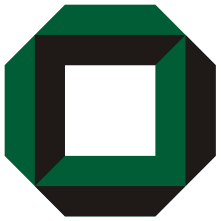
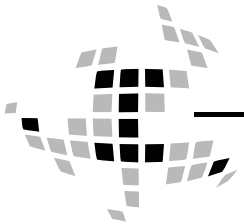


Universität Karlsruhe (TH)
Forschungsuniversität · gegründet 1825

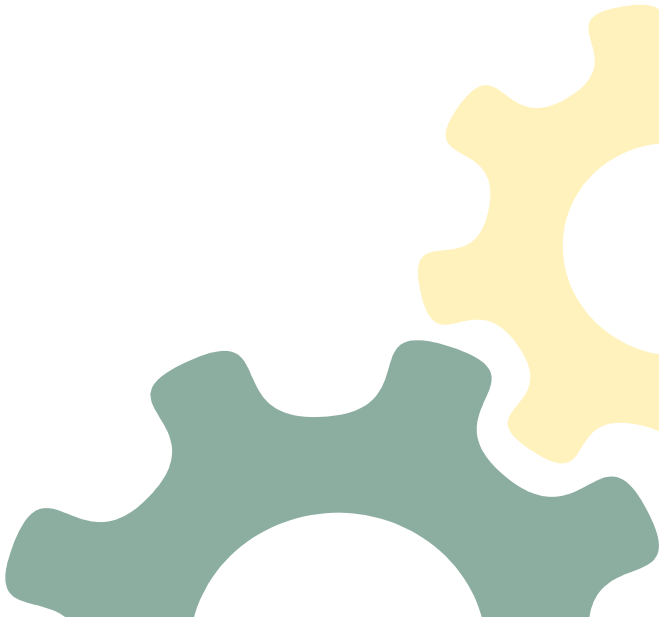
Kapitel 6.2
Testwerkzeuge

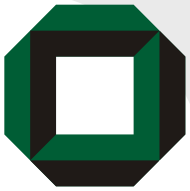
SWT I – Sommersemester 2009

Walter F. Tichy
Andreas Höfer
David J. Meder



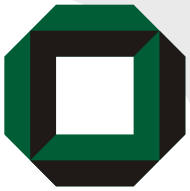
Fakultät für **Informatik**
Lehrstuhl für Programmiersysteme





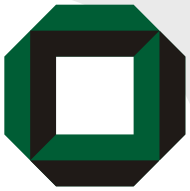
Testwerkzeuge der Software-Amateure

- Testen mit Ausgabe-Anweisungen im Programm
  - drucken Variablen an bestimmten Stellen im Programm aus
  - werden nach Test gelöscht oder auskommentiert
  - besser: per Übersetzerdirektiven (`#ifdef` in C) zu- und ausschalten.



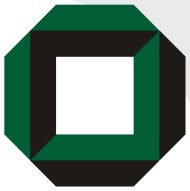
Testwerkzeuge der Software-Amateure

- Testen mit einem interaktiven Debugger
  - Haltepunkte festlegen
  - Variablen interaktiv an den Haltepunkten inspizieren
  - Information über Untersuchung ist nach Debuggerlauf verloren



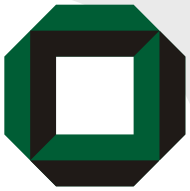
Testwerkzeuge der Software-Amateure

- Testen mit Test-Skripten
  - laufen automatisch ab
  - verknüpft mit `print`-Anweisungen
- Allen gemeinsam:
  manuelle Überprüfung der Ausgaben



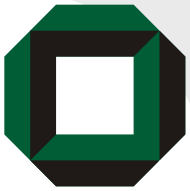
Nachteile dieser Methoden

- Bei Programmänderungen müssen Testfälle wiederholt werden
  - Mühsam und unpraktisch und oft unmöglich
    - Ausgabe-Anweisungen sind oft nicht mehr vorhanden
    - wenn doch vorhanden, sind die richtigen Auskommentierungen zu entfernen (und später wieder einzusetzen)
    - bei Testen mit Debugger ist keinerlei Information vorhanden, welche Variablen inspiziert werden sollen.



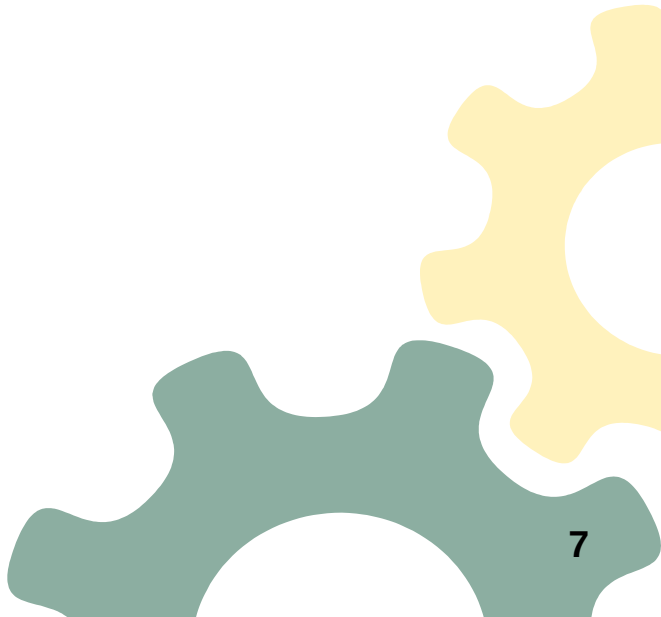
Weitere Nachteile dieser Methoden

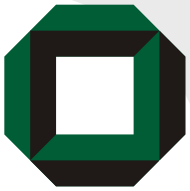
- Ergebnisse müssen manuell überprüft werden
  - nur der Programmierer weiß, was die Ausgaben bedeuten und wann sie richtig sind
  - selbst dieser verliert dieses Wissen mit der Zeit.
- Es ist unpraktikabel, Testfälle für viele Komponenten zusammenzufassen und gesammelt auszuführen
  - technisch schwierig (Testaufbauten unterschiedlich)
  - zu viele Ausgabedaten zu überprüfen.



Alternativen

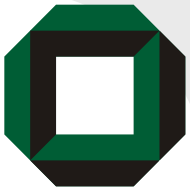
- Benutze Zusicherungen (engl. assertions)
- Schreibe automatisch ablaufende Testfälle, die sich selbst überprüfen.





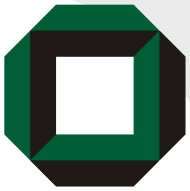
Zusicherungen (Assertions)

- Zusicherungen
  - Boolesche Funktionen
    - Vor- und Nachbedingungen (z.B. dass ein übergebener Parameter positiv sein muss oder eine Referenz nicht unbestimmt sein darf)
    - Invarianten einer Datenstruktur
  - Werden zur Laufzeit ausgeführt;
- Im Fehlerfall melden sie sich mit einer Ausnahme oder Fehlermeldung.



Zusicherungen

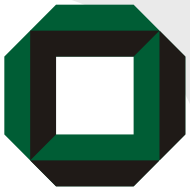
- Bei Eiffel, Java (ab 1.4) und C# sind Zusicherungen in die Sprache integriert.
- Bei C und C++ kann man sich mit Makros behelfen.
- Wichtig: Zusicherungen können zu- und abgeschaltet werden.



Zusicherungen in Java

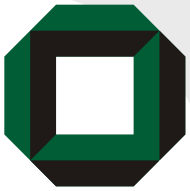
```
AssertStatement:
   assert Expression1;
   assert Expression1 : Expression2;
```

- Semantik
  - Verletzte Zusicherungen (evaluiert zu „falsch“) löst Ausnahme **`AssertionError`** aus
- Semantik Doppelpunkt
  - **`Expression1: boolean`**, **`Expression2:`** nicht **`void`**
  - **`Expression2`** wird ausgeführt wenn **`Expression1`** „falsch“ liefert
  - Ergebnis von **`Expression2`** wird an Konstruktor von **`AssertionError`** übergeben
  - Programm endet mit Ergebnis von **`Expression2`** und Aufrufstapelabzug.



Benutzung von Zusicherungen

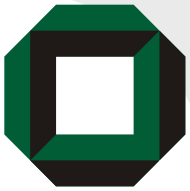
- Aufgaben von Zusicherungen
  - Am Anfang einer Methode werden Vorbedingungen an Parameterwerten geprüft, am Ende Ergebnisse und Invarianten.
- Beispiele
  - Am Ende einer Sortiermethode könnte z.B. eine Zusicherung überprüfen, ob die zu sortierenden Daten wirklich aufsteigend angeordnet sind.
  - Um nachzuprüfen, ob vor und nach einer Manipulation ein Baum noch balanciert ist, könnte man die Funktion `istBalanciert()` schreiben und in Zusicherungen aufrufen:
    - `assert istBalanciert();`



Benutzung von Zusicherungen

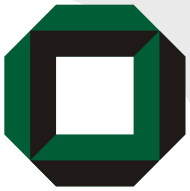
- Unerreichbare Codesegmente können mit einer Zusicherung abgesichert werden:

```
switch(farbe) {
   case GRÜN:
        ...
        break;
   case ROT:
        ...
        break;
   default:
        assert false;  // unerreichbar
        break;         // zur Vollständigkeit
}
```



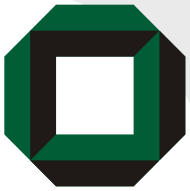
Benutzung von Zusicherungen

- Konvention für **öffentliche** Methoden
  - Überprüfung der Eingabeparameter nicht mit Zusicherungen, sondern mit `IllegalArgumentException`
  - Folge: Klientenprogramme können darauf reagieren



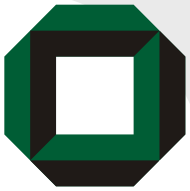
Benutzung von Zusicherungen

- Konvention für **private** Methoden
  - Eingabeparameter, Nachbedingungen und Invarianten aller privater Methoden mit Zusicherungen überprüfen
  - Grund: Verletzung ist unerwarteter Fehler
  - Aber: Falsche Parameter bei öffentlichen Methoden sind nicht unerwartet.



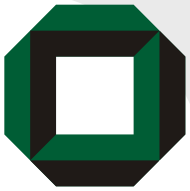
Benutzung von Zusicherungen

- Mit Zusicherungen können Missverständnisse und Fehlinterpretationen der Programmierer rasch aufgedeckt werden.
- In Produktionsläufen werden Zusicherungen aus Leistungsgründen abgeschaltet (Bei Java auch selektiv für einzelne Klassen).
- Bei Auftreten eines Fehlers oder beim Testen von Programmänderungen werden die Zusicherungen wieder zugeschaltet.



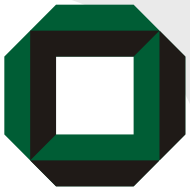
Benutzung von Zusicherungen

- Zusicherungen sind keine Testfälle
  - es werden lediglich bestimmte Bedingungen im Programmlauf überprüft
  - Sofern sie zugeschaltet sind, werden sie beim Ablauf von Testfällen mit ausgeführt.



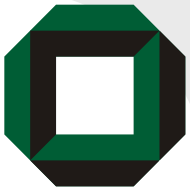
Automatisch ablaufende Testfälle

- Testfall
  - Ausführen eines Programmes unter bekannten Bedingungen
  - Eingabedaten gegeben
  - Ergebnis bekannt
  - Ziel: Aufdecken von Fehlern



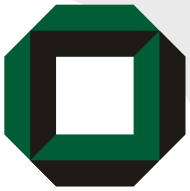
2. Automatisch ablaufende Testfälle

- Automatisch ablaufende Testfälle
  - Überprüfen Ergebnis ihrer Ausführung selbst
  - Rückgabe: boolescher Wert (erfolgreich oder fehlgeschlagen)
  - Optional: Auslösen einer Ausnahme bei Fehlschlag
  - Können zu großen Testmengen zusammengesetzt werden.
  - Können beliebig oft ausgeführt werden (täglich, stündlich, bei jeder Änderung).
  - Können als Regressionstest benutzt werden.



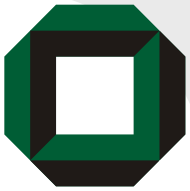
Rahmenarchitektur für autom. ablaufende Testfälle

- Warum Rahmenarchitektur?
  - Vereinfacht das Schreiben der Testfälle
  - Zusammenfassen von Testfällen zu Testmengen
  - Automatisches Ablaufen ganzer Testmengen; nur Versager werden gemeldet.
  - Wirkt standardisierend (jeder macht´s gleich, Zusammenfassung von Testmengen damit möglich).



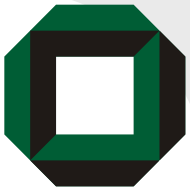
JUnit 4: Ein Testrahmen für Java

- Erlaubt schnellen und hierarchischen Aufbau von Testmengen
- liefert die fehlgeschlagenen Testfälle
- standardmäßig nur textuelle Anzeige, grafische Anzeige wird von der Programmierumgebung, z.B. Eclipse realisiert
- Autoren: Erich Gamma, Kent Beck
- Wo: http://www.junit.org
- Ähnliche Werkzeuge existieren für eine Vielzahl anderer Programmiersprachen



JUnit 4: Testklasse

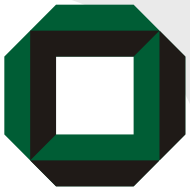
- Enthält (zusammengehörige) Testfälle in Form von Methoden.
- Hält i.d.R. Referenzen auf die zu prüfenden Testobjekte
- Die Vergleiche von Soll- und Ist-Werten findet mittels Zusicherungen aus der Klasse `org.junit.Assert` statt.
- Kennzeichnung der Test-Methoden erfolgt mit Annotationen.



Exkurs: Annotationen in Java 5

- Annotationen sind Meta-Informationen, die zum Quelltext hinzugefügt werden können.
- Sie bleiben im Byte-Code erhalten, beeinflussen aber nicht direkt die Ausführung des Programms.
- Sie können zur Übersetzungszeit durch Werkzeuge und zur Laufzeit durch Reflexion (engl. reflection) ausgelesen und verarbeitet werden.
- Annotationen dürfen in jeder Deklaration vorkommen: Paket, Klasse, Schnittstelle, Instanz-/Klassenvariable, Methode, Parameter, Konstruktor, lokale Variable.
- Beispiel: markiere Methode als veraltet

```
@Deprecated public void tuWas() { … }
```



JUnit 4: Beispiel

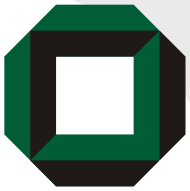
```
package demo;
import static org.junit.Assert.assertTrue;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class BibliothekTest {
  private Bibliothek bib;

  @Before public void baueAuf() {
    bib = new Bibliothek();
  }

  @Test public void buchIstInBibliothek() {
    boolean b = bib.pruefeVerfuegbarkeit("Faust");
    assertTrue("Faust muss in der Bibliothek sein.", b);
  }

  @After public void raeumeAuf() {
    bib = null;
  }
}
```

Zusicherungen werden per **`import static`** eingeblendet.

Methoden mit **`@Before`** dienen zum Aufbau von Testressourcen.

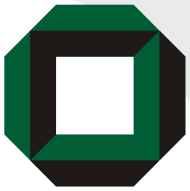
`@Test` markiert einen Testfall.

Methoden mit **`@After`** dienen zum Abbau von Testressourcen.



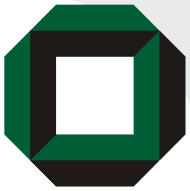
JUnit 4: Einige Assert-Methoden

- `assertEquals(Object soll, Object ist)`
  Schlägt fehl, wenn der Aufruf der `equals`-Methode von `soll` mit dem Parameter `ist` den Wert falsch zurück gibt.
- `assertSame(Object soll, Object ist)`
  Schlägt fehl, wenn die Referenzen auf unterschiedliche Objekte zeigen.
- `assertEquals(double soll, double ist, double delta)`
  Schlägt fehl, wenn sich `soll` und `ist` um mehr als `delta` unterscheiden.
- `assertTrue(boolean ausdruck)`
  `assertFalse(boolean ausdruck)`
  Schlägt fehl, wenn der Ausdruck falsch/wahr ist.
- `assertNull(Object referenz)`
  `assertNotNull(Object referenz)`
  Schlägt fehl, wenn die Referenz nicht `null`/`null` ist.
- Zu all diesen Methoden gibt es eine Variante mit einer Zeichenkette für eine Nachricht im Fehlerfall.



JUnit 4: Annotationen

- `@Test` – Markiert einen Testfall
  - Erwartete Ausnahmen können wie folgt angegeben werden:
    `@Test(expected=SQLException.class)`
  - Es kann auch die maximale Laufzeit in Millisekunden festgelegt werden:
    `@Test(timeout=42)`
- `@Ignore` – Markiert einen Testfall, der nicht ausgeführt werden soll
  - Die Bergründung wird als Zeichenkette angegeben:
    ```
    @Ignore("Funktionalität fehlt noch.")
    @Test public void buchAusleihen() { … }
    ```
- `@Before` – Markiert eine Methode, die vor jedem Testfall ausgeführt wird.
- `@After` – Markiert eine Methode, die nach jedem Testfall ausgeführt wird.
- `@BeforeClass` – Markiert eine Klassenmethode, die vor Instanziierung der Testklasse ausgeführt wird.
- `@AfterClass` – Markiert eine Klassenmethode, die nach allen anderen bisher genannten Methoden ausgeführt wird.



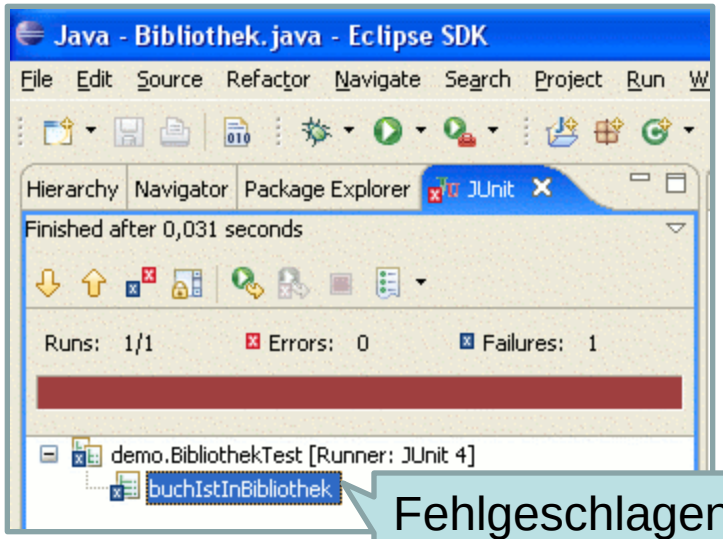
JUnit 4: Ausführungsreihenfolge

```
[ … ]

public class BibliothekTest {
    [ … ]

1   @BeforeClass public static void verbindeMitDB() { … }

2 5 @Before public void baueAuf() { … }

3   @Test public void buchIstInBibliothek() { … }

6   @Test public void buchAusleihen() { … }

4 7 @After public void raeumeAuf() { … }

8   @AfterClass public static void trenneVonDB() {

}
```

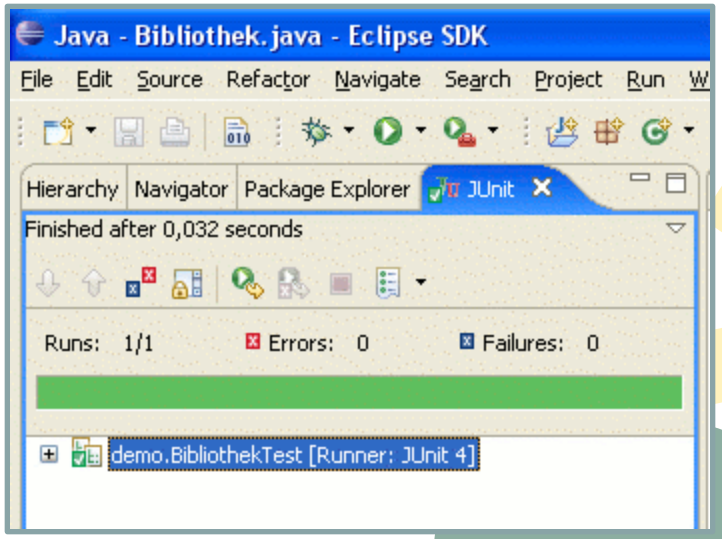
Die Reihenfolge der Testfälle ist nicht garantiert. Die Punkte 3 und 6 könnten auch vertauscht sein.

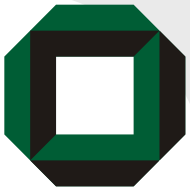
Ausführung in Eclipse

- In Eclipse reicht es, im Kontextmenü einer Testklasse „Run As“ → „JUnit Test“ zu wählen; dann werden die Testmethoden ausgeführt.
- Testklassen können auch in Paketen zusammengefasst werden, und diese wiederum in größere Pakete; bei Auswahl mit „Run As“ → „JUnit Test“ erfolgt das Ausführen aller enthaltenen Testmethoden.
- Die Anzeige erfolgt in einem eigenen Fenster.



Fehlgeschlagener Testfall





Zusammenfassung

- Automatisch ausführbare Testfälle sind besonders für den Regressionstest wichtig.
- Regressionstests helfen verhindern, dass alte Fehler wieder auftauchen.
- JUnit und ähnliche Regressionstest-Werkzeuge sind in der Praxis extrem wichtig.
- Sie erfordern, dass Testfälle sich selbst überprüfen.
- Können mit Zusicherungen kombiniert werden.