Testen

Testen und Softwarequalität (1)

- Testfälle können nicht alles abdecken
- gefundene Fehler $\Leftrightarrow$ vorhandene Fehler (Beurteilung)
- 1 Fehler ausgebessert $\Rightarrow$ 10 Fehler eingebaut
- Fehler in jeder Softwareentwicklungsphase
- nicht alle Fehler werden sichtbar (aber für wen?)
- seltene Fehler sind großes Sicherheitsrisiko (Eindringen)
- „Fehler“ können Absicht sein (Eindringen ermöglichen)

Testen und Softwarequalität (2)

- Testen kann Problemstellen aufdecken
- aber Qualitätssteigerung nur durch statisches Verstehen (z.B. Code Reviews)
- Testfälle zur Spezifikation verwendbar

Teststufen

- Unittest
- Integrationstest
- Systemtest
- Abnahmetest

Testmethoden

- Black-Box-Test (Testen am Ende)
- White-Box-Test (Programm und Testfälle gleichzeitig)
- Grey-Box-Test (Testfälle zuerst)

Testarten

- Funktionaler Test
- Nichtfunktionaler Test
- Schnittstellentest
- Oberflächentest
- Stresstest (z.B. Crasch- oder Lasttest)
- Sicherheitstest
- Regressionstest

Laufzeitmessungen

Arten von Laufzeiten

```
time javac Hello.java

real    0m0.863s
user    0m1.036s
sys     0m0.064s
```

- Problem: Messergebnisse jedesmal anders

Einflüsse auf Laufzeiten

- Latenz (Einweglatenz, Round-Trip-Time)
- Bandbreite
- feingranularer Parallelismus
- Thread-Parallelismus
- Datenmengen, Rechnerbelastung, Betriebssystem, Compiler, Interpreter, Hardware, . . .

Häufiger Fehler:

- Hochrechnen auf andere Datenmenge

Echtzeitsysteme

- Zeitüberschreitung = Fehler
- harte versus weiche Echtzeitsysteme
- harte Echtzeitsysteme häufig sicherheitskritisch
- Laufzeitmessungen + Beweisverfahren, oft mit Redundanz
- Java dafür kaum geeignet, eher C

Nachvollziehen des Programmablaufs

Stack-Trace (1)

```
public class Rec {
    public static void main (String[] args) {
        rec(2);
    }
    private static int rec (int x) {
        assert x > 0 : "x = " + x;
        return rec (x - 1);
    }
}
```

```
Exception in thread "main" java.lang.AssertionError: x = 0
    at Rec.rec(Rec.java:6)
    at Rec.rec(Rec.java:8)
    at Rec.rec(Rec.java:8)
    at Rec.main(Rec.java:3)
```

Stack-Trace (2)

```
Exception in thread "main" java.lang.StackOverflowError:
    at Rec.rec(Rec.java:8)
    at Rec.rec(Rec.java:8)
    at Rec.rec(Rec.java:8)
    at Rec.rec(Rec.java:8)
    at Rec.rec(Rec.java:8)
    at Rec.rec(Rec.java:8)
    at Rec.rec(Rec.java:8)
    at Rec.rec(Rec.java:8)
    at Rec.rec(Rec.java:8)
    ...
```

Debug-Anweisung

```
public class Rec {
    public static void main (String[] args) {
        rec(2);
    }
    private static int rec (int x) {
        System.out.println("x = " + x);
        return rec (x - 1);
    }
}
```

- `if(debug) System.out.println(...)`
- Logdatei (Protrokoll-Datei)

Debugger

Werkzeug unterstützt Programmierer durch

- Unterbrechung des Programmablaufs bei *Breakpoint*
  - an bestimmten Programmstellen setzbar
    (z.B. Methodenanfang)
  - bei Zugriff auf bestimmte Variablen
  - bei Änderung des Wertes bestimmter Variablen
- während Unterbrechung Variableninhalte untersuchen und Variablenwerte ändern sowie Breakpoints (zurück)setzen
- schrittweise Programmausführung (step into, step over)

Probleme beim Debuggen

- wissen nicht, worauf wir achten müssen
- Änderungen von Programmen (Debug-Anweisungen) bzw. Variablenwerten wirken sich oft anders aus als erwartet
- Verwendung des Debuggers ändert Programmverhalten
- Fehler zeigen sich nicht dort, wo sie passieren („Nadel im Heuhaufen“)
- betroffene Programmstellen nur statisch zu verstehen

Eingrenzen von Fehlern

- Orientierung
- Hypothese
- Planung
- Durchführung
- Auswertung

Fehlermeldungen

- Compiler erkennt Inkonsistenzen, aber keine Fehler
- daher können Fehlermeldungen irreführend sein
- Fehlermeldung nicht als Handlungsanweisung verstehen, sondern

  Fehlerursache finden!