Praktikum „SEP: Java-Programmierung“ SS 2019

Programmierstil

Karlheinz Friedberger und Thomas Bunk

Camel Code

Source:

`https://www.perlmonks.org/?node_id=45213`

Programmierstil

- Programme sollen nicht nur funktionieren, sondern auch lesbar sein
  - Können andere den Code verstehen? Stichwort: Teamarbeit!
  - Kann man selbst den Code noch verstehen in zwei Stunden/Wochen/Monaten/Jahren?
  - Quelltext ist die einzig zuverlässige Dokumentation
  - Quelltext wird (häufig) mit ausgeliefert
  - JAVA Coding Conventions (JCC) $\rightarrow$ internationaler Standard
  - Revisionsverwaltung (z.B. SVN, MERCURIAL, GIT)

Kriterien

- Sprechende Bezeichner
- Qualität der Dokumentation: Genügend, sinnvoll und korrekt
- Formatierung/Layout unterstützt Programmstruktur
- Sinnvolle und feingranulare Aufteilung, keine Monstermethoden oder Monsterklassen
- Geheimnisprinzip (Information Hiding)
- Erweiterbarkeit
- Das Eine-Zuständigkeit-Prinzip (Single Responsibility): Jede Klasse hat **eine** Aufgabe
- Konsistenz

Bezeichner

- Sprechende Bezeichner:
  Nicht `int a, b;` sondern `int width, height;`
- Einzelne Buchstaben nur in trivialen Fällen:
  z.B. Laufindex `i` oder `double mean(int x, int y)`
- Keine Typinformationen im Namen:
  z.B. `int lengthInt;` oder `Node rootNode;`
  $\rightarrow$ überflüssig, da über die IDE jederzeit einsehbar!
- Groß- und Kleinschreibung in Java:

`StringBuilder`	Klasse, Substantiv, jeder Wortteil beginnt mit Großbuchstaben
`childCount`	Variable, Substantiv, erster Wortteil klein
`getPoints`	Methode, Verb, erster Wortteil klein
`NUMBER_OF_CHARS`	Konstante, Großbuchstaben und Unterstriche

Dokumentation

- Ziel: Erklären, **was** mit **wem** gemacht wird (und ggf. **warum**), wenn es **Mehrwert** bringt und **nicht offensichtlich** ist
- Sinnlos: `frobulateWidgets(); // frobulates the widgets`
- Möglichkeiten für Dokumentation:
  - JavaDoc
  - Kommentare
  - Code
- Dokumentation und Kommentare **aktuell** halten (schwierig!)
- **Lügende Dokumentation ist noch schlimmer als gar keine!**

Dokumentation: JavaDoc

- Für Methoden und Klassen
- **Was** passiert
- **Nicht wie** ist es umgesetzt
- Mindestens alles, was public und protected
- **Immer**, wenn Methodenname nicht ausreicht, um Verhalten zu verstehen

Good:

```
/**
 * Check if enough tokens are
 * provided. If not, a
 * corresponding error message
 * is printed.
 *
 * @param ...
 */
boolean hasCorrectLength(
    String[] tokens,
    int expectedLength) {
  ...
}
```

Bad:

```
/**
 * Check if enough tokens are
 * provided by calling
 * tokens.len and checking that
 * it is the same value as
 * expectedLength.
 * If not, we use
 * System.out.println and
 * both values to print a
 * meaningful error message.
 * ...
 */
```

Dokumentation: Kommentare

- Kommentare für zusätzliche Informationen innerhalb von Methoden
- **Vor** dem Kommentierten
- **Kurze** Kommentare am Ende der selben Zeile
- **Wichtig: Wieso** etwas (so) gemacht wird
- „Was mit wem“ gemacht wird sollte schon in Javadoc stehen!

```
try {
    // points is a user-input and hasn't been checked
    // yet, so it may not be a valid String and
    // a NumberFormatException can be thrown here
    return Integer.parseInt(points);

} catch (NumberFormatException e) {
    showError("Could not parse number " + points + ".");
}
```

Layout

- Nicht mehr als eine Anweisung pro Zeile
- Zeilen nicht zu lang werden lassen (z.B. maximal 80 oder 100 Zeichen)
- Leerzeilen
- Einrückung (Leerzeichen, keine Tabs)
- Konsistente Positionen von Klammern
- Boolesche Variablen wenn möglich positiv formulieren

```
public static void main(String[] args) throws IOException {
  BufferedReader in
      = new BufferedReader(new InputStreamReader(System.in));
  boolean run = true;

  while (run) {
    ...
  }
  ...
}
```

Layout

- Mehrwertige Boolesche Ausdrücke immer eindeutig Klammern
- Booleans nie gegen Konstanten `true` oder `false` vergleichen
- Wo und wie werden überlange Zeilen umgebrochen:
  - Hinter Kommata und öffnenden Klammern
  - Vor Operatoren
  - Besser „außen“ als „innen“ bei geschachtelten Ausdrücken
  - Neue Zeile nach umgebendem Ausdruck ausrichten
  - Konkatenation (+) bei überlangen Zeichenketten (z.B. bei `private static final String HELP_TEXT = "..."`)
- Methodenlänge (Richtwert): Nicht mehr als ca. 20 Zeilen (oft weniger)
- Klassenlänge: Schwieriger festzulegen, aber nicht mehr als 1000 Zeilen

Geheimnisprinzip (Information Hiding) und Zugriffsmodifikatoren

- Jedes Attribut, jede Methode, jeder Konstruktor nur so lokal wie möglich zugreifbar
- Nur was geheim ist, kann beliebig verändert werden
- Nur was geheim ist, ist kontrollierbar
- Jeder (potentielle) Zugriff potenziert die Komplexität
- Lesezugriff auf Attribute von außen nur über Getter und nur wenn nötig: `getXXX`, `isXXX`, `hasXXX`
- Schreibzugriff auf Attribute von außen nur über Setter und nur wenn unbedingt nötig: `setXXX`
- Variablen, die nach der Initialisierung nie mehr verändert werden immer `final`
- Statische Attribute sollten immer `final` sein
- Idealerweise sind Objekte unveränderlich (immutable), z.B. `String` (in Aufgabe 1 aber nur schwer umsetzbar)

Erweiterbarkeit

- Grundprinzip: Antizipation des Wandels
- **Naheliegende** Veränderungen müssen leicht umsetzbar sein
- Umsetzung durch
  - Geheimnisprinzip (siehe vorherige Folie)
  - Vermeidung von (insbesondere globalem oder veränderbaren) Zustand wo immer möglich (siehe vorherige Folie)
  - Verwendung gängiger Programmiermuster
  - Vermeidung von Redundanz: DRY (don't repeat yourself)
  - Sprechende Konstanten statt „Magic Numbers“
- Nicht sinnvoll: Programmieren für in der Zukunft **vielleicht** notwendige Features
  (YAGNI: you ain't gonna need it, KISS: Keep it simple, stupid)

Interessante Literatur

- Jon Bentley: „Bumper-Sticker Computer Science“(kurze Tipps zu Programmierung & Co.): `http://www.bowdoin.edu/~ltoma/teaching/cs340/spring05/coursestuff/Bentley_BumperSticker.pdf`
- Google Java Code Style: `http://google.github.io/styleguide/javaguide.html`
- Joshua Bloch: Effective Java (3rd Edition!)