Bundesverwaltungsamt

IsyFact-Standard

Java Programmierkonventionen

Version 2.6
13.05.2015

IsyFact Standards

„Java Programmierkonventionen“
des Bundesverwaltungsamts ist lizenziert unter einer
Creative Commons Namensnennung 4.0 International Lizenz.

Die Lizenzbestimmungen können unter folgender URL heruntergeladen werden: http://creativecommons.org/licenses/by/4.0

Ansprechpartner:

Referat Z II 2
Bundesverwaltungsamt
E-Mail: isyfact@bva.bund.de
Internet: www.isyfact.de

Dokumentinformationen

Dokumenten-ID:	Java-Programmierkonventionen.docx

Inhaltsverzeichnis

1. Einleitung

In diesem Dokument werden die Java-Programmierkonventionen der IsyFact beschrieben.

Die Einhaltung der Programmierkonventionen fördert die Wartbarkeit und Weiterentwicklungsfähigkeit der nach IsyFact-Standards erstellen Software.

2. Motivation und Referenzen

Von den Lebensphasen einer Software macht die Erstellungsphase gegenüber der Erweiterungs- und Wartungsphase nur einen kleinen Teil aus. Es ist davon auszugehen, dass im Laufe der Zeit unterschiedliche Personen, Teams oder Dienstleister an einem Softwaresystem arbeiten. Allen Beteiligten wird die Einarbeitung deutlich erleichtert, wenn die Software „aus einem Guss“ erscheint. Hält sie sich noch an allgemein etablierte und bekannte Standards, verkürzt sich die Einarbeitungszeit noch weiter. Um das zu erreichen, müssen Konventionen für die Programmierung festgelegt werden.

Die in diesem Dokument beschriebenen Richtlinien sind für die Erstellung von Programmen mit der IsyFact verbindlich. Sie bauen auf [Vermeulen2000] auf und präzisieren sie an einigen Stellen. Die in [Vermeulen2000] beschriebenen Konventionen beruhen auf den Sun Coding Conventions [Sun1997] und den Coding Standards der Firma Ambysoft [Ambler1999].

3. Grundsätzliches

PROGRAMMIERE IMMER IM STIL DES ORIGINALS!

Wird bestehender Programmcode verändert, dann werden die Änderungen immer im Stil des schon vorhandenen Codes programmiert, auch wenn der Code dadurch nicht die unten folgenden Richtlinien erfüllt. Bestehender Code, der nach anderen Richtlinien erstellt wurde, wird nicht im Rahmen von Wartungsmaßnahmen an andere Richtlinien angepasst, nur der Richtlinien wegen.

DOKUMENTIERE ABWEICHUNGEN!

Falls bestimmte Richtlinien nicht angewendet werden können/sollen, ist der technische Chef-Designer des Projektes zu involvieren. Er entscheidet darüber, ob die Abweichung zulässig ist. Abweichungen müssen immer im Entwicklerhandbuch des Projektes mit Begründung dokumentiert werden. Bevor eine Richtlinie verletzt wird, sollte man sicher sein, dass man die Motivation der Regel verstanden hat und die Konsequenzen der Nicht-Einhaltung beurteilen kann.

4. Namenskonventionen

4.1. Sprache

Die Sprache von Bezeichnern hängt davon ab, was mit ihnen referenziert wird.

Die Sprache ist eine Mischung aus deutsch und englisch. Für technische Bezeichner wird Englisch verwendet, für fachliche Bezeichner Deutsch. In Bezeichnern werden keine Umlaute und kein ß verwendet.

Beispiele: **`setMeldung()`**, **`suchePerson()`**

Motivation

Der Bruch zwischen den Sprachen fällt so mit dem Bruch zwischen technischem und fachlichem Code zusammen. Komplett deutsche oder komplett englische Bezeichner hätten dagegen folgende Nachteile:

- Komplett englische Bezeichner würden es erfordern, alle Fachbegriffe zu übersetzen. Alle am Projekt beteiligten Personen müssten diese „Vokabeln“ neu lernen.
- Komplett deutsche Bezeichner wirken sehr verkrampft, wenn sie Bibliotheksklassen mit englischen Bezeichnern nutzen oder (z. B. durch Ableitung) erweitern.
- Komplett deutsche Bezeichner führen zu Irritationen und Problemen, da Java bestimmte Namenskonventionen zum Beispiel bei Beans voraussetzt (getXXX und setYYY).

Java erlaubt zwar Umlaute in Bezeichnern, im Falle von Klassennamen müssen dann jedoch auch die Dateinamen Umlaute enthalten. Dies wird nicht von allen Betriebssystemen unterstützt beziehungsweise führt beim Übertragen von Dateien zwischen Systemen leicht zu Problemen.

4.2. Allgemeine Regeln

SPRECHENDE NAMEN WÄHLEN!

Es sollen möglichst „sprechende“ (selbsterklärende) Namen verwendet werden. Abkürzungen, zum Beispiel durch das Weglassen von Vokalen, sind grundsätzlich zu vermeiden. Ausnahmen dürfen bei temporär verwendeten Variablen (zum Beispiel Zählervariablen in einer **`for`**-Schleife) gemacht werden, wenn die Verwendung im Kontext klar ist.

GÄNGIGE BEZEICHNER BENUTZEN!

Es sollen nur gängige Bezeichner verwendet werden. Existiert ein Glossar, sind die dort aufgeführten Bezeichner zu verwenden.

NUR DER ERSTE BUCHSTABE EINER ABKÜRZUNG GROß!

Zur besseren Unterscheidung der Namensbestandteile eines Bezeichners wird bei Abkürzungen nur der erste Buchstabe der Abkürzung groß geschrieben.

Falsch: `loadXMLDocument()`
Richtig: `loadXmlDocument()`

Das erleichtert die Lesbarkeit, insbesondere wenn zwei Abkürzungen hintereinander folgen.

KEINE UNTERSCHEIDUNG BEI GROß-/KLEINSCHREIBUNG!

Es dürfen nicht mehrere Namen verwendet werden, die sich ausschließlich durch abweichende Groß-/Kleinschreibung unterscheiden.

4.3. Dateinamen

Um Einheitlichkeit bei den Dateinamen sicherzustellen, sind Namenskonventionen für Dateiendungen und gebräuchliche Dateinamen notwendig.

Dateinamen enthalten keine Sonderzeichen, Umlaute oder Leerzeichen. Erlaubt sind die Buchstaben von A-Z (groß und klein), Ziffern, der Unterstrich, der Mittelstrich und der Punkt. Datumsangaben werden dem Dateinamen vorangestellt und erfolgen im Format <JJJJ-MM-TT_Dateiname>.

Tabelle 1: Dateiendungen

Endung	Typ bzw. Beschreibung
`.properties`	Datei mit Konfigurationsparametern

In der nachfolgenden Tabelle werden gebräuchliche Dateinamen aufgeführt.

Tabelle 2: Gebräuchliche Dateinamen

Dateiname	Typ bzw. Beschreibung
`index.html`	Der Name der Datei, in der eine zusammenfassende Beschreibung des Dateiverzeichnisses steht, das kein Package ist.
`package.html`	Der Name der Datei, in der eine zusammenfassende Beschreibung des Packages steht.

4.4. Bezeichner und Kommentare

Mehrere Wörter werden bei zusammengesetzten Bezeichnern direkt aneinander geschrieben und nicht durch Sonderzeichen getrennt. Die einzelnen Wörter beginnen jeweils mit einem Großbuchstaben.

Falsch: **`Number_Formatter`**
Richtig: **`NumberFormatter`**

Ansonsten gelten die Regeln der nachfolgenden Abschnitte.

4.4.1 Package-Namen

STANDARD-PACKAGE-STRUKTUR VERWENDEN!

Die Package-Struktur folgt einer Konvention, die aus der fachlichen und technischen Referenzarchitektur hergeleitet wird:

```
<organisation>.<domäne>.<anwendungssystem >.<layer>.<subsystem/
komponente>. ...
   <domäne>
      = (Name gemäß fachlicher Architektur, z. B. „cd“)
   <anwendungssystem->
      = (Name gemäß fachlicher Architektur, z.B. „registercd“)
   <layer>
      = common | gui | batch | service | core | persistence
   <subsystem/komponente>
      =  <für Anwendungssystem> | <subsystem/komponente>
         <-
         <für Anwendungssystem>
           = ... (Name der Fachanwendung bzw. querschnittlichen Komponente
gemäß fachlicher Architektur)
```

Unterhalb von `<subsystem/komponente>` werden die Packages projektspezifisch strukturiert.

KEINE SONDERZEICHEN VERWENDEN!

Der Anwendungsname in **`<anwendungssystem->`** wird an Java-Package-Konventionen angepasst. Leer- und Sonderzeichen in den Anwendungsnamen werden gestrichen.

PACKAGE-NAMEN IMMER KLEIN SCHREIBEN!

Namen von Packages dürfen nur Kleinbuchstaben enthalten.

RICHTIG

`de.bund.bva.cd.registercd.persistence.meldung`

`de.bund.bva.cd.registercd.service.auskunft`

`de.bund.bva.pliscommon.logging.common.layout`

Falsch

```
de.bund.bva.cd.CDRegister.persistence.meldung

de.bund.bva.cd.register.cd.persistence.meldung

de.bund.bva.cd.registercd.admin.service
```

4.4.2 Klassen- und Interface-Namen

ERSTER BUCHSTABE IMMER GROß!

Bei Klassen- und Interface-Namen wird der erste Buchstabe jedes Teilwortes immer groß geschrieben.

Beispiele: **`DemoClass`**, **`PrintStream`**, **`ActionListener`**

SUBSTANTIVE ALS KLASSENNAMEN!

Für die Namen von Klassen sind Substantive zu verwenden. Besteht der Zweck der Klasse ausschließlich oder zum größten Teil aus der Implementierung eines Interfaces, dann wird die Klasse so genannt wie das Interface, ergänzt um das Suffix „**`Impl`**".

Beispiele: **`Meldung`**, **`KundeDaoImpl`**

PLURAL FÜR ZUSAMMENFASSUNGEN!

Für Klassen, die Dinge zusammenfassen, soll der Plural verwendet werden.

Beispiele: **`LineMetrics`**, **`Beans`**, **`Types`**, **`Sachverhalte`**

BEI INTERFACES SUBSTANTIVE ODER ADJEKTIVE VERWENDEN!

Bei Interfaces soll der Bezeichner ein Substantiv oder ein Adjektiv sein. Namen von Interfaces werden NICHT mit dem Präfix „**`I`**" versehen. Wenn Interface und Implementierung eigentlich gleiche Namen haben, wird die Implementierung mit dem Suffix „**`Impl`**“ versehen.

Beispiele: **`ActionListener`**, **`Runnable`**, **`Accessable`**

4.4.3 Methodennamen

METHODENNAMEN BEGINNEN IMMER MIT EINEM VERB!

Methodennamen sind Verben und beginnen immer mit einem Kleinbuchstaben. Danach wird der erste Buchstabe eines jeden Teilwortes groß geschrieben. Teilworte werden nicht durch Sonderzeichen getrennt, insbesondere nicht durch Unterstriche. Nachfolgende Teilworte können Substantive sein.

Beispiel: **`doSomething`**, **`getStrasse`**, **`setName`**

JAVABEANS-KONVENTIONEN EINHALTEN!

Die JavaBeans-Konventionen müssen eingehalten werden: Lesen von Eigenschaften mittels **`getProperty()`** bzw. **`isProperty()`** für Booleans, Schreiben von Eigenschaften mittels **`setProperty()`**.

4.4.4 Variablennamen

Für die Vergabe von Variablennamen gilt: Je globaler die Sichtbarkeit einer Variable ist, desto aussagekräftiger (und ggfls. länger) sollte der Name sein. Das schließt nicht aus, das ein für drei Zeilen gültiger Schleifenzähler „**`i`**“ heißt.

ALS VARIABLENNAMEN SUBSTANTIVE VERWENDEN UND IMMER KLEIN BEGINNEN!

Variablennamen beginnen immer mit einem Kleinbuchstaben und sind ein Substantiv. Danach wird der erste Buchstabe eines jeden Teilwortes groß geschrieben. Teilworte werden nicht durch Sonderzeichen getrennt.

Beispiel: **`mySampleVariable`**

PLURAL FÜR BEZEICHNER VON SAMMLUNGEN!

Für die Bezeichner von Sammlungen sind Namen im Plural zu verwenden.

Beispiel: **`auftraege`**, **`auftragsPositionen`**, **`kunden`**

STANDARDS FÜR TEMPORÄRE VARIABLEN EINSETZEN!

Folgende Bezeichner sind für die Bezeichner von temporären Variablen zu verwenden:

Integer	**`i, j, k`**
Character	**`c, d, e`**
Koordinaten	**`x, y, z`**
Object	**`o`**
Stream	**`in, out, inOut`**
String	**`s, t`**

KEINE PRÄFIXE AUßER `THIS.` VERWENDEN!

Außer dem Präfix **`this.`** bei Instanzvariablen werden keine Präfixe für Klassen-, Instanzvariablen und für Parameter verwendet.

4.4.5 Konstanten

KONSTANTEN IMMER GROß!

Die Bezeichner von Konstanten werden nur mit Großbuchstaben geschrieben. Jedes Teilwort wird durch einen Unterstrich getrennt. Bei jeder Konstante ist zu überlegen, ob sie nicht durch das Typesafe-Enum-Pattern oder eine Konfigurationsvariable aus einer Datei/Datenbank ersetzt werden kann.

Beispiele: **`A_MAGIC_NUMBER`**, **`MAX_VALUE`**, **`MIN_VALUE`**

5. Formatierung

Hinweis

Die Formatierung des Quellcodes gemäß der nachfolgenden Regeln kann durch den Eclipse Code Formatter automatisch vorgenommen werden.

5.1. Einrückungen und Klammerposition

Für das Einrücken sind immer **vier Leerzeichen** zu verwenden. Bei Code-Blöcken wird die öffnende Klammer „**{**" immer als letztes Zeichen der Zeile gesetzt, die den Codeblock einleitet. Die schließende Klammer „**}**" wird immer in einer neuen Zeile nach dem Block positioniert und links an dem ersten Zeichen der einleitenden Zeile ausgerichtet.

Zur Einrückung des Textes **niemals Tabulatoren**, sondern immer Leerzeichen verwenden, da Tabulatoren von verschiedenen Werkzeugen unterschiedlich interpretiert werden können.

Geschweifte Klammern sollen auch dann verwendet werden, wenn innerhalb eines Blocks nur ein Statement vorhanden ist und somit syntaktisch auf deren Verwendung verzichtet werden könnte.

Beispiele für Anwendung der Formatierungsregeln (• steht für ein Leerzeichen):

```
public•class•MyClass•{
••••statements;
}
```

```
if•(condition)•{
••••statements;
}•else•{
••••statements;
}
```

5.2. Leerzeichen und Leerzeilen in Kommandos und Ausdrücken

Es wird empfohlen, Leerzeichen wie folgt zu verwenden:

- zwischen einem Schlüsselwort und einer direkt darauf folgenden „**{**" Klammer
- zwischen der Klammer „**)**" bzw. „**}**" und einem direkt darauf folgenden Schlüsselwort
- zwischen einer Klammer „**)**" und einer direkt darauf folgenden Klammer „**{**"
- nach einem Komma (z. B. bei einer Methode mit mehreren Parametern)

- zwischen einem binären (ternären) Operator (außer dem Punktoperator) und dem vorausgehenden und dem nachfolgenden Ausdruck
  Beispiel: **double•length•=•Math.sqrt(x•*•x•+•y•*•y);**

  Dies gilt insbesondere für „•=•" und „•==•"

Leerzeilen eignen sich zur Trennung logischer unabhängiger Teile des Codes. Methoden werden durch eine Leerzeile voneinander getrennt. Auch innerhalb einer Methode können Leerzeilen die logische Trennung von Blöcken verdeutlichen.

5.3. Aufteilen langer Codezeilen

Die Zeichen pro Zeile sind auf eine lesbare Anzahl zu begrenzen. Es sollten niemals mehrere Anweisungen in einer Zeile codiert werden. Wenn ein Ausdruck nicht in eine Zeile passt, so ist sie so zu trennen, dass das Ergebnis sinnvoll lesbar ist. Es folgen Hinweise, wo unter Umständen sinnvoll getrennt werden und wie getrennte Zeilen formatiert werden könnten: Eine Zeile in der Eclipse-Entwicklungsumgebung fasst bei Verwendung der default-Einstellungen ca. 110 Zeichen.

- Hinter einem Komma
- Vor einem Operator (+, -, etc.)
- Bei geschachtelten Ausdrücken möglichst weit außen
- Die neue Zeile wird so eingerückt, dass sie unter dem Anfang des Teilausdrucks steht, den man trennt

Falls die obigen Regeln zu unleserlichen Einrückungen führen, wird einfach um acht Zeichen eingerückt

Das Umbrechen einer Zeile geschieht möglichst nach einem Komma. Die nächste Zeile wird dann an dem Ausdruck vor dem Komma ausgerichtet.

6. Dokumentation

Folgende Grundsätze sind beim Schreiben von Dokumentation und Kommentaren zu befolgen:

CODE UND DOKUMENTATION MÜSSEN IMMER ÜBEREINSTIMMEN!

> Wenn Code verändert wird, muss sichergestellt werden, dass die entsprechenden Kommentare und die Dokumentation weiter zum Code passen. Nach jedem Refactoring muss ein Überprüfen und eventuelles Anpassen der Dokumentation erfolgen.

KOMMENTARE DEUTLICH IN AKTIVER SPRACHE FORMULIEREN UND FLOSKELN VERMEIDEN!

> Für die technische Dokumentation hat sich eine klare und schnörkellose Sprache bewährt.

Bei der Dokumentation von Programmcode sind zwei Adressatenkreise zu unterscheiden:

Personen, die den Code einsetzen, d. h. nutzen wollen. Sie sind an den öffentlichen Programmierschnittstellen der Packages und der Klassen bzw. Interfaces interessiert, also an der **Außensicht**.

Personen, die den Code warten und weiterentwickeln müssen. Sie sind auch an den öffentlichen Programmierschnittstellen interessiert, aber vor Allem auch an den privaten Schnittstellen und der internen Implementierung, also der **Innensicht**.

Hinweis

Beim Schreiben der Dokumentation sollte man immer davon ausgehen, dass der Leser zwar Java programmieren kann, sich aber nicht mit dem Code und den Zusammenhängen auskennt. Wenn die Software lange nicht mehr "angefasst" werden musste, kann das sogar der Autor der Software selbst sein, der sich anhand der Dokumentation wieder "hineindenken" muss.

In Java wird zwischen Dokumentationskommentaren (`/**...*/`) und Implementierungskommentaren (`/*...*/`, `//`) unterschieden.

6.1. Dokumentationskommentare (Javadoc)

Prinzipiell müssen alle Klassen, Interfaces und Methoden einen Dokumentationskommentar (eine Außensicht) enthalten. Ausnahmen sind im Einzelfall anonyme innere Klassen und ihre Methoden sowie Implementierungsklassen von Interfaces (dort mit @see auf die Interface-Dokumentation verweisen). Es wird empfohlen, je Package eine Datei „package-info.java" zu erzeugen, die das Zusammenspiel von Klassen/Interfaces in dem Package erläutert.

Für die Erstellung von Dokumentationskommentaren gelten die folgenden Regeln:

ALLE DOKUMENTATIONSKOMMENTARE WERDEN EINHEITLICH FORMATIERT!

- Schlüsselworte und Bezeichner im beschreibenden Text werden mit dem HTML-Tag **<code>...</code>** formatiert.
- Programmcode wird im beschreibenden Text mit dem HTML-Tag **<pre>...</pre>** formatiert. Damit wird gewährleistet, dass eine Darstellung des Codes in „dicktengleicher Schrift“ (Nichtproportionalschrift, Festbreitenschrift oder Monospaced Font) erfolgt und Einrückungen so wiedergegeben werden, wie sie beim Editieren eingegeben wurden. Es ist darauf zu achten, dass alle Leerzeichen berücksichtigt werden und kein automatischer Zeilenumbruch erfolgt.
- Nicht mehr zu verwendende Konstrukte werden als **@deprecated** gekennzeichnet.

Beispiel:

```
/**
 * Beschreibender Text für zu kommentierendes Element.
 *
 * @tag Beschreibender Text für dieses Tag
 */
```

DER ERSTE SATZ EINES DOKUMENTATIONSKOMMENTARS MUSS ALLEINE STEHEN KÖNNEN!

Javadoc verwendet den ersten Satz in einer Beschreibung als Kurzbeschreibung des zu dokumentierenden Elements (Klasse, Schnittstelle, Methode, Attribut).

JAVADOC TAGS WERDEN IN EINER EINHEITLICHEN REIHENFOLGE VERWENDET!

Zu jeder Klasse und jeder Schnittstelle (Interface) wird mindestens ein Autor (**`@author`**) genannt, der für diese Klasse der Ansprechpartner ist. Mehrere Autoren werden in chronologischer Reihenfolge aufgeführt (Ersteller zuerst). Werden mehrere Autoren angegeben, ist der aktuelle Ansprechpartner immer der zuletzt genannte Autor. Es wird ein **`@version`**-Tag gesetzt, das über das Konfigurationsmanagement automatisch aktualisiert wird.

Beispiel:

```
/**
 * Beschreibung...
 * Das folgende Beispiel benutzt ein
 * <code>Class</code> Objekt um den Klassennamen
 * des Objekts auszudrucken:
 *
 * <pre>
 * void printClassName(Object o) {
 *
 *
 *
 *
 System.out.println("Die Klasse von "
                      + o
                      + " ist "
                      + o.getClass().getName());
 * </pre>
 * ...
 *
 * @author
 * @version
 *
 * @see
 * @since
 * @deprecated
 */
```

Jeder Parameter einer Methode wird mit einem **`@param`**-Tag beschrieben. Das **`@return`**-Tag wird nur verwendet, wenn der Rückgabewert der Methode ungleich **`void`** ist. Jede checked Exception, die in der **`throws`**-Klausel der Methode aufgeführt ist, wird mit einem **`@exception`**-Tag kommentiert.

Beispiel:

```
/**
 * Beschreibung.
 *
 * @param
 * @return
 * @exception
 *
 * @see
 * @since
 * @deprecated
 */
```

Kommentare zu Attributen sehen zum Beispiel wie folgt aus:

```
/**
 * Beschreibung.
 *
 * @see
 * @since
 * @deprecated
 */
```

`@see`-Tags sind sparsam zu verwenden, denn diese Verlinkung muss manuell gepflegt werden. Mehrere **`@see`**-Tags werden gemäß ihrer „Entfernung" von der aktuellen Stelle aufgeführt (Dokumenten-Navigation, Namensqualifikation). Innerhalb einer Gruppe überladener Methoden werden die Methoden gemäß der Anzahl Parameter aufgelistet.

Beispiel:

```
/**
 * ...
 * @see #field
 * @see #Constructor()
 * @see #Constructor(Type...)
 * @see #method()
 * @see #method(Type...)
 * @see Class
 * @see Class#field
 * @see Class#Constructor()
 * @see Class#Constructor(Type...)
 * @see Class#method()
 * @see Class#method(Type...)
 * @see package.Class
 * @see package.Class#field
 * @see package.Class#Constructor()
 * @see package.Class#Constructor(Type...)
 * @see package.Class#method()
 * @see package.Class#method(Type...)
 * @see package
 * @see <a href="URL#label">label</a>
 * @see "String"
 */
```

6.2. Implementierungskommentare

Code sollte immer so geschrieben werden, dass er selbsterklärend ist und nicht oder nur sehr wenig inline kommentiert werden muss. Implementierungskommentare begründen Designentscheidungen, die aus dem Code nicht allein ersichtlich sind, oder sie erklären aufwändige Algorithmen. Sie wiederholen nicht den Code in Prosa (Negativbeispiel: "Erhöhe Schleifenzähler um 1."). Implementierungskommentare werden für folgende Zwecke eingesetzt:

- Erklärung spezieller oder komplizierter Ausdrücke
- Erläuterung von Designentscheidungen auf Code-Ebene
- Quellenhinweise für komplexe Algorithmen
- Erläuterung von Fehlerbehebungen und Workarounds
- Hinweis auf Notwendigkeit zur Optimierung und Überarbeitung
- Benennung bekannter Probleme und Limitierungen
- Verzierungen (aus "*****" gemalte Rechtecke oder Trennlinien) sind zu unterlassen.

Wenn eine Stelle im Code noch nicht fertig ist und später kontrolliert oder überarbeitet werden soll, so ist sie mit „**`// TODO: Grund`**" zu markieren. Eclipse zeigt diese speziellen Kommentare analog zu Fehlern in einer ToDo-Liste an.

6.3. Änderungshistorie

Es wird immer der Ist-Zustand beschrieben. Änderungen werden in der Änderungshistorie beschrieben. Kommentare werden nicht dazu verwendet, alte Versionen des Codes zu deaktivieren. Wenn man alte Versionen wiederherstellen möchte, so ist dazu auf das Konfigurationsmanagement zurückzugreifen. Für die Analyse von Codeänderungen sind die bekannten DIFF-Werkzeuge[1] einzusetzen.

Änderungen werden mittels eines Version Control System (z. B. Subversion) nachverfolgt. Dazu ist es zwingend erforderlich, die durchgeführten Änderungen bzw. die Ursache dafür beim Check-In ausreichend zu dokumentieren. Die Dokumentationssprache hierfür ist deutsch. Darüber hinaus werden keine Versionsinformationen im Source Code gepflegt.

[1] Werkzeug, mit dem sich Unterschiede zwischen zwei Textdateien synoptisch darstellen lassen (diff unter Unix; .UltraEdit)

7. Codestruktur

7.1. Imports

Import-Statements werden nach Packages sortiert und gruppiert. Folgende Import-Statements sind zu vermeiden:

- *-Importe
- sun.*-Importe
- redundante oder nicht genutzte Importe

7.2. Deklarationen in Klassen

- Variablenbezeichner dürfen sich nicht überdecken. Zum Beispiel darf eine lokale Variable nicht genauso heißen wie ein Attribut der Klasse.
- Lokale Variablen sind in dem Scope zu deklarieren, in dem sie auch verwendet werden.
- Falls die Variable nicht selbsterklärend ist, wird sie mit einem kurzen einzeiligen Kommentar beschrieben.

7.3. Verwendung der Kurzschreibweisen

Seit Java 1.5 gibt es eine Kurzschreibweise für verschiedene Programmierkonstrukte (z.B. Bedingungen, Schleifen). Folgende Kurzschreibweisen sind erlaubt:

```
for (TYPE item : list) {
  ...
}
```

und

```
"..." + (VAR==null)?"DEFAULT":VAR
```

Letzteres ist erlaubt, sollte aber vermieden werden.

Nicht erlaubt ist die Kurzschreibweise für Bedingungen, also das Weglassen der Klammern:

```
if (COND)
  STATEMENT
```

8. Quellenverzeichnis

[Ambler1999]

S. W. Ambler, Writing Robust Java Code. The AmbySoft Inc. Coding Standards for Java v17.01d 1998-1999.
http://www.ambysoft.com/downloads/javaCodingStandards.pdf.

[Sun1997]

Sun, Java Code Conventions. Sun Microsystems, Inc.. , 1997.
http://java.sun.com/docs/codeconv/.

[Vermeulen2000]

A. Vermeulen, S. W. Ambler, G. Baumgardner, E. Metz, T. Misfeldt, J. Shur und P. Thomson, The Elements of Java Style, Cambridge University, 2000.

9. Tabellenverzeichnis