

# Kapitel 5.2 – Testwerkzeuge

**SWT I – Sommersemester 2021**

**Walter F. Tichy, Christopher Gerking, Tobias Hey**



# 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 („*Stack trace*“).

# 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
- 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

- Siehe Kapitel 0.1 zu JUnit



### 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

- Einrücken nicht korrekt
- JavaDoc-Kommentare vergessen
- Methodenparameter nicht als `final` deklariert

#### ■ Defekte anhand von Fehlermustern finden

SpotBugs

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

# Prüfprogramme für Eclipse – SpotBugs (1)

- SpotBugs sucht mit Hilfe sog. Fehlermuster nach möglichen Defekten im Code.
- Ein **Fehlermuster** (engl. *bug pattern*) beschreibt wiederkehrende Beziehungen zwischen gefundenem Versagen und den zugrundeliegenden Defekten.
- <https://spotbugs.github.io/>

# Prüfprogramme für Eclipse – SpotBugs (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 – SpotBugs (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.**