SE2, Quelltextkonventionen für Java

Modul: Softwareentwicklung II – Sommersemester 2015

Richtlinien für die Gestaltung von Java Quelltexten

CommSy-Projektraum ..... SE2 CommSy SoSe 2015
Ausgabedatum ..... 02. April 2015

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. Konventionen erleichtern das Verständnis von Quelltext, denn eine einheitliche Gestaltung erlaubt es dem Leser, sich auf die wesentlichen Aspekte eines Quelltextes zu konzentrieren.

Diese Konventionen basieren überwiegend auf den englischsprachigen Konventionen der Firma Sun Microsystems (heute Oracle)[1]. Sie geben diese Richtlinien jedoch nicht vollständig wieder, teilweise weichen sie auch von diesen ab, an anderen Stellen gehen sie über diese hinaus.

2 Quelltextorganisation

Ein Quelltextbeispiel zur Orientierung:

```
/**
 * Speichert eine Zahl. Diese Klasse ist sinnlos und dient nur als Beispiel
 * für die Quelltextkonventionen.
 *
 * @author SE2-Team
 * @version April 2015
 */
class Speicherplatz
{
    /**
     * Die Antwort auf die eine Frage.
     */
    public static final int DIE_ANTWORT = 42;

    private int _inhalt;

    /**
     * Initialisiert einen Speicherplatz.
     *
     * @param inhalt der anfängliche Inhalt des Speichers
     */
    public Speicherplatz(int inhalt)
    {
        _inhalt = inhalt;
    }

    /**
     * Setzt den Inhalt dieses Speicherplatzes.
     */
    public void setInhalt(int inhalt)
    {
        _inhalt = inhalt;
    }

    /**
     * Gibt den Inhalt dieses Speicherplatzes zurück.
     */
    public int getInhalt()
    {
        return _inhalt;
    }
}
```

[1] Oracle, *Code Conventions for the Java Programming Language,* http://www.oracle.com/technetwork/java/codeconv-138413.html, 1999.

2.1 Das Quelltextbeispiel zeigt die **Reihenfolge**, in der die Teile einer Klassendefinition geordnet sein sollten:

Symbolische Konstanten
Exemplarvariablen (Zustandsfelder)
Konstruktoren
Methoden

2.2 Klassenrümpfe, Methodenrümpfe, geschachtelte Blöcke etc. werden **um 4 Leerzeichen eingerückt**, um die Struktur des Quelltextes erkennbar zu machen. Die öffnende geschweifte Klammer steht dabei in einer eigenen Zeile.

Tabulatorzeichen sollten nicht zur Einrückung verwendet werden, weil sie auf verschiedenen Systemen unterschiedlich dargestellt werden. Eclipse kann so konfiguriert werden, dass beim Drücken der Tabulator-Taste anstelle eines Tabulatorzeichens die passende Anzahl Leerzeichen eingefügt wird.

2.3 Falls in einer Klassendefinition Klassen aus anderen Packages benutzt werden, stehen ihre Import-Anweisungen immer zu Anfang der Datei. Dabei sollten Sammelimporte vermieden werden (etwa `import java.util.*`), stattdessen sollte jede Klasse explizit einzeln importiert werden (z.B. `import java.util.Date`).

Eclipse bietet eine Funktion, die die Import-Anweisungen automatisch so organisiert: „Source – Organize Imports“ (Tastenkürzel: Strg+Umschalt+O). Die Import-Anweisungen werden dabei außerdem sortiert, gruppiert und unnötige Importe entfernt.

2.4 Eine Zeile sollte **nicht mehr als 80 Zeichen** lang sein. Das erleichtert beispielsweise den Vergleich verschiedener Versionen einer Datei, wenn man diese nebeneinander anzeigt.

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.

3.1 Wir unterscheiden **zwei wichtige Arten von Kommentaren**: Implementationskommentare, mit denen technische Hinweise auch zu einzelnen Anweisungen gegeben werden, und Schnittstellenkommentare, die sich auf eine ganze Klasse oder eine ganze Methode beziehen.

3.2 **Schnittstellenkommentare** sollten lediglich Auskunft über das „Was“ (einer Klasse, einer Methode, eines Konstruktors) geben, nicht aber über das „Wie“. Sie sollten so formuliert sein, dass ein Klient der Klasse weiß, wie er die Klasse oder Methode 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.

Schnittstellenkommentare sollten in Java-Quelltexten als Javadoc-Kommentare geschrieben werden. Dazu dienen die Kommentarzeichen `/**` (Beginn) und `*/` (Ende).

3.3 **Implementationskommentare** dokumentieren technische Details der Implementierung. Mit ihnen kann auch das „Wie“ einer Methode oder Klasse beschrieben werden, also etwa eine Erläuterung des verwendeten Algorithmus.

Für Implementationskommentare sollten die Java-Kommentarzeichen `//` verwendet werden. Der Compiler ignoriert von dieser Zeichenfolge an den Rest der Zeile.
Implementationskommentare können auch verwendet werden, um Programmteile kurzzeitig „abzuschalten“. In Eclipse steht die Funktion „Source – Toggle Comments“ (Tastenkürzel: Strg+7) zur Verfügung, um Quelltextzeilen und -abschnitte aus- und einzukommentieren.

3.4 Bei der Kommentierung von Quelltexten ist es hilfreich, die Metaphern von Lärm und Schweigen im Hinterkopf zu haben. Unter **Lärm** sind Kommentare zu fassen, die lediglich das wiederholen, was bereits unmittelbar aus dem Quelltext ersichtlich ist. Ein Beispiel:

```
zaehler = 0;  // zaehler auf Null setzen     Lärm!
```

Unter **Schweigen** hingegen wird verstanden, wenn Aspekte des Quelltextes, die nicht unmittelbar ersichtlich sind, nicht explizit gemacht werden. Wenn beispielsweise in einer Methode ein bekannter Algorithmus umgesetzt ist, dann sollte dieser Algorithmus auch in einem Kommentar genannt werden.

3.5 **Alle Klassendefinitionen werden mit einem Schnittstellenkommentar dokumentiert.** Der Kommentar sollte beschreiben, welche Funktion ein Exemplar der Klasse hat.

3.6 **Alle Methoden und Konstruktoren werden mit einem Schnittstellenkommentar dokumentiert,** auch private Methoden. Der Kommentar sollte mindestens einen Satz umfassen, der die Funktion der Methode kurz beschreibt.

Ebenso werden alle weiteren Bestandteile einer Klasse, die zu ihrer Schnittstelle gehören (z.B. nicht-private symbolische Konstanten) mit einem Schnittstellenkommentar dokumentiert.
Interne (private) Bestandteile sollten dokumentiert werden, wenn dies für das Verständnis hilfreich ist.

3.7 Methodenparameter sowie der Rückgabewert einer Methode sollten in ihrem Schnittstellenkommentar dokumentiert werden. Javadoc stellt dafür die Javadoc-Tags @param und @return zur Verfügung. Wenn die Kommentare allerdings keine zusätzlichen Informationen gegenüber den Bezeichnern und Typen liefern würden, kann auf diese verzichtet werden. Das ist häufig bei Getter- und Setter-Methoden der Fall (zum Beispiel auch im einleitenden Quelltext-Beispiel, siehe dort die Kommentare der Methoden `setInhalt` und `getInhalt`).

4 Blöcke, Deklarationen, Anweisungen

4.1 Pro Zeile sollte nur eine Deklaration oder Anweisung stehen.

4.2 Lokale Variablen sollten bei ihrer Deklaration initialisiert werden.

4.3 Bei bedingten Anweisungen (if-Anweisungen) sollten **immer Blockklammern** verwendet werden. Dies ist einheitlicher und robuster gegenüber Änderungen. Ebenso sollte auch als Schleifenrumpf immer ein Block verwendet werden.

Wenn eine if-Anweisung einen else-Zweig besitzt, sollte das Schlüsselwort **`else`** in einer eigenen Zeile stehen.

```
if (bedingung)
{
    macheEtwas();
}
else
{
    macheEtwasAnderes();
}
```

4.4 **Boolesche Ausdrücke** sind **defensiv** zu **klammern**, d.h. das Ergebnis sollte unabhängig von den Operatorpräzedenzen sein. So bleibt der Ausdruck gut lesbar, ohne dass der Leser sämtliche Operatorpräzedenzen auswendig wissen muss.

```
(x == y) && (x == z)
```

5 Benennungsregeln

Gut gewählte Namen, aus denen die Funktion eines Elements klar hervorgeht, können wesentlich dazu beitragen, die Lesbarkeit von Quelltexten zu verbessern.

In Java sind Groß- und Kleinschreibung signifikant (Java ist „case-sensitive“). Die Bezeichner `b` und `B` beispielsweise sind deshalb verschieden. Wenn ein Bezeichner sich aus mehreren Worten zusammensetzt, ist es in Java üblich, diese zusammen zu schreiben und jedes weitere Teilwort mit einem Großbuchstaben zu beginnen, `zumBeispielWieHier`; diese Konvention wird auch **camelCase** genannt.

5.1 Als Bezeichner für eine **Klasse** sollte ein Nomen verwendet werden, das beschreibt, worum es sich bei einem Exemplar der Klasse handelt. Bezeichner für Klassen beginnen mit einem Großbuchstaben. Beispiel: `Wecker`.

5.2 In einem Bezeichner für eine **Methode** sollte ein Verb auftauchen, das beschreibt, was die Methode tut. Bezeichner für Methoden beginnen mit einem Kleinbuchstaben. Beispiel: `schalteAlarmAus`.

5.3 **Variablen** sollten möglichst sprechend benannt werden, in der Regel mit einem Nomen oder einem Adjektiv. Bezeichner für Variablen beginnen mit einem Kleinbuchstaben. Beispiel: `weckzeit`, `istAktiv`.

Bezeichner von **Exemplarvariablen** – und nur diese – sollten mit einem Unterstrich beginnen. Wenn man dieser Konvention folgt, ist es sinnvoll, Eclipse entsprechend zu konfigurieren, um die Werkzeugunterstützung zu verbessern. Die entsprechende Einstellung kann in den Preferences vorgenommen werden unter „Java – Code Style“. Hier muss in die „Prefix list“ für „Fields“ ein Unterstrich eingetragen werden.

5.4 Bezeichner für **symbolische Konstanten** werden abweichend von den üblichen Namenskonventionen durchgängig aus Großbuchstaben gebildet, einzelne Teilwörter werden mit einem Unterstrich getrennt.
Beispiel: `MINUTEN_PRO_TAG`.

Achtung: Nicht jede Variable, die mit `final` deklariert wird, ist eine symbolische Konstante! Beispiel:
`private static final StringBuilder logger = new StringBuilder();`
logger ist kein Stellvertreter für eine Konstante wie 42, sondern referenziert ein veränderliches Objekt.

5.5 Für die **Sprache** von Bezeichnern gelten folgende Richtlinien:

- **Technische Begriffe** aus Java sollten **nicht eingedeutscht** werden.
- Umgekehrt sollten Begriffe, die aus der fachlichen Sprache des Anwendungsbereichs stammen, nicht unnötig ins Englische übersetzt werden. **Wenn** die **Fachsprache Deutsch** ist, sollten also für fachliche Objekte und Methoden auch **deutsche Bezeichner** verwendet werden.
- Der Name von Getter- und Setter-Methoden sollten jedoch immer mit `get` oder `set` beginnen, auch bei fachlichen Methoden. Diese Konvention ist in Java etabliert und wird teilweise von unterstützenden Werkzeugen vorausgesetzt.

5.6 Um Kompatiblitätsprobleme zu vermeiden, sollten für **Bezeichner nur lateinische Buchstaben und Ziffern** verwendet werden. In Kommentaren können auch andere Zeichen (zum Beispiel Umlaute) verwendet werden, wenn im Projekt eine geeignete, einheitliche Zeichenkodierung festgelegt wurde (in SE2 verwenden wir **UTF-8**).

6 Allgemeine Entwurfsregeln

6.1 **Exemplarvariablen** sollten **immer private** deklariert und damit nur innerhalb der Klasse zugreifbar sein.

6.2 **In längeren Methoden** sollten **return-Anweisungen** nur ganz oben oder ganz unten auftreten.

7 Konventionen für Tests

7.1 Der Bezeichner von **Testklassen** endet mit dem Wort „**Test**“. Üblicherweise testet eine Testklasse das Verhalten genau einer getesteten Klasse, in diesem Fall sollte der Name der Testklasse gebildet werden, indem an den Namen der getesteten Klasse das Wort „Test“ angehängt wird. Beispiel: `Liste` und `ListeTest`.

7.2 Der Bezeichner von **Testmethoden** sollte mit „**test**“ beginnen. Die Annotation `@Test` sollte auf einer eigenen Zeile vor dem Rest des Methodenkopfes stehen.

7.3 Bei Testmethoden ist **kein Schnittstellenkommentar** erforderlich.

7.4 Werden Methoden geschrieben, die vor oder nach jedem Test ausgeführt werden (Annotationen `@Before` und `@After` in JUnit 4), so sollten diese **`setUp`** bzw. **`tearDown`** genannt werden.

7.5 Bei der Benutzung von `assertEquals` sollte beachtet werden, dass der **erwartete Wert als erster Parameter** anzugeben ist.

```
@Test
public void testNeueListeIstLeer()
{
    Liste liste = new Liste();
    assertEquals(0, liste.anzahlElemente());
}
```