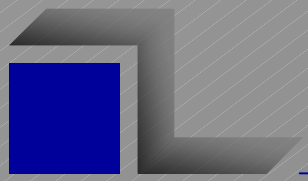
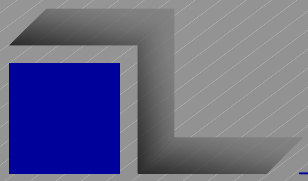


Testen mit JUnit

- Motivation
  - Test First
  - Design for Test
- Vorbereitung in Eclipse
- Testfall (eigentlich: Testklasse) zu einer Klasse erstellen
- Beispiel zur Demonstration
- Test-Szenario
- Prüf-Methoden
- Diverse Ergänzungen

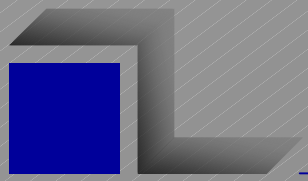


Test First

Testen mit JUnit

- Motivation
- Eclipse-Vorbereitung
- Testfall erstellen
- Testklasse ausführen
- Demo-Beispiel
- Test-Szenario
- Prüf-Methoden
- Diverse
- ...
- ...
- ...
- ...

- "Immer dann, wenn Du in Versuchung kommst, etwas wie eine print-Anweisung oder einen Debugger-Ausdruck zu schreiben, schreibe das stattdessen als Test." [Martin Fowler]
- Beispiele
  - Während der Entwicklung: Schreibe vor dem Hinzufügen neuer Funktionalität zuerst Code, der diese neue Funktionalität benutzt
  - Bei der Fehlersuche, beim Debugging: Wenn ein Fehler entdeckt wurde schreibe Testcode, der den Fehler nachvollzieht, also zuerst fehlschlägt, und erst dann funktioniert, wenn der Fehler beseitigt wurde

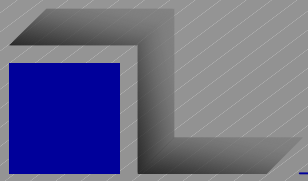


Partner-Diskussion: Test First

Testen mit JUnit

- Motivation
- Eclipse-Vorbereitung
- Testfall erstellen
- Testklasse ausführen
- Demo-Beispiel
- Test-Szenario
- Prüf-Methoden
- Diverse
- ...
- ...
- ...
- ...

- Diskutieren Sie mit einem Partner
  - Welche Vorteile bietet es, wenn man Tests unmittelbar beim oder sogar vor dem Programmieren schreibt?
  - Welche Nachteile stehen dem gegenüber?
- Dauer: 3 Minuten



Design for Test

Testen mit JUnit

- Motivation
- Eclipse-Vorbereitung
- Testfall erstellen
- Testklasse ausführen
- Demo-Beispiel
- Test-Szenario
- Prüf-Methoden
- Diverse
- ...
- ...
- ...
- ...

- Design for Test bedeutet
  - Der Fokus beim Entwickeln liegt auf der Schnittstelle, nicht (nur) der Funktionalität
  - Der Code muss bequem aufrufbar sein
  - Der Code muss testbar sein

- "Möglicherweise ist der wichtigste Vorteil von all diesen Tests die Auswirkung auf Architektur und Design. Um ein Modul oder eine Anwendung testbar zu machen, muss sie geeignet entkoppelt sein. *Je testbarer* es ist, desto modularer (desto besser gekapselt) ist die Software.
  Die Berücksichtigung umfassender Akzeptanz- und Unit-Tests hat einen deutlich positiven Effekt auf die Struktur der Software."

  [Robert Martin, Agile Software Development. Prentice Hall 2003]

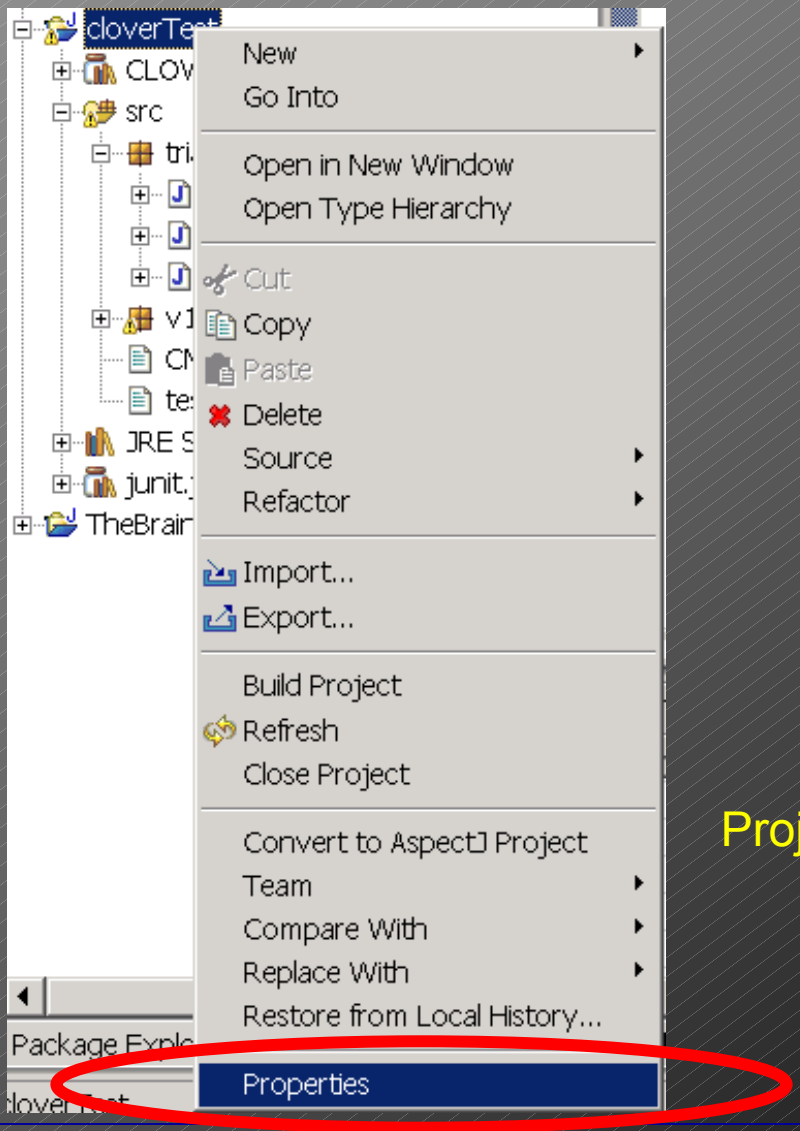
Fragen?

- Demo
  - JUnit in Eclipse-Projekt einbinden
  - OO -> test.junit -> Calculations

Vorbereitung in Eclipse: Bibliothek einbinden 1/3

Testen mit JUnit

- Motivation
- Eclipse-Vorbereitung
- Testfall erstellen
- Testklasse ausführen
- Demo-Beispiel
- Test-Szenario
- Prüf-Methoden
- Diverse
- ...
- ...
- ...
- ...

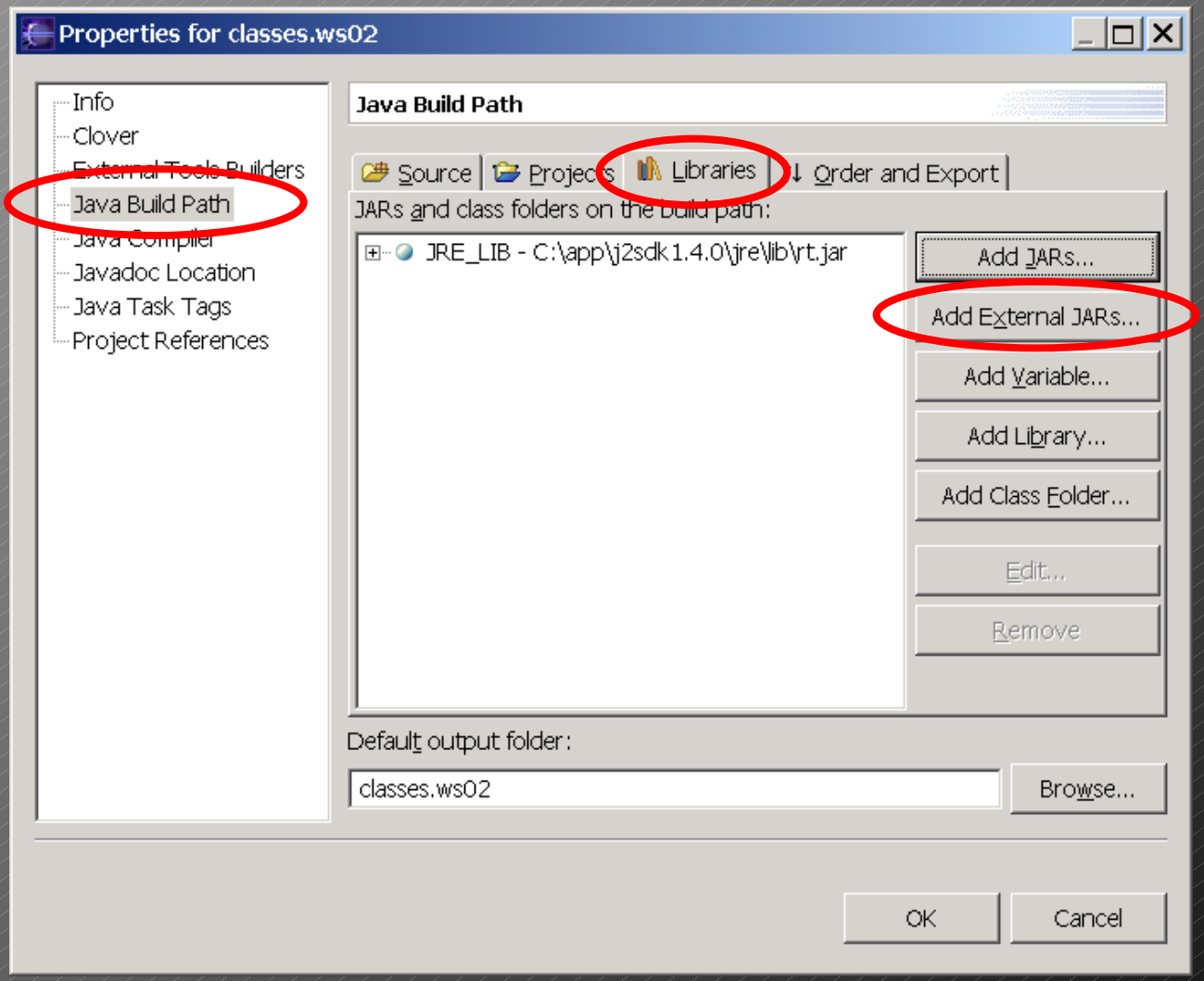


Project -> Properties

Vorbereitung in Eclipse: Bibliothek einbinden 2/3

Testen mit JUnit

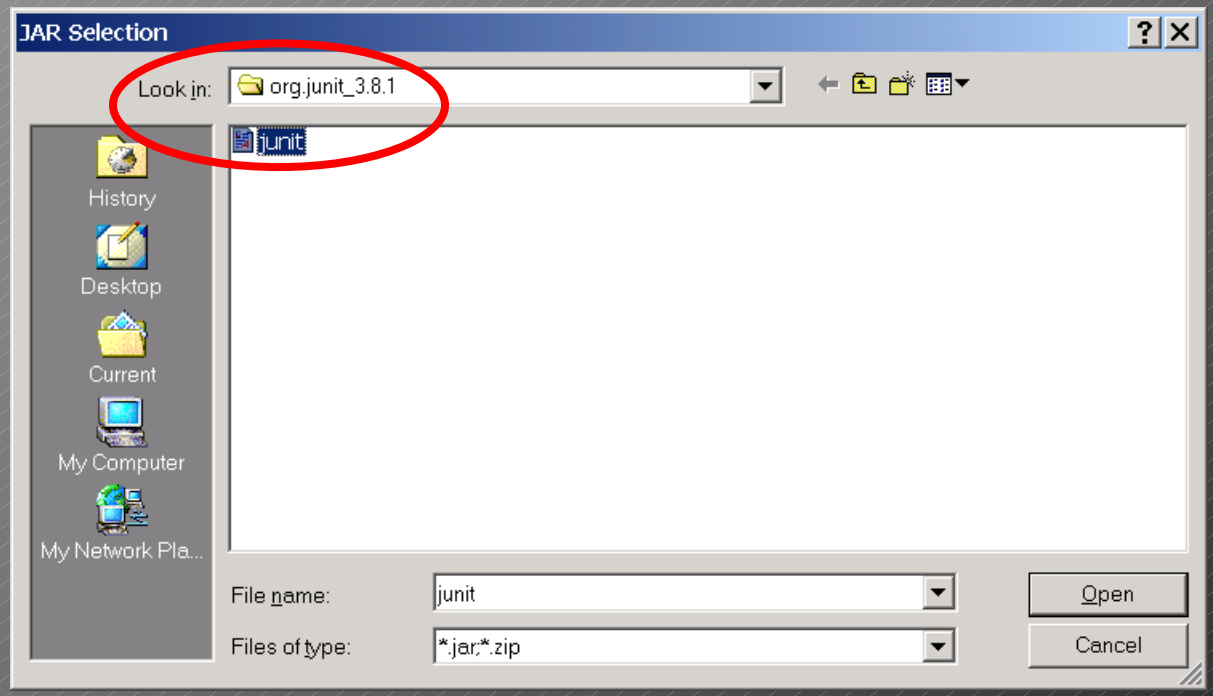
- Motivation
- Eclipse-Vorbereitung
- Testfall erstellen
- Testklasse ausführen
- Demo-Beispiel
- Test-Szenario
- Prüf-Methoden
- Diverse
- ...
- ...
- ...
- ...



Vorbereitung in Eclipse: Bibliothek einbinden 3/3

Testen mit JUnit

- Motivation
- Eclipse-Vorbereitung
- Testfall erstellen
- Testklasse ausführen
- Demo-Beispiel
- Test-Szenario
- Prüf-Methoden
- Diverse
- ...
- ...
- ...
- ...

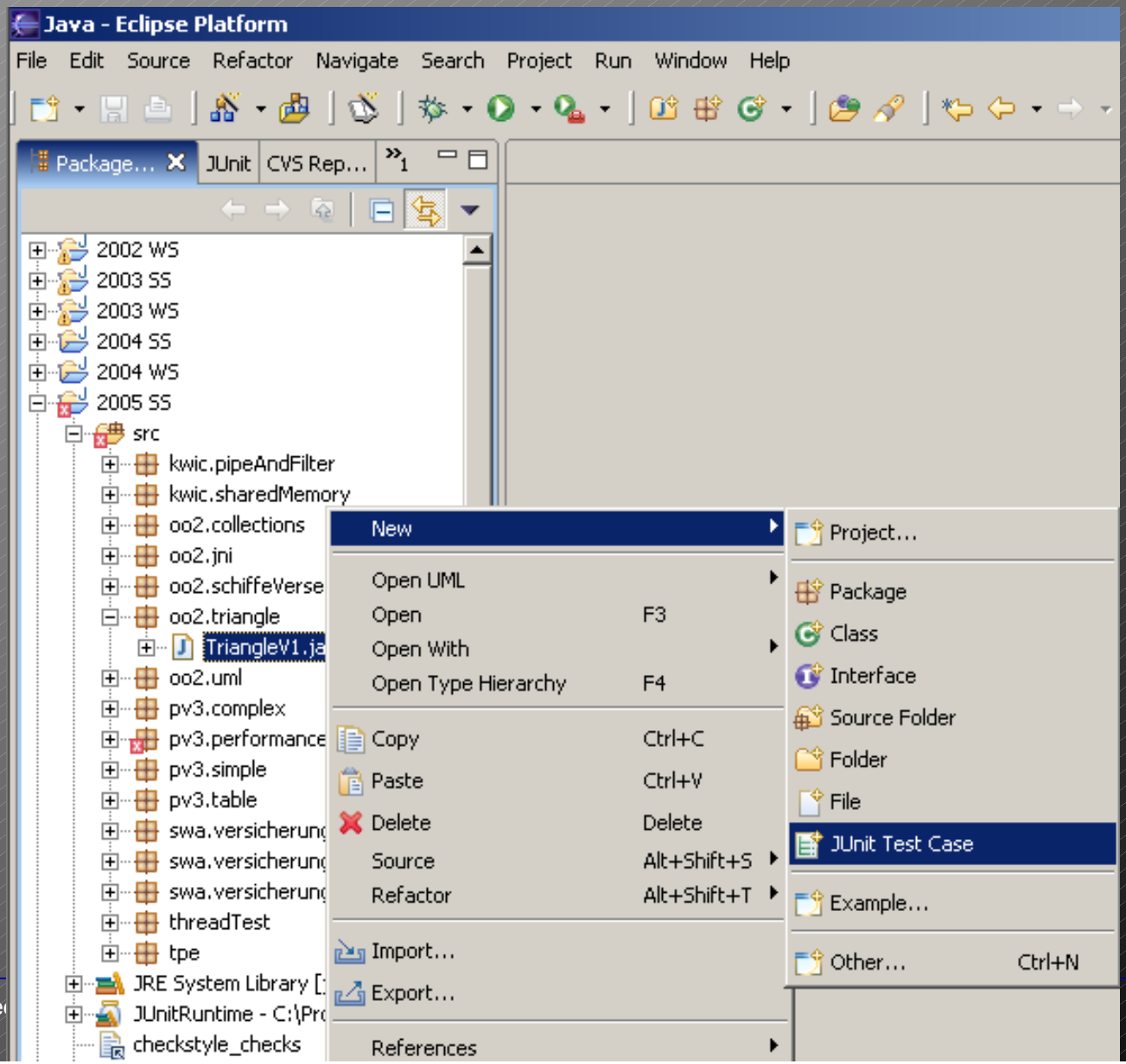


C:\Program Files\eclipse\plugins

Testfall zu einer Klasse erstellen (eigentlich: Test*klasse* mit Testfällen erstellen)

Testen mit JUnit

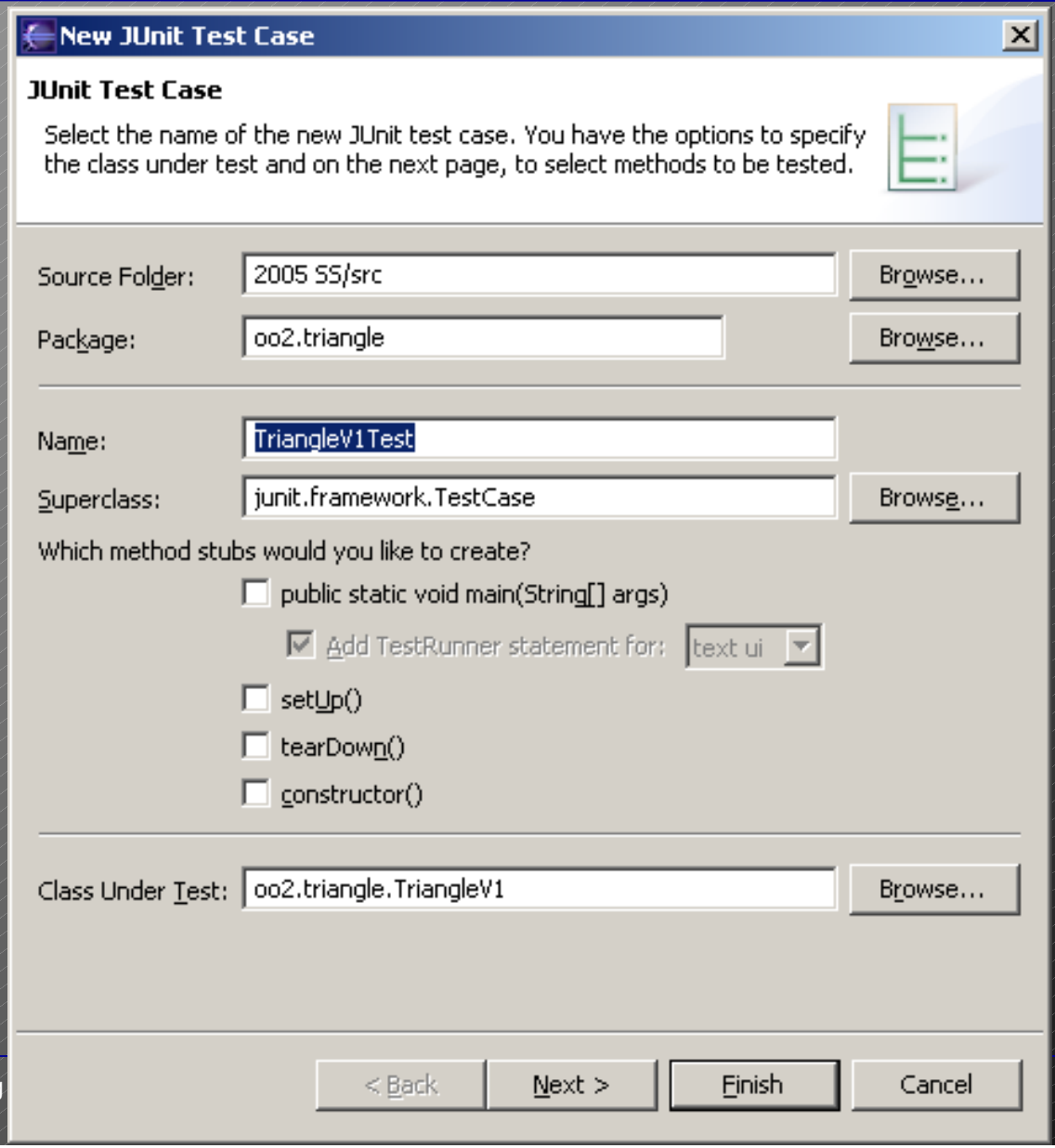
- Motivation
- Eclipse-Vorbereitung
- Testfall erstellen
- Testklasse ausführen
- Demo-Beispiel
- Test-Szenario
- Prüf-Methoden
- Diverse
- ...
- ...
- ...
- ...



Testfall zu einer Klasse erstellen: Name der Testklasse wählen

Testen mit JUnit

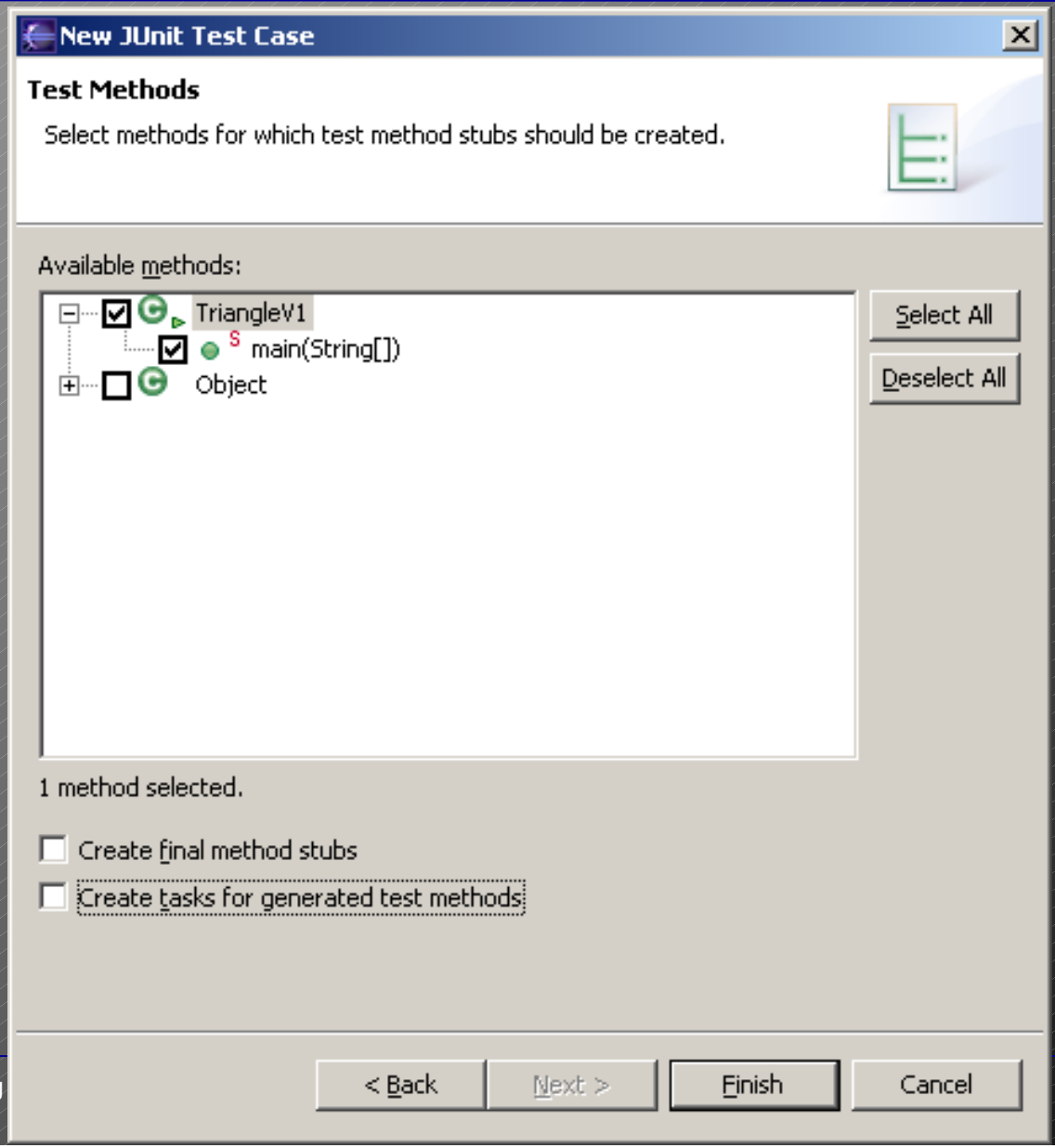
- Motivation
- Eclipse-Vorbereitung
- Testfall erstellen
- Testklasse ausführen
- Demo-Beispiel
- Test-Szenario
- Prüf-Methoden
- Diverse
- ...
- ...
- ...
- ...



Testfall zu einer Klasse erstellen: Methoden auswählen, die getestet werden sollen

Testen mit JUnit

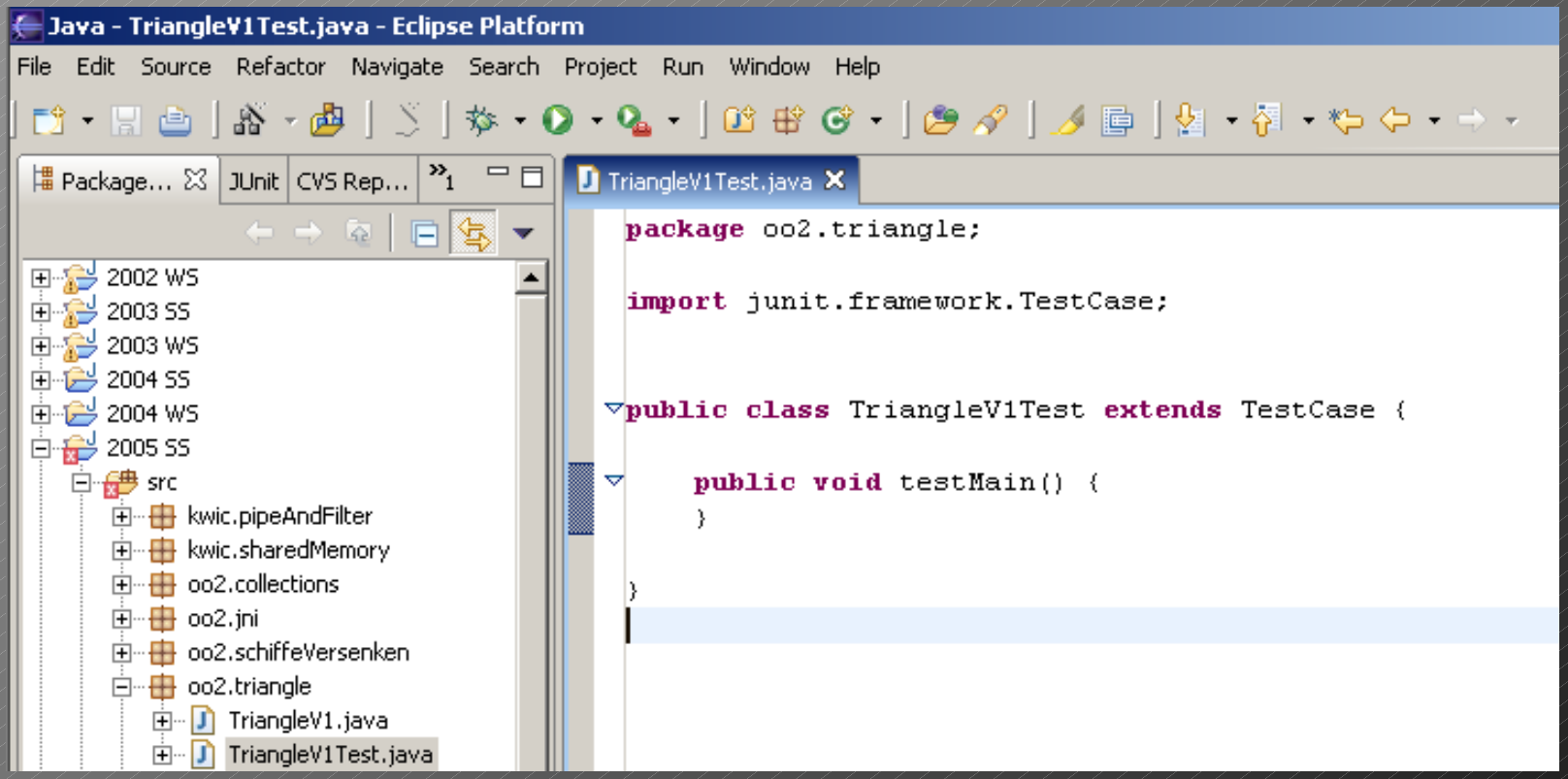
- Motivation
- Eclipse-Vorbereitung
- Testfall erstellen
- Testklasse ausführen
- Demo-Beispiel
- Test-Szenario
- Prüf-Methoden
- Diverse
- ...
- ...
- ...
- ...



Testfall zu einer Klasse erstellen: generierter Programmcode

Testen mit JUnit

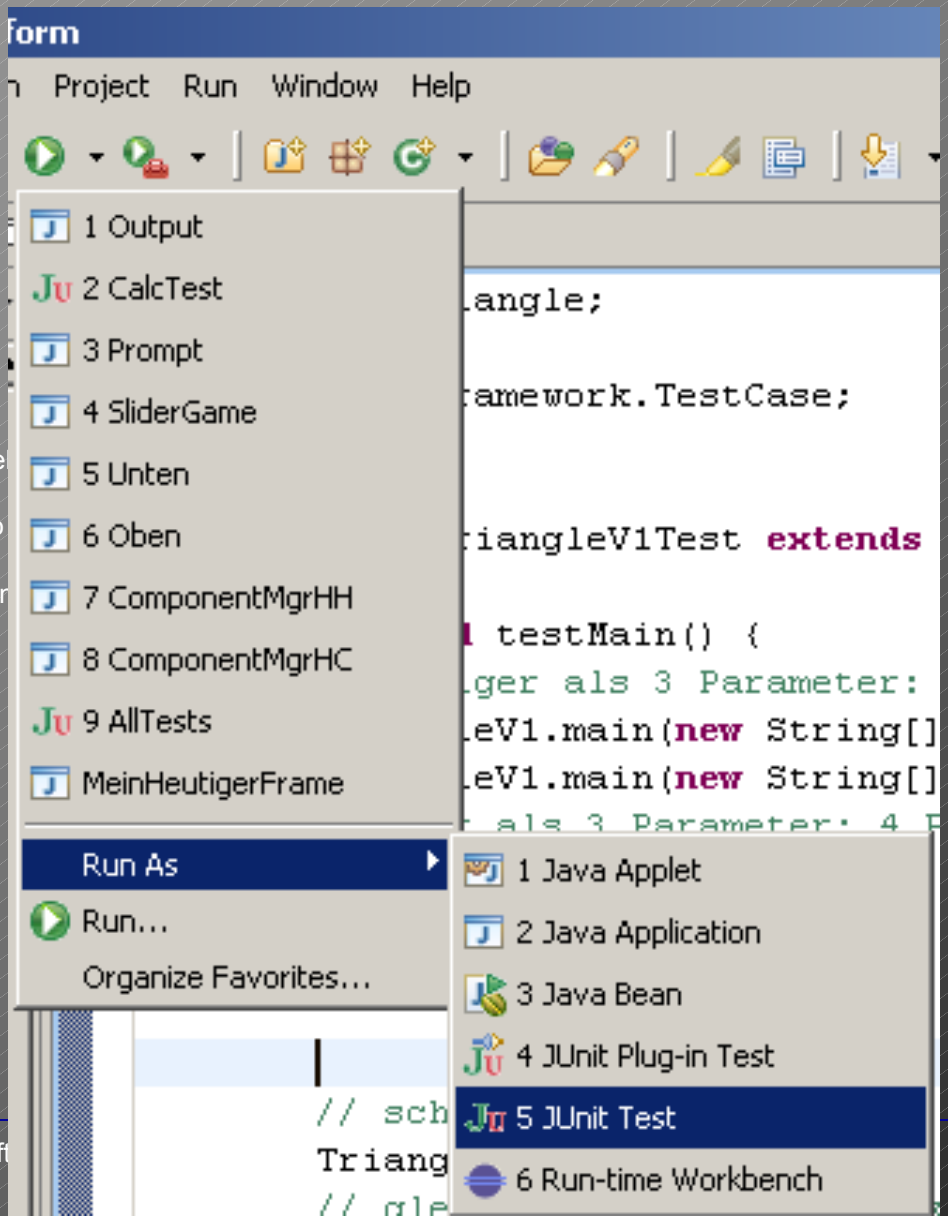
- Motivation
- Eclipse-Vorbereitung
- Testfall erstellen
- Testklasse ausführen
- Demo-Beispiel
- Test-Szenario
- Prüf-Methoden
- Diverse
- ...
- ...
- ...
- ...



Testfall zu einer Klasse ausführen

Testen mit JUnit

- Motivation
- Eclipse-Vorbereitung
- Testfall erstellen
- Testklasse ausführen
- Demo-Beispie
- Test-Szenario
- Prüf-Methoder
- Diverse
- ...
- ...
- ...
- ...



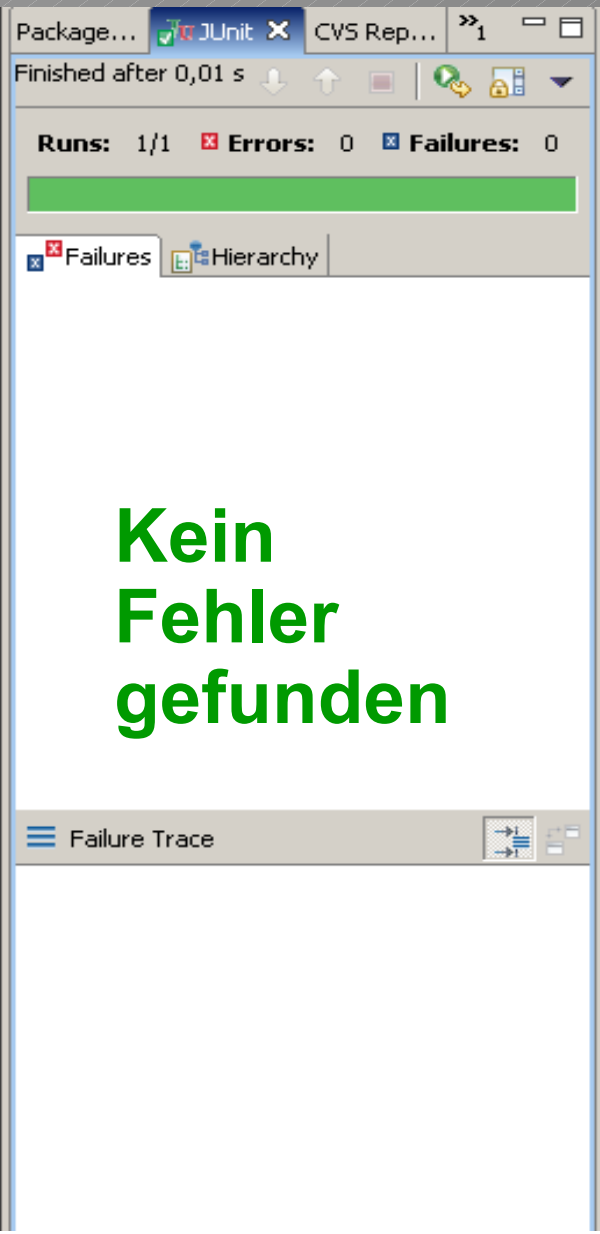
Alle Methoden der Klasse, die als

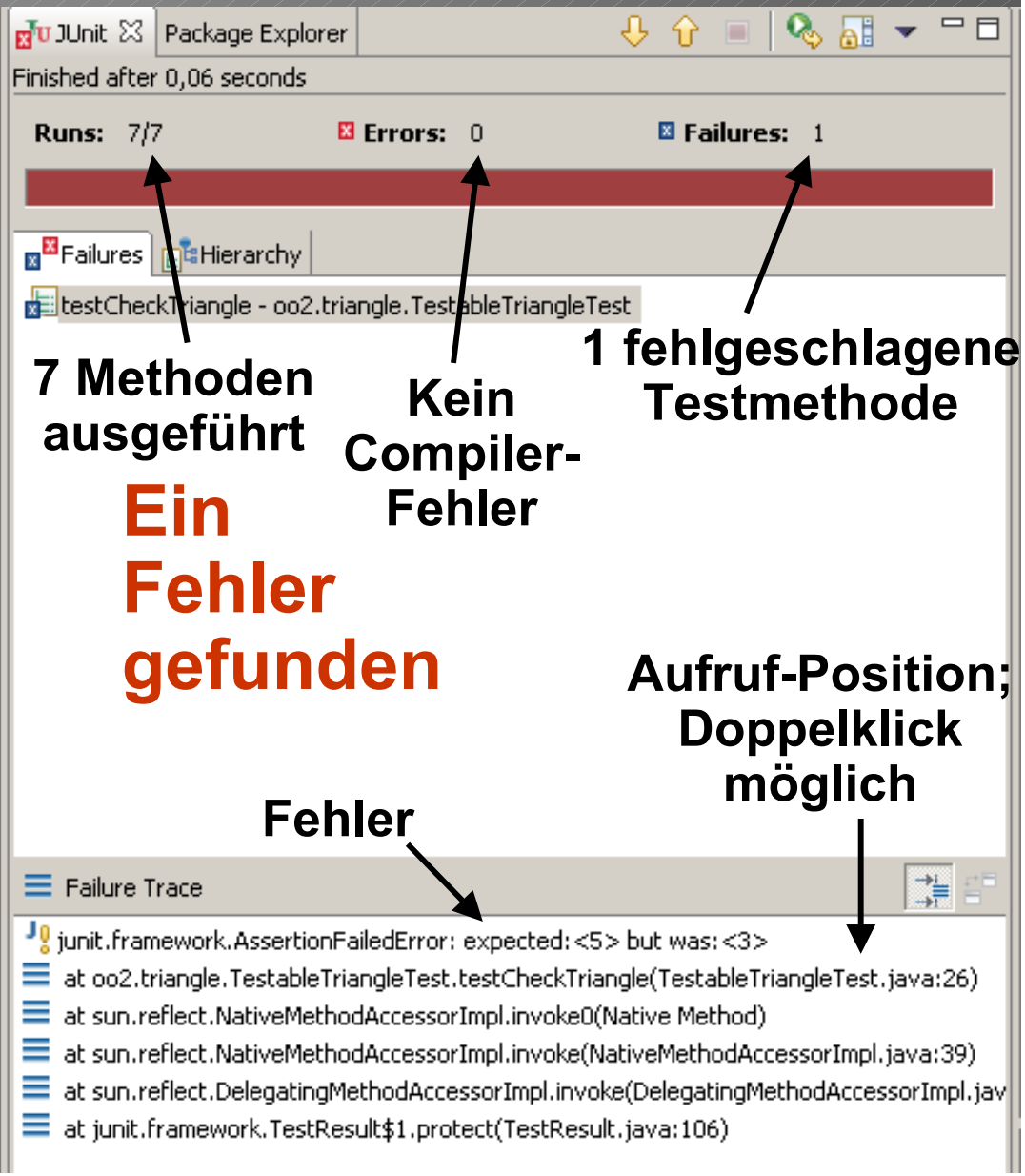
public void test...() { ... }

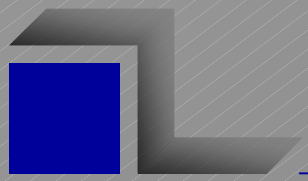
deklariert sind,
werden nacheinander ausgeführt

Testfall zu einer Klasse ausführen: negatives / positives Ergebnis

Testen mit JUn
- Motivation
- Eclipse-Vorbereitu
- Testfall erstellen
- Testklasse ausführen
- Demo-Beis
- Test-Szena
- Prüf-Metho
- Diverse
- ...
- ...
- ...
- ...







- Motivation
- Eclipse-Vorbereitung
- Testfall erstellen
- Testklasse ausführen
- Demo-Beispiel
- Test-Szenario
- Prüf-Methoden
- Diverse
- ...
- ...
- ...
- ...

Fragen?

Beispiel: eine zu testende Klasse mit zwei Methoden

Testen mit JUnit

- Motivation
- Eclipse-Vorbereitung
- Testfall erstellen
- Testklasse ausführen
- Demo-Beispiel
- Test-Szenario
- Prüf-Methoden
- Diverse
- ...
- ...
- ...
- ...

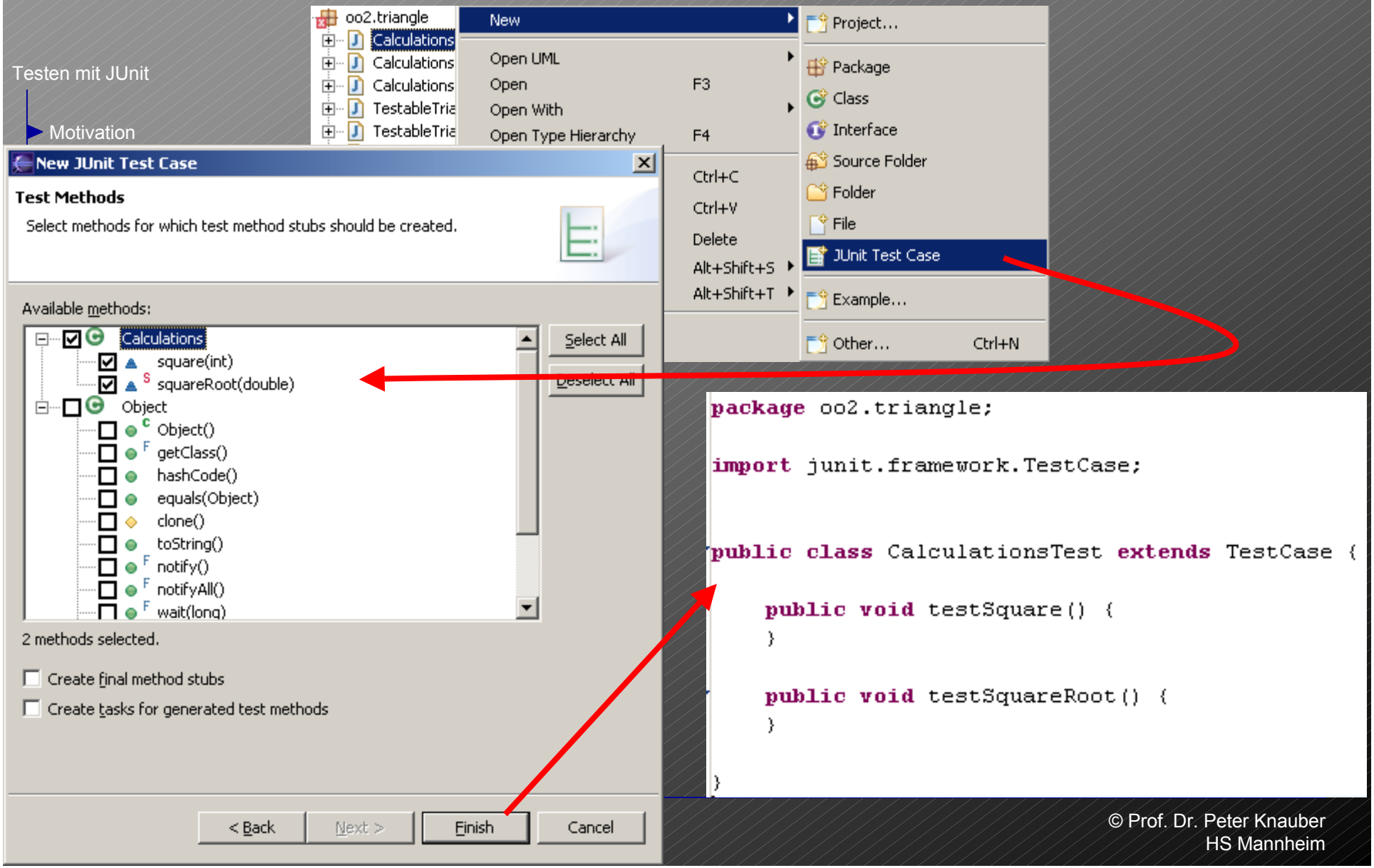

```
package oo2.triangle;


public class Calculations {

    int square( int x ) {
        return x * x;
    }

    static double squareRoot( double x ) {
        return Math.sqrt(x)  - 0.00000001;
    }

}
```

Nur zu Demo-Zwecken: einmal dynamisch, einmal statisch

Idee: kleiner Rechenfehler, wie sie bei float/double manchmal vorkommen

- Demo
  - Programm zeigen
  - Testfälle erstellen und ausführen

Erstellen der Testklasse

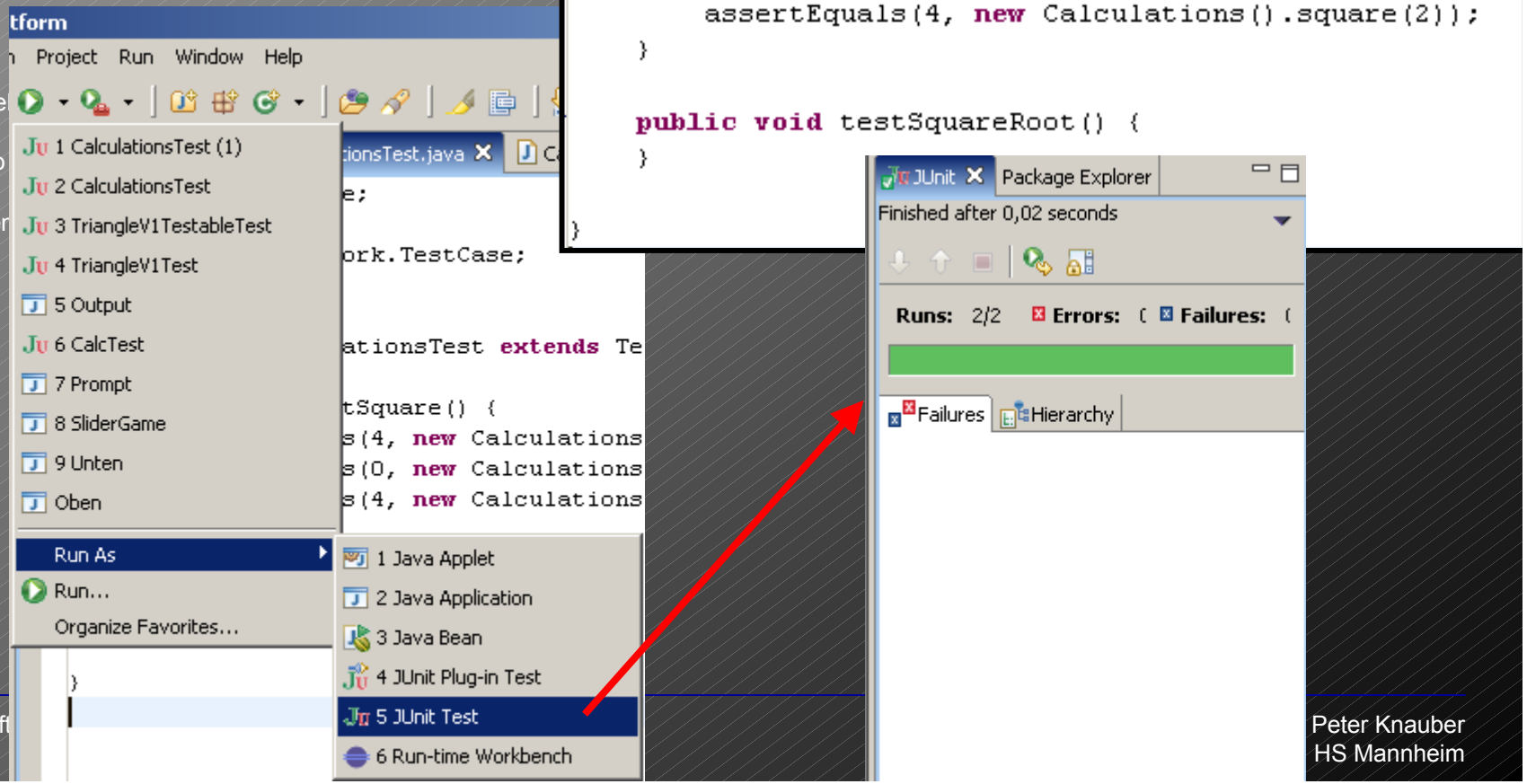


Ausführen der Testklasse

Testen mit JUnit

- Motivation
- Eclipse-Vorbereitung
- Testfall erstellen
- Testklasse ausführen
- Demo-Beispiel
- Test-Szenario
- Prüf-Methoden
- Diverse
- ...
- ...
- ...
- ...


```
package oo2.triangle;

import junit.framework.TestCase;


public class CalculationsTest extends TestCase {

    public void testSquare() {
        assertEquals(4, new Calculations().square(-2));
        assertEquals(0, new Calculations().square(0));
        assertEquals(4, new Calculations().square(2));
    }

    public void testSquareRoot() {
    }

}
```



"Gründliches" Testen der Wurzel-Methode 1/2

Testen mit JUnit

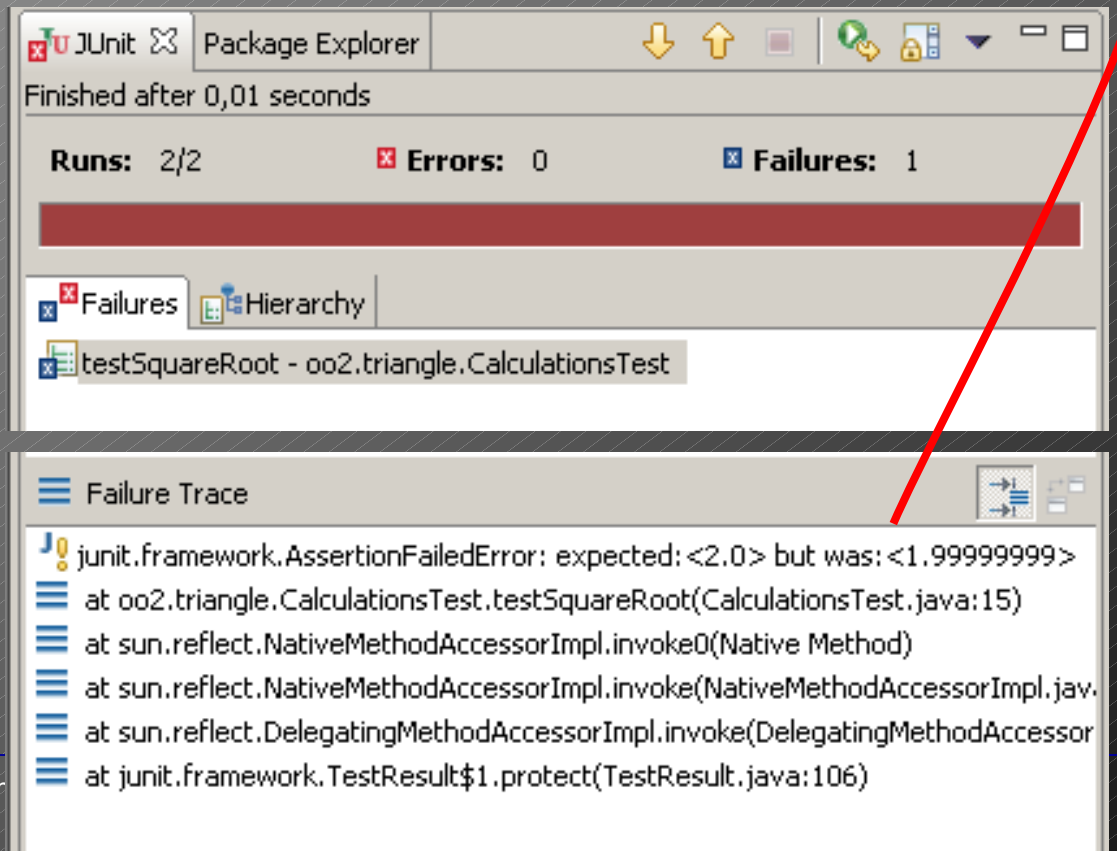
- Motivation
- Eclipse-Vorbereitung
- Testfall erstellen
- Testklasse ausführen
- Demo-Beispiel
- Test-Szenario
- Prüf-Methoden
- Diverse
- ...
- ...
- ...
- ...


```
public void testSquareRoot() {
    assertEquals(2, Calculations.squareRoot(4), 0);
    assertEquals(0, Calculations.squareRoot(0), 0);
}
```

Keine Fehlertoleranz

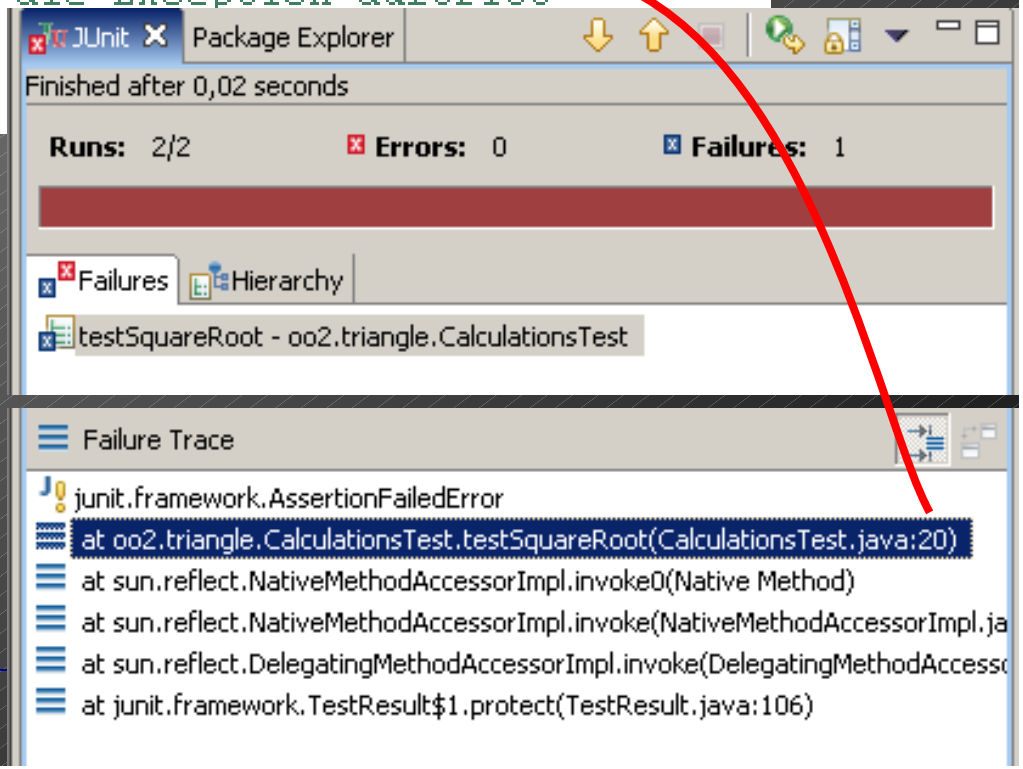
Besser:

```
, 0.0001);
, 0.0001);
```



"Gründliches" Testen der Wurzel-Methode 2/2

```
public void testSquareRoot() {
    assertEquals(2, Calculations.squareRoot(4), 0.0001);
    assertEquals(0, Calculations.squareRoot(0), 0.0001);

    try {
        Calculations.squareRoot(-4);
        fail();
    } catch (ArithmeticException e) {
        // alles ok, wenn die Exception auftritt
    }
}
```

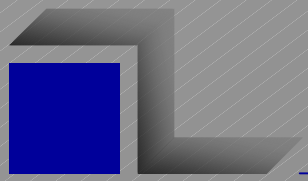


Testen mit JUnit

- Motivation
- Eclipse-Vorbereitung
- Testfall erstellen
- Testklasse ausführen
- Demo-Beispiel
- Test-Szenario
- Prüf-Methoden
- Diverse
- ...
- ...
- ...
- ...

Verbesserter Code der Wurzel-Methode


```
double squareRoot( double x ) {
    if (x <= 0)
        throw new ArithmeticException("Kein Wurzelziehen von Zahlen kleiner 0 möglich");
    else
        return Math.sqrt(x) + 0.00000001;
}
```



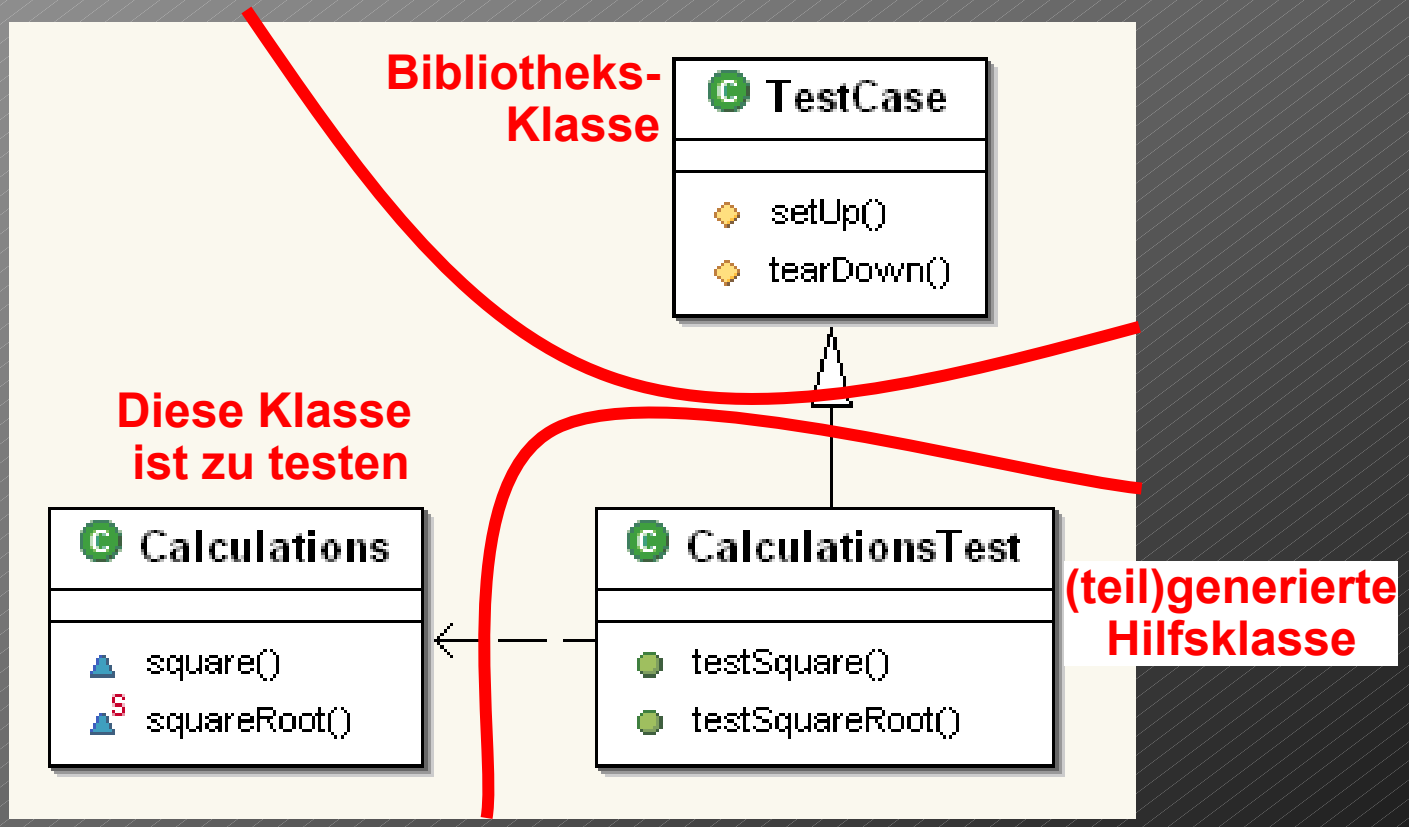
- Motivation
- Eclipse-Vorbereitung
- Testfall erstellen
- Testklasse ausführen
- Demo-Beispiel
- Test-Szenario
- Prüf-Methoden
- Diverse
- ...
- ...
- ...
- ...

Fragen?

Test-Szenario

Testen mit JUnit

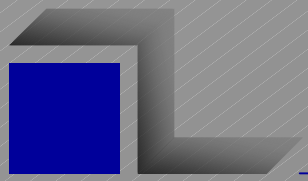
- Motivation
- Eclipse-Vorbereitung
- Testfall erstellen
- Testklasse ausführen
- Demo-Beispiel
- Test-Szenario
- Prüf-Methoden
- Diverse
- ...
- ...
- ...
- ...



Prüfmethoden (geerbt von *TestCase*)

- Konzept:
  An bestimmten Stellen in einem (Test-)Programm soll sichergestellt (engl.: *assert*) werden, dass bestimmte Bedingungen erfüllt sind
- Die Klasse *TestCase* bietet viele Prüf-Methoden
  - assertTrue(*Boole'sche Bedingung*)
  - assertFalse(*Boole'sche Bedingung*)
  - assertEquals(*Soll-Wert*, *Ist-Wert*); mögliche Typen:
    boolean, byte, char, int, long, short; Object, String (es wird mit *equals* verglichen)
  - assertEquals(*Soll-Wert*, *Ist-Wert*, *delta*); mögliche Typen:
    double, float
  - assertNull(*Objekt*)
  - assertNotNull(*Objekt*)
  - assertSame(*Objekt1*, *Objekt2*)
  - assertNotSame(*Objekt1*, *Objekt2*)
  - fail()
- Ist eine der Prüfbedingungen nicht erfüllt, wird ein *AssertionFailedError* ausgelöst
- Alle Methoden existieren auch mit einem zusätzlichen ersten Parameter (Typ String) für eine (hilfreiche!) Meldung

Testen mit JUnit
Motivation
Eclipse-Vorbereitung
Testfall erstellen
Testklasse ausführen
Demo-Beispiel
Test-Szenario
Prüf-Methoden
Diverse
...
...
...
...



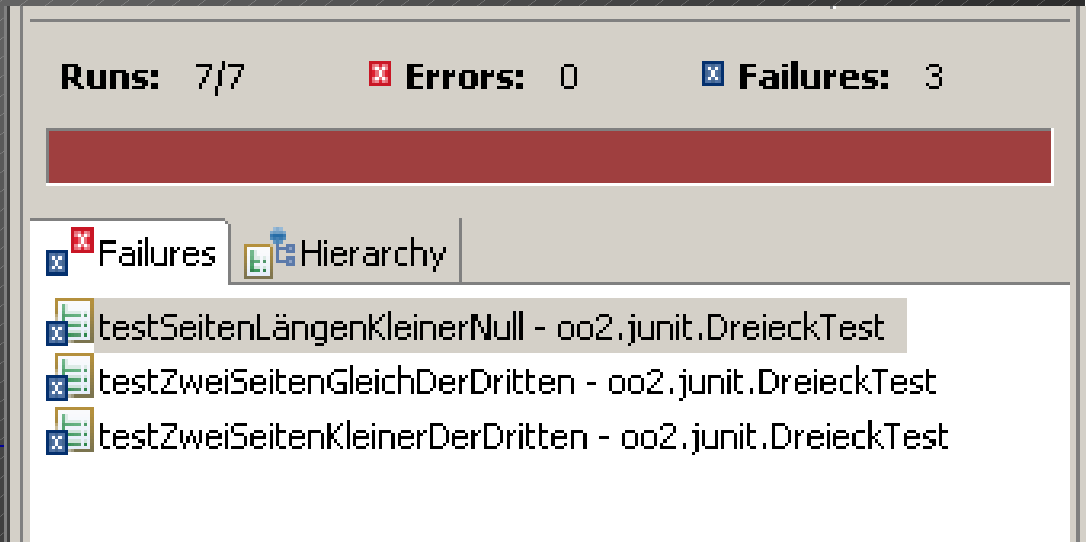
- Motivation
- Eclipse-Vorbereitung
- Testfall erstellen
- Testklasse ausführen
- Demo-Beispiel
- Test-Szenario
- Prüf-Methoden
- Diverse
- ...
- ...
- ...
- ...

Fragen?

Testfälle entwerfen

Testen mit JUnit
- Motivation
- Eclipse-Vorbereitung
- Testfall erstellen
- Testklasse ausführen
- Demo-Beispiel
- Test-Szenario
- Prüf-Methoden
- Diverse
- ...
- ...
- ...
- ...

- Jede *test...*-Methode sollte nur einen Testfall prüfen, aber den komplett:
  - Einen geeigneten Zustand vor der zu testenden Methode herstellen
  - Die zu testende Methode aufrufen
  - Das Ergebnis (= Rückgabewert) der Methode überprüfen
  - Den Zustand nach dem Methodenaufruf überprüfen
- Jeder Testfall sollte unabhängig vom Vorgänger funktionieren können
- Die *test...*-Methoden sollten geeignet benannt sein



Vor- und Nachbereitung fürs Testen

Testen mit JUnit
- Motivation
- Eclipse-Vorbereitung
- Testfall erstellen
- Testklasse ausführen
- Demo-Beispiel
- Test-Szenario
- Prüf-Methoden
- Diverse
- ...
- ...
- ...
- ...

- Folgende Methode wird *vor jeder test...-Methode* ausgeführt:

  protected void setUp() throws Exception

- Folgende Methode wird *nach jeder test...-Methode* ausgeführt:

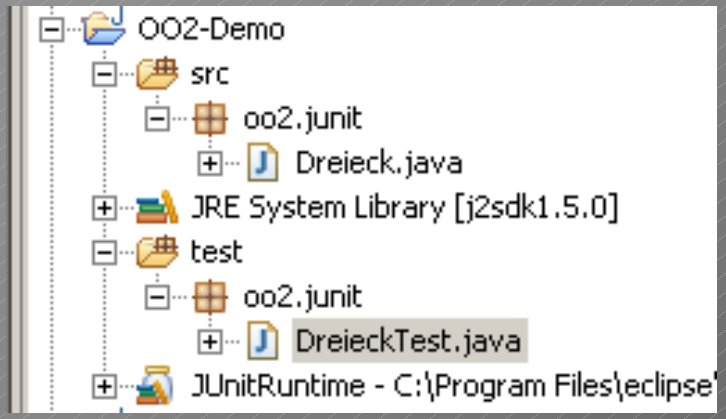
  protected void tearDown() throws Exception

- Man sagt, es wird eine *Test-Fixture* auf- bzw. abgebaut

Trennen von Produktiv- und Test-Code

Testen mit JUnit

- Motivation
- Eclipse-Vorbereitung
- Testfall erstellen
- Testklasse ausführen
- Demo-Beispiel
- Test-Szenario
- Prüf-Methoden
- Diverse
- ...
- ...
- ...
- ...



- Gleiche Package-Struktur in beiden Ordnern
- Export JAR-Datei:
  nur die Packages
  (= diejenigen Package-Anteile)
  im src-Ordner exportieren

```
package oo2.junit;

import junit.framework.TestCase;

public class DreieckTest extends TestCase {
```

```
package oo2.junit;

public class Dreieck {

    static final int GLEICHSEITIG = 1;
    static final int GLEICHSCHENKLIG = 2;
    static final int SCHIEF = 3;
    static final int UNGÜLTIG = -1;
```