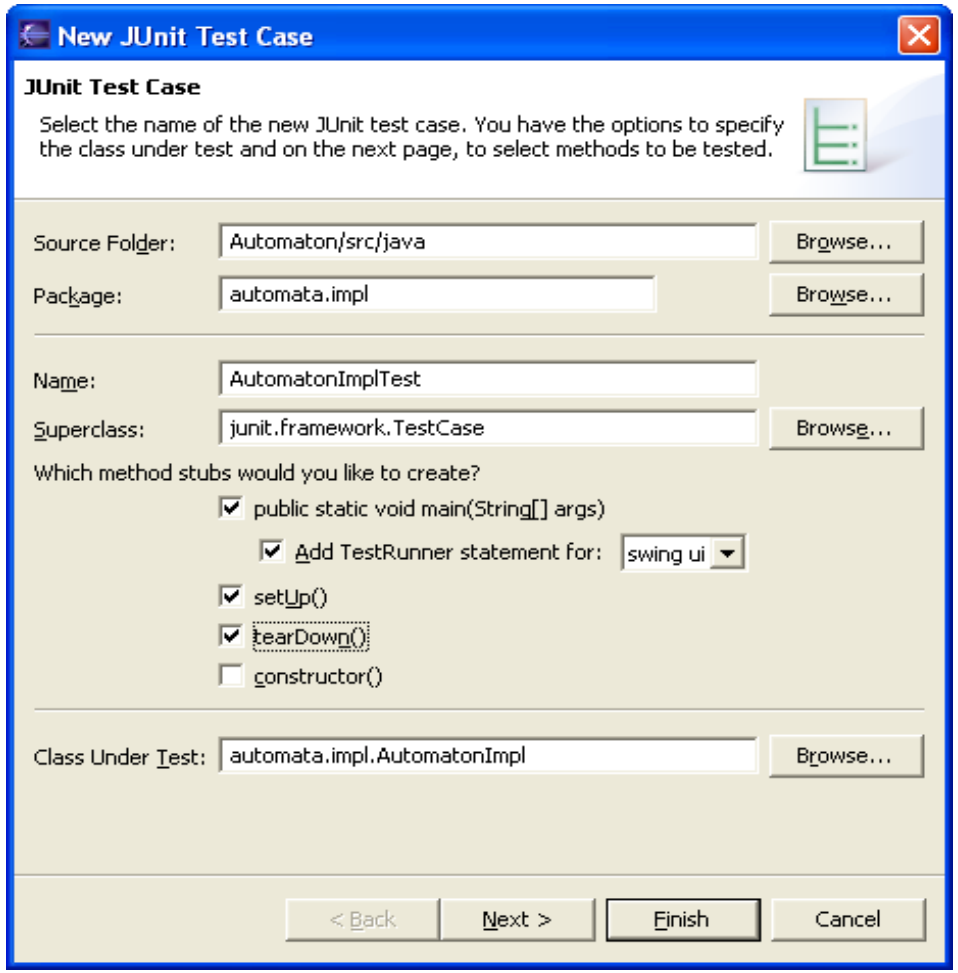
```
@RunWith(Suite.class)
@SuiteClasses({ ComputeTrianglePropertiesTest.class })
public class AllTests {

}
```

Entwicklung und Tests

- Wann sollte getestet werden?
  - *Möglichst gleich nach dem Kompilieren.*
  - *Auf diese Weise wissen wir schnell, wo der Code funktioniert, und wo nicht.*
- Wie soll man entwickeln?
  - *Ein wenig Testen, ein wenig Entwickeln.*
  - *Die Anforderungen an den zu schreibenden Code sind nie mehr so genau bewusst, wie während des Programmierens.*
  - *Deshalb: Test schreiben, Code schreiben, Kompilieren und gleich Testen.*
- Parallele Teststruktur anlegen:
  - *src: enthält die eigentlichen Sourcen*
  - *test: enthält die gleiche Paketstruktur wie src*
  - *Name der Testklasse: <Klasse>Test*

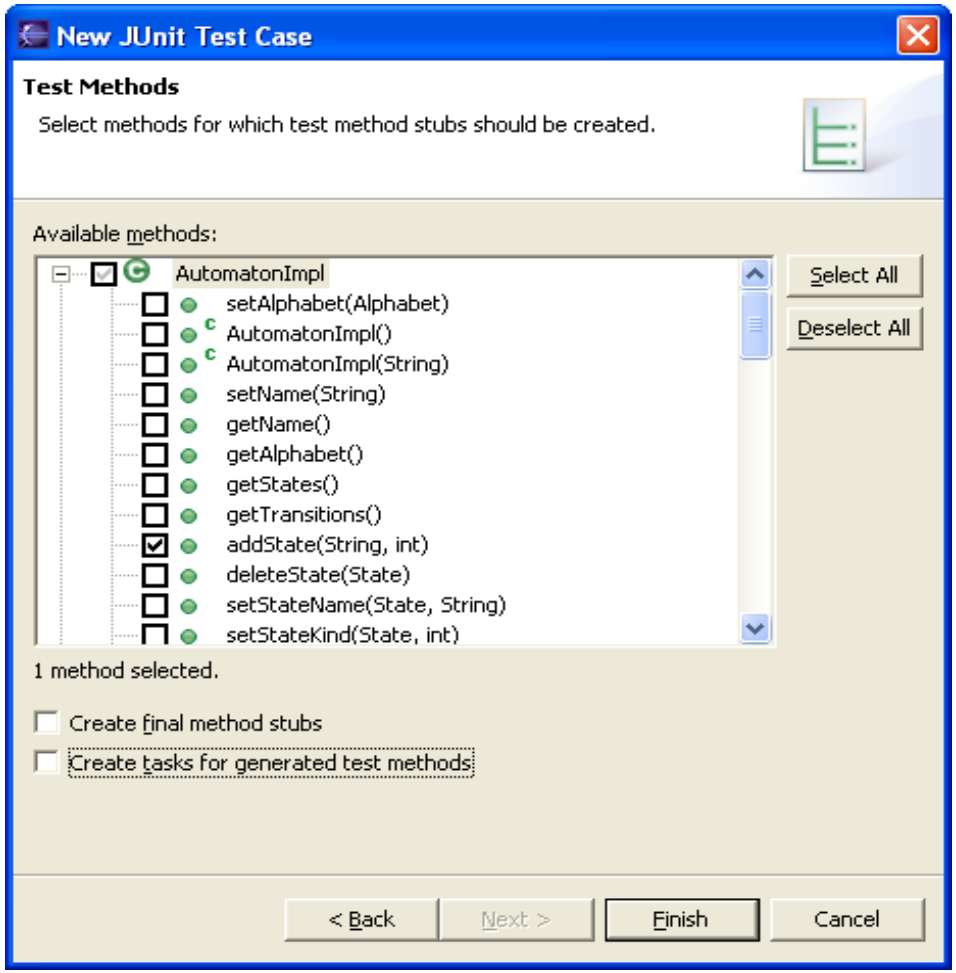
JUnit und Eclipse

- Testklasse anlegen:
  - *File → new → JUnit Test Case*



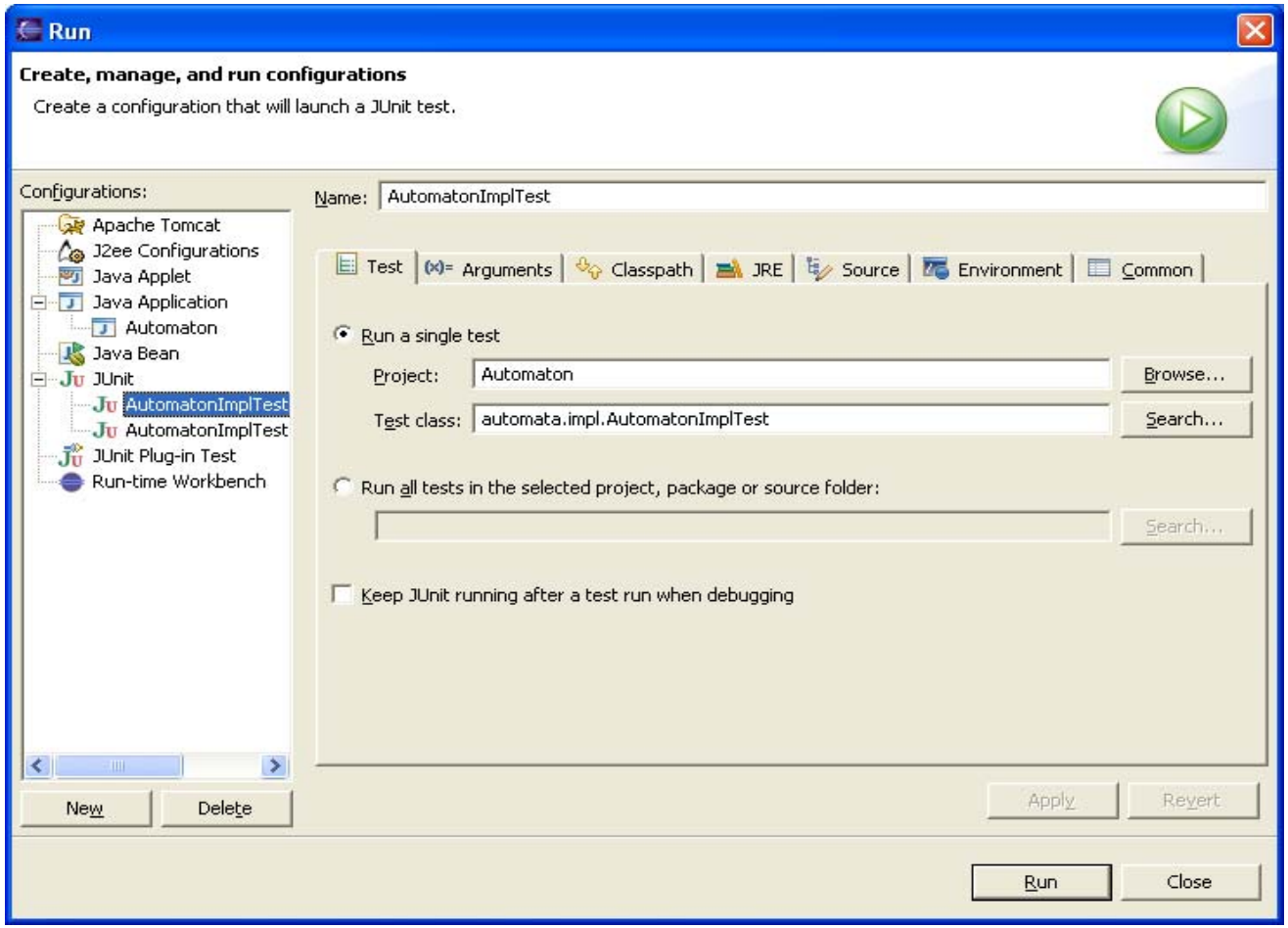
JUnit und Eclipse

- Auswahl der zu testenden Methoden



JUnit und Eclipse

- Durchführen von Tests:
- zu testende Klasse:
- Run → Run As → JUnit Test
- Alternativ auch zu testen:
  - *Paket*
  - *Methode*



Zusammenfassung

- Teststrategien:
  - *Blackbox-Test: Testen des Ein-/Ausgabeverhaltens*
  - *Whitebox-Test: Testen des bekannten Codes*
- Testen im Softwareentwicklungsprozess:
  - *Anwendungsfälle → Testfälle*
  - *Test-First-Ansatz: Erst Testen, dann Implementieren.*
- Automatisierte Tests unterstützen den Entwickler während der gesamten Entwicklung:
  - *Schnelle Wiederverwendung von Tests nach Codeänderung*
  - *Automatisierte Tests verlangen, nachdem sie einmal verfasst wurden, vom Entwickler kein aktives Denken mehr (außer im Fehlerfall).*
  - *Tests sind vom eigentlichen Code getrennt*
- JUnit: Defacto-Standard für die Definition von automatisierten Testfällen