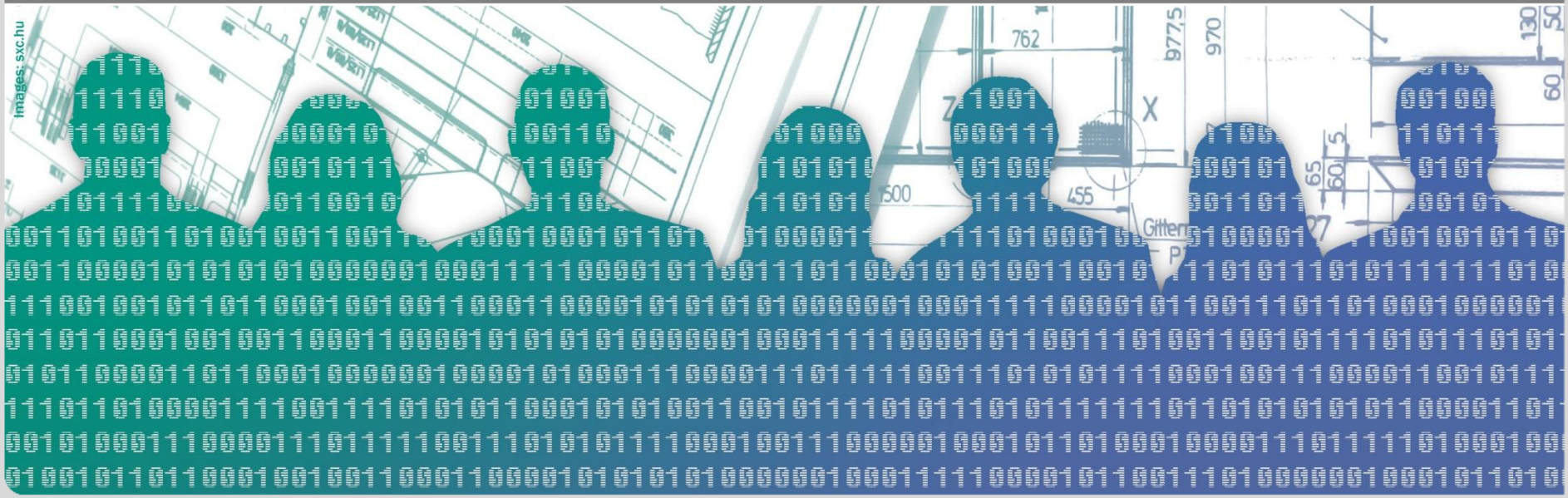
Kapitel 5.2
Testwerkzeuge

SWT I – Sommersemester 2010

Walter F. Tichy, Andreas Höfer, Korbinian Molitorisz

IPD Tichy, Fakultät für Informatik



Testwerkzeuge der Softwareamateure

- Testen mit Ausgabeanweisungen 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 Softwareamateure

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

Testwerkzeuge der Softwareamateure

- 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.
- Benutze Prüfprogramme, die Software auf Schwachstellen untersuchen.

1. 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 `assert Expression1;`
  - Falls `Expression` „falsch“ ergibt, wird Ausnahme `AssertionError` ausgelöst.
- 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 Anfang einer Sortiermethode würde überprüft, ob das übergebene Feld nicht null ist, und ob die Grenzen (falls angegeben) stimmen.
  - 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 Defekt
  - 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 Defekts 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.

2. Automatisch ablaufende Testfälle

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

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.

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

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: 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.

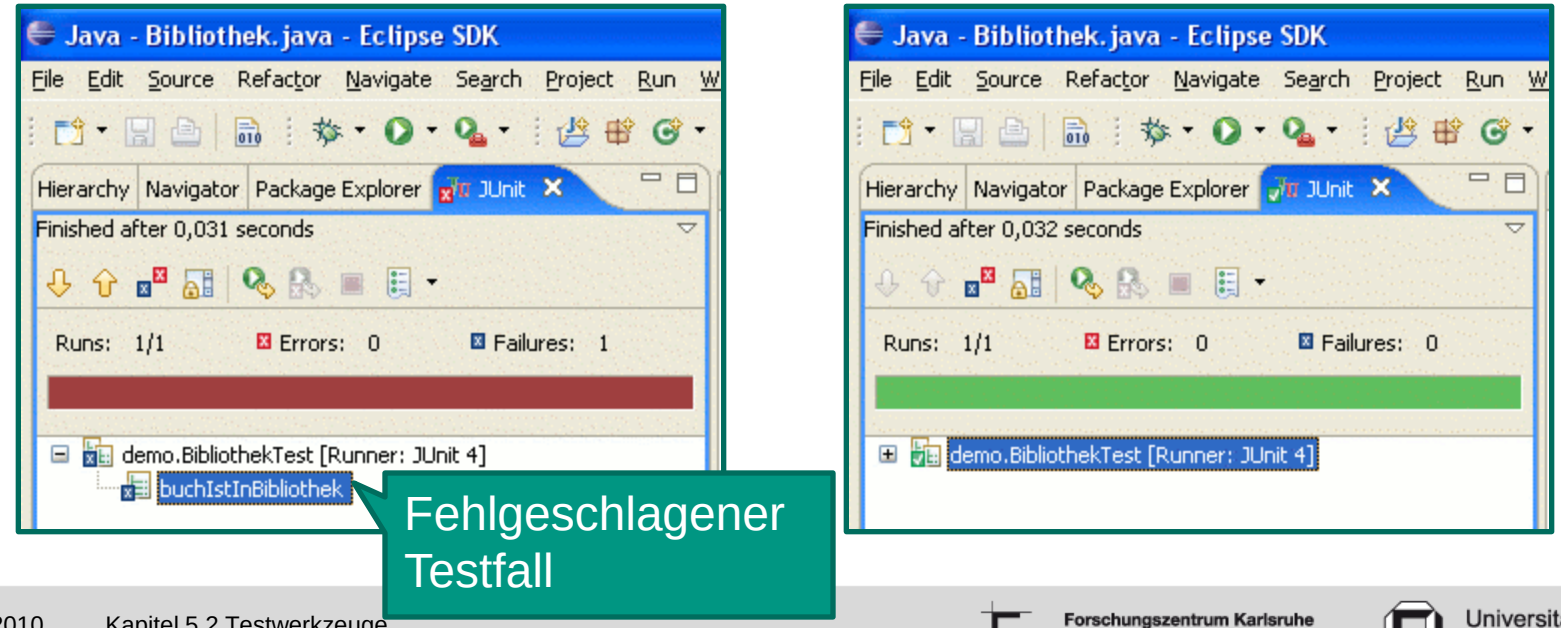
JUnit 4: Ausführungsreihenfolge

```
[ … ]

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.



Zusammenfassung

- Automatisch ausführbare Testfälle sind besonders für den Regressionstest wichtig
- Regressionstests helfen zu verhindern, dass alte Defekte 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

3. Prüfprogramme

- Warnungen und Defekte
  - Werden von der Entwicklungsumgebung angezeigt. Darunter fallen z.B.:
    - Evtl. nicht initialisierte Variablen
    - Nicht erreichbare Anweisungen
    - Unnötige Anweisungen
- Programmierstil überprüfen

Checkstyle

  - Tabulatoren anstelle von Leerzeichen verwendet
  - JavaDoc-Kommentare vergessen
  - Methodenparameter nicht als `final` deklariert

FindBugs

- Defekte anhand von Fehlermustern finden
  - Mit Hilfe einer statischen Analyse können Defekte anhand bestimmter Muster gefunden werden.

Prüfprogramme für Eclipse – Checkstyle

- Checkstyle unterstützt den Programmierer bei der Einhaltung von Programmierstilen bei Java-Code.
  - Der einzuhaltende Programmierstil kann mit einer Konfigurationsdatei festgelegt werden.
- Checkstyle bietet dazu eine Vielzahl von verschiedenen Prüfungen, welche miteinander kombiniert werden können.
- Beispiel:
  JavadocMethod – Überprüft, ob alle Methoden und Konstruktoren vollständige JavaDoc-Kommentare besitzen.

- http://checkstyle.sourceforge.net/

Prüfprogramme für Eclipse – FindBugs (1)

- FindBugs sucht mit Hilfe sog. Fehlermuster nach möglichen Defekten im Code.
- Ein **Fehlermuster** (engl. *bug pattern*) beschreibt wiederkehrende Beziehungen zwischen gefundenem Versagen und den zugrunde-liegenden Defekten.

- http://findbugs.sourceforge.net/

Prüfprogramme für Eclipse – FindBugs (2)

- Schlechte Angewohnheiten (engl. *bad practice*)
  - **Klasse definiert clone(), implementiert aber Cloneable nicht.**
    Das kann in Ordnung sein (z.B. wenn man kontrollieren möchte, wie Unterklassen klonieren), aber überprüfe, ob das so gewollt ist.
- Korrektheit (engl. *correctness*)
  - **Verwirrende Methodennamen**
    Methodennamen unterscheiden sich nur durch Groß/Kleinschreibung. Liegt hier ein Fall von verpasster Überschreibung vor, und ist die richtige Methode aufgerufen?
- Internationalisierung (engl. internationalization)
  - **Benutze einen Lokalitäts-Parameter in Methode**
    Eine Zeichenreihe wird auf Groß/Kleinschreibung umgewandelt, wobei die Standard-Konvertierung der Plattform benutzt wird. Das kann mit internationalen Schriftsätzen schief gehen.
- Code-Angreifbarkeit (engl. malicious code vulnerability)
  - **Ein statisches Attribut sollte nur im Paket sichtbar sein,**
    andernfalls könnte es von außen oder aus Versehen geändert werden.

Prüfprogramme für Eclipse – FindBugs (3)

- Korrektheit bei mehrfädigen Anwendungen (engl. *multithreaded correctness*)
  - **notify() sollte durch notifyAll() ersetzt werden**
    Java Monitore werden meist für mehrere Bedingungen benutzt. `notify()` aktiviert einen beliebigen Faden, aber das könnte der Falsche sein, der auf eine andere Bedingung wartet.
- Ausführungsgeschwindigkeit (engl. *performance*)
  - **toString() wird an einer Zeichenreihe ausgeführt.**
    Das ist redundant.
- Sicherheit (engl. *security*)
  - **Leeres Kennwort**
    Software erzeugt eine Datenbank mit leerem Kennwort. Das könnte bedeuten, dass die Datenbank nicht Kennwort-geschützt ist.
- Fragwürdiger Code (engl. *dodgy*):
  - **Überprüfung einer Variable auf null, von der man weiß, dass sie null ist.**

Prüfprogramme für Eclipse – EMMA (1)

- EMMA ermittelt die Anweisungsabdeckung von Testfällen für Java-Anwendungen und erstellt daraus Statistiken.
  - Es wird dazu kein Java-Code benötigt (es genügen .class oder .jar Dateien).
- Die überprüfbare Anweisungsabdeckung reicht von ganzen Paketen, Klassen und Methoden bis hin zu einzelnen Codezeilen.
  - Wird eine Codezeile nur teilweise ausgeführt, so wird dies von EMMA registriert.
- http://emma.sourceforge.net/
- http://www.eclemma.org/ (Eclipse-Plugin)

Prüfprogramme für Eclipse – EMMA (2)