Quelltextkonventionen für Java

Softwareentwicklung I

Richtlinien für die Gestaltung von Java-Quelltexten (Level 1)

1 Warum Quelltextkonventionen?

„Eine Quelltextzeile wird nur einmal geschrieben, aber hundertmal gelesen.“ Dieser Satz veranschaulicht die statistisch nachgewiesene Aussage, dass ca. 80% der Kosten von Software in die Wartung fließen. Nur sehr selten sind der Autor eines Programms und der Wartungsprogrammierer dieselbe Person. Quelltextkonventionen erleichtern deshalb die Wartung, denn eine einheitliche Gestaltung erlaubt es dem Leser, sich auf die wesentlichen Aspekte eines Quelltextes zu konzentrieren.

Für SE1 gilt: Nur ein Quelltext, der wirklich lesbar ist – sich an die hier genannten Konventionen hält – kann angemessen bewertet werden.

Die hier aufgeführten Konventionen fassen überwiegend die englischsprachigen Java-Konventionen der Firma Sun Microsystems (`http://java.sun.com/docs/codeconv/`) zusammen.

2 Klassenstruktur

2.1 Vor einer Klassendefinition steht immer ein **Klassenkommentar**, der den Zweck der Klasse beschreibt. Vor jeder Methode einer Klasse steht stets ein **Methodenkommentar**, der den Dienst beschreibt, den diese Methode einem Klienten bietet. Diese Kommentare nennen wir **Schnittstellenkommentare**.

Ein Quelltextbeispiel zur Orientierung:

```
import java.util.Date;

/**
 * Eine Klasse, die ein einfaches Modell von Girokonten
 * implementiert.
 * Diese Klasse dient speziell als Beispiel für die
 * Quelltextkonventionen und hat somit einige Punkte, die
 * in einer "echten" Kontoverwaltung anders gelöst werden
 * würden.
 *
 * @author Petra Becker-Pechau, Axel Schmolitzky
 * @version Oktober 2009
 */
class Girokonto
{
    // Der Saldo des Kontos in Cent
    private int _saldo;

    // Das Datum der letzten Änderung des Saldos
    private Date _datumLetzteAenderung = new Date();

    /**
     * Erzeuge ein Girokonto mit einem Eröffnungssaldo.
     * @param eroeffnungssaldo der Eröffnungssaldo in Cent.
     */
    public Girokonto(int eroeffnungssaldo)
    {
        _saldo = eroeffnungssaldo;
    }
```

```
    /**
     * Zahle einen Betrag auf das Konto ein.
     * @param betrag der einzuzahlende Betrag in Cent
     */
    public void einzahlen(int betrag)
    {
        _saldo = _saldo + betrag;
        _datumLetzteAenderung = new Date();
    }

    // Hier wurden für das Beispiel überflüssige Methoden herausgeschnitten

    /**
     * @return Saldo des Kontos in Cent.
     */
    public int gibSaldo()
    {
        return _saldo;
    }

    /**
     * @return das Datum der letzten Saldo-Änderung.
     */
    public Date gibDatumLetzteAenderung()
    {
        return _datumLetzteAenderung;
    }
}
```

2.2 Dieses Beispiel verdeutlicht die **Reihenfolge**, in der die Teile einer Klassendefinition aufgeführt sein sollten:

(Zustands-)Felder (Exemplarvariablen)
Konstruktoren
Methoden

2.3 Das Beispiel verdeutlicht auch, dass **diese Teile** gegenüber der öffnenden geschweiften Klammer der Klassendefinition **um 4 Zeichen eingerückt** sein sollten.

2.4 Inhaltlich verschiedene Abschnitte sollten **durch Leerzeilen voneinander getrennt** werden (Felder von Konstruktoren mit zwei Leerzeilen, Methoden voneinander mit einer etc.).

2.5 **Felder sollten immer `private`** deklariert und damit nur innerhalb der Klasse zugreifbar sein.

2.6 Es ist oft nützlich, **Felder speziell** zu **benennen**, im obigen Beispiel etwa mit einem Unterstrich beginnend. Andere Konventionen setzen einheitlich einen festen Buchstaben an den Anfang (etwa ein „f“ für *field* bzw. Feld), um nicht mit der Unterstrich-Regelung der Sun-Konventionen in Konflikt zu geraten. Durch eine solche Regelung ist aus dem Quelltext sehr leicht ersichtlich, an welchen Stellen auf Felder zugegriffen wird.

2.7 Eine Zeile sollte **nicht mehr als 70 Zeichen** lang sein – bis zu 80 sind im Extremfall zulässig. Dies ist eine Konvention, die in erster Linie lesbaren Ausdrucken dient.

3 Kommentierung

Kommentare sind technisch gesehen Abschnitte im Quelltext, die vom Compiler ignoriert werden. Sie sind also niemals Teil einer Programmausführung, sondern dienen ausschließlich der Dokumentation des statischen Quelltextes.

3.1 **Schnittstellenkommentare (für Klassen und Methoden)** sollten lediglich Auskunft über das „Was“ (einer Klasse, einer Methode) geben, nicht aber über das „Wie“. Sie sollten so formuliert sein, dass ein Klient der Klasse weiß, wie er die Klasse oder Methode (welche Parameter zu welchem Zweck etc.) benutzen soll und mit welchen Ergebnissen er bei einer Benutzung rechnen kann. Sie sollten jedoch nicht Details der Implementation verraten, die für einen Klienten irrelevant sind. Für Schnittstellenkommentare sollten die Java-Kommentarzeichen `/**` (Beginn) und `*/` (Ende) verwendet werden. Diese sollten nicht zum Auskommentieren längerer Programmabschnitte verwendet werden, da die Gefahr besteht, dass Zeilen nicht unmittelbar als auskommentiert erkennbar sind.

4 Blöcke, Deklarationen, Anweisungen

4.1 **Geschachtelte Blöcke** sollten stets um 4 Leerzeichen eingerückt sein.

4.2 **Tabulatoren** sollten bei der Einrückung vermieden werden, da diese auf jedem System anders dargestellt werden. In BlueJ ist ihre Benutzung unproblematisch.

4.3 **Blockklammern** (geschweifte Klammern) stets in **eigene Zeilen** schreiben.

4.4 **Pro Zeile sollte nur eine Deklaration oder Anweisung** stehen.

4.5 **Lokale Variablen sollten bei ihrer Deklaration initialisiert werden**, notfalls mit dem jeweiligen Standard-Wert.

4.6 **Variablendeklarationen** sollten immer am Beginn des jeweiligen Blockes stehen, in dem Sie verwendet werden. Eine Mischung von Deklarationen und Anweisungen erschwert die Lesbarkeit.

4.7 Bei gewöhnlichen Auswahlanweisungen – `if`-Anweisungen – sollte ein ggf. vorhandenes `else` stets **in einer eigenen Zeile** stehen. Allg. gilt: Verwende immer Blockklammern bei `if`-Anweisungen, diese sind zwar bei genau einer auszuführenden Anweisung nicht notwendig, erhöhen jedoch die Lesbarkeit des Textes. Außerdem ist dies **robuster** gegenüber Änderungen: Wenn beim Eintreten der Bedingung nicht nur eine, sondern auch eine weitere Anweisung ausgeführt werden soll, kann die zweite leicht hinzugefügt werden, ohne dass Klammern eingefügt werden müssen.

4.8 **Boolesche Ausdrücke** sollten immer in der Art geklammert werden, dass jeweils nur eine Prüfung, oder eine Verknüpfung zweier Prüfungen sich innerhalb eines Klammerpaares befindet.

Beispiel:

```
if ((x == y) && (x == z))
{
    ....
}
else
{
    ....
}
```

5 Benennungsregeln

In Java sind Groß- und Kleinschreibung signifikant (Java ist „case-sensitive“). Die Bezeichner `eins` und `Eins` beispielsweise sind deshalb verschieden.

5.1 **Bezeichner von Feldern** – und nur genau diese – können mit einem Unterstrich beginnen. Auf diese Weise sind Felder sofort im Quelltext erkennbar. Alle anderen Bezeichner enthalten keine Unterstriche. Wenn dieser Konvention gefolgt wird, sollte sie auch durchgängig (innerhalb eines Projektes etwa) eingehalten werden.

5.2 „**Neue“ Worte innerhalb eines Bezeichners** beginnen mit einem großen Buchstaben. Im Allgemeinen beginnen Bezeichner mit einem Kleinbuchstaben, lediglich Klassenbezeichner beginnen mit einem Großbuchstaben.

5.3 Für **Bezeichner von Wertkonstanten** gelten eigene Konventionen. Sie werden durchgängig aus Großbuchstaben gebildet, einzelne Wörter werden hier – und nur hier – mit einem Unterstrich getrennt.

5.4 **Klassen, Methoden und Variablen sollten „sprechend“ benannt werden**, etwa mit ganzen Begriffen oder Kurzsätzen, die die Funktion des Bezeichneten deutlich machen. **Methoden** sollten stets nach der Dienstleistung benannt werden, die sie liefern oder die sie anbieten.

Beispiel: `public String nameMitAnrede() {...};`