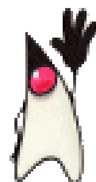
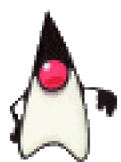
Entwicklungsrichtlinien für Java-Software

Forschungsgruppe Requirements Engineering

Institut für Informatik

Universität Zürich





INHALTSVERZEICHNIS

1 EINLEITUNG UND ORGANISATORISCHES 2

1.1 Grundidee, Aufgabe der Entwicklungsrichtlinien ... 2
1.2 Verbindlichkeit der Richtlinien ... 2
1.3 Abweichung von den Richtlinien ... 3
1.4 Aktualisierung alten Codes ... 3
1.5 Inhaltsübersicht ... 3

2 PROGRAMMIERHINWEISE FÜR JAVA 4

2.1 Konfigurationsmanagement, Software-Verwaltung ... 4
2.2 Codierregelungen ... 4
2.3 Pakete ... 8
2.4 Importe ... 8
2.5 Klassen ... 8
2.6 Interfaces ... 11
2.7 Methoden ... 11

3 EINRÜCKUNGEN UND LAYOUT 15

3.1 Grundidee, Einrückungen und Layout ... 15
3.2 Einrückung bei Blöcken ... 16
3.3 Reihenfolge der Deklaration der Klassenelemente ... 16
3.4 Einrückung der Klassendeklaration, des Methodenkopfs und der Kommentare ... 17
3.5 Methodenrumpf ... 18

4 NAMEN UND BEZEICHNER 22

4.1 Grundidee, Bezeichnerwahl ... 22
4.2 Sprache für Bezeichner und Kommentare ... 22
4.3 Bezeichner für Klassen und Interfaces ... 23
4.4 Bezeichner für Konstanten und Variablen ... 24
4.5 Benennung von Methoden ... 25
4.6 Benennung von Paketen ... 26

5 DOKUMENTIEREN VON CODE 28

5.1 Grundidee, Kommentardichte ... 28
5.2 Allgemeines zur Kommentierung mit JavaDoc ... 29
5.3 Kommentierung der Projektübersicht mit JavaDoc ... 30
5.4 Kommentierung von Paketen ... 31
5.5 Kommentieren von Klassen ... 32

5.6 Kommentieren von Interfaces ... 33
5.7 Kommentieren von Methoden ... 33
5.8 Kommentieren von Attributen mit JavaDoc ... 37
5.9 Alle JavaDoc Tags für die Kommentierung in einer Übersicht ... 38

6 EINSATZ VON TOOLS **42**

6.1 Konfigurationsmanagement mit CVS ... 42
6.2 IContract ... 44

A LITERATUR **I**

B INDEX **II**

Entwicklungsrichtlinien für Java-Software

Forschungsgruppe Requirements Engineering, Institut für Informatik der Universität Zürich

http://www.ifi.unizh.ch/groups/req/publications/java.html

Projektübergreifende Entwicklungsrichtlinien der Forschungsgruppe Requirements Engineering für Software-Entwicklungen in der Programmiersprache Java.

Vorwort

Die Entwicklungsrichtlinien sind ursprünglich entstanden als Aufarbeitung und Anpassung des Style Guides für Smalltalk-Entwicklungen (Schneider 1994) an die Bedürfnisse einer Software-Entwicklung mit Java. Eine weitere, recht zentrale und ergiebige Quelle waren die Codierrichtlinien von Doug Lea (Lea 1996).

Diese Richtlinien sind in erster Linie gedacht und gemacht für gruppeninterne (Forschungsgruppe Requirements Engineering) Software-Projekte und gelten als verbindlich für studentische Arbeiten, die im Rahmen der Forschungsgruppe Requirements Engineering entstehen; also für Semesterarbeiten, Diplomarbeiten aber auch fürdas Software Praktikum (SOPRA).

Für die Version 3.0 wurde vor allem das Kapitel 5 Kommentierung gründlich überarbeitet und ein neues Kapitel über die Integration von Tools angehängt.

Erste Version erstellt am:	1996/06/07
Erste Version erstellt von:	SB
Zuletzt geändert am:	2002/10/29
Zuletzt geändert von:	CS, NM, SM
Version:	3.0
Mitarbeiter:	Martin Arnold (MA), Stefan Berner (SB), Martin Glinz (MG), Stefan Joos (SJ), Johannes Ryser (JR), Silvio Meier (SM), Nancy Merlo-Schett (NM), Christian Seybold (CS)

Copyright © ’96-’02, Forschungsgruppe Requirements Engineering
Dieses Dokument ist urheberrechtlich geschützt.
Alle Copyrights sind das Eigentum der jeweiligen Inhaber.

1 Einleitung und Organisatorisches

1.1 Grundidee, Aufgabe der Entwicklungsrichtlinien

Wenn man ein Dokument liest, dessen Aufbau man bereits kennt, erschliesst sich auch der Inhalt umso leichter. Dieser Effekt ist besonders ausgeprägt bei Code-Dokumenten. Durch schlechte, oder besser:

- Durch ungewohnte Formatierung von Code kann dessen Verständnis erheblich erschwert werden.

Wir massen uns nicht an, "guten" Programmierstil definieren zu können; sehr wohl können wir aber einheitlichen Stil vereinbaren und für den Gebrauch in allen Projekten der Gruppe *Requirements Engineering* vorschreiben. Zumindest kann sich dann jeder im Code jedes anderen leichter orientieren.

1.2 Verbindlichkeit der Richtlinien

Stilanleitungen sind immer eine heikle Sache. Jeder glaubt, der eigene Stil sei besser als alle anderen Versuche. Wer der Ansicht ist, dass beispielsweise seine Formatierung der von uns vorgegebenen weit überlegen ist, soll sich an uns wenden und das mit uns diskutieren. Sind die Änderungsvorschläge hinreichend konkret und konstruktiv, so wird in der Regel im Rahmen einer Inspektion Art und Umfang der Aktualisierung der "alten" Richtlinien erarbeitet.

Wir sind natürlich immer bestrebt, möglichst sinnvolle Vorgaben zu machen. Solange aber nichts abweichendes schriftlich vereinbart ist, gelten die in diesem Dokument aufgeführten Minimalrichtlinien als absolut verbindlich.

Gegen die Richtlinien in einer studentischen Arbeit systematisch zu verstossen, hat Abzüge in der (B-)Note zur Folge, wenn folgendes nicht eingehalten wird:

An einigen Stellen wird "soll" als Modalverb verwendet. Von solchen Richtlinien *kann* abgewichen werden, wenn ein guter Grund vorliegt, der aber dokumentiert werden *muss*.

Wird hingegen das Modalverb "muss" angewendet oder ist aus dem Kontext ersichtlich, dass es sich bei der jeweiligen Richtlinien um eine "Muss"-Richtlinie handelt, gilt es, den Grund des Nichteinhaltens der Richtlinie -unter vorheriger Rücksprache und mit dem Einverständnis des jeweiligen Auftraggebers- genauestens zu dokumentieren (siehe Kapitel 1.3).

Wir sind uns bewusst, dass kein Standard perfekt und in allen Situationen anwendbar ist. Ambler (1997) beschreibt die "legale" Möglichkeit, bewusst gegen eine Richtlinie zu verstossen mit der *prime directive:*

prime directive for standards

"When you go against a standard, document it."

Die Standards schreiben – wie schon erwähnt – einen gewissen Minimalrahmen fest; es sind bei weitem nicht alle Fälle geregelt. Es ist aber die Aufgabe *jedes* Dokumentierenden, seine Erzeugnisse möglichst verständlich und dabei knapp vorzustellen. Wenn dazu Mittel erforderlich sind, die über die beschriebenen Standards hinausgehen, sollen sie eingesetzt werden (Schneider 1994).

1.3 Abweichung von den Richtlinien

Wenn von den Richtlinien abgewichen werden soll, muss dies *vor* Beginn der Arbeit mit dem Auftraggeber vereinbart und schriftlich festgehalten werden.

1.4 Aktualisierung alten Codes

Die vorliegende Version 3.0 dieser Richtlinien stellt die derzeit aktuelle Version dieser Richtlinien dar.

Wird in einem Projekt Code verwendet, der nach älteren Versionen dieser Richtlinien dokumentiert ist, *darf* dieser Code bei einer Änderung der betreffenden Stelle in das hier beschriebene, aktuelle Format überführt werden. Die Änderung *muss* aber dokumentiert werden. Durch dieses Vorgehen wird älterer Code nach und nach in das jeweils aktuelle Format überführt.

1.5 Inhaltsübersicht

In Kapitel 2 finden sich allgemeine Programmierhinweise. Die Richtlinien in diesem Kapitel betreffen primär Software-Verwaltung sowie Programmierstil oder -technik, also in erster Linie inhaltliche Aspekte und weniger formale Aspekte wie Formatierung von Code oder Kommentaren etc.

Die Richtlinien in Kapitel 3 regeln das Layout bzw. die äussere Form des Programmcodes, betreffen also Aspekte wie Einrückung und Formatierung von Programmteilen etc.

Kapitel 4 behandelt die Bezeichnerwahl. Hier finden sich Regeln und einige Hinweise, nach welchem Schema Klassen, Methoden und Variablen zu benennen sind, so dass möglichst intuitiv vom Bezeichner auf die Bedeutung des Bezeichneten geschlossen werden kann.

Das Dokumentieren und Kommentieren von Code, d.h. was muss wie beschrieben und kommentiert werden und welche äussere Form muss der Kommentar haben, ist in Kapitel 5, “Dokumentieren von Code” Seite 28 (JavaDoc) geregelt.

2 Programmierhinweise für Java

2.1 Konfigurationsmanagement, Software-Verwaltung

In jedem Softwareprojekt sind die Basisdienste des Konfigurationsmanagements (→ Identifikation, Änderungslenkung, Buchführung/Dokumentation, Synthese und Planung/Darlegung) sicherzustellen. Projektabhängig obliegt jeweils die Skalierung des Konfigurationmanagements der jeweiligen Projektplanung und ist hier nicht weiter geregelt. Während es für ein Ein- oder Zwei-Personenprojekte meist vom Aufwand her nicht vertretbar und auch wenig sinnvoll ist, ein voll institutionalisiertes Konfigurationsmanagement zu fordern, ist dies für Projekte mit drei oder mehr Personen in der Regel unverzichtbar. In Kapitel 6, “Einsatz von Tools” Seite 42 wird eine Übersicht über einige in diesem Kontext empfehlenswerte Tools gegeben.

2.2 Codierregelungen

Nachfolgend werden einige Codierregelungen aufgeführt. Codierregelungen sind gesammelte und aufbereitete Erfahrungen aus der Anwendung mit Sprachkonstrukten der Programmiersprache. Zur Realisierung eines Problems gibt es viele gangbare Lösungswege, nicht alle davon sind vorteilhaft, keiner davon ist aber grundsätzlich falsch. So wird nachfolgend beispielsweise vor der Verwendung potentiell fehlerträchtiger Anweisungen gewarnt oder auf deren Probleme aufmerksam gemacht. Auch werden Empfehlungen für bestimmte Anwendungen aufgeführt. Alle diese Regelungen sind mit Vorsicht zu geniessen. Es sind keine allumfassenden Lösungen, die in jeder Situation angewendet werden können oder müssen.

main Methode

- In jeder Klasse kann zu Testzwecken eine **main** Methode erzeugt werden. Dies erlaubt, die Funktionalität in Form eines Unittests oder Demos zu prüfen. Insbesondere beim Setzen von Werten bei Klassenvariablen ist Vorsicht geboten, da dies isoliert im Test zwar einwandfrei funktioniert, beispielsweise aber bei einer Integration im Gesamtsystem jedoch Probleme bereiten kann.
- Bei eigenständigen Applikationsprogrammen sollte die Klasse mit der **main** Methode von den anderen "normalen" Klassen getrennt werden. Ein möglicher Bezeichner diese Klasse wäre *Applikation*. Mittels der klaren Abtrennung wird generell die Wiederverwendung der Klassen erhöht.

Design Patterns

- Design Patterns oder "pattern language" ist die erfolgreiche Antwort auf alltägliche Software Probleme in einem spezifischen Kontext. Mit Hilfe von Pattern lassen sich häufig auftretende Entwurfsstrukturen und deren Absicht benennen und katalogisieren, womit das gemeinsame Vokabular gefördert, die Kommunikation erleichtert und die Wiederverwendbarkeit unterstützt wird. Das Verwenden von Patterns wird

honoriert, was aber nicht heisst, dass Patterns fast wahllos eingesetzt werden sollen. Folgendes gilt zu beachten: wo Patterns verwendet werden, muss das im Klassenkopf auch dokumentiert werden (siehe Kapitel 5.9; **@responsibilities**). Die Benennung von Klassen mit dem Patternnamen ist nicht sinnvoll, da beispielsweise die Java Klassenbibliothek als solches eine Klasse namens **Observer** zur Verfügung stellt.

Überladen von Methoden

- Das *Überladen von Methoden* innerhalb derselben Klasse ist gestattet, falls diese semantisch das gleiche tun. Dies ist ein häufig verwendetes Verfahren, das es erlaubt, Gruppen von Methoden, die ähnlichen Zwecken dienen, den gleichen Namen zu geben. Das Überladen von Argumenten hingegen ist nicht legal, wenn diese semantisch nicht das gleiche tun.

BEISPIEL 1. Überladen von Methoden. Zwei Methoden dürfen in derselben Klasse den gleichen Bezeichner zugeteilt bekommen, wenn sie semantisch das gleiche tun.

richtig:

```
open(File f);    // opens a file for reading byte by byte
open(Stream s);  // opens a stream for reading
```

falsch:

```
add(int n);      // adds n
add(String name);// concats strings
```

Identitäts- oder Wertvergleich

- Der Vergleich von Objekten (*Identitätsvergleich*) erfolgt durch den Vergleichsoperator ==. Hier wird überprüft ob zwei Variablen auf das gleiche Objekt verweisen, und nicht, ob beide Objekte den gleichen Wert enthalten. Um zu prüfen, ob zwei Objekte tatsächlich gleich sind, müssen sie eine speziell für diesen Objekttyp geschriebene Methode verwenden (`equals()`). Beim *Wertvergleich* muss demnach diese Methode `equals(Object)` überschrieben werden. Wertvergleiche sind meist bei der Arithmetik anzutreffen: der Wertvergleich aller Attribute zweier Objekte ist selten. Der Vergleichsoperator == muss insbesondere beim Vergleich von Strings *vorsichtig* angewandt werden.

Zuweisungsoperator

- Zuweisungen in Bedingungen sind verboten. = darf innerhalb der Bedingungen von **if** und **while** Schleifen nicht angewandt werden. In der Klammer müssen die Ausdrücke folgendermassen festgehalten werden (Beispiel 2a). Die Ausdrücke in Beispiel 2b sind nicht erlaubt.

BEISPIEL 2. Zuweisungen in **if** und **while** Schleifen...

Beispiel 2a)erwünscht sind folgende Ausdrücke**:**

```
if (i == 5) { ...... }
```

oder

```
while (!cancel ){ ...... }
```

Beispiel 2b) nicht erlaubt sind u.a. folgende Ausdrücke:

```
if (cancel== true) { ....}
```

oder

```
if (i = 5) { ...}
```

- Java hat ausser Arrays und Klassen keine benutzerdefinierbaren Typen und somit auch keine Aufzählungstypen oder Bereichstypen. Dieser Mangel ist sicherlich nicht substantiell, da er durch die Verwendung von Klassen kompensiert werden kann, was jedoch aufwendiger ist. Das Vorhandensein von Aufzählungstypen in einer Programmiersprache erleichtert generell deren Verständlichkeit, da Ausdrücke wie in (a) abgebildet möglich sind. Sieht man von einer Verwendung von Klassen ab, dann würde ein äquivalentes Code-Stück in Java in etwa so aussehen wie in (b) abgebildet.

Substituieren von Aufzählungstypen oder Bereichstypen

BEISPIEL 3. Substituieren von Aufzählungstypen

`TYPE // (a)`
`    TrafficLightType =(red, green, yellow);`
 	
`VAR`
`    trafficLight : TrafficLightType;`
`BEGIN`
`    …`
`    IF (trafficLight == red)`
`    THEN`
`    …`
`    END`
`    …`
`END`
	`    // (b)`
	`final int red = 0;`
	`final int yellow = 1;`
	`final int green = 2;`
	`…`
	`void foo()`
	`    {`
	
	`    int light;`
	`    …`
	`    if (light ==  red)`
	`        {`
	`        …`
	`        }`
	`    …`
	`    }`
		`class TrafficLight     // (c)`
		`    {`
		`    boolean isRed() { .. return ..;}`
		`    boolean isYellow() { .. return ..;}`
		`    boolean isGreen() { .. return..;}`
		`    …`
		`    void fooToo( … )`
		`        {`
		`        …`
		`        if (this.isRed())`
		`            {`
		`            …`
		`            }`
		`        …`
		`        }`
		`    }`

Das Code-Fragment (b) hat im Vergleich zu (a) den Nachteil, dass es weniger gut verständlich ist, insbesondere wenn Variablendeklaration und -verwendung weit auseinander liegen. Ferner kann »trafficLight« nur einen vordefinierten und keinen eigenen, benutzerdefinierten Typ haben. Es können daher der Variablen »trafficLight« beliebige, also auch nicht mehr sinnvoll interpretierbare Werte vom Typ `int` zugewiesen werden, z.B. 17. Ist man nicht bereit, diese Nachteile bezüglich Lesbarkeit und Ausführungssicherheit zu akzeptieren, dann muss eine eigene Klasse TrafficLight geschaffen werden (c).

- *Generell gilt*: Für alle nicht privaten Klassen soll – obwohl aufwendiger – eine Lösung nach Schema (c) vorgezogen werden, da sie zu robusterem Code führt, die Entwurfsentscheidung, wie »trafficLight« realisiert ist, kapselt und über die Schnittstelle nur sinnvoll interpretierbare Objektzustände erlaubt …

lokale Variablen

- Die Deklaration einer lokalen Variablen soll dort stattfinden, wo sie verwendet werden. Dies ist besonders bei Schleifen der Fall.
- Lokale (temporäre) Variablen sollten lieber neu deklariert und initialisiert werden, als Bestehende zu überschreiben/wiederzuverwenden. Fehlerquellen können so minimiert werden.

Deklarieren von Arrays

- Arrays werden mit Type[] arrayName deklariert. Dieser Vermerk steht für alle unverbesserlichen C Programmierer!

Casts in Bedingungen einbetten

Casts sollten in Bedingungen eingebunden werden. Dies verhilft beispielsweise zu einer typsicheren Konvertierung:

BEISPIEL 4. Casts in Bedingungen zur Veranschaulichung.

```
…
C cx= null;
   if (x is instance of C)
      /* Problemlose Konvertierung */
      {
      cx = (C)x;
      }
   else
      /* Reaktion falls Bedingung falsch */
      {
      evasiveAction();
      }
```

Anstelle mit einer Exception (*ClassCastException*) zu reagieren oder darauf zu warten, dass das Laufzeitsystem eine solche wirft, kann mittels *casts* eine spezielle Methode aufgerufen werden, wenn wie im vorherigen Beispiel das Objekt keine Instanz der erwarteten Klasse ist.

Blöcke in Schleifen und Bedingungen

- **break** muss in Schleifen umsichtig verwendet werden
- **continue** darf nicht verwendet werden.
- **switch**-Anweisungen sollen einen **default case** beinhalten.
- **break** muss jeden **case** einer **switch**-Anweisung beenden.

2.3 Pakete

Pakete sind ein geeignetes Mittel zur Gliederung von Code in Einheiten, welche auf bestimmte Art und Weise miteinander in engerem logischen Zusammenhang stehen. Eine Gliederung des Java Codes nach Paketen ist besonders sinnvoll bei grösseren Projekten, deren Ziel es u.a. ist, den erstellten Code wiederverwenden zu können. Dabei sollte zudem ein entsprechendes Konfigurationsmanagement eingesetzt werden, um die Java-Pakete zu verwalten.

Ein Paket kann aus mehreren Quelldateien bestehen. Eine Java-Quellcodedatei wird durch die Verwendung des Konstruktes **package** <Paketname>; am Beginn des Quellcodes einem bestimmten Java-Paket zugeordnet. Die Benennung von Paketen ist genauer unter Kapitel 4.6 beschrieben.

2.3.1 Dokumentation von Paketen

(siehe auch Kapitel 5.4, »Kommentierung von Paketen«)

2.4 Importe

Importangaben dokumentieren diejenigen Ressourcen, die ein Paket zur Erfüllung seiner Aufgaben und Dienste benötigt. Neben ihrer Funktion für den Übersetzer sind Importangaben auch für den Entwickler hilfreich und notwendig, um den Kontext und die Abhängigkeiten eines Pakets und der darin enthaltenen Klassen besser zu verstehen.

- Das wahllose Importieren sämtlicher Klassen eines Pakets (mittels *) sollte vermieden werden.
- Für den Fall, dass ein Import nicht eindeutig ist (beispielsweise wenn es zwei Klassen mit gleichem Bezeichner in unterschiedlichen Paketen gibt), muss die exakte Lokation der zu verwendeten Klasse mittels der Punkt-Notation angegeben werden (z.B. `java.awt.Frame`), auch wenn ein Test des Programms ergibt, dass dies nicht notwendig wäre. Hierbei ist besonders zu beachten, dass die volle Qualifikation einer Klasse dazu führen kann, dass Code an mehreren Stellen geändert werden muss, wenn der Name des Pakets sich ändert.

2.5 Klassen

2.5.1 Klassenkopf

Neben Klassenbezeichnung, Klassenkommentar und ggf. Schnittstellen wird im Klassenkopf auch die Attributierung der Klasse geregelt, d.h. es wird geregelt, ob die Klasse abstrakt ist (Schlüsselwort **abstract**), ob es Unterklassen geben darf (Schlüsselwort **final**)

und wie die Sichtbarkeit des Klassenbezeichners ist (Schlüsselwort **public**). Hierbei sollten die folgenden Regeln beachtet werden:

Abstrakte Klassen

- Eine Klasse sollte nur dann abstrakt sein, wenn sie "teilweise abstrakt" ist, d.h. wenn sie eine bestimmte Funktionalität implementiert, die sie mit ihren (potentiellen) Unterklassen gemeinsam hat. Wenn Unsicherheit oder Unklarheit darüber besteht, in welcher Form die Funktionalität zu implementieren ist oder wenn es darum geht, ein bestimmtes Protokoll oder eine bestimmtes Schnittstellenformat einzuhalten, dann sollten Interfaces und nicht abstrakte Klassen verwendet werden. Interfaces sind wesentlich flexibler als abstrakte Klassen. Sie unterstützen Mehrfachvererbung und können u.a. auch dazu benutzt werden, ansonsten unzusammenhängenden Klassen ähnliche Funktionalität zu geben.

Minimieren der public oder protected Methoden (Sichtbarkeit) reduziert die Koppelung, erhöht die Erlernbarkeit und Kapselung

- Die Grösse der *Schnittstelle* einer Klasse ist u.a. massgebend für ihre Erlernbarkeit und Werbeartikel. Die Minimierung der **public** und **protected** Methoden ist aus nachfolgenden Gründen anzustreben:
  - *Erlernbarkeit*: Versteht man die öffentliche Schnittstelle einer Klasse, weiss man, wie sie anzuwenden ist. Je kleiner resp. kürzer die öffentliche Schnittstelle desto schneller der Lernprozess.
  - *Vermindertes Coupling*: Wenn immer eine Instanz einer Klasse eine Mitteilung einer Instanz einer anderen Klasse oder der Klasse selbst übermittelt, werden diese zwei Klassen gekoppelt. Wird die öffentliche Schnittstelle vermindert, wird auch die Möglichkeit für die Koppelung reduziert.
  - *Grössere Flexibilität durch Kapselung*: Steht im Zusammenhang mit Koppelung. Muss eine Methode in der öffentlichen Schnittstelle geändert werden, müssen alle Codezeilen, die einen potentiellen Zugriff auf die Methode haben, verändert werden. Je kleiner die öffentliche Schnittstelle desto grösser die Kapselung und somit desto grösser der Freiraum oder die Flexibilität zur Änderung ohne grossen Aufwand oder schwerwiegenden Folgen.

2.5.2 Klassenrumpf

Klassenvariablen

Klassenvariablen sollten sparsam und umsichtig verwendet werden, da sie dazu führen können, dass Klassen zunehmend kontextabhängig werden und/oder Seiteneffekte verbergen. Klassenvariablen können ausserdem die Änderbarkeit von Klassen erschweren, da ihr Typ in Unterklassen nicht mehr geändert werden kann.

Zugriff auf Instanz- und Klassenvariablen

Der Zugriff auf Variablen erfolgt grundsätzlich nur über die Methoden, die diesen Namen (siehe auch Kapitel 4.5.1) tragen, beispielsweise für eine Klasse `SomeThing` mit einer Instanzvariablen `name` also `SomeThing.name(aName)` zum Setzen und `SomeThing.name()` zum Lesen. In Übereinstimmung mit den Sun-Konventionen dürfen die Präfixe **set** und **get** für die Benennung von Zugriffsmethoden ebenfalls verwendet werden (siehe Kapitel 4.5).

Auch Instanzen dürfen ihre eigenen Instanzvariablen beim Schreiben nur mit Hilfe einer Methode verändern, beim Lesen hingegen darf direkt zugegriffen werden. Dies stellt u.a. sicher, dass sich sowohl Synchronisations- wie auch Benachrichtungsmechanismen einfach ändern lassen, wenn dies notwendig werden sollte.

- Sämtliche Instanz- und Klassenvariablen sind grundsätzlich **private** (oder zumindest **protected**) nie aber mit **public** zu deklarieren. Wenn Variablen **public** deklariert sind, dann sind diese öffentlich zugreifbar und damit ist die interne Struktur der Klasse nicht mehr frei wählbar oder änderbar. Auch Methoden können somit nicht davon ausgehen, dass diese mit **public** deklarierten Variablen gültige Werte besitzen (siehe Kapitel 2.5.4).

statische Initialisierer

- Werden Klassenvariablen als non-private **static** deklariert, muss sichergestellt werden, dass sie *von Anfang an einen sinnvoller Wert* zugewiesen bekommen auch wenn nie eine Instanz der Klasse erstellt wird. Es kann nicht vorausgesetzt werden, dass auf non-private **static** Klassenvariablen nur nach der Instantiierung zugegriffen wird. Die Sicherstellung eines adäquaten Wertes kann entweder bei der Deklaration oder durch einen *static initializer-Block* erfolgen.

2.5.3 Kommentierung von Klassen

(siehe auch Kapitel 5.5, “Kommentieren von Klassen” Seite 32)

2.5.4 Allgemeine Hinweise

Nachfolgend finden sich einige Faustregeln (vgl. Lorenz 1993), die bei Entwurf und Prüfung von Klassen hilfreich sein können. Diese sind nicht als absolute Richtlinie anzusehen, welche unter allen Umständen einzuhalten ist, sondern vielmehr als ein Hilfsmittel, um potentielle Problembereiche zu identifizieren.

- Ein Methode sollte nicht mehr als 30 Zeilen Code umfassen.
- Methoden, welche eine Länge von 40 Zeilen Code überschreiten, sollten neu entworfen werden.
- Eine Klasse sollte nicht mehr als 15 Methoden umfassen (hierbei nicht gerechnet, die Methoden zum direkten Erzeugen einer Instanz).
- Höhere Durchschnittswerte können darauf hindeuten, dass zuviel Funktionalität in zu wenig Klassen steckt.
- Ein Klasse sollte nicht mehr als 6 Instanzvariablen pro Klasse umfassen.
- Mehr als 10 Variablen können darauf hindeuten, dass eine Klasse mehr tut als sie sollte.

2.6 Interfaces

Pro Anwendungsbereich sollen alle Konstanten in einem Interface gesammelt werden. Dieser Ansatz verhilft Konsistenz zu bewahren.

BEISPIEL 5. Konstantendeklaration in einem Interface, welches dann von diversen Klassen implementiert wird

```
public interface Constants
   {
   final int RED = 0;
   final int YELLOW = 1;
   final int GREEN = 2;
   }

public class TrafficLight implements Constants
   {
   ....
   }

public class TrafficController implements Constants
   {
   ....
   }
```

Attributierung von **public** bei Methoden in einem Interface kann weggelassen werden, da dies überflüssig ist.

2.7 Methoden

2.7.1 Methodenkopf

Der Methodenkopf umfasst Methodendeklaration inklusive der Attributierung und den Kopfkommentar. Er definiert und dokumentiert die Schnittstelle der Methode und deren Sichtbarkeit nach aussen.

Diejenigen Methoden in Basisklassen, die im Fall der Spezialisierung der Klasse von Unterklassen zu implementieren sind, sollten als abstrakte Methode realisiert werden (daher auch **abstract** attributiert sein) und nicht mittels einer Methode realisiert werden, die einfach "nichts tut". Bei **abstract** attributierten Methoden stellt der Übersetzer sicher, dass die Methode in einer Unterklasse auch implementiert wird. Bei einer Methode, die "nichts tut", ist dies nicht der Fall.

2.7.2 Methodenrumpf

- *Als Leitsatz gilt, kurze und hoch kohäsive Methoden zu schreiben.*

Kombiniertes Setzen von Instanz- oder Klassenvariablen

Es kann vorkommen, dass bestimmte Klassen- oder Instanzvariablen grundsätzlich nur zusammen mit anderen Klassen- oder Instanzvariablen gesetzt (oder abgefragt) werden müssen. Wenn die betreffenden Variablen *wirklich* nur auf diese Art und Weise benutzt

werden, ist es ausdrücklich erlaubt, hierfür kombinierte Zugriffsmethoden zur Verfügung zu stellen. Die elementaren Zugriffsmethoden sollten, soweit vorhanden und möglich, entweder **protected** oder **private** attributiert werden.

Meist ist es verständlicher, kombinierte Zugriffsmethoden nicht nach den betroffenen Instanzvariablen zu benennen, sondern nach dem eigentlichen Zweck bzw. der Aufgabe, die die Methode hat; z.B. `dateOfBirth(...)` statt `dayMonthYear(...)` (siehe Kapitel 4.5.1).

2.7.3 Unterscheidung von Methoden, Methodenarten

Jede Methode erfüllt einen Zweck bzw. stellt einen Dienst zur Verfügung. Aus dem Namen der Methode sollte ihr (Haupt-)Dienst, aber nicht ihre Implementierung ersichtlich sein. Grundsätzlich können Methoden nach der Art Dienstes bzw. der Aufgabe, welche sie zu erfüllen haben, kategorisiert werden. Werden Methoden gleicher Art konsequent und einheitlich nach dem selben Schema benannt, so verbessert dies die Lesbarkeit und Verständlichkeit von Code beträchtlich. Unterschieden wird hier grundsätzlich zwischen folgenden Methodenarten:

(a) Methoden, deren Hauptaufgabe darin besteht, Objektzustände oder Objekte bereitzustellen, aber die Aktionen, die dafür notwendig sind eigentlich nicht interessieren (dürfen) und daher besser verborgen bleiben (siehe auch Beispiel 14 auf Seite 25). Eine Methode dieser Art wird hier *Zustands-* oder *Zugriffsmethode* genannt.

Zustandsmethoden und Zugriffsmethoden

(b) Methoden, die Auskunft über die Existenz einer Beziehung oder das "Erfüllt-Sein" einer Bedingung zwischen Werten oder Objekten geben. Eine Methode dieser Art wird hier *Vergleichs-* oder *Prädikatmethode* (kurz *Prädikat*) genannt.

Vergleichsmethoden und Prädikatemethoden

(c) Methoden, bei welchen in erster Linie nicht der Wert oder das Objekt von Interesse ist, mit welchem die Methode antwortet, sondern vielmehr die Aktion, die die Methode ausführen soll oder der Effekt, welchen die Methode hat, primär von Interesse ist. Eine Methoden dieser Art wird hier Aktionsmethode genannt.

Aktionsmethoden

Es gibt sicherlich weitere Methodenarten, welche unterscheidenswert sein könnten. Wir haben uns aber aus Gründen der einfachen Handhandbarkeit und Anwendbarkeit der Richtlinien auf drei, zugegeben etwas gröbere Kategorien beschränkt.

Wird eine Methode entworfen oder erstellt, so sollte sich der Entwerfer zuerst überlegen, was für eine Methodenart es sich handelt bzw. was für eine Methodenart benötigt wird. Konkrete Richtlinien für die Benennung der Methoden einer Methodenart finden sich in Kapitel 4.5, »Benennung von Methoden«.

2.7.4 Kommentierung von Methoden

(siehe Kapitel 5.7, »Kommentieren von Methoden«)

2.7.5 Exceptions

Anwendung von Exceptions

Mittels Exceptions und Exception Handlern soll(t)en unerwartete – nicht aber unvorhersehbare – Bedingungen/Fehler/Situationen/Zustände, so abgefangen/behandelt werden, dass ein Programm möglichst nicht abbricht, sondern ab einem definierten *Wiedereinstiegspunkt* weiterlaufen kann.

- Code für Fehlerbehandlung wird separiert vom Code für regulären Programmablauf; der Code wird besser verständlich.

Exceptions sind nicht dazu da, Bedingungen/Situationen/Zustände zu behandeln, die bei jedem regulären Programmablauf zu erwarten sind.

- Code für Fehlerbehandlung würde nun nicht separiert vom Code für regulären Programmablauf.
- in diesem Fall bieten Exceptions keine Vorteile gegenüber verschachtelten **if**-Anweisungen.
  - Code wird zunehmend schwierig(er) zu verstehen, da der jeweilige Kontext schlechter zu erfassen ist.
  - effiziente Optimierung des Übersetzers ist (fast) nicht mehr möglich, da nun zwei "reguläre" Kontroll- und Datenflüsse bestehen würden.

Nachfolgend wird ein Beispiel aufgeführt, das verdeutlichen soll, dass Exceptions -wie im vorangegangenen Teil schon erwähnt- nicht vorhersehbare Fehler abfangen sollen. Vorhersehbar ist im unten aufgeführten Code-Fragment, dass die Konstante `MAX_VALUE` der Klasse `Integer` grösser sein kann, als die Anzahl Elemente des Arrays `anArray`. Anstelle diesen Fehler mit einer Exception abzufangen, sollte eine Überprüfung der Anzahl Array Elemente und der Konstanten `Integer.MAX_VALUE` vorweggehen. Die Umstände, die zum Fehler beitragen, sind hier klar ersichtlich und einfach verhinderbar. Das Beispiel ist insofern schlecht, da keine Ausnahmesituation vorliegt, um eine Exception aufrufen zu müssen.

BEISPIEL 6. Schlechtes Beispiel für Exceptionsaufruf

```
...
int[] anArray = { 3, 1, 2, 5, 4 };
int sum = 0;
try
   {
      for (int i = 0; i < Integer.MAX_VALUE; i = i + 1)
      {
      sum = sum + anArray[i];
      }
   }
catch( ArrayIndexOutOfBoundsException e )
   {}
```

Exceptions und Errors

Eine Exception in Java zeigt eine Ausnahmesituation an, die im regulären Programmbetrieb nicht vorgesehen ist und besonderer Behandlung bedarf, damit das Programm weiterlaufen kann.

Ein Error in Java zeigt einen schwerwiegenden Fehler oder ein Systemversagen an, welches in den allermeisten Fällen dazu führt, dass eine Programmausführung abgebrochen werden muss.

- Errors sollten – auch wenn dies möglich ist – *nicht* abgefangen werden, da sich das System in einem undefinierten Zustand befinden kann.

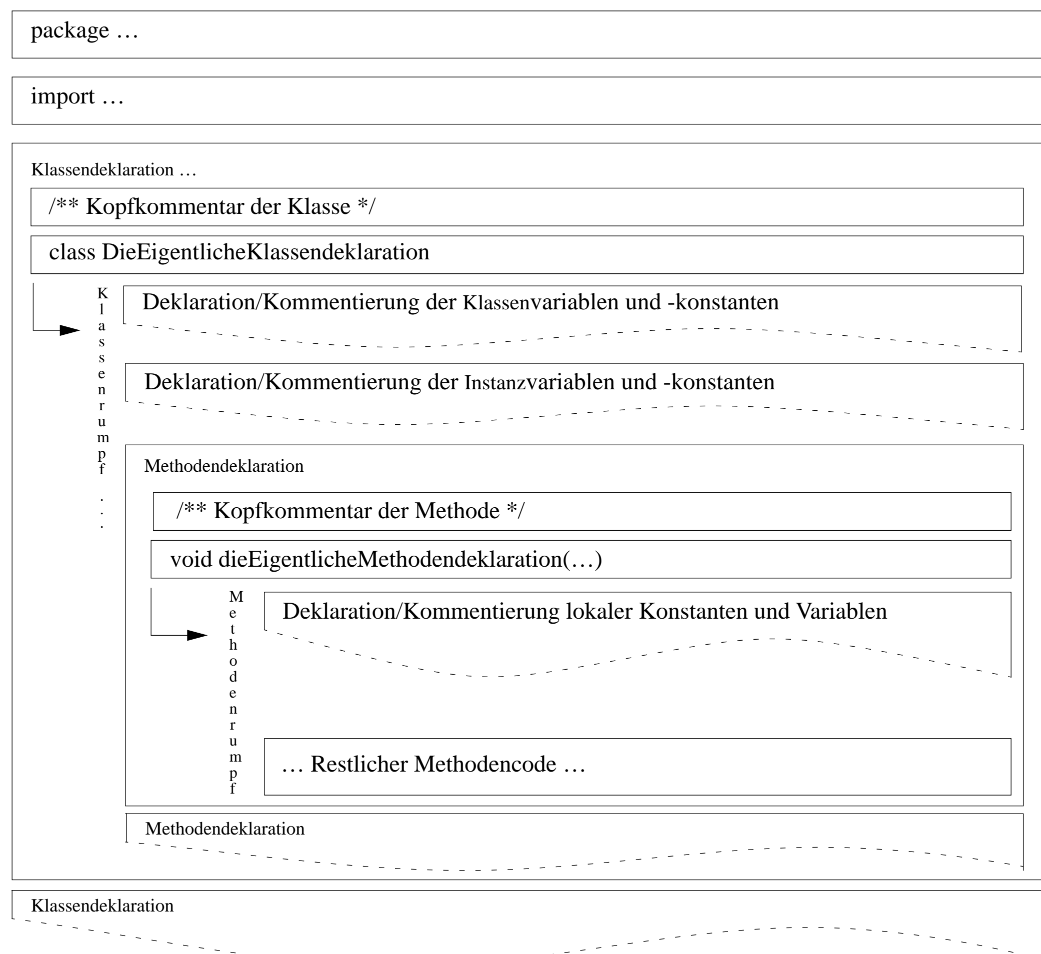
3 Einrückungen und Layout

3.1 Grundidee, Einrückungen und Layout

Strukturell untergeordnete Konstrukte *müssen* gegenüber dem jeweils nächst übergeordneten Konstrukt um mindestens eine Tabulatorstufe eingerückt werden. Ausschliesslich für den unwahrscheinlichen Fall, dass der verwendete Texteditor oder Browser nicht über eine Tabulatorfunktion verfügt, entspricht ein Tabulator vier Leerzeichen. Einrückungen von mehr als sieben Tabulatorstufen sollen vermieden werden; ggf. sollten Teile der Methode delegiert werden. Wenn notwendig kann man nur für diesen Zweck private Methoden schreiben.

Leerzeilen helfen, den Text zu gliedern und sollten nicht zu sparsam eingesetzt werden. Die Zeilenbreite soll nicht 80 Zeichen überschreiben. In den folgenden Kapiteln werden einige Fälle genauer behandelt, in denen diese grundsätzliche Festlegung nicht eindeutig ist.

BEISPIEL 7. Allgemeines Schema für Gliederung und Layout von Java-Code



3.2 Einrückung bei Blöcken

Das Einrücken bei Schleifen wird u.a. von der verwendeten Programmierumgebung geprägt, kann aber auch eine Art 'Handschrift' des Verfassers sein. Nachfolgend werden vier verschiedene Einrückungsmöglichkeiten aufgeführt, wobei -falls möglich und auch hier in diesen Richtlinien als Leitsatz angewandt- soll die Variante drei (Variante III) verwendet werden. Wichtig ist, dass, wenn eine Formatierungsart gewählt, diese konsistent eingehalten wird.

BEISPIEL 8. Einrückungen bei Schleifen

Variante I): Öffnende Klammer auf der **if**-Zeile; schliessende Klammer auf **if** Höhe

```
if (cancel) {
   ..... /* Schleifenrumpf um eine Tabulatorbreite vom if eingerückt */
}
```

Variante II): Öffnende Klammer auf der nächsten Zeile, nicht eingerückt; schliessende
Klammer auf gleicher Höhe wie öffnende Klammer

```
if (cancel)
{
   ..... /* Schleifenrumpf um eine Tabulatorbreite vom if eingerückt */
}
```

Variante III): Öffnende Klammer auf der nächsten Zeile, eingerückt; schliessende Klammer ebenfalls eingerückt auf gleicher Höhe wie öffnende Klammer

```
if (cancel)
   {
   ..... /* Schleifenrumpf um eine Tabulatorbreite vom if eingerückt */
   }
```

Variante IV): Öffnende Klammer auf der nächsten Zeile, eingerückt; schliessende Klammer ebenfalls eingerückt auf gleicher Höhe wie öffnende Klammer

```
if (cancel)
   {
         ... /* Schleifenrumpf um zwei Tabulatorbreiten vom if eingerückt */
   }
```

3.3 Reihenfolge der Deklaration der Klassenelemente

Verläuft die Deklaration der einzelnen Klassenelemente in einer schematisierten Reihenfolge und nicht willkürlich, kann die Lesbarkeit der Klasse erhöht, ein einheitlich konsistenter Stil approximiert und das schnelle Verstehen der Essenz gefördert werden. Nachfolgend wird die Reihenfolge der Auflistung der Bestandteile einer Klasse aufgeführt:

TABELLE 1. Gliederung der Reihenfolge der Deklaration der Klassenelemente

Reihenfolge	Element
1	*Klassenvariablen*
	Instanzvariablen
2	*Konstruktor*
	finalizer Methode
3	*Klassenmethoden*
	Instanzmethoden

Wird die Klasse so gross, dass oben aufgeführte Gliederung zu unübersichtlich wird, kann zusätzlich noch in der Attributierung unterschieden werden:

- **public**
- **protected**
- **private**

3.4 Einrückung der Klassendeklaration, des Methodenkopfs und der Kommentare

- Die Paket- oder Importdeklaration wird nicht eingerückt.
- Die Klassen- oder Interfacedeklaration wird nicht eingerückt.
- Der Methodenkopf wird gegenüber der Klassen- oder Interfacedeklaration um eine Tabulator-Stufe eingerückt. Der Kopfkommentar wird auf der selben Höhe wie der zugehörige Methodenkopf plaziert, unmittelbar vor dem Methodenkopf (keine Leerzeile).
- Alle folgenden Teile (also der Methodenrumpf) sind relativ zum zugehörigen Methodenkopf um mindestens eine Tabulator-Stufe eingerückt.
- Kommentare (siehe auch Kapitel 5) werden jeweils soweit eingerückt wie der Codeteil, auf den sie sich beziehen.
- Der Kopfkommentar einer jeden Methode muss Informationen über den aktuellen Änderungsstand der Methode enthalten (siehe Kapitel 5.7, »Kommentieren von Methoden«).

3.5 Methodenrumpf

3.5.1 Zu lange Nachrichten

Wenn Ausdrücke sehr lang werden, beispielsweise durch lange Bezeichner, viele Parameter oder Kaskaden, sollen sie an logisch sinnvollen Stellen aufgeteilt und so eingerückt werden, dass die inhaltliche Struktur des Ausdrucks auch optisch sichtbar bleibt, z.B. geklammerte Ausdrücke in je eine Zeile (siehe Beispiel 9).

- Ist eine zusammengesetzte Nachricht (siehe Beispiel 9 auf Seite 18 `myView.doThis(...)   ...`) zu lang, so soll der Adressat und ggf. der erste Teil (`myView.doThis(...)`) in die erste Zeile, dann – um eine Stufe eingerückt – jeweils ein Teil (`.andThat(...), .andSomethingOther(...)`) pro Zeile darunter auftauchen.
- Oft ist es zweckmässig, vor und nach solch einem Konstrukt eine Leerzeile zu lassen.

```
…
myView.doThis( withThat … )
   .andThat( withThose … )
   .andSomethingOther( withThoseFollowing … );
…
```

BEISPIEL 9. Gliederung von zusammengesetzten Nachrichten.

3.5.2 Bedingte Anweisungen und Fallunterscheidungen

Das einleitende Schlüsselwort **if** der **if**-Anweisung und die zugehörige Bedingung wird an die Position geschrieben, an der auch ein gewöhnliches Statement beginnen würde. Darüber steht evtl. ein Kommentar, der sich auf die Fallunterscheidung als Ganzes bezieht (siehe Beispiel 10 und Beispiel 11):

```
…
/* Kommentar */
if (<Bedingung>)
   {  /* ggf. Kommentar */
   Anweisung_1;
   Anweisung_2;
   …
   Anweisung_n;
   }
else if (<Bedingung>)
   {  /* ggf. Kommentar */
   Anweisung_1;
   …
   Anweisung_n;
   }
```

BEISPIEL 10. Allgemeine Form für die Gliederung von Fallunterscheidungen

```
…
else
   {  /* ggf. Kommentar */
   Anweisung_1;
   …
   Anweisung_n;
   }
…
```

Der zur Anweisung gehörende Block (oder zumindest dessen erste Anweisung) beginnt im allgemeinen auf einer neuen Zeile und ist noch einmal um mindestens einen Tabulator eingerückt.

BEISPIEL 11. Kommentierung und Kopfzeile der Fallunterscheidung

```
…
/* if x lies out of screen, set x to smallest possible value*/
if (preferredWindowSize.x() < screenSize.x())
   {
   preferredWindowSize.move(screenSize.x(),
                  referredWindowSize.y());
   …
```

Die Alternative (**else**) steht in jedem Fall in einer eigenen Zeile und ist genau soweit eingerückt wie das einleitende **if** der zugehörigen **if**-Anweisung.

BEISPIEL 12. Einrücken von Blöcken bei einer Fallunterscheidung.

```
…
// Eine erste Fallunterscheidung
if (aCondition1)
   {
   /* Meist sind Blöcke um einen Tabulator eingerückt … */
   Anweisung_1;
   …
   Anweisung_n;
   }
else{ /* Nur extrem kurze Blöcke passen mit in die Zeile */ }
…

// Eine zweite Fallunterscheidung
if (aCondition2){
      /* … es dürfen aber auch mehr Einrückungen sein */
      Anweisung_1;
      …
      Anweisung_n;
      }
else{ /* Nur extrem kurze Blöcke passen mit in die Zeile */ }
```

3.5.3 Blöcke in Schleifen, Bedingungen etc.

Bei Blöcken als Parameter (wie beispielsweise bei einer Schleife, einer Bedingung oder bei der Behandlung von Exceptions) sind folgende Fälle zu unterscheiden:

- Der gesamte Block passt *bequem* in eine Zeile. Dann kann er auch in eine Zeile geschrieben werden. Dies ist schon dann nicht mehr der Fall, wenn mindestens zwei Kommandos in einem Block stehen.
- Auch Blöcke, die nur aus einer Anweisung bestehen, *müssen* geklammert werden, auch wenn dies rein syntaktisch gesehen nicht notwendig wäre. Dadurch wird die *Dangling Else-Mehrdeutigkeit* der **if**-Anweisung umgangen und explizit gemacht, zu welcher **if**-Anweisung ein **else** gehört. Ferner wird der häufig gemachte Flüchtigkeitsfehler vermieden, die Klammerung zu vergessen, wenn nachträglich der Bedingungsrumpf erweitert wird.
- Passt der Block nicht mehr in eine Zeile, umfasst er mindestens zwei Kommandos oder eine Variablendeklaration, so wird er eingerückt gegenüber dem Konstrukt, das ihn verwendet.
- In der ersten Zeile steht die öffnende Klammer und evtl. ein Blockkommentar. Dann folgt in einer neuen Zeile (Ausnahme siehe Beispiel 12) ein Kommando pro Zeile (es sei denn, ein Kommando ist zu lang und muss geteilt werden, siehe auch Kapitel 3.5.1).

BEISPIEL 13. Korrektes Einrücken von Blöcken

```
…
/* Die Blöcke passen in eine Zeile da sie nur aus einer Anweisung bestehen,
müssen aber immer geklammert werden */
if (rectangle.x() > rectangle.y())
   { rectangle.invert(); }
else
   { rectangle.copy(); }
…

/* Der True-Block ist zu lang für eine Zeile, der False-Block nicht. */
if (rectangle.x() > rectangle.y())
   {  /* True-Zweig der Bedingung */
   rectangle.invert();
   rectangle.placeUpRight();
   }
else  {return  false;}
…

/* Mehr als ein Kommando im Block */
for  (int i = 0; i < 23; i++)
   {
   System.out.println( … );
```

```
   System.out.println( … );
   …
   }
…

/* Weitere Blöcke im Schleifenrumpf */
while (!this.available)
   {
   try
      {
      wait();
      …
      }
   catch (InterruptedException anException e)
      {
      …
      }
   }
…
```

4 Namen und Bezeichner

4.1 Grundidee, Bezeichnerwahl

Die Verständlichkeit der Bezeichnung bzw. der Bezeichner von Klassen, Konstanten, Methoden und Variablen erleichtert wesentlich das Verständnis von Code. Daher sollte die Wahl von Bezeichnern stets mit viel Sorgfalt und Überlegung erfolgen. Die Aussagekraft eines jeden Bezeichners ist mit einem impliziten Kommentar zu vergleichen und hat sinngemäss den gleichen Stellenwert wie dieser (siehe auch Kapitel 5, »Dokumentieren von Code«).

- Alle Bezeichner sind so zu wählen, dass sie möglichst selbsterklärend sind, unabhängig davon, ob “nur“ eine temporäre Variable oder eine zentrale Klasse bezeichnet wird. Der Verwendungszweck und die Rolle die "dem Bezeichneten" im Anwendungsbereich zukommt, sollten unmittelbar aus dem Bezeichner ersichtlich sein. Die Auswahl eines Bezeichner erfordert vor allem Sorgfalt und Zeit.
- Ist ein Bezeichner aus mehreren Wörtern zusammengesetzt, wie beispielsweise `handleEvent(......)`, so wird dieser durch die Verwendung von Grossbuchstaben gegliedert. Mit Ausnahme von Bezeichnern für Konstanten (siehe Beispiel 4.4.1) ist die Verwendung von untergesetzten Strichen (engl. Underscores) zu vermeiden, also nicht `handle_event(...)` etc.
- Gleichartige Bezeichner innerhalb des selben Gültigkeitsbereichs dürfen sich nicht nur durch Gross-Kleinschreibung unterscheiden. So darf beispielsweise eine Klasse nicht zwei verschiedene Instanzvariablen haben, wenn die eine Variable x**p**os und die andere Variable x**P**os benannt ist.

4.2 Sprache für Bezeichner und Kommentare

Alle Kommentare, alle Bezeichner sowie lokale und temporäre Variablen sind in Englisch abzufassen. Für jedes Projekt kann ausdrücklich eine andere Sprache vereinbart werden, die dann allerdings projekteinheitlich zu verwenden ist (hierfür unbedingt Kapitel 1.2, »Verbindlichkeit der Richtlinien« berücksichtigen). Bei der Wahl der Bezeichner in Englisch sollte in Zweifelsfällen ein Lexikon zu Rate gezogen werden:

- Es sollen ausschliesslich *eingeführte* Fachwörter verwendet werden, wo solche existieren.
- Nur dort, wo eine international übliche Begriffsbildung nicht zu erkennen ist, dürfen eigene Wortschöpfungen oder "Hau-Ruck-Übersetzungen" eingesetzt werden.

4.3 Bezeichner für Klassen und Interfaces

4.3.1 Bezeichner für Klassen, Klassenvariablen

Bezeichner für Klassen und Klassenvariablen

- Bezeichner für Klassen und Klassenvariablen beginnen grundsätzlich mit einem Grossbuchstaben.
- Der Bezeichner einer Klasse sollte möglichst wenig über deren Realisierung bzw. Implementierung verraten, sondern vielmehr etwas über die Bedeutung ihrer Objekte. So sollte beispielsweise eine Klasse, welche für die Verwaltung der Symboltabelle für einen Übersetzer zuständig ist, nicht `TokenArray`, `ArrayOfSymbols` oder `SymbolTableCollection` genannt werden, sondern `SymbolTable`.
- Bezeichner von Klassenvariablen müssen umsichtig gewählt werden. Das Vergeben desselben Namens einer Klassenvariablen wie in der Oberklasse sollte grundsätzlich vermieden werden (*hiding names*), da dies zu Fehlern beiträgt. Wird dies doch getan, muss das Vorhaben verdeutlicht werden.

Bezeichner für Exceptions

- Bezeichner von Klassen, die eine spezielle Bedeutung haben, können durch die Verwendung von bestimmten Endungen bezeichnet werden, sofern dies dazu beträgt, dass die angestrebte Bedeutung besser zum Ausdruck kommt. Die folgende Liste soll einen Überblick darüber geben, welche Endungen für welche Arten von Klassen zu verwenden sind:

TABELLE 2. Bezeichnerkonvention von Klassen mit spezieller Bedeutung

Klassenendung	Art der Klasse	Beispiele
Adapter	Klassen, welche ein EventListener-Interface mit leeren Methoden implementieren, bezeichnet man mit der Endung Adapter.[a]	*MouseAdapter* *WindowAdapter* *MyEventAdapter*
Event	Klassen, welche von java.awt.Event abstammen und Ereignisse darstellen, müssen die Endung Event tragen.[a]	*UserChooseEvent* *UserCancelEvent*
Exception	Bezeichnet Klassen, die von der Klasse java.lang.Exception abstammen.[a]	*NoSuchElementException* *AbortException*
Listener	Bezeichnet spezielle Interfaces, welche eine (abstrakte) Schnittstelle für die Ereignisbehandlung bereitstellen. Das Verwenden dieser Endung hilft Ereignisbehandlungsschnittstellen von normalen Interfaces zu unterscheiden. (Ereignis-)Adapter implementieren die Schnittstellen von Listeners leer, um in bestimmten Fällen den Implementierungsaufwand zu vermindern.[a]	*MouseListener* *ActionListener* *MyEventListener*

a. Diese Namenskonvention entspricht derjenigen von Sun.

4.3.2 Bezeichner für Interfaces

- Wie bei Klassen beginnen die Bezeichner von Interfaces mit einem Grossbuchstaben. Zudem werden meist deskriptive Adjektive zur Bezeichnung angewendet wie z.B. `Runnable`, `Cloneable` sowie deskriptive Nomen wie `Singleton` oder `DataInput`.

4.4 Bezeichner für Konstanten und Variablen

4.4.1 Bezeichner für Konstanten

- Bezeichner für Konstanten dürfen keine Kleinbuchstaben enthalten, d.h. `PI` und `P1` sind korrekte Bezeichner für eine Konstante, hingegen ist `Pi` als Bezeichner für eine Konstante nicht korrekt.
- Konstante Grössen sind zu Beginn einer Klasse oder Methode als *symbolische Konstanten* mit einem selbsterklärenden Namen zu vereinbaren. Konstanten werden grundsätzlich **final** deklariert.

4.4.2 Bezeichner für Instanzvariablen und lokale Variablen

- Bezeichner für Instanzvariablen und lokale Variablen beginnen mit einem Kleinbuchstaben.
- Namen, die aus drei oder weniger Zeichen bestehen, sollen nur für lokale Aufgaben (Schleifenindices etc.) verwendet werden. Ihre Bedeutung ist bei der Deklaration zu dokumentieren (siehe Kapitel 4.4.3).

4.4.3 Bezeichner für lokale Variablen

Nachfolgend werden einige Abkürzungen, die sich als Standard für lokale Variablen etabliert haben, aufgeführt.

- *Exceptions*: Für die Bezeichnung einer temporären Exception gilt der Kleinbuchstabe **e** als akzeptiert.
- *Schleifenzähler*: Als Schleifenzähler können folgende Kleinbuchstaben gelten:
  i, j, k
- *Streams*: Streams werden in Bezug auf ihre Tätigkeit benannt. Dabei gelten die Abkürzungen: **in**, **out** und **inOut** als anerkannter Standard.

Die Anwendung der *ungarischen Notation* für die Vergabe von Variablennamen widerspricht dem Polymorphiekonzept und ist daher nicht erlaubt.

4.5 Benennung von Methoden

4.5.1 Bezeichnerwahl für eine Methode einer Methodenart

Die nachfolgend aufgeführten Regeln sollten bei der Wahl der Methodennamen Berücksichtigung finden (im Übrigen gelten sinngemäss die Ausführungen von Beginn des Kapitels 4 auf Seite 22).

- Zustands- oder Zugriffsmethoden (siehe Methoden der Kategorie (a) in Kapitel 2.7.3, »Unterscheidung von Methoden, Methodenarten«) werden nur mit einem Substantiv benannt. Bei Bedarf kann ein Adjektiv oder eine entsprechende Präposition vorangestellt werden. Das Substantiv bezeichnet hierbei die Rolle oder den Typ des Objekts oder Werts, mit welchem die Methode antwortet. Beispiele: `length(...)`, `size(...)`, `area(...)`, `union(...)`, `currentState(...)`, `previousState(...)`, `asSquareMeters(...)` etc.

BEISPIEL 14. Insbesondere bei langen oder kaskadierenden Nachrichten sind deklarative Methodennamen meist einfacher zu lesen als imperative.

```
…
/* leicht(er) lesbar */
Circle.area().asSquareMeters(); // so
Circle.area().toSquareMeters(); // oder so
…

/* schwierig(er) zu lesen und sollte vermieden werden */
Circle.calculateArea().convertToSquareMeters();
…
```

- Sämtliche Methoden zum Setzen und Lesen von Variablen fallen unter die Kategorie Zustands- und Zugriffsmethoden (siehe Methoden der Kategorie (a) in Kapitel 2.7.3) und *müssen* wie die betroffenen Variablen bezeichnet werden, also zum Beispiel `name()` oder `name(...)` heissen, wenn damit eine Instanzvariable `name` gelesen oder geschrieben werden soll. Die Vor- oder Nachsilben **get** und **set** sind, um mit ***Sun*** konsistent zu sein, zusätzlich erlaubt (also `setNameTo(...)` oder `getName()`). Nach aussen nicht zugängliche Lese- oder Schreibmethoden werden **private** deklariert.
- Vergleichs- oder Prädikatmethoden (siehe Methoden der Kategorie (b) in Kapitel 2.7.3) werden unter Benutzung eines Verbs als Fragment eines Fragesatzes benannt, z.B. `intersects()`, `isInState()`, `isEquivalenceRelation()` etc.
- Aktionsmethoden (siehe Methoden der Kategorie (c) in Kapitel 2.7.3) werden unter Benutzung eines Verbs im Imperativ als Fragment eines Befehlssatzes benannt, z.B. `update(...)`, `redraw(...)`, `handleEvent(...)`, `drawLine(...)`, `addTax(...)` etc.

4.5.2 Bezeichner für formale Parameter von Methoden

Eine Methode mit Parametern, wie beispielsweise **void** `aMethod( aType aParameter, ... )`, muss in der Kopfzeile formale Bezeichner für ihre formalen Parameter angeben. Als Bezeichner ist üblicherweise zu wählen:

- Die Bezeichnung der Rolle, die der formale Parameter für die Methode jeweils spielt (siehe Beispiel 15a).
- Alternativ kann auch der Klassenname, deren Instanz als aktueller Parameter übergeben wird, verwendet werden. Können Instanzen einer Klasse oder ihrer Unterklassen übergeben werden, so ist die oberste bzw. die allgemeinste Klasse zu nennen, von der die Parameter sein können. Dieser Klassenname wird vom unbestimmten, englischen Artikel "a" oder "an" eingeleitet: `aNumber` (siehe Beispiel 15b).

BEISPIEL 15. Bezeichnung von formalen Parametern

```
/** (a) parameters are named as the role the are playing in this function*/
void reshape (int xPosition,
              int yPosition,
              int width,
              int height)
   …

/** (b) - formal parameters are name as their type */
void name (String aString)
   …
```

4.6 Benennung von Paketen

Für die Benennung von Paketen gilt generell, dass sämtliche Teile des Paketnamens in Kleinschrift darzustellen sind. Es dürfen keine Unterstriche, Zahlen oder Sonderzeichen verwendet werden. Die einzelnen Teile des Paketnamens (durch Punkte abgegrenzt) sind möglichst kurz und treffend zu wählen, so dass es anhand des Paketnamens möglich ist, auf den Funktionsumfang des entsprechenden Paketes zu schliessen.

Es existieren zwei mögliche Konventionen für die Benennung von Paketen, welche nachfolgend in 4.6.1und 4.6.2 aufgezeigt werden.

4.6.1 Paketbenennung von Organisationen mit eigener Internetdomäne

Paketnamen können nach der Konvention von Sun (Sun 1997) durch die Verwendung des in umgekehrter Reihenfolge[1] geschriebenen Domänennamens der Organisation, welche das Java-Paket erstellt, eingeleitet werden. Durch die Verwendung der Internetdomäne

wird eine Eindeutigkeit der Paketbenennung zwischen verschiedenen Softwareherstellern erreicht. Die Eindeutigkeit der Paketbenennung innerhalb der Pakete eines Softwareherstellers muss dieser selbst durch ein geeignetes Konfigurationsmanagement durchsetzen. Anschliessend an den Domänennamen erfolgt die weitere Unterteilung mit weiteren Paketnamen. Zur Veranschaulichung ist auf Beispiel 16 verwiesen.

BEISPIEL 16. Paketname mit der Verwendung des umgekehrten Internetdomänennamens.

```
/*
   Example for naming a package created within our institute using
   its domain name ifi.unizh.ch. The class Foo is
   accessable over the full qualification of the class name including
   the packet name.
*/
package ch.unizh.ifi.packagename.subpackagename;

…
public class Foo
   {
   public Foo()
      {
      …
      }
   …
   }
```

4.6.2 Paketbennenung von Organisationen ohne eigene Internetdomäne

Neben der unter 4.6.1 beschriebenen Konvention für die Benennung von Paketen können Pakete auch ohne die Verwendung der in umgekehrter Reihenfolge geschriebenen Internetdomäne benannt werden. Dies ist vor allem dann der Fall, wenn die Organisation, welche das Java-Paket entwickelt, keine eigene Internetdomäne besitzt oder diese unpraktisch für die Verwendung in einem Paketnamen ist (z.B. bei sehr langen Internetdomänen). In diesem Fall kann auch der Name der Organisation einfach als Einleitung des Paketnamens gewählt werden. Es sollte aber darauf geachtet werden, dass einerseits die Eindeutigkeit des Paketnamens gegenüber anderen Organisationen gewährleistet ist und dass es andererseits trotzdem möglich ist, mit dem einleitenden Namen des Paketes die Organisation eindeutig zu identifizieren.

[1] D.h. ch.unizh.ifi wenn die Domäne ifi.unizh.ch heisst.

5 Dokumentieren von Code

5.1 Grundidee, Kommentardichte

Zu den gebräuchlichsten und auch wichtigsten Dokumentationsmitteln für Code zählt der Kommentar. Kommentare tragen entscheidend zum schnellen Verständnis von Code und damit zum einfachen Ändern, Erweitern, Lesen, Portieren, Warten etc. von Software bei. Kommentare sind daher obligatorisch und müssen gleich sorgfältig wie der Code behandelt werden! Ungenügend dokumentierter Code ist grundsätzlich ein kritischer Befund in jeder Code-Inspektion.

Ein guter Kommentar versucht nicht das zu erklären, was bereits unmittelbar aus dem Code hervorgeht, sondern gibt die nötige Zusatz- bzw. Kontextinformation, um den kommentierten Code schnell und umfassend zu verstehen.

Erfahrungsgemäss gibt es eher zu wenige Kommentare und viel zu wenige Leerzeilen, als zu viel davon. Üblicherweise stehen Kommentare in einer eigenen Zeile *vor* dem kommentierten Abschnitt.

JavaDoc, Dokumentieren der Schnittstellen einer Software

Ein wünschenswertes Ziel bei der Erstellung einer Dokumentation ist es, die Schnittstellen der erstellten Software unabhängig von Implementierung dokumentieren zu können. Dies ermöglicht es, die Schnittstellen nach aussen zu beschreiben, ohne den dahinterstehenden Quellcode offenzulegen. Eine mögliche Lösung besteht darin, die Schnittstellendokumentation losgelöst vom Quellcode zu erstellen. Dabei ist die Gefahr von Inkonsistenzen zwischen Quellcode und der dazugehörenden Dokumentation aber sehr gross.

Ein anderer Ansatz verfolgt die Dokumentierung der Schnittstellen im Code. Mit einem entsprechenden Werkzeug kann aus der, im Quellcode dokumentierten Schnittstelle eine vom Quellcode unabhängige Schnittstellendokumentation erstellt werden. Dieser Ansatz wird durch das Werkzeug JavaDoc (JavaDoc 1999) verfolgt, welches im JDK[2] von Sun verfügbar ist. Mit JavaDoc ist es möglich, aus eingebetteten Kommentaren eine Quellcodedokumentation in HTML zu erstellen.

JavaDoc stellt einen Defacto-Standard bei der Dokumentation von Software, welche in Java entwickelt wurde, dar, daher müssen alle Kommentare in diesen Entwicklungsrichtlinien gemäss den Konventionen von JavaDoc abgefasst werden.

Die nachfolgenden Ausführungen beziehen sich auf die Verwendung von JavaDoc 1.2. Es können aber auch Vorgängerversionen mit gewissen Einschränkungen eingesetzt werden.

[2] Java Development Kit: Dabei handelt es sich um eine Sammlung von Werkzeugen zur Entwicklung von Java-Programmen.

5.1.1 Sprache für Kommentare

Um einen möglichst guten Lesefluss im Programmcode zu erreichen, sollte die Sprache für die Kommentare dieselbe sein wie diejenige für die Bezeichner (siehe hierfür Kapitel 4.2, »Sprache für Bezeichner und Kommentare«), im Normalfall ist die Kommentarsprache also Englisch.

5.2 Allgemeines zur Kommentierung mit JavaDoc

JavaDoc-Kommentare werden durch /** eingeleitet. Das Ende wird durch das Kommentarende */ bezeichnet.

Ein JavaDoc-Kommentar steht unmittelbar vor dem Codeelement, dessen Schnittstelle beschrieben werden soll (mit Ausnahme der Projektübersicht und der Paketbeschreibung). Es können folgende Elemente in JavaDoc kommentiert werden:

- Projektübersicht
- Pakete
- Klassen und Interfaces
- Attribute
- Methoden

JavaDoc verwendet bestimmte Tags, um Eigenschaften eines Kommentars zu definieren. Ein JavaDoc-Tag verwendet das @-Zeichen und ein zusätzliches Schlüsselwort.

Eine Übersicht und Beschreibung über alle JavaDoc Tags, welche im Rahmen dieser Richtlinien verwendet werden dürfen, ist in Kapitel 5.9 zu finden.

Die Reihenfolge der in den folgenden Kapiteln beschriebenen Tags ist nicht alphabetisch, sondern nach der Wichtigkeit der einzelnen Tags geordnet. Die jeweils vorgegebene Reihenfolge der beschriebenen Tags ist innerhalb der erstellten Dokumentation genau einzuhalten.

Gewisse Tags innerhalb dieser vorgegebenen Reihenfolge sind obligatorisch, andere fakultativ. Die fakultativen Tags können, falls sie nicht benutzt werden, vollständig aus der Dokumentation weggelassen werden. Die obligatorischen Tags hingegen müssen in jedem Fall in der Dokumentation in der vorgegebenen Reihenfolge aufgeführt werden. Je nach Kontext werden in den folgenden Kapiteln die jeweils obligatorischen und fakultativen Tags explizit aufgeführt.

In der Dokumentation werden Umlaute aus Gründen der Lesbarkeit und Formatierung mit zwei Buchstaben geschrieben, d.h. ae für ä, oe für ö und ue für ü. Spezielle Zeichen, wie

z.B. das < oder das > werden mit Hilfe der sogenannten HTML-Entities dargestellt, d.h. < wird als < und > wird als > geschrieben.

Wo dies nötig oder sinnvoll ist, sollen mit Hilfe von HTML-Tags Formatierungen vorgenommen werden.

5.3 Kommentierung der Projektübersicht mit JavaDoc

In JavaDoc ist es möglich, eine Projektübersicht über die erstellte Software einzubinden. Eine Projektübersicht dient als Einleitung für die Dokumentation der Software und wird als Einstiegsseite in der HTML-Dokumentation mit der Liste der Pakete angezeigt. Die Projektübersicht wird wie die Paketübersicht (siehe Kapitel 5.4) in einer eigenen HTML-Datei beschrieben und für JavaDoc bereitgestellt.

Alles, was sich innerhalb des HTML-Body Abschnittes von JavaDoc vorgefunden wird, wird als Übersicht in die Softwaredokumentation übernommen. Eine Übersicht über alle JavaDoc Tags ist zu finden in Kapitel 5.9.

Obligatorische Tags: @project, @java

Fakultative Tags: @copyright, @history, @version, @see, {@link}

Die Projektübersicht sollte weiter einen kurze Beschreibung des vorliegenden Softwareprojekt enthalten. Folgendes Beispiel soll die Verwendung einer Projektübersicht illustrieren:

```
<HTML>
   <HEAD>
   </HEAD>
   <BODY>
      This software project aimed in the implementation of
      a Java Virtual Machine which is fully implemented in
      Java itself.
      <br>…<br>
      Especially important are also the classes in the package
      {@link jvm.threading} where the component responsible for
      the multi threading is located.
      <br>…<br>
      @project   Java in Java JVM
      @java      JDK, 1.2
      @copyright Department for Computer Science, Requirement
                 Engineering Research Group
      @history   2000-02-01 SM 1.0 first version was built, jvm.types
                 jvm.memorymanagement inserted
                 2000-03-25 PH 1.1 second release built
                 2000-04-13 SM 1.0 third release build, jvm.threading
                 inserted
```

BEISPIEL 17. Eine Projektübersicht für eine von JavaDoc generierte Softwaredokumentation

```
        @version   2000-04-13, SM, 1.2

        @see       jvm.threading
        @see       jvm.types
        @see       jvm.memorymanagement
    </BODY>
```

</HTML>

5.4 Kommentierung von Paketen

Pakete können in Java auf mehrere (physikalische) Dateien verteilt sein. Es macht daher wenig Sinn (aber viel Mühe), Pakete durch Kommentare im Code zu dokumentieren, da bei einer evtl. Änderung gleich mehrere Dateien geändert werden müssen, die aber alle die gleiche Dokumentation bzw. Kommentierung enthalten. Pakete werden daher durch eine eigene Datei dokumentiert. Zu jedem Paket muss es daher eine entsprechende Dokumentation nach folgendem Vorbild geben (siehe Beispiel 18), welche auf dem aktuellen Stand zu halten ist:

BEISPIEL 18. Eine Paketbeschreibung, die von JavaDoc erzeugt wurde.

```
<HTML>
   <HEAD>
   </HEAD>

   <BODY>
      This package consists of several classes which
      provide the functionality of a Java Virtual Machine
      interpreter. It holds several sub packages. It uses the services
      provided by the {@link jvm.memory} package.
      …<br>
      @version   2000-04-13, PH, 1.1

      @see       jvm.threading
   </BODY>
</HTML>
```

Paketübersichten können in JavaDoc über sogenannte Paket-Dokumentationsdateien erstellt werden. Dazu wird die Dokumentation des Paketes in einer Datei namens *pakkage.html* im Verzeichnis des Pakets abgelegt. Diese Datei ist eine in HTML formatierte Datei, deren Body-Teil in die Dokumentation der Software integriert wird.

Der erste Satz der Paketbeschreibung sollte im Wesentlichen kurz und bündig den Zweck des Paketes erläutern. Dieser Satz wird als Abstract in der Projektübersicht verwendet. Erwähnungen von Klassen, Interfaces, Exceptions (des Paketes) und des Paketnamens sind innerhalb von JavaDoc überflüssig, da diese Informationen automatisch von JavaDoc aus dem Quellcode extrahiert und mitsamt Links in der erzeugten Paket-, Klassen- und der Projektübersicht aufgelistet werden.

Eine Paketdokumentation kann verschiedene JavaDoc-Tags enthalten:

Obligatorische Tags: -

Fakultative Tags: @history, @version, @see, {@link}

5.5 Kommentieren von Klassen

Eine vernünftig entworfene Klasse stellt nach aussen hin eine bestimmte Art von Diensten zur Verfügung und übernimmt die Verantwortung für deren Bereitstellung und Ausführung. Gleichzeitig verbirgt bzw. kapselt sie möglichst viel Wissen darüber, wie dieses Dienstleistungangebot klassenintern realisiert ist. Daher soll folgendes dokumentiert werden:

- Die Art von Dienst(en), die die Klasse zur Verfügung stellt, z.B. Bereitstellung und Manipulation einer Symboltabelle …
- Das konkrete Wissen über den jeweiligen Entwurf und dessen Realisierung, welches die Klasse kapselt.

Jede Klasse *muss* einen Klassenkommentar haben, welcher auf dem aktuellen Stand zu halten ist! Der Zweck und die Verantwortlichkeiten einer Klasse wird als erstes in einem prägnaten Abstract festgehalten, der mehrere Sätze umfassen kann. Nach dieser einleitenden Zweckbeschreibung der Klasse können verschiedene JavaDoc-Tags verwendet werden, um weitere Kommentarelemente anzugeben. Folgende JavaDoc-Tags müssen bzw. können verwendet werden:

Obligatorische Tags: @author, @history, @version, @responsibilities.

Fakultative Tags: @copyright, @invariant, @obligation, @since, @see, {@link}, @deprecated.

Folgendes Beispiel soll das Aussehen eines Klassenkommentares verdeutlichen (siehe Beispiel 19 auf Seite 32) :

BEISPIEL 19. Kommentierung einer Klassenbeschreibung mit JavaDoc

```
/**
   The class JvmInterpreter interprets Java bytecode instructions
   and several instructions into platform dependent calls.
   It uses for this purpose the class {@link jvm.ThreadScheduler}.

   @author        Peter Heiner
   @copyright     Department for Computer Science, Requirement
                  Engineering Research Group
   @history       2000-01-12 PH First version
                  2000-03-03 SM Review results corrected
                  2000-04-01 SM Invokevirtual problem corrected
   @version       2000-04-01, SM, 1.1
```

```
   @responsibilities
                  This class executes several bytecode instructions by
                  sing the interpreter an threading mechanisms.

   @see           jvm.ThreadScheduler
   @see           jvm.MonitorManager
   @see           jvm.interpretercore
*/
public class JvmInterpreter
   {
   …
   }
```

5.6 Kommentieren von Interfaces

Analog der Vorgehensweise zur Dokumentation von Klassen mit JavaDoc ist die Kommentierung von Interfaces aufgebaut, d.h. es werden dieselben Tags in derselben Reihenfolge verwendet, wie in Kapitel 5.5 beschrieben. Zusätzlich sollte aber das Anwendungsgebiet bzw. die intendierte Verwendung des Interfaces im Abstract genau erläutert werden.

Verantwortlichkeiten

5.7 Kommentieren von Methoden

Kopfkommentare sind obligatorisch. Für *jede* Methode, also auch für sehr einfache Methoden muss ein Kopfkommentar vorliegen, wie beispielsweise

```
aType aMethod()
   {
   return this.anObject;
   }
```

.Der Kopfkommentar einer Methode steht unmittelbar vor der Zeile des Methodenkopfs (keine Leerzeile dazwischen) und gibt an, welche Dienste die Methode unter welchen Voraussetzungen garantiert. Bekannte Probleme bei der Verwendung der Methode müssen ebenfalls angegeben werden.

Als Faustregel ist eine Methode mit mehr als 20 Zeilen als unterteilungsbedürftig anzusehen. Dabei soll die Methode in logische Abschnitte mit genügend vielen Leerzeilen unterteilt. Diese logischen Abschnitte sollen kommentiert werden. Der Implementierer sollte beim Kommentieren denken müssen, nicht der Leser beim Durchsehen!

Der Kopfkommentar ist üblicherweise nicht sehr lang und geht nicht auf Details der Implementierung ein. Diese sind evtl. abzutrennen und in einen eigenen Kommentar zu stecken. Der Kopfkommentar gehört noch zur Schnittstelle der Methode nach aussen. Wie

gesagt, er beschreibt *was* die Methode unter welchen Voraussetzungen tut, und *nicht wie* sie es tut.

Der allgemeine Kopfkommentar beschreibt den Zweck der Methode, d.h. den Dienst, welche die Methode anbietet. Handelt es sich um eine abstrakte Methode, so ist insbesondere zu kommentieren, wie und wozu diese in einer Unterklasse zu implementieren ist.

Der erste Satz dieses Kopfkommentars wird als Abstract verwendet, wenn die Methode von anderen JavaDoc-Elementen her referenziert wird. Daher muss dieser erste Satz bereits den Zweck und die Verwendung kurz und prägnant ausdrücken.

Die wichtigsten Teile des Methodenkopfkommentares sind nachfolgend erläutert:

- Vorbedingung `@pre` (siehe Kapitel 5.7.1)
- Nachbedingung `@post` (siehe Kapitel 5.7.2)
- Verpflichtungen `@obligation` (siehe Kapitel 5.7.3)

Bei der Kommentierung von Methoden gilt die Verwendung folgender Tags:

Obligatorische Tags: @pre, @post, @param (falls Parameter vorhanden), @return (falls Rückgabewert vorhanden)

Fakultative Tags: @invariant, @obligation, @throws, @since, @see, {@link}, @deprecated

5.7.1 Angabe der Vorbedingung im Kopfkommentar

Beim nachfolgend vorgestellten Kopfkommentar (siehe Beispiel 20) wird die Vorbedingung formal beschrieben. Ebenso wäre es möglich die Vorbedingung natürlichsprachlich zu spezifizieren, wenn möglich sollte aber die formale Variante gewählt werden.

Es soll ebenfalls dokumentiert werden, dass keine Vorbedingung existiert (durch `@pre-`). Existieren eine oder mehrere Vorbedingungen, so *müssen* diese im Kopfkommentar auch mit angegeben werden (siehe Beispiel 20). Folgende Regeln für die Kommentierung einer Vorbedingung sind gegeben:

- Die Vorbedingung wird mittels dem JavaDoc @pre Tag dokumentiert.
- In der Vorbedingung sind zumindest diejenigen Voraussetzungen zu dokumentieren, die zwar erfüllt sein müssen, damit die Methode ihre Aufgabe erfüllen kann, die aber nicht speziell von der Methode überprüft werden.
- Voraussetzungen, welche der Übersetzer (oder der Binder) bereits kontrolliert und somit auch deren Einhaltung garantiert, werden als Vorbedingung nicht extra aufgeführt. Ein Beispiel für eine solche Vorbedingungen ist: "der Parameter `aNumber` muss vom Typ `real` sein (siehe Beispiel 20)".

- Bei der Verwendung eines Tools, welches die Verletzung von Vor- und Nachbedingungen überprüft, muss die Beschreibung formal erfolgen (siehe auch Kapitel Tools).

BEISPIEL 20. Angabe der Vor- und Nachbedingung im Kopfkommentar

```
/**
Pops the two top most elements of the stack and returns them in a
array of size 2.
@pre       ((returnArray != false) && (returnArray.length >= 2) &&
            (this.stack.size()) > 1)
@post      stack.elementAt(stack.size() - 2)@pre != this.stack.top()
@param     returnArray The Array which should be filled with two top
           most elements. This array may not be null and it mus
           contain at least 2 elements.
@return    Returns the two top most elements of the stack
*/
protected Object[] popTwoElements(Object[] returnArray)
{
...
}
```

5.7.2 Angabe der Nachbedingung im Kopfkommentar

Im Allgemeinen wird die Nachbedingung gemäss folgenden Richtlinien dokumentiert:

- Es wird das @post Tag in JavaDoc verwendet.
- Das Tag enthält eine Beschreibung der Bedingung, welche nach dem Ausführen der Methode erfüllt sein muss.
- Die Beschreibung soll am Besten formal durch einen logischen Ausdruck, wie er auch in Java verwendet wird, beschrieben werden oder alternativ informal durch einen natürlichsprachlichen Text, wobei die formale Beschreibung vorzuziehen ist. Bei einer natürlichsprachlichen Formulierung soll darauf geachtet werden, dass nicht die Zweckbeschreibung der Methode in der Nachbedingung auftaucht.
- Bei der Verwendung eines Tools, welche die Verletzung von Vor- und Nachbedingungen überprüft, muss die Beschreibung formal erfolgen (siehe auch Kapitel Tools).

5.7.3 Angabe von Verpflichtungen

Sind mit der Benutzung einer Methode bestimmte Verpflichtungen verbunden, so sollte dies im Methodenkopf explizit dokumentiert werden (siehe Beispiel 21). Im Gegensatz zu den Vorbedingungen muss nicht dokumentiert werden, dass keine Verpflichtungen existieren, der betreffende Teil im Kopfkommentar wird dann weggelassen. Die Kommentierung erfolgt mittels dem JavaDoc @obligation Tag.

```
/**
   Opens a file with the filename accessing the data with random
      access for read and write.
   @pre     -
   @post    this.isFileOpen()
   @obligation
            The calling method has to take care that the opened file is al
            so closed after usage.
   @param   accessMode Describes the mode with which the file is opend for
            random access. One of these arguments are allowed for this parameter:
            <i>r</i> for writing, <i>rw</i> für schreibenden und lesenden Zugriff
            sein.
   @param   fileName Filename of the file to open.
   @throws  IOException if an IO error occurs
   @throws  IllegalArgumentException if the argument for accessMode is
            not correct.
   @since   1.1
*/
public RandomAccessFile(String fileName, String accessMode)
   throws IOException
   {
   ...
   }
```

BEISPIEL 21. Eine Konstruktormethode ohne zu dokumentierende Vorbedingungen, aber mit umfangreicher Nachbedingung und einer dokumentierten Verpflichtung.

5.7.4 Kommentierung von Code in Methoden

Bei der Kommentierung des Methodenrumpfs kommen keine JavaDoc Tags zum Einsatz, da JavaDoc zum Ziel hat die (Blackbox-)Funktionalität von Schnittstellen zu dokumentieren, nicht aber deren Implementierung. Daher erfolgt die Kommentierung mittels normalen Java-Kommentaren. Diese Kommentare werden entweder durch **//** (einzeilig) oder durch **/* ... */** (mehrzeilig) beschrieben. Es gibt dabei weiter folgendes zu beachten:

- Kommentare in der selben Zeile hinter Programmanweisungen dürfen nur zur Erklärung dieser einzelnen Anweisungen verwendet werden.
- Logische zusammenghörende Zeilen (Sinnabschnitte) innerhalb einer Methode sind durch Leerzeilen voneinander zu trennen. Nach der Leerzeile soll ein kurzer Kommentar den Sinn des folgenden Abschnitts angeben und evtl. unverständliche Befehle klären.
- Paraphrasieren von Anweisungen macht wenig Sinn. Vielmehr sollte der Kontext einer Anweisung beschrieben werden.

Folgendes Beispiel soll die Kommentierung innerhalb eines Methodenrumpfs beispielhaft illustrieren:

BEISPIEL 22. Dokumentation innerhalb eines Methodenkörpers.

```
/**
...
*/
public void addHashedElement(Object o)
   {
   // get default hash code of the object
   int hashCode = o.hashCode();

   // compute first position, which potentially contains no element
   int index = hashCode % hashedElements.length;
   ...
   if (hashedElements[index] != null)
      {
      // free array element found, set object
      hashedElements[index] = o;
      }
   else // element was not free
      {
      // compute collision index
      index = (index + 11 + hashcode) % hashedElements.length;
      }
   }
```

5.8 Kommentieren von Attributen mit JavaDoc

Die Kommentierung von Attributen mit JavaDoc folgt im Wesentlichen den Konventionen, welche in den vorhergehenden Abschnitten für Klassen und Methoden beschrieben wurden. Jedes Attribut muss mit einem Kommentar versehen werden, welcher den Verwendungszweck des Attributes beschreibt. Der erste Satz wird als Abstract in der Klassendokumentation verwendet. Er sollte deshalb kurz und prägnant sein.

Folgende Tags von JavaDoc können zur Dokumentation von Attributen verwendet werden. Diese Tags sollten im Normalfall nicht verwendet werden, können aber in Ausnahmesituationen helfen, mehr Klarheit bezüglich des Attributes zu schaffen:

Obligatorische Tags: -

Fakultative Tags: @since, @see, {@link}, @deprecated

Das folgende Beispiel illustriert die Verwendung der oben beschriebenen Konventionen:

BEISPIEL 23. Kommentierung eines Attributes mit JavaDoc.

```
/**
   Represents the current instruction offset where the next instruction to
execute is
   located.
   @see jvm.JvmInterpreter#executeByteCode(byte[], int)
```

```
*/
protected int programmingCounter;
```

5.9 Alle JavaDoc Tags für die Kommentierung in einer Übersicht

Folgende alphabetische Auflistung soll alle verwendbaren JavaDoc Tags für die Kommentierung von Methoden beschreiben. Im Anschluss daran findet sich eine Tabelle, welche nochmals zusammenfassend bezeichnet, welche Tags wo verwendet werden dürfen oder müssen:

- **@author <Vorname>, <Name>** : Beschreibt den Namen des Verantwortlichen für den entsprechenden Teil der Software (Klasse, Interface).
- **@copyright <Copyright Hinweis>**[3]**:** Dieses Tag bezeichnet den Inhaber des Urheberrechts in der Projekt und der Klassenbeschreibung.
- **@deprecated :** Dieses Tag bezeichnet Methoden, Attribute oder Klassen, welche veraltet sind. Es sollte nur in Ausnahmesituationen verwendet werden.
- **@history <Datum> <Namenskürzel> <Kommentar>**[3] : Dieses Tag enthält ein Log über sämtliche Änderungen, die an der Software vollzogen wurden. <Datum> beschreibt den Zeitpunkt der Änderung, <Namenskürzel> ist das Namenskürzel des Autors und <Kommentar> bezeichnet eine genaue Beschreibung was geändert wurde. In dieser Beschreibung kann mittels des {@link} Tags direkt auf die betroffenen Elemente im Code verlinkt werden.
- **@java <Java-Version>**[3] : Beschreibt die verwendete Java Version für das Produkt. Diese Versionsangabe stellt eine Minimalanforderung dar, damit das Projekt ausgeführt werden kann.
- **{@link <Referenz auf andere Elemente der Dokumentation>}**: Dieses Tag ermöglicht es, im Fliesstext der Klassenbeschreibung, Methodenbeschreibung, etc. weitergehende Referenzen anzugeben. Zu der Syntax der Referenzen siehe das @see Tag.
- **@obligation <Verpflichtung(en)>**[3] **:** Beschreibt diejenigen Verpflichtungen, die der Aufrufer der Methode explizit übernehmen muss. @obligation unterscheidet sich von @pre dadurch, dass mit diesem Tag Verpflichtungen dokumentiert werden, die aus einem grösseren Anwendungskontext hervorgehen.

[3] Dieses Tag ist in der Standardversion von JavaDoc nicht vorgesehen, aber es ist möglich, durch die Verwendung eines selbstentwickelten Zusatzes (Doclet) für die JavaDoc-API dieses Tag auszuwerten (siehe dazu auch Suns JavaDoc-Dokumentation (JavaDoc 1999)).

- **@param <Name eines Methodenparameters> <Beschreibung des Methodenparameters>**: Dieses Tag beschreibt einen Methodenparameter
- **@invariant <Invariantebedingung>**: Dieses Tag beschreibt eine Bedingung, welche nach der Ausführung der betroffenen Klasse, des Interfaces oder der Methode ausgeführt wird. Die Bedingung kann wie beim @post Tag formal oder informal beschrieben werden, wobei eine formale Beschreibung vorgezogen wird.
- **@pre <Vorbedingung(en)>**[3] **:** Beschreibt die Menge der Voraussetzungen (preconditions), deren Erfüllung vor Ausführung der Methode garantiert sein müssen, damit die Nachbedingung (@post) gilt. Falls keine explizit definierte Vorbedingung existiert, wird dies durch @pre - dokumentiert. Die Bedingung kann wie das @post Tag formal oder informal beschrieben werden, wobei eine formale Beschreibung vorgezogen wird.
- **@post <Nachbedingung>**[3] **:** Dieses Tag beschreibt die Menge der Nachbedingungen, welche eine Methode nach der Ausführung erfüllt, sofern alle Verpflichtungen und die Vorbedingung eingehalten wurde. Die Bedingung kann formal oder informal beschrieben werden, wobei eine formale Beschreibung vorgezogen wird.
- **@project <Projektname>**[3] : Bezeichnet den Namen des Java Projektes.
- **@responsibilities <Erklärung der Verantwortlichkeiten>**[3]: Dieses Tag bezeichnet die Verantwortlichkeiten und Entwurfsgeheimnisse einer Klasse. Benutzte Design Patterns werden hier erwähnt.
- **@return <Beschreibung des Rückgabewertes>:** Dieses Tag beschreibt den Rückgabewert einer Methode.
- **@since <Versionsnummer der Software, seit der diese Methode existiert>**: Dieses Tag ermöglicht es anzugeben, seit wann diese Klasse, Methode oder Attribut in der dokumentierten Software vorhanden ist.
- **@see <Referenz auf andere Elemente der Dokumentation>:** Dieses Tag ermöglicht es, im Anschluss an die Beschreibung eines Projektes, Paketes, Klasse Methode oder eines Attributes weitergehende Referenzen anzugeben. Eine Referenz für ein anderes Element der Dokumentation wird durch den Namen des Elementes bezeichnet. Eine Referenz auf eine Klasse wird durch ihren voll qualifizierten Namen dargestellt. Eine Referenz auf eine Methode oder ein Attribut wird durch den voll qualifizierten Klassennamen inklusive dem Methodennamen (abgetrennt durch das Nummernsymbol [#]) beschrieben. Werden Attribute oder Methoden in derselben Klassse referenziert, so kann auch nur der Methodenname bzw. der Attributname angeführt von einem Nummernsymbol (#) verwendet werden. Beispiele von Referenzen sind:

TABELLE 3. Referezen

Paket:	*paketname.unterpaketname*
Klasse:	*package.subpackage.ClassName*
Methode:	*package.subpackage.ClassnameName#method(int)*
Methode (in gleicher Klasse):	*#method(int, double)*
Attribut:	*package.subpackage.ClassName#anAttribute*
Attribut (in gleicher Klasse):	*#anAttribute*

- **@throws <Exceptionname>:** Beschreibt eine Exception, die von der Methode geworfen werden kann. Äquivalent zu @throws ist das JavaDoc-Tag @exception verwendbar.
- @version <Datum> <Namenskürzel> <Revision>[3]: Dieses Tag bezeichnet die vorliegende Version des betroffenen Code-Elementes in Java. Datum bezeichnet den Zeitpunkt, an welchem die Verison angelegt wurde. <Namenskürzel> bezeichnet den Namen der Person, welche die Änderung ausgeführt hat und <Revision> bezeichnet die Versions- und Revisionsnummer. Bei der Verwendung eines Versionierungstools (z.B. CVS) ist es möglich, diese Angaben durch das Tool eintragen zu lassen. Hinweise zur Verwendung von CVS sind im Kapitel Tools zu finden.

Nachfolgend sind die in den unterschiedlichen Teilen der Dokumentation einzusetzenden obligatorischen und fakultativen JavaDoc-Tags aufgelistet. Die Tags sind in derselben Reihenfolge in der Dokumentation zu verwenden, wie sie in der untenstehenden Tabelle aufgeführt sind.

Tag / Dokumentations-kontext	Projekt-übersicht	Pakete	Klassen	Interfaces	Methoden	Attribute
@author	-	-	**O**	**O**	-	-
@project	**O**	-	-	-	-	-
@java	**O**	-	-	-	-	-
@copyright	*F*	-	*F*	-	-	-
@history	*F*	*F*	**O**	**O**	-	-
@version	*F*	*F*	**O**	**O**	-	-
@responsibilities	-	-	**O**	**O**	-	-
@pre	-	-	-	-	**O**	-
@post	-	-	-	-	**O**	-
@invariant	-	-	*F*	*F*	*F*	-
@obligation	-	-	*F*	*F*	*F*	-
@param	-	-	-	-	**O**	-
@return	-	-	-	-	**O**	-
@throws	-	-	-	-	*F*	-
@since	-	-	*F*	*F*	*F*	*F*
@see	*F*	*F*	*F*	*F*	*F*	*F*
{@link}	*F*	*F*	*F*	*F*	*F*	*F*
@deprecated	-	-	*F*	*F*	*F*	*F*

O = **obligatorisch**
F = fakultativ
- = nicht verwendet in diesem Kontext

6 Einsatz von Tools

Einige zusätzliche Tools seien hier erwähnt, deren Einsatz optional bleibt, aber dringend empfohlen wird. Es handelt sich hierbei weder um eine Installations- oder Bedienungsanleitung dieser Tools. Es geht vielmehr darum wie durch die Verzahnung dieser Tools der Entwicklungsprozess stärker verbessert wird, als die Bausteine isoliert betrachtet bewirken könnten.

6.1 Konfigurationsmanagement mit CVS[4]

Die Verwendung eines Konfigurationsmanagement Tool ist generell sinnvoll. Der Austausch von Files wird vereinfacht, Änderungen werden protokolliert und alte Versionen können jederzeit wieder hergestellt werden. Hier ist beschrieben, welche Dinge beim Einsatz von CVS beachtet werden müssen. Bei den **fettgedruckten** Wörtern handelt es sich im Folgenden um Kommandos von CVS.

6.1.1 Revision and Releases

Revision X.Y: die Version einer einzelnen Datei, z.B. 1.56

Release ABC: quasi die Version des gesamten Projektes, also die Summe aller Revisions zu einem bestimmten Zeitpunkt. Ein Release wird erzeugt, in dem die aktuelle Revision aller zugehörigen Dateien mit einem Tag (nicht zu verwechseln mit JavaDoc-Tags) gekennzeichnet wird, also einem symbolischen Namen, z.B. REL_1.0. Das aktuelle Release ist die Summe aller neuesten Revisionen.

Während Revisionen bei Änderungen automatisch erzeugt werden, so müssen Releases manuell angelegt werden. Neben den tatsächlich abgeschlossenen Releases macht es Sinn, je nach Größe des Projektes Releases zusätzlich täglich oder wöchentlich zu kennzeichnen (z.B. mit dem Datum), um einfach auf einen alten Stand zurückkehren zu können. Damit kann zum Beispiel das Vorkommen eines Fehlers in einem älteren Release geprüft werden.

Bei der Verwendung von CVS gibt es unterschiedliche Philosophien. Bei großen Arbeitspaketen macht unter Umständen die zweite Vorgehensweise Sinn. Grundsätzlich sollte jedoch zunächst die erste in Betracht gezogen werden. In jedem Fall muss die Methode vor dem Projektstart bestimmt werden.

(1) Das aktuelle Release ist immer funktionstüchtig. Es ist Aufgabe derjenigen Person, die gerade Änderungen vornimmt, das sicherzustellen. Dazu wird nachdem die vorzunehmenden Änderungen durchgeführt wurden, jedoch vor einem **Commit,** der

[4] http://www.cvshome.org

Code nochmals aktualisiert (**Update**). Gibt es dabei keine Konflikte, lässt sich die lokale Version jetzt immer noch kompilieren und laufen auch die Testfälle lokal ohne Probleme, dürfen die Änderungen festgelegt (**committed)** werden.

(2) Bei großen Arbeitspaketen muss der Entwickler die Möglichkeit haben, Zwischenstände zur eigenen Sicherheit (Backup) im Repository abzulegen, schon bevor andere Entwickler damit arbeiten können. Dadurch wird das aktuelle Release in vielen Fällen unbrauchbar. Deshalb führt man ein Tag ein, das das letzte lauffähige Release bezeichnet, zum Beispiel OK. Das bedeutet, dass grundsätzlich auf das Release OK upgedatet wird, nicht mehr auf das aktuelle Release. Ist eine Änderung vollständig fertig gestellt und besteht sie auch den Check gegen das Release OK (siehe Punkt 1: **update**, kompilieren, testen), so dürfen die geänderten Dateien mit einem OK gekennzeichnet (**getaggt**) werden (natürlich nachdem sie eingecheckt wurden).

6.1.2 Schlüsselwörter

Angaben in einer Klasse bezüglich der aktuellen Revision oder dem Datum und Autor der letzten Änderung sind sehr sinnvoll. Allerdings sind solche Angaben häufig nicht brauchbar, da die Pflege vernachlässigt oder vergessen wird. CVS bietet dafür Unterstützung an, diese Informationen automatisch aktuell zu halten. Bei jedem **Commit** werden die sogenannten Schlüsselwörter (z.B. $Date$) durch aktuelle Werte ersetzt. Dadurch ergibt sich eine ideale Kombination von CVS und JavaDoc, in dem man CVS Schlüsselwörter bei JavaDoc-Tags hinterlegt. Bei der Verwendung von CVS wird deshalb der Syntax des @version Tags wie untenstehend geändert. In Zukunft wird es von CVS auf dem aktuellen Stand gehalten. Manuell darf hier nichts mehr editiert werden.

```
@version $Date$, $Author$, $Revision$
```

6.1.3 Kommentare beim Commit

Von der Durchführung eines Arbeitspaketes, also der kleinsten funktionalen Änderung, so dass das Programm wieder läuft, sind meist mehrere Dateien betroffen. Nur zusammengenommen machen sie Sinn. Das soll sich - ähnlich den Tags bei den Releases – hier in den verwendeten Kommentaren beim **Commit** bemerkbar machen. Idealerweise werden alle geänderten Dateien gemeinsam **committed**, so dass alle Dateien denselben Kommentar aufweisen, der die Änderung beschreibt. Wird zusätzlich ein Fehlermanagementtool wie zum Beispiel *Bugzilla* eingesetzt, so muss ein Hinweis im Kommentar auf die Fehlerbezeichnung dort enthalten sein. Beispiel: *Bug 27: notification of the observer was missing.*

6.2 IContract[5]

IContract ist ein Tool, das es erlaubt, formale Invarianten, Pre- und Postconditions automatisch auszuwerten. Mittels eines Precompilers wird für die im Kommentar hinterlegten Bedingungen Code generiert, der bei entsprechenden Verletzungen mit Exceptions reagiert. Es bietet sich also die Möglichkeit, die vertraglichen Bedingungen bei Methodenaufrufen (Design by Contract) zu überprüfen. Damit wird es möglich eine instrumentierte Versionen des Programms herzustellen:

- die unveränderte Version, die die Einhaltung des Vertrages voraussetzt und
- eine instrumentierte Version, die zu Testzwecken verwendet werden kann, um die Einhaltung aller Verträge kontinuierlich während der Ausführung zu prüfen. Dadurch können Fehler - vor allem nach durchgeführten Änderungen - automatisch und dem nahe dem tatsächlichen Fehlerherd entdeckt werden.

Bedingungen müssen formal, das heißt in Javacode ähnlicher Form beschrieben und bei den entsprechenden Tags hinterlegt werden, z.B.:

```
@prevalue > 5
@post@return != null
@invariantobjectList.size() > 0
```

Bei Auswertung mittels eines Tools dürfen keine textuelle Beschreibungen bei einem dieser JavaDoc-Tags verwendet werden. Diese können separat im allgemeinen Kommentarteil angehängt werden. All drei Tags sollen im Methodenkommentar verwendet werden; das Tag @invariant ist sogar im Klassenkommentar einsetzbar. Man beachte, dass diese Tags ebenfalls an Subklassen vererbt werden. Diese dürfen also dort nicht wiederholt werden. Um eine echte Subtyp-Beziehung zu programmieren, was dringend empfohlen wird, dürfen in der Subklasse

- Preconditions höchstens abgeschwächt und
- Postconditions höchstens verstärkt werden.
- Invarianten bleiben für alle Klassen unverändert bestehen und können damit in der Superklasse hinterlegt werden.

[5] http://www.reliable-systems.com/tools/iContract/iContract.htm

Anhang A Literatur

Ambler, Scott W.(1997): Java Coding Standards, [http://www.AmbySoft.com/javaCodingStandards.pdf], 10.07.1997.

Lea, Doug (1996): Draft Java Coding Standard. [http://g.oswego.edu/dl/html/javaCodingStd.html].

Lorenz, M. (1993): Object-Oriented Software Development – Practical Guide. Prentice Hall, Englewood Cliffs, 1993.

Schneider, K. (1994): Styleguide für Smalltalk-Entwickllungen – Version 3 und 4. Abt. Software Engineering, Institut für Informatik, Universität Stuttgart, 1992 und 1994.

Weitere Literatur (in den Richlinien nicht direkt zitiert)

Berner et al. (1996): Entwicklungsrichtlinien für Java-Software-Version 1.4.2; Institut für Informatik der Universität Zürich, 1996. [http://www.ifi.unizh.ch/groups/req/publications/selected_publications.html].

Berner et al. (1997): Skript zum Kurs Programmieren in Java, V1/SS97; Forschungsgruppe Requirements Engineering; Institut für Informatik der Universität Zürich, 1997.

dubhe (1997): Java Style Guide, [http://dubhe.cc.nps.navy.mil/~java/course/styleguide.html].

Meyer, B. (1988): Object-Oriented Software Construction. Prentice Hall International, London, 1988.

Sandvik, Kent (1996): Java Coding Style Guidelines, [http://reality.sgi.com/sandvik/JavaGuidelines.html], 02.02.1996.

Skibinski, Jan (1996): Comments on Java programming style, [http://www.numeric-quest.com/nesw/NQ-comments.html]; 1996.05.06 for Java 1.0.

Anhang B Index

A

abstract 8, 9, 11, 34
abstrakte Klasse 9
Abweichung von den Richtlinien 3
Aktionsmethode 12
Aktualisierung alten Codes 3
Änderung
- der Entwicklungsrichtlinien 2
- Dokumentierung von Änderungen 17

Änderungsvorschläge 2
Attributierung 11
Attributierung der Klasse 8
Aufgabe der Entwicklungsrichtlinien 2
Aufteilung von langen Ausdrücken 18
Aufzählungstypen 6

B

Bedingung 18
Benennung von Methoden 25
Bereichstypen 6
Bezeichner 23
- Bezeichnerwahl 22
- für formale Parameter 26
- für Instanzvariablen und lokale Variablen 24
- für Klassen und Klassenvariablen 23
- für Konstanten 24
- gleichartige Bezeichner 22
- Gliederung von Bezeichnern 22

Bezeichner für Exceptions 23
Bezeichner für Interfaces 24
Bezeichner für Klassenvariablen 23
Bezeichner für lokale Variablen 24
- Exceptions 24
- Schleifenzähler 24
- Streams 24

Bezeichnerwahl 22
Blöcke 20
Blockkommentar 20
Browser 15

C

Codierregeln 4
- Deklaration von Arrays 7
- Deklaration von lokalen Variablen 7
- Identitäts- oder Wertvergleich 5
- Methode equals 5
- Methode main 4
- Variable 7

Copyright 1

E

Einrückung 15, 17
- der Importdeklaration 17
- der Klassendeklaration 17
- der Paketdeklaration 17
- des Kopfkommentars 17
- des Methodenkopfs 17
- des Methodenrumpfs 17
- einer zusammengesetzten Nachricht 18
- von Blöcken 20
- von Kommentaren 17

else 19
else if 18
Error 14
Errors 14
Exception 14
Exceptions 13, 14
- Anwendung 13

F

Fallunterscheidung 18
Faustregel 25
- für Klassenentwurf 10
- für Methodenentwurf 11
- Kommentierung von Methoden 33

Fehlen der Tabulatorfunktion 15
final 8, 24
Forschungsgruppe Requirements Engineering 1

G

Gleichartige Bezeichner 22

Gliederung von Java-Code 15
Gross-Kleinschreibung 22

H

Hiding names 23

I

if 18, 19
if-Anweisung 18
    Dangling-Else 20
    Zuweisungsoperator 5
Import 8
    importieren mittels * 8
    nicht eindeutiger Import 8
    volle Qualifikation 8
Inhaltsübersicht 3
Instanzvariable 10
Interface 9, 11
    Einrückung 17

J

Java-Beans 4

K

Kaskaden 18, 25
Klasse 8
    abstract 9
    abstrakte Klasse 8, 9
    Anzahl Instanzvariablen 10
    Anzahl Methoden 10
    Attributierung 8
    Bezeichnerwahl 22
    Einrückung 17
    Grösse der öffentlichen Schnittstelle 9
    Instanzvariable 10
    Klassenkopf 8
    Klassenvariable 9, 10
    teilweise abstrakt 9
    Vermindertes Coupling 9
    Zugriff auf Klassen- und Instanzvariablen 9
Klassenkommentar 32
Klassenvariable 9, 10
Kommentar 3, 17, 28
    in der selben Zeile 36
    Kopfkommentar 11, 33
    Sprache für Kommentare 29
Kommentardichte 28
Kommentierung
    der if-Anweisung 18
    einer abstrakten Methode 34
    von Klassen 32
    von Methoden 33
    von Paketen 31
Konfigurationsmanagement
    Basisdienste 4
Kopfkommentar 11, 33
    Angabe der Nachbedingung 35
    Angabe der Verpflichtung 35
    Angabe der Vorbedingung 34
    Angaben zum Änderungsstand 17
    Nachbedingung 34
    Verpflichtung 34
    Vorbedingung 34

L

Lange Ausdrücke 18
Layout 15
Leerzeichen 15
Leerzeile 28
Leerzeilen 15

M

Mehrfachvererbung 9
Methode
    abstrakte Methode 34
    Aktionsmethode 12
    Attributierung 11
    Aufteilung von langen Ausdrücken 18
    Bezeichnerwahl 22
    Einrückung einer zusg. Nachricht 18
    equals 5
    kaskadierende Methodenaufrufe 18
    Kopfkommentar 34, 35
    Länge einer Methode 10
    Main 4
    Methodenkopf 11
    Nachbedingung 35
    Prädikatmethode 12

statische Initialisierer 10
Stubs 11
Substituieren von Aufzählungstypen 6
Substituieren von Bereichstypen 6
Überladen 5
Überladen von Methoden 5, 6
Vergleichsmethode 12
Vorbedingung 34
Wertvergleich 5
Zugriffsmethode 12
Zustandsmethode 12
Methoden
Aktionsmethoden 25
Prädikatmethoden 25
Zugriffsmethoden 25
zum Setzen und Lesen von Variablen 25
Methodende
Einrückung 17
Methodenkopf 11
Methodennamen 25
Minimalrahmen 3

N

Nachbedingung 34, 35
non-private static 10

O

Obligation 34, 35

P

Paket
Kommentierung 31
Prädikatmethode 12
prime directive 3
private 10, 15, 25
protected 10
public 36

Q

Quellen 1

S

Schnittstelle 9
der Methode 33
Setzmethode 12
Sichtbarkeit des Klassenbezeichners 9
static 9
statische Initialisierer 10
subClassResposibility 11
Synchronisation 10

T

Tabulator 15
Tabulatorstufe 15
Texteditor 15

U

Ungenügend dokumentierter Code 28

V

Variable 9
Bezeichnerwahl 22
Deklaration 7
Instanzvariable 10
Klassenvariable 9, 10
kombiniertes Setzen von Variablen 11
Zugriff auf Variablen 9
Variablen
gleichartige Bezeichner 22
Verantwortlichkeiten 33
Verbindlichkeit der Richtlinien 2
verborgene Seiteneffekte 9
Vergleichsmethode 12
Vergleichsoperator
Identitätsvergleich 5
Verpflichtung 34, 35
Version der Richtlinien 3
Vorbedingung 34
Vorwort 1

W

while-Anweisung
Zuweisungsoperator 5

Z

Zugriffsmethode 9, 12
Zugriffsmethoden 25
Zustandsmethode 12