Java-Codierrichtlinien für den Fachbereich MNI

Dieses Dokument fasst die Codierrichtlinien für Java in der Lehre für den Fachbereich MNI zusammen.

1 Formatierung der Quellen

Der folgende Abschnitt fasst die wichtigsten Konventionen für die Formatierung von Java-Code zusammen und gibt Empfehlung bezüglich der gemeinsamen Arbeit am Code. Als Grundlage dienen die Sun Java Code Conventions [4].

- Als Zeichencodierung verwenden wir „Latin-1“ oder „UTF-8“[1].
- Jede Variable sollte in einer separaten Zeile stehen und initialisiert werden.
- Öffnende geschweifte Klammern beginnen immer in der selben Zeile wie das vorausgehende Statement, schließende stehen in einer separaten Zeile.
- Um die Lesbarkeit zu erhöhen, sollten lieber zu viele als zu wenige Klammern verwendet werden (vgl. [4, 10.5.1.]).
- Die Klammer- und Einrückungsregeln für Schleifen und Verzweigungen der Sun Java Code Conventions sollten beachtet werden [4, 7.4 - 7.9]. Bei der Verwendung der `switch`-Anweisung sollte zusätzlich jedes `case` eingerückt werden.
- Verwendet eine Entwicklergruppe ein Versionverwaltungssystem wie svn o.ä., dann sollte die Umformatierung von Quellcode möglichst unterbleiben, damit Versionen noch vernünftig vergleichbar bleiben. Verwendet man Eclipse, kann man dies durch folgendes Vorgehen erreichen:
  - Es sollte eine gemeinsame Formatvorlage in Eclipse verwendet und diese angewendet werden, bevor veränderter Code wieder eingecheckt wird.
  - Eclipse überprüft die Reihenfolge der Objektvariablen und Methoden und sortiert sie ggf. nach einem gemeinsam festgelegten Schema.
  - Auch die Importdateien können automatisch sortiert werden.

2 Namensgebung

- Grundsätzlich wird die Sprache der Anwendungsdomäne verwendet, außer es handelt sich um eingebürgerte Fachtermini der Informatik und Softwareentwicklung. Kommentare werden in der gleichen Sprache verfasst.
- Es sollten treffende Namen verwendet werden.
- Nomen sollten die Namen von Klassen bilden.

[1] Sind mehrere Entwickler beteiligt, muss sich auf ein gemeinsames Encoding geeinigt werden.

- Namen von Methoden sollten mit einem Verb beginnen.
- Interfaces werden nicht besonders kennzeichnet, sondern die verschiedenen Implementierungen werden als solche kenntlich gemacht. (z.B. `Connector` und `ConnectorImpl`)
- Package-, Methoden- und Variablennamen beginnen immer mit einem Kleinbuchstaben, Klassennamen werden aber am Anfang großgeschrieben.
- Zusammengesetzte Namen werden mit Binnenmajuskeln geschrieben.
- Die Verwendung des default-Packages ist zulässig z.B. bei Übungsaufgaben. Schreibt man aber Code, der auch in einem anderen Kontext eingesetzt werden soll, sollten aussagekräftige Paketnamen verwendet werden.
- Abkürzungen sollten nur verwendet werden, falls sie fachlich üblich sind (z.B. `XML`).
- Akronyme, die als Wort gesprochen werden, werden, bis auf den ersten Buchstaben, klein geschrieben (z.B. `Jar`), außer es sind andere Konventionen (z.B. `SAXParser`) üblich.

3 Spezifikation & Kommentierung

Bei der Kommentierung von Java-Sourcecode wird zwischen Implementierungs- und Dokumentationskommentaren unterschieden. Wir verwenden Dokumentationskommentare nur für die Spezfikation von Klassen, Interfaces und Schnittstellen von Methoden; Implementierungskommentare werden nicht mit Javadoc-Annotationen versehen – wer die Implementierungskommentare sehen möchte, soll gleich die Quelle studieren.

3.1 Implementationskommentare

Implementierungskommentare dienen dazu den Code für andere (oder auch für einen persönlich) an schwierigen Stellen verständlicher zu machen. Am besten der Code erklärt sich selbst, deshalb werden offensichtliche Dinge nicht als Kommentar wiederholt.

Implementierungskommentare werden als Blockkommentare durch `/*...*/` bzw. als Zeilenkommentare durch `//...` gekennzeichnet. Blockkommentaren sollte der Vorzug gegeben werden.

3.2 Dokumentationskommentare

Dokumentationskommentare werden von Javadoc erkannt und sollten für eine Spezifikation der Schnittstelle verwendet werden. Sie werden durch `/**...*/` kenntlich gemacht.

Die Javadoc-Kommentare sollten keinesfalls schon geschriebenen Code wiedergeben, sondern die äußere (abstrakte) Sicht beschreiben, die die Verwendung für den Benutzer erläutert.

Die folgende Liste hält sich im wesentlichen an die Empfehlungen von Sun [2], erweitert sie aber um Regeln für die Spezifikation:

- Der erste Satz des Kommentars einer Javaquelle wird von Javadoc als Zusammenfassung über die Klasse in der Übersichtstabelle angesehen. Dabei wird nach einem Punkt gefolgt von Leerzeichen oder Absatz gefolgt von einen Großbuchstaben geparst. In diesem Satz sollte auf weitere Punkte verzichtet werden, um die Erkennung sicher zu stellen.
- Da die Javadoc-Kommentare in HTML umgewandelt werden, können zur Strukturierung und Formatierung HTML-Tags verwendet werden. Hierbei sollte man sich auf einfache Tags wie `<br/>`, `<ul> <li>...</li>...</ul>` oder ähnliche beschränken.
- Klassen, Interfaces:
  - Der Kommentar sollte mit einer Formulierung beginnen, die das Konzept darstellt, etwa: „Ein Objekt dieser Klasse repräsentiert . . . “.
  - Für die Beschreibung des Konzepts wird die Sichtweise des Verwenders eingenommen, nicht die der Implementierung. Je nach Art kann sich das Konzept stark oder weniger stark von der Implementierung unterscheiden. Starke Unterscheidung ist etwa gegeben bei einem abstrakten Datentyp, der ein mathematisches Konzept repräsentiert wie eine Menge, weniger stark ist der Unterschied bei Klassen, die Entitätstypen entsprechen, wie z.B. einem Kunden-Datensatz.
  - In der Beschreibung des Konzepts kann deshalb auf bekannte mathematische Konzepte zurückgegriffen werden oder es kann Bezug genommen werden auf Attribute, sofern es für diese öffentliche Get- bzw. Set-Methoden gibt.
    Beispiel 1: „Ein Objekt dieser Klasse repräsentiert eine Menge ganzzahliger Werte M $= \{$a, b, c,. . . $\}$“
    Beispiel 2: „Ein Objekt dieser Klasse repräsentiert einen Kunden mit den Attributen Kundennummer, Name,. . . “ – vorausgesetzt es gibt Methoden `getKundennummer`, `getName`, . . .
  - Die Klasseninvariante beschreibt Eigenschaften des abstrakten Konzepts.
- Bei der Dokumentation der Methoden und der Konstruktoren sollten folgende Javadoc-Tags verwendet werden:
  - `@param`: Erläuterung der Parameter.
  - `@pre`: Vorbedingungen.
  - `@post`: Nachbedingungen, sofern durch `@return` nicht hinreichend erklärt.
  - `@return`: Rückgabewert der Methode.
  - `@modifies`: Seiteneffekte und Änderungen, die außerhalb der Methode durch sie entstehen können.
  - `@throws`: Exceptions und ihr Grund (siehe auch Abschnitt 4)

Es folgen zwei Beispiele:

Objekte der Klasse IntMenge sind Mengen von Integern. Das würde man in Zeiten der Java Collections natürlich nicht mehr selbst implementieren (siehe Regel: kenne und verwende die Bibliotheken), es dient hier nur der Illustration der Regeln zur Dokumentation.

```java
/*
  ------------------------------------------------------------------------
  Copyright (c) 2007 by NN, Fachhochschule Gießen-Friedberg.
  All rights reserved. -- oder eine Lizenz
  $Id: $
  ------------------------------------------------------------------------
  Historie:
  2007-11-30 erste Fassung
  ------------------------------------------------------------------------
*/

package mni;

import java.util.*;

/**
 *  Objekte der Klasse IntMenge repräsentieren Mengen ganzzahliger Elemente.
 *  <br/>
 *  Eine IntMenge M = {a, b, c, ...} enthält Integers a, b, c, die allesamt
 *  voneinander verschieden sind und keine Reihenfolge haben. Mit IntMengen kann
 *  man tun, was man mit Mengen so tun kann: Vereinigung, Durchschnitt usw.
 *
 *  <br/>
 *  Die Klasse IntMenge ist eine zustandsorientierte Klasse.
 *
 * @author NN
 *
 */

public final class IntMenge {

    /*
     * Die Menge wird intern durch eine sortierte Liste repräsentiert.
     *
     * Repräsentationsinvariante: die Implementierung sorgt dafür, dass die
     * Elemente in der Liste eindeutig und sortiert sind.
     *
     * Abstraktionsfunktion: [a, b, c ...] -> {a, b, c, ...}
     */
    private List<Integer> elemente = new Vector<Integer>();

    /**
     * @param x
     *                Wert, der zur Menge hinzugefügt werden soll
     * @post Die Menge this' = die Menge this vereinigt mit {x}
     * @modifies this
     */
    public void addElement(int x) {
        if (!isIn(x)) {
            elemente.add(x);
            Collections.sort(elemente);
        }
    }

    /**
     * @param x
     *                Wert, dessen Vorkommen interessiert
     * @return true, wenn x in der Menge enthalten ist, false sonst
     */
    public boolean isIn(int x) {
        return elemente.indexOf(x) >= 0;
    }
}
```

Objekte der Klasse Mixer repräsentieren Mixer für Cocktails oder Milchmixgetränke oder so. Hier nur zur Demo der Regeln für die Dokumentation.

```
1  /*
2    ------------------------------------------------------------------------
3    Copyright (c) 2007 by NN, Fachhochschule Gießen-Friedberg.
4    All rights reserved. -- oder eine Lizenz
5    $Id: $
6    ------------------------------------------------------------------------
7    Historie:
8    2007-11-30 erste Fassung
9    ------------------------------------------------------------------------
10 */
11
12 package mni;
13
14
15 /**
16  *
17  * Objekte der Klasse Mixer repräsentieren Mixer, mit denen man Cocktails und
18  * ähnliches herstellen kann.
19  * <br/>
20  * Mixer haben eine Geschwindigkeit "speed" und
21  * können befüllt sein "full" oder "!full".
22  * <br/><br/>
23  * Klasseninvariante:<br/>
24  * <ul>
25  *  <li>speed ist in {0, 1, 2, 3}</li>
26  *  <li>!full impliziert speed == 0</li>
27  * </ul>
28  * <br/><br/>
29  * Die Klasse Mixer ist eine zustandsorientierte Klasse.
30  *
31  * @author NN
32  */
33
34 public final class Mixer {
35
36     private int speed = 0;
37     private boolean full = false;
38
39     /**
40      * Liefert die Geschwindigkeit des Mixers.
41      * @return Geschwindigkeit speed des Mixers, speed in {0, 1, 2, 3}
42      */
43     public int getSpeed() {
44         return speed;
45     }
46
47     /**
48      * Verändert die Geschwindigkeit des Mixers in Einer-Schritten.
49      * Berücksichtigt, ob der Mixer befüllt ist.
50      * @param  speed Geschwindigkeit des Mixers
51      * @pre    speed in {0, 1, 2, 3} && <br/>
52      *       (speed == getSpeed()+1 || speed == getSpeed()-1) <br/>
53      *       speed != 0 nur wenn isFull()
54      * @post   speed' ist die gewünschte Geschwindigkeit
55      * @modifies this
56      */
57     public void setSpeed(int speed) throws IllegalArgumentException {
58         this.speed = speed;
59     }
60
61     /**
62      * Füllzustand des Mixers
63      * @return full -- Mixer befüllt?
```

```
64      */
65     public boolean isFull() {
66         return full;
67     }
68 
69     /**
70      * Füllt den Mixer; Mixer muss dabei leer sein
71      * @pre    !full
72      * @post   full == true
73      * @modifies this
74      * @throws   IllegalArgumentException – wenn Vorbedingung nicht erfüllt ist
75      */
76     public void fill() throws IllegalArgumentException {
77         if (this.full == true) {
78             throw new IllegalArgumentException("Vorbedingung von fill verletzt");
79         }
80         this.full = true;
81     }
82 
83     /**
84      * Leert den Mixer; ein bereits leerer Mixer bleibt leer
85      * @post   full == false
86      * @modifies this
87      */
88     public void empty() {
89         this.full = false;
90     }
91 
92 }
```

Um die Dokumentation mit unseren speziellen Tags (vgl. [3]) zu generieren, muss Javadoc mit folgende Kommandozeilenparameter aufgerufen werden:

`javadoc -public` *filename* `-tag pre:tcm:"Pre:" -tag post:tcm:"Post:"`
`-tag modifies:tcm:"Modifies:"`

4 Vertrag der Schnittstelle und Exceptions

Was die Verwendung von Vorbedingungen und ihre Überprüfung angeht, gibt es zwei Auffassungen:

1. Der Verwender einer Schnittstelle *muss* die Vorbedingung sicherstellen, ehe er eine Methode aufruft, infolgedessen sollte die Methode selbst die Vorbedingung nicht prüfen, ja sie *darf* sie nicht prüfen (Bertrand Meyer).
2. Der Verwender einer Schnittstelle *muss* die Vorbedingung sicherstellen, trotzdem *darf* eine Methode überprüfen, ob sie erfüllt ist und geeignet reagieren, z.B. durch einen Eintrag in ein Fehlerlog und eine ungeprüfte (*unchecked*) Exception wie `IllegalArgumentException`. Dies sollte man natürlich nicht tun, wenn man dadurch den Witz der Methode (man denke an binäre Suche!) zerstören würde.

Wir denken, dass beide Auffassungen ihre Berechtigung haben. In einem sehr disziplinierten Umfeld kann (1) sehr vernünfitg sein, insbesondere, wenn es auf Performanz ankommt. Unter anderen Umständen kann das Vorgehen nach (2) hilfreich sein, wenn die Verwender von Methoden sich nicht an den Vertrag halten und dann die explizite Exception, die auf die Verletzung aufmerksam macht, bei der Fehlersuche hilft.

Auch was die Verwendung von Exceptions angeht, gibt es unterschiedliche Auffassungen:

1. Eine Exception darf *nur* dann auftreten, wenn eine Methode ein „abnormales“ Ergebnis zur Folge hat. Insbesondere darf niemals eine Exception auftreten, wenn die Vorbedingung der Methode erfüllt ist.
2. Es kann durchaus sein, dass eine geprüfte (*checked*) Exception auftritt, auch wenn die Vorbedingung erfüllt ist, z.B. wenn eine „ungewöhnliche“ Situation auftritt.

Auch hier gibt es Argumente für beide Auffassungen, oder anders gesagt, Situationen, wo (1) angemessen ist und andere wo (2) passt.

Für die Spezifikation der Vertrags und das Verhalten der Methode ergeben sich aus dieser Diskussion folgende Möglichkeiten:

1a Die Vorbedingung ist *erfüllt* und der Verlauf der Methode ist normal:
die Methode erfüllt die Nachbedingung und es entsteht keine Exception.

1b Die Vorbedingung ist *erfüllt* und der Verlauf der Methode ist ungewöhnlich:
die Methode wirft eine geprüfte Exception und dokumentiert diese als Teil des Vertrags.

2a Die Vorbedingung ist *nicht erfüllt* und die Methode überprüft dies:
die Methode wirft eine ungeprüfte Exception und dokumentiert dies nach außen.

2b Die Vorbedingung ist *nicht erfüllt* und die Methode überprüft dies nicht, oder doch, gibt es aber nicht nach außen bekannt:
das Verhalten der Methode ist *undefiniert*, sie kann z.B. eine (ungeprüfte) Exception werfen, aber auch unsinnige Werte produzieren...

Beispiele für die Verwendung von Exceptions aus der Klassenbibliothek des JDK:

- Ein Beispiel für eine ungewöhnliche Situatation, also Werfen einer geprüften Exception, ohne dass die Vorbedingung verletzt ist:

  Die Methode

```
1  public static void Thread.sleep(int dauer) throws InterruptedException;
```

  der Klasse `java.lang.Thread` blockiert den aktuellen Thread `t`. Die Blockade endet entweder (1) abnormal, wenn vor Ablauf der angegebenen Dauer ein anderer Thread diesen Thread mittels `t.interrupt()` unterbrechen möchte, oder (2) normal, wenn die angegebene Zeitdauer ohne Unterbrechungsanforderung durch andere verstrichen ist.

  Anmerkung: Wäre man konsequent gewesen, müsste jede blockierende Methode die InterruptedException werfen. Dies ist aber nicht der Fall, z.B. bei `java.net.ServerSocket.accept()` muss man vor Aufruf der Methode `accept()` mittels `setSoTimeout(int dauer)` die accept()-Methode dazu bringen, nach Ablauf der angegebenen Zeitdauer eine geprüfte Ausnahme, SocketTimeoutException, zu werfen.

- Ein weiteres Beispiel für eine ungewöhnliche Situation, diesmal ohne explizite Verwendung von Nebenläufigkeitskonstrukten: Die Methode

```
1  public String readLine() throws IOException
```

  der Klasse `java.io.BufferedReader`.

In beiden Beispielen ist die Methode blockierend und der Blockadezustand wird aufgehoben durch die geworfene Exception.

5 Grundlegende Designempfehlungen

Dieser Abschnitt enthält einige grundlegende Empfehlungen zum Design von Java-Code, sie stammen weitgehend aus dem Buch „Effective Java“ von Joshua Bloch [1] (das ausdrücklich zur Lektüre empfohlen wird).

- Die Objektvariablen einer Klasse sollten `private` sein. Falls `public` oder `protected` als Sichtbarkeitsattribut genutzt wird, muss die Objektvariable (1) in der Schnittstelle dokumentiert werden und (2) der Grund dieser Entscheidung muss durch einen Kommentar erläutert werden. [Item 13 in Bloch's Buch]
- Klassen, die nicht zur Ableitung vorgesehen sind, sollten `final` sein.
- Interfaces werden in den Methodenköpfen ohne die Schlüsselwörter `abstract` und `public` verfasst.
- Nicht mehr benötigte Objektreferenzen sollten auf `null` gesetzt werden: Wer möchte, dass der Müll (vom Garbage Collector) abgeholt wird, muss ihn schon rausstellen. [Item 6]
- Falls eine Klasse die Methode `equals()` überschreibt (was bei werteorientierten Klassen die Regel ist), muss sie auch `hashCode()` überschreiben – und umgekehrt. [Item 9]
  Bei dieser Gelegenheit sollte auch erwogen werden, `compareTo` zu implementieren. [Item 12]
- Die Methode `toString()` sollte stets überschrieben werden – jeder Verwender wird es danken. [Item 10]
- Wer `clone()` überschreibt, muss genau wissen, was er tut. Ein Kopierkonstruktor ist häufig die bessere Lösung. [Item 11]
- Wenn Methoden einer Oberklasse überschrieben werden, sollte `@Override` verwendet werden, denn dadurch werden Tippfehler erkannt. Ab Java 1.6 ist dies sowohl für Klassen als auch für Interfaces[2] möglich. [Item 36]
- Die Verwendung einer vorhandenen Bibliothek ist stets der Eigenprogrammierung vorzuziehen, also: kenne und verwende Bibliotheken. (Dies gilt natürlich nicht, wenn im Programmierkurs die Aufgabe genau darin besteht, etwas zu implementieren, was es auch in Bibliotheken gibt). [Item 47]

[2] Wer Kompatibilität mit Java 1.5 gewährleisten möchte, muss darauf verzichten.

- Verwende für die Konkatenation von Strings nicht `String`, sondern `StringBuilder` oder `StringBuffer`. [Item 51]

```
1   // besser nicht:
2   String s = "";
3   for ( int i = 0; i < numItems(); i++
4     s += lineForItem(i);
5   }
6   // sondern:
7   StringBuffer sb = new StringBuffer();
8   for ( int i = 0; i < numItems(); i++
9     sb.append( lineForItem(i) );
10  }
11  String s = sb.toString();
```

- Verwende stets Referenzen vom Typ des Interfaces, nicht vom Typ einer Implementierung. [Item 52]

```
1   // schlecht
2   Vector<String> Kunden = new Vector<String>();
3   // gut
4   List<String> Kunden = new Vector<String>();
```

- Verwende Collections immer parametrisiert. Auf diese Weise werden Laufzeit-Typfehler [Item 23] vermieden (siehe vorheriges Beispiel).
- Löse sämtliche Unchecked Warnings auf [Item 24]. Kann dies nicht allein durch den Code geschehen, stelle sicher, dass die Warnung unberechtigt ist und Typsicherheit besteht und füge `@SuppressWarnings("unchecked")`ein. Es sollte nie ein entsprechender Kommentar fehlen.
- Verwende Exceptions nur für die Fehlerbehandlung, nicht für die Ablaufsteuerung. [Item 57]

```
1   // niemals
2   try {
3     int i = 0;
4     while (true)
5       a[i++].f();
6   } catch (ArrayIndexOutOfBoundException e){
7   }
8   //sondern
9   for ( int i = 0; i < a.length; i++ )
10    a[i].f();
```

- Standardexceptions sollte man verwenden, wenn sie passen; man muss dann keine eigenen erfinden. [Item 60]
- Exceptions nicht ignorieren oder verstecken. [Item 65]

```
1   // ist sehr suspekt
2   try {
3     ....
4   } catch (SomeException e) {
5   }
```

- Wird eine Klasse in einem nebenläufigen Kontext verwendet, muss sich der Verwender um die Threadsicherheit Gedanken machen. Der Entwickler einer Klasse sollte seine Entscheidungen dokumentieren, insbesondere, ob die Klasse threadsicher ist. Findet man keine solche Dokumentation, muss man davon ausgehen, dass die Klasse *nicht* threadsicher ist.

Metaregel

Diese Regeln können natürlich kreativ angewendet werden, also auch im Sinne des Erfinders übertreten werden, wenn gute Gründe dafür sprechen.

Nachbemerkung

Kommentare, Kritik und Verbesserungsvorschläge unbedingt erwünscht, wenden Sie sich an einen der Autoren.

P.S. Um die Richtlinie kurz zu halten, haben wir versucht nur das Wichtigste zu nennen, die Auswahl hat sicherlich was Subjektives.

Literatur

[1] Joshua Bloch. *Effective Java - Second Edition*. Addison-Wesley, 2008.

[2] Sun Microsystems. *How to Write Doc Comments for the Javadoc Tool*. URL `http://java.sun.com/j2se/javadoc/writingdoccomments/index.html`.

[3] Sun Microsystems. *Javadoc - The Java API Documentation Generator*. URL `http://java.sun.com/j2se/1.5.0/docs/tooldocs/windows/javadoc.html`.

[4] Sun Microsystems. *CodeConventions*, 1999. URL `http://java.sun.com/docs/codeconv/CodeConventions.pdf`.

Autor: Berthold Franzen, Bodo Igler, Nadja Krümmel, Thomas Letschert, Burkhardt Renz, Institut für SoftwareArchitektur.