Tutorium zur Vorlesung Programmieren

7. Tutorium

Joshua Gleitze

12. Dezember 2016

IPD Koziolek

Lehren aus dem 2. Übungsblatt

Zur Korrektur

Überblick über die Punkteverteilung in verlinkter Tabelle.

Dinge, für die es Punktabzug gab, habe ich im Kommentar mit (-) versehen (Keine Garantie auf Vollständigkeit).

Notenverteilung

Durchschnitt:
~12,4 Punkte

≥ Punkte	… entsprechen ungefähr der Note
19	1
18	1,3
17	1,7
16	2
15	2,3
14	2,7
13	3
12	3,3
11	3,7
9	4
0	n. b.

Meine Beispiellösung

Eine Lösung für die volle Punktzahl, die aber nicht (viel) mehr als notwendig tut:

joshuagleitze.de/tutorien/programmieren1617/code/Beispiellösung2

Code-Duplikation

Code-Duplikate sind zu vermeiden. Dazu muss man manchmal kreativ sein. Das sollte man üben.

Länge

Kürzerer Code ist oft verständlicher und weniger fehleranfällig.

Aber: nicht übertreiben! Ziel ist Verständlichkeit.

(Übrigens ist kürzerer Code nicht schneller. Oft sind auf Performance optimierte Lösungen länger.)

Beispiel: Konstruktoren

In der Regel gibt es nur einen Konstruktor, der tatsächlich Arbeit ausführt:

java.util.Scanner (OpenJDK 8)

```
304 public final class Scanner implements Iterator<String>, Closeable {
    ...
530     private Scanner(Readable source, Pattern pattern) {
531         assert source != null : "source should not be null";
532         assert pattern != null : "pattern should not be null";
533         this.source = source;
534         delimPattern = pattern;
535         buf = CharBuffer.allocate(BUFFER_SIZE);
536         buf.limit(0);
537         matcher = delimPattern.matcher(buf);
538         matcher.useTransparentBounds(true);
539         matcher.useAnchoringBounds(false);
540         useLocale(Locale.getDefault(Locale.Category.FORMAT));
541     }
```

Beispiel: Konstruktoren

In der Regel gibt es nur einen Konstruktor, der tatsächlich Arbeit ausführt:

java.util.Scanner (OpenJDK 8)

```
550     public Scanner(Readable source) {
551         this(Objects.requireNonNull(source, "source"), WHITESPACE_PATTERN);
552     }
        ...
562     public Scanner(InputStream source) {
563         this(new InputStreamReader(source), WHITESPACE_PATTERN);
564     }
        ...
610     public Scanner(File source) throws FileNotFoundException {
611         this((ReadableByteChannel)(new FileInputStream(source).getChannel()));
612     }
        ...
701     public Scanner(String source) {
702         this(new StringReader(source), WHITESPACE_PATTERN);
703     }
```

Beispiel: Date vergleichen

```
1  public boolean isAfter(Date other) {
2      return other.isBefore(this);
3  }
```

Datentkapselung & Geheimnisprinzip

Andere Klassen sind schwarze Kisten: Wir wissen über sie nur, was sie nach außen dokumentieren.

Bis zu einem gewissen Grad gilt das auch für andere Methoden.

Beispiel: Duplikation & Geheimnisprinzip

```
1  public boolean isEqual(DateTime other) {
2      return this.date.getYear() == other.getYear()
3          && this.date.getMonthValue() == other.getMonthValue()
4          && this.date.getDayOfMonth() == other.getDayOfMonth()
5          && this.time.getHour() == other.getHour()
6          && this.time.getMinute() == other.getMinute()
7          && this.time.getSecond() == other.getSecond();
8  }
```

besser:

```
1  public boolean isEqual(DateTime other) {
2      return this.date.isEqual(other.date)
3          && this.time.isEqual(other.time);
4  }
```

Im nächsten Übungblatt: Abzug für Duplikation!

Duplikation muss aber nicht „auf Teufel komm raus“ vermieden werden:
Manchmal ist ein wenig kopierter Code leichter verständlich, als Methoden,
die versuchen, zu viele, zu verschiedene Fälle zu lösen.

Vergleich mit boolean

booleans sind Werte!

Ein Vergleich mit **boolean** ist immer überflüssig!

```
1  if (this.isGood() == false) {
2      this.makeGood();
3  }
```

Vergleich mit boolean

booleans sind Werte!

Ein Vergleich mit **boolean** ist immer überflüssig!

```
1  if (!this.isGood()) {
2      this.makeGood();
3  }
```

Vergleich mit boolean

booleans sind Werte!

Ein Vergleich mit **boolean** ist immer überflüssig!

```
1  if (this.isEnabled() == true) {
2      return true;
3  } else {
4      return false;
5  }
```

Vergleich mit `boolean`

`boolean`s sind Werte!

Ein Vergleich mit `boolean` ist immer überflüssig!

```
1   return this.isEnabled();
```

Modulo

`%` ist *nicht* der Modulo-Operator!

`%` berechnet den Rest der Division:

$$24 \equiv 4 \mod 10 \qquad \texttt{24 \% 10 == 4}$$
$$10 \equiv 0 \mod 10 \qquad \texttt{10 \% 10 == 0}$$
$$-4 \equiv 6 \mod 10 \qquad \texttt{-4 \% 10 == -4}$$
$$-10 \equiv 0 \mod 10 \qquad \texttt{-10 \% 10 == 0}$$

Modulo-Operation mit `Math#floorMod(int, int)` (Javadoc)

`equals` und `hashCode`

`equals`

`#equals` vergleicht `this` mit einem anderen Objekt. Die Methode sollte so implementiert sein, dass die enstprechende Relation diese Eigenschaften hat:

- reflexiv
- transitiv
- symmetrisch
- $\forall$ `x != null`: `x.equals(null) == false`

hashCode

`#hashCode` bildet ein Objekt auf einen `int` ab. Die Methode wird in verschiedenen Algorithmen verwendet, um effizient nach Objekten suchen zu können. Die Methode muss so implementiert werden, dass Folgendes gilt:

$$\forall \texttt{x,y}: \texttt{x.equals(y)} \implies \texttt{x.hashCode()} == \texttt{y.hashCode()}$$

Wenn `equals` überschrieben wird muss auch `hashCode` überschrieben werden!

`equals` UND `hashCode`

`#equals` und `#hashCode` implementiert man normalerweise, indem man die Generatorfunktion der IDE verwendet.

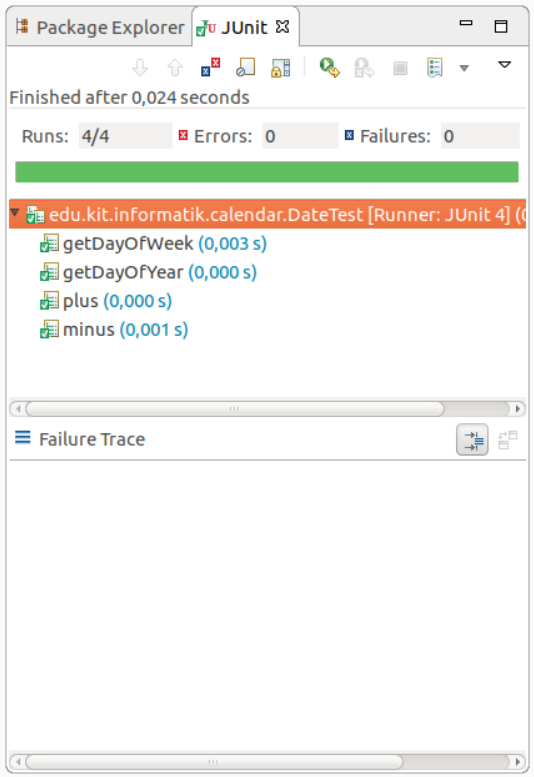
In Eclipse ist das: Rechtsklick → Source → Generate equals and hashCode

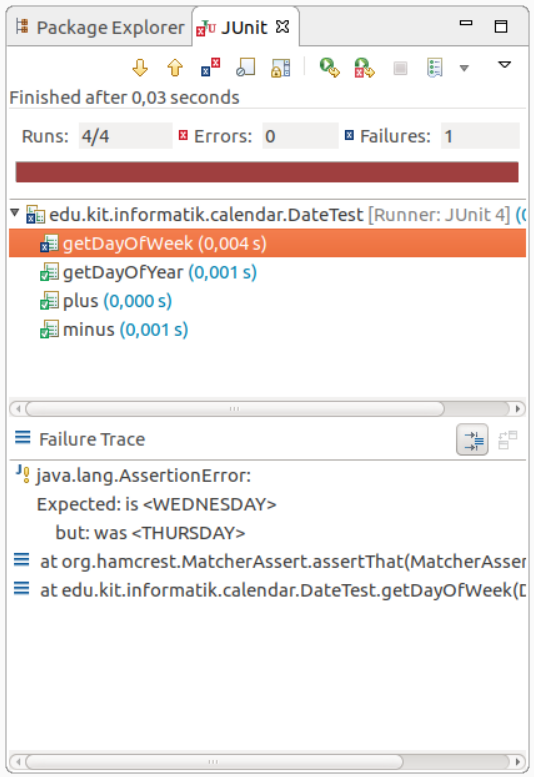
Testen und Debuggen

Testen mit JUnit

JUnit ist das verbreitetste Rahmenwerk zum testen von Java-Programmen.

Gut in Eclipse integriert.





Imports für JUnit

Die folgenden imports können jeder JUnit-Testklasse hinzugefügt werden:

```
import static org.hamcrest.CoreMatchers.*;
import static org.junit.Assert.assertThat;
import static org.junit.Assert.fail;
import org.junit.*;
```

Dadurch stehen alle gebräuchlichen Methoden und Klassen zur Verfügung.

Testfälle in JUnit

Testfälle stehen in Methoden, werden in Klassen zusammengefasst und mit `@Test` markiert.

```
1  public class DateTimeTest {
2
3      @Test
4      public void plusDate() {
5          // Ein Testfall
6      }
7
8      @Test
9      public void plusTime() {
10         // Noch ein Testfall
11     }
12 }
```

JUnit: weitere Annotations

Methoden können auch noch diese Markierungen tragen:

`@Ignore`	Diesen Testfall nicht ausführen
`@BeforeClass`	Einmalig vor allen Testfällen ausführen
`@Before`	Vor jedem Testfall in dieser Klasse ausführen
`@After`	Nach jedem Testfall dieser Klasse ausführen
`@AfterClass`	Nach allen Testfällen ausführen

Testen mit assertThat und Matchern

Um zu prüfen, verwenden wir diese Methoden:

```
assertThat(Prüfausdruck, Matcher);
assertThat(Nachricht, Prüfausdruck, Matcher);
```

- `Prüfausdruck` liefert einen Wert, der geprüft werden soll
- `Matcher` beschreibt die Logik, mit der der Wert geprüft wird
- `Nachricht` wird ausgegeben, falls der Test fehlschlägt

Weiterführendes

Dokumentation der wichtigsten Matcher: org.hamcrest.CoreMatchers

Alle Hamcrest-Matcher: org.hamcrest

Eigene Matcher schreiben: „Create your own hamcrest matcher“ (planetgeek.ch)

Testen: Best Practices

- Ein Testfall prüft nur eine Eigenschaft
- Ein Testfall ist sehr leicht zu lesen
    - In Tests ist Code zu duplizieren nicht so schlimm, wenn es der Lesbarkeit dient
- Ein Testfall ist nicht randomisiert

Debugging

Debugger erlauben es, die Ausführung von Code Schritt für Schritt nachzuvollziehen.

Um „einzusteigen“ werden Breakpoints gesetzt. Kommt die Ausführung des Programms an einen Breakpoint wird die Ausführung eingefroren und der Zustand des Programmes kann inspiziert werden. Von dort aus kann auch Schrittweise weiter ausgeführt werden.

Tutorial zu Debugging in Eclipse

Weitere Informationen und fortgeschrittenes Debugging:

Java Debugging with Eclipse - Tutorial (Vogella GmbH)

Evaluation

Evaluation

http://t1p.de/evaluationTutoriumJoshua