Entwicklungsrichtlinien

für

Java-Software

Forschungsgruppe Requirements Engineering

Institut für Informatik

Universität Zürich

INHALTSVERZEICHNIS

Entwicklungsrichtlinien für Java-Software

Stefan Berner, Martin Glinz, Stefan Joos, Johannes Ryser, Nancy Schett
Forschungsgruppe Requirements Engineering

Institut für Informatik der Universität Zürich

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

Erste Version erstellt am:	1996/06/07/1051 (Freitag, 7. Juni 1996, 10:51 Uhr)
Erste Version erstellt von:	SB
Zuletzt geändert am:	1998/12/08/1122 (Dienstag, 8. Dezember 1998, 11:22 Uhr)
Zuletzt geändert von:	SB, NS
Projekt:	projektübergreifend
Version:	2.0.1
Mitarbeiter:	Martin Arnold (MA), Stefan Berner (SB), Martin Glinz (MG), Stefan Joos (SJ), Johannes Ryser (JR), Nancy Schett (NS)
Status:	geprüft 1998/05/18, herausgegeben 1998/05/18
Prüfteam:	MG, MR, SJ, JR

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).

Gedankt werden soll an dieser Stelle den kritischen Prüfteams, welche aus Martin Arnold, Martin Glinz, Stefan Joos, Mathias Richter sowie Johannes Ryser bestanden und sich der aufwendigen und mühevollen Prozedur unterzogen haben, diese Richtlinien für die verschiedenen Versionen gründlich zu "reviewen".

Diese Richtlinien sind in erster Linie gedacht und gemacht für gruppeninterne (Forschungsgruppe Requirements Engineering) Software-Projekte; d.h. es ist derzeit unbeantwortet, ob und wie sinnvoll sie in "fremden" Projekten angewandt werden können.

Die Entwicklungsrichtlinien dürfen von jedermann frei kopiert und benutzt werden, aber nur in ihrer vollständigen Form. Die Benutzung dieser Entwicklungsrichtlinien erfolgt auf eigene Gefahr, d.h. die Autoren haften ausdrücklich nicht (weder direkt noch indirekt) für etwaige Schäden.

Wir möchten Sie ferner bitten, sich (ab und zu) mit uns wegen der neuesten Version der Richtlinien in Verbindung zu setzen, damit – wenn irgendwie möglich – immer nur diese Version weitergegeben wird. Änderungs-, Korrektur- und/oder Verbesserungsvorschläge sind jederzeit willkommen.

Die aktuelle Version finden Sie auch immer an folgender Stelle im World Wide Web:
http://www.ifi.unizh.ch/staff/berner.html

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 dann 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 Betreuers- 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 Betreuer vereinbart werden. Die Abweichungen müssen schriftlich festgehalten werden.

1.4 Aktualisierung alten Codes

Die vorliegende Version 2.0.1 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 (siehe auch Kapitel 2.1, »Änderungen und Erweiterungen von bestehendem Code«). 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 geregelt.

2 Programmierhinweise für Java

2.1 Änderungen und Erweiterungen von bestehendem Code

Änderungen an vorgefundenen Klassen, müssen immer auch als Java-Quell-Code verfügbar gemacht werden (nur die *Class-Datei* ist explizit nicht ausreichend, die *Java-Datei* muss ebenfalls verfügbar sein). Verfügbar gemacht werden heisst, der geänderte Code muss mittels Übersetzung in ein Java-Projekt übernommen und integriert werden können, wobei hierfür nur die minimale Code-Menge vorliegen darf. Minimale Menge heisst, es dürfen nur die geänderten oder erweiterten Klassen oder Methoden in der Änderungsdatei vorhanden sein.

Jede Änderung vorgefundener Klassen *muss* ausserdem über eine *ReadMe-Datei* dokumentiert werden. Damit soll sichergestellt werden, dass der eigentlich zuständige Autor zuverlässig erkennen kann, dass sein Code verändert wurde. Alle Änderungen sind zentral für das Projekt an einer Stelle abzulegen (und unterliegen dem Konfigurationsmanagement, siehe Kapitel 2.1.1).

Vorgefundene Klassen sind Systemklassen oder Klassen anderer Projektbeteiligter. Vorgefundene Klassen dürfen nur in Ausnahmefällen verändert werden:

- Systemklassen (→ ausnahmslos alle *API-Klassen*) dürfen nicht geändert werden auch wenn der Quellcode prinzipiell zur Verfügung steht; das heisst u.a. alle dort bereits bestehenden Variablendeklarationen oder Methoden dürfen nicht verändert werden.
- Wird in einer vorgefundenen Klasse ein Fehler gefunden, so darf er nur nach Abstimmung mit einem Betreuer korrigiert werden, weil oft Fehler vermutet werden, die dann doch keine sind.
- Im Anschluss an eine Änderung muss der Bearbeiter bzw. der Betreuer die Verbesserung an alle anderen Betroffenen weitermelden und ihnen die verbesserte Klasse anbieten (nicht aufdrängen).
- Sollen vorgefundene Klassen um neue Funktionalität erweitert werden (wobei keine bestehenden Methoden verändert werden), so ist zunächst kritisch zu prüfen, ob die neue Funktionalität wirklich der vorgefundenen Klasse selbst zugeordnet werden muss, oder ob nicht sinnvollerweise eine Unterklasse zu bilden ist.

2.1.1 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.

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.

Java-Beans

- Mittels der *Default-Konstruktormethode* `newInstance()` eines `Class`-Objekts können Objekte erzeugt werden. Dies ermöglicht Klassen, deren Typ zur Kompilierzeit nicht definiert oder unbekannt ist, dynamisch währen der Laufzeit zu laden und zu instantiieren. Dieses Vorgehen wird z.B. beim Laden von unbekannten Applets in Html-Seiten angewandt.

Überladen von Methoden

➢ Das *Überladen von Methoden* innerhalb derselben Klasse ist gestattet, falls diese aus gleich vielen Parametern bestehen und 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);   // öffnet eine Datei für byteweises Lesen
     open(Stream); // öffnet einen Stream für byteweises Lesen

falsch:
     add(int n);   // addiert n zum Empfänger hinzu
     add(String name);// Stringkonkatenation
```

Identitäts- oder Wertvergleich

➢ Der Vergleichen 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()` überschrieben werden. Wertvergleiche sind meist bei der Arithmetik anzutreffen: der Wertvergleich für zwei komplette 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.

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

```
if (i==true)
     {    /* richtig, da vom Typ boolean */
     …
     }
else if (i=true)
     {    /* falsch, da Zuweisung */
     …
     }
```

...

Substituieren von Aufzählungstypen oder Bereichstypen

➢ 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.

BEISPIEL 3. Substituieren von Aufzählungstypen

```
TYPE        // (a)
  TrafficLightType
    = ( red, green, yellow );

VAR
  trafficLight
    : TrafficLightType;

BEGIN
  ...
  IF ( ampel = red ) THEN
    ...
  END
  ...
END
```

```
            // (b)
...
void foo()
  {
  /* 0=red, 1=yellow,
  2=green */
  int trafficLight;
  ...
  if ( trafficLight == 0 )
    {
    ...
    }
  ...
  }
```

```
class TrafficLight// (c)
  {
  boolean isRed() { ... };
  boolean isYellow() { ... };
  boolean isGreen() { ... };
  ...
  void fooToo( ... )
    {
    TrafficLight trafficLight
      = new Ampel();
    ...
    if (ampel.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 Ampel geschaffen werden (c).

➢ *Generell gilt*: Für alle nicht privaten Klassen ist – obwohl aufwendiger – eine Lösung nach Schema (c) vorzuziehen, 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 muss dort stattfinden, wo ihr *Initialwert* bekannt ist. Dies ist besonders bei Schleifen der Fall.
- ➢ Lokale 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

- ➢ **`Continue`** und **`break`** müssen in Schleifen umsichtig verwendet werden.
- ➢ **`Switch`**-Anweisungen müssen immer einen **`default case`** beinhalten.

➢ **`Break`** muss jeden **`case`** einer **`switch`**-Anweisung beenden.

2.3 Pakete

… wird bei Bedarf ergänzt …

2.3.1 Dokumentation von Paketen

(siehe auch Kapitel 5.2, »Kommentierung von Paketen«)

2.4 Importe

Importangaben dokumentieren diejenigen Ressourcen, die eine 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.

"final" Klassen

➢ Eine Klasse sollte nur dann **`final`** attributiert sein, wenn sie Unterklasse oder Implementation einer Schnittstelle ist, welche bereits alle Methoden definiert, die nicht implementierungsspezifisch für die jeweilige Klasse sind. Wenn eine Klasse "nur" **`final`** attributiert wird, dann kann von dieser Klasse keine Unterklasse mehr gebildet werden. Existiert zu dieser Klasse eine Basisklasse, die nicht **`final`** attributiert ist, aber gleiche Funktionalität hat, dann besteht zumindest die Möglichkeit, von dieser Klassen eine Unterklasse zu bilden.

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 Instanzvariable `name` also `aThing.name(aName)` zum Setzen und `aThing.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 nur mit Hilfe dieser Methoden verändern und nicht direkt in anderen Methoden auf sie zugreifen. Dies stellt u.a. sicher, dass sich sowohl Synchronisations- wie auch Benachrichtungsmechanismen einfach ändern lassen, wenn dies notwendig werden sollte (siehe Kapitel 2.7.5, “Synchronisation”).

- 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).

- Nur in äussert seltenen Ausnahmefällen, wenn es beispielsweise auf das letzte Bisschen Rechenzeit ankommt, darf hiervon abgesehen werden und Variablen dürfen **`public`** deklariert sein. Diese Ausnahmen sind – insbesondere auch im Klassenkommentar – sorgfältigst zu dokumentieren *und* bedürfen ggf. der Zustimmung eines Betreuers. Zusätzlich sollte der betroffene Feinentwurf oder Code einer Inspektion bzw. einem Review unterzogen werden.

pseudo finals

- Für Klassen- oder Instanzvariablen, deren Wert sich nach der ersten Zuweisung nicht mehr ändert, die aber nicht final deklariert werden können (sogenannte *pseudo* **`finals`**), z.B. weil zum Zeitpunkt der Objekterzeugung der Wert noch nicht bekannt ist, sollte dokumentiert werden, dass der Wert, wenn er einmal gesetzt worden ist, nicht mehr geändert wird bzw. werden kann. Dies kann u.a. dadurch geschehen, dass dies mit den Worten "pseudo final" dokumentiert wird.

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 5.3, »Kommentieren von Klassen«)

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.

Minimieren der public oder protected Schnittstelle 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`** oder **`protected`** Schnittstelle 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.

2.6 Interfaces

… wird bei Bedarf ergänzt …

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.

2.7.1.1 Methodendeklaration

Diejenigen Methoden im 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.

2.7.1.2 Kopfkommentar

Vorbedingung im Kopfkommentar

Die Vorbedingung (kurz *Pre* für engl. Pre Condition) dokumentiert diejenigen Voraussetzungen, die erfüllt sein müssen, damit eine Methode ihre Aufgabe spezifikationsgemäss erfüllen und somit auch die Nachbedingung garantiert werden kann (siehe Beispiel 20). Es ist üblich, dass der Aufrufer das Erfülltsein der Vorbedingung sicherstellt und nicht der Aufgerufene. Die Methode selbst ist nicht verpflichtet die Vorbedingung zu überprüfen oder deren Einhaltung sicherzustellen, sondern darf davon ausgehen, dass sie erfüllt sind. Konkrete Richtlinien für die Dokumentation der Vorbedingung finden sich in Kapitel 5.5.2, »Angabe der Vorbedingung im Kopfkommentar«.

Nachbedingung in Kopfkommentar

Die Nachbedingung (kurz *Post* für engl. Post Condition) dokumentiert diejenigen Dienste, welche die Methode nach aussen hin anbietet und garantiert zu erfüllen. Für jede Methode ist die Nachbedingung zu dokumentieren. Konkrete Richtlinien für die Dokumentation der Nachbedingung finden sich in Kapitel 5.5.1, »Angabe der Nachbedingung im Kopfkommentar«.

Verpflichtung im Kopfkommentar

Insbesondere bei der Vergabe von Betriebsmitteln kann es vorkommen, dass mit dem Aufruf einer Methode der Aufrufer automatisch bestimmte Verpflichtungen (kurz *Obligation* für engl. Obligations) eingeht bzw. eingehen muss, beispielsweise wenn ein durch den Methodenaufruf belegtes Betriebsmittel später wieder frei gegeben werden muss. Konkrete Richtlinien für die Dokumentation von Verpflichtungen finden sich in Kapitel 5.5.3, »Angabe von Verpflichtungen«.

2.7.2 Methodenrumpf

… wird bei Bedarf ergänzt …

➢ *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).

Erfordert das Setzen einer Variable aufwendige Operationen, so sollten wo immer möglich diese Operationen in der Setzmethode erfolgen und keine externe Aufbereitung erfordern. Wenn scheinbar ein Name als String gesetzt wird, intern aber ein eigenes Namensobjekt geschaffen wird, sähe das so aus (siehe Beispiel 5):

BEISPIEL 5. Setzen von Variablen über eine Setzmethode. Es wird scheinbar ein Name als String gesetzt, intern aber ein eigenes Namesobjekt geschaffen.

```
/**  … Kopfkommentar …

Erste Version von Kurt Schneider, 13.03.1993
Letzte Änderung von Stefan Berner, 17.07.1994

Belegt den Namen */
void name( String aName )
     {
     /* Temporäre Variable */
     NameObject aNameObject;

     /* Zuerst Namensobjekt anlegen, dann benennen */
     aNameObject = new NameObject();
     aNameObject.doSomething( aName );

     /* direkter Zugriff auf <name> NUR hier */
     name = aNameObject;
     };
…
```

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:

Zustandsmethoden und Zugriffsmethoden

(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 35). Eine Methode dieser Art wird hier *Zustands-* oder *Zugriffsmethode* genannt.

Vergleichsmethoden und Prädikatemethoden

(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.

Aktionsmethoden

(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.

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 5.5, »Kommentieren von Methoden«)

2.7.5 Synchronisation

synchronized

➢ *Synchronized Methoden* sind *synchronized Blöcken* vorzuziehen. Dies ergibt eine bessere Kapselung, weniger Problemen mit Subklassen und fördert die Effizienz.

Notifyall versus notify

➢ `NotifyAll()` sollte anstelle von `notify()` oder `resume()` verwendet werden. `Notify()` weckt nur einen zufällig ausgewählten Thread auf, während `notifyAll()` *alle* Threads aufweckt. Wird nur ein Thread aufgeweckt, so ist die Gefahr einer Verklemmung grösser. Das Aufwecken aller Threads kostet in der Regel nur wenig mehr Rechenzeit und da die Gefahr einer Verklemmung geringer ist, wenn jeder Thread die Möglichkeit hat, aktiv zu werden, sollte `notifyAll()` bevorzugt werden.
Klassen, die nur `notify()` verwenden, können normalerweise nur maximal eine Art von wait-Bedingung über alle Methoden in der Klasse und allen Subklassen unterstützen. Mit `notifyAll()` hingegen werden alle mit der Klasse in Verbindung stehenden Referenzen / Objekte benachrichtigt.

wait bei Methoden

➢ Wenn eine Methode `wait` aufruft, muss dies dokumentiert werden.

wait in while Schleifen

➢ `Wait` Statements sind in **`while`** Schleifen einzubauen. Ein **`if`** alleine reicht nicht aus, da jedesmal beim Aufwachen überprüft werden muss, ob die Bedingung zum Weitermachen gegeben ist.

2.7.6 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.

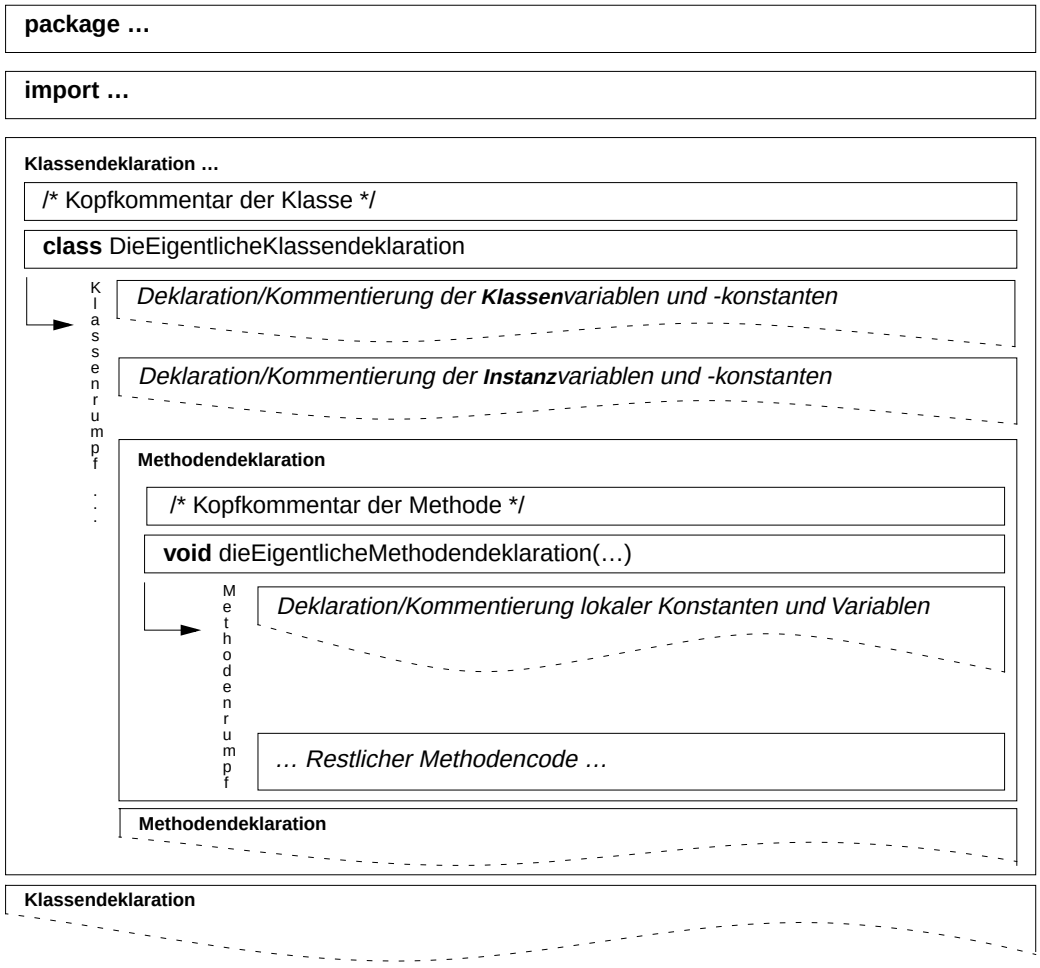
3 Einrückungen und Layout

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. 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 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

3.3 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 Beispiel 8, Beispiel 18, Beispiel 19 und Kapitel 5.5, »Kommentieren von Methoden«)

BEISPIEL 8. Angabe des Änderungsstandes einer Methode im Kopfkommentar der Methode.

```
/**  … Kopfkommentar …

Erste Version von <Name>, <Datum>
Zuletzt geändert von <Name>, <Datum> */
void aMethod()

     {    /* BEGIN aMethod */
     …
```

3.4 Methodenrumpf

3.4.1 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 25 `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.

BEISPIEL 9. Gliederung von zusammengesetzten Nachrichten.

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

3.4.2 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):

BEISPIEL 10. Allgemeine Form für die Gliederung von Fallunterscheidungen

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

…

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

```
…
/* Wenn die X-Koordinate nicht mehr auf dem Bildschirm liegt,
dann die X-Koordinate auf minimal mögliche X-Koordinate setzen
*/
if ( preferredWindowSize.x() < screenSize.x() )
     {
     preferredWindowSize.move(    screenSize.x(),
                                  preferredWindowSize.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 (siehe Beispiel 10 und Beispiel 12).

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 */ };
…
```

3.4.3 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 3.4.1).

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 = false )
     {
     try
          {
```

```
                wait();
                …
                }
            catch { InterruptedException anException }
                {
                …
                }
            };
…
```

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 `xpos` und die andere Variable `xPos` benannt ist.

4.2 Sprache für Bezeichner und Kommentare

Alle Bezeichner sind in Englisch abzufassen. Auch lokale und temporäre Variablen haben die englische Schreibweise zu beachten. Alle Kommentare sind in Deutsch 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 für Klassen, die eine Exception repräsentieren, enden mit dem Vermerk *Exception*, d.h. `ClassNameEndsWith`**`Excepion`**. Somit ist die Verwendung aus der Endung des Klassennamens sofort ersichtlich.

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.

Weitere Standards für lokale Variablen: Einige dieser Vorgaben für lokale Variablen werden insbesondere von ***Sun*** forciert:

TABELLE 2. Bezeichner für lokale Variablen

Typ der Variable	*Vorgeschlagene Namenskonvention*
offset	*off*
length	*len*
byte	*b*
char	*c*
double	*d*
float	*f*
long	*l*
integer	*int*
Object	*o*

Typ der Variable	*Vorgeschlagene Namenskonvention*
String	*s*
Arbitrary value	*v*
Point	*p*
Dimension	*d*
Graphics	*g*
Rect	*r*
Image	*img*

Die Namenskonvention schreibt für die Variablen-Typen `double` und `dimension` den Buchstaben **`d`** vor. Da es sich um lokale Variablen handelt und die Abkürzung je nach Kontext vergeben wird, ist diese Überschneidung von keiner grossen Bedeutung.

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 31).

- 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) Formale Parameter sind wie die Rolle bezeichnet,
welche diese in der Methode innehaben */
void reshape( int xPosition,
              int yPosition,
              int width,
              int height)
    …

/* (b) - Formale Parameter sind nach dem Basistyp bzw. der
Basisklasse bezeichnet */
void name( String aString )
    …
```

5 Dokumentieren von Code

5.1 Grundidee, Kommentardichte

Zu den gebräuchlichsten und auch wichtigsten Dokumentationsmitteln für Code zählt der Kommentar. Mitunter tragen Kommentare 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.

Für *jede* Methode, also auch für sehr einfache Methoden, wie beispielsweise:

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

muss ein Kopfkommentar (siehe Kapitel 5.5) vorliegen.

Als Faustregel ist eine Methode mit mehr als 20 Zeilen als unterteilungs- und kommentierungsbedürftig anzusehen. Der Implementierer sollte beim Kommentieren denken müssen, nicht der Leser beim Durchsehen!

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. Gibt es mehrere denkbare Lösungsalternativen, um Code zugänglicher zu machen, sollte man sich für einen Ansatz entscheiden und diesen auch in Zukunft anwenden. Auch bei der Kommentierung gilt es, konsistent vorzugehen.

Javadoc

Alle Kommentare in diesen Entwicklungsrichtlinien sind zwar in Bezug auf Einrückung der Kommentare überarbeitet worden, die Konvertierung in **Javadoc** wurde jedoch unterlassen. Jedermann *kann* und *darf* **Javadoc**-Konventionen bei der Kommentierung anwenden, sollte aber bestrebt sein, auch hier wieder einheitlich vorzugehen.

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. Kommentare in der selben Zeile hinter Programmanweisungen dürfen nur zur Erklärung dieser einzelnen Anweisungen verwendet werden.

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.

5.1.1 Sprache für Kommentare

(siehe hierfür Kapitel 4.2, »Sprache für Bezeichner und Kommentare« und Kapitel 1.2, »Verbindlichkeit der Richtlinien«)

5.2 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 folgenden Vorbild geben (siehe Beispiel 16), welche auf dem aktuellen Stand zu halten ist.

BEISPIEL 16. Dokumentation bzw. Kommentierung von Paketen.

```
…
VERSION: <Versionsnummer> vom <TT.MM.JJJJ>

ZWECK
<Erklärung des Einsatzgebietes, der Struktur und der Aufgabe
des Pakets>

KLASSEN
<Liste der im Paket enthaltenen Klassen>

INTERFACES
<Liste der deklarierten Interfaces>

EXCEPTIONS
<Liste der deklarierten Exceptions>

ERRORS
<Liste der deklarierten Errors>
…
PAKET:    <name.des.pakets>
```

Der Name der Paket-Dokumentations-Datei beginnt genau wie das zugehörige Paket. Eine Paketdokumentation sollte entweder in Form eines ASCII-Textes oder als HTML vorliegen.

5.3 Kommentieren von Klassen

Jede Klasse *muss* einen Klassenkommentar nach folgendem Vorbild (siehe Beispiel 17 auf Seite 40) haben, welcher auf dem aktuellen Stand zu halten ist.

BEISPIEL 17. Kommentierung von Klassen, Klassenkommentar

```
…
/*-------------------------------------------------------------
AUTOR:          <Ausgeschriebener Name>
PROJEKT:        <Projektname>
Java:           Sun's Java Development Kit, Version 1.0.2
Copyright:      Forschungsgruppe Requirements Engineering,
                Institut fuer Informatik, Universitaet Zuerich
KLASSE:         <Name der Klasse>
VERSION:        <Versionsnummer>[1] von <Name>[2] am <TT.MM.JJJJ>
-------------------------------------------------------------

ZWECK
<Erklärung des Einsatzgebietes und der Aufgabe der Klasse>

VERANTWORTLICHKEITEN
<Erklärung der Verantwortlichkeiten/Responsibilities/
Entwurfsgeheimnisse etc. der Klasse> */

public class Foo   // hier die eigentliche Klassendeklaration
     {
     /* --- KLASSEN-VARIABLEN --- */
     <Typ> <Name>  /* <Erklärung des Verwendungszwecks … */
     /* Die <Erklärung> kann auch in die Zeile darüber */
     <Typ> <Name>

     …

     /* --- INSTANZ-VARIABLEN --- */
     <Typ> <name>       /* <Erklärung des Verwendungszwecks
                        für die Instanzvariable> */
     <Typ> <name>       // <Erklärung für die Instanzvariable>
     /* <Erklärung für die Instanzvariable> */
     <Typ> <name>
```

[1] Beispiele für Versionsnummern: 1,2,3 oder 1.1, 3.2 etc.

[2] Name oder Namenskürzel der Person, welche die Klasse geändert hat.

...

Verantwortlichkeiten

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. verkapselt sie möglichst viel Wissen darüber, wie dieses Dienstleistungangebot klassenintern realisiert ist. Bei den Verantwortlichkeiten soll 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 verkapselt.

5.4 Kommentieren von Interfaces

Analog der Vorgehensweise zur Dokumentation von Klassen ist die Kommentierung von Interfaces aufgebaut. Zusätzlich sollte aber das Anwendungsgebiet bzw. die intendierte Verwendung des Interfaces genau erläutert werden.

5.5 Kommentieren von Methoden

Kopfkommentare sind obligatorisch. Für *jede* Methode (siehe auch Kapitel 5.1) *muss* ein Kopfkommentar vorliegen.

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

BEISPIEL 18. Der Kopfkommentar einer Methode

```
/* ...
PRE  -

POST
Gibt zurück, ob der Empfänger die Eigenschaften
einer Äquivalenzrelation hat.

Erste Version von Kurt Schneider, 14.03.1993
Letzte Änderung von Stefan Berner, 10.07.1996 */
boolean isEquivalenceRelation()
     {    /* Es wird getestet, ob reflexiv, transitiv
          und symmetrisch. */
```

```
        ...
    }
```

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 Kopfkommentar gliedert sich grob in vier Teile. Diese sind:

- Vorbedingung `PRE` (siehe Kapitel 5.5.2)
- Nachbedingung `POST` (siehe Kapitel 5.5.1)
- Verpflichtungen `OBLIGATION` (siehe Kapitel 5.5.3)
- Versions- und Änderungsinformation (nicht weiter behandelt)

Entsprechend der Häufigkeit ihres Vorkommens wird in den nachfolgenden Kaptiteln auf Nachbedingung, Vorbedingung und Verpflichtungen eingegangen.

5.5.1 Angabe der Nachbedingung im Kopfkommentar

Im Allgemeinen wird die Nachbedingung dokumentiert durch:

- Eine kurze Beschreibung dessen, was die Methode tut, d.h. worin ihr Dienst besteht (siehe Beispiel 18, 19, 20 oder 21) und ggf. welche Ausnahmebehandlung sie selbst durchführt (siehe Beispiel 21).
- Ferner sollten die Aufgaben der Parameter grob skizziert sein (siehe Beispiel 19 und Beispiel 21). Wird ein Parametername erwähnt, so steht er zur Hervorhebung im Kopfkommentar in spitzen Klammern. In anderen Kommentaren sind die spitzen Klammern freigestellt.
- Die Nachbedingung darf ausdrücklich deskriptiv oder prozedural formuliert sein. Wenn sie prozedural angegeben wird, sollte darauf geachtet werden, dass sie möglichst lösungsneutral ist. Wenn mit jeder Änderung des Methodenrumpfes auch die Nachbedingung geändert werden muss, dann deutet dies u.a. darauf hin, dass diese nicht genügend lösungsneutral ist.
- Handelt es sich um eine abstrakte Methode, so ist insbesondere zu kommentieren, wie und wozu diese in einer Unterklasse zu implementieren ist.

```
/* ...
PRE  -
POST
```

BEISPIEL 19. Der Kopfkommentar einer Methode

```
Addiert <aNumber> zum Empfänger und gibt
das Ergebnis zurück.

Erste Version von Kurt Schneider, 13.03.1993
Letzte Änderung von Stefan Berner, 10.07.1996*/
real add( real aNumber)
     {…
     }
```

5.5.2 Angabe der Vorbedingung im Kopfkommentar

Bei den bisher vorgestellten Kopfkommentaren (siehe Beispiel 18 oder 19) wird im wesentlichen die Nachbedingung der Methode angegeben, da keine zu dokumentierende Vorbedingung existiert. Es wird aber in jedem Fall dokumentiert, dass keine Vorbedingung existiert (durch `PRE -`). Existieren eine oder mehrere Vorbedingungen, so *müssen* diese im Kopfkommentar auch mit angegeben werden (siehe Beispiel 20).

- 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. Beispiele für solche Vorbedingungen sind: "der Parameter `aNumber` muss vom Typ `real` sein (siehe Beispiel 19)" oder "ein Objekt des Typs Stack existiert (siehe Beispiel 20)".

BEISPIEL 20. Angabe der Vor- und Nachbedingung im Kopfkommentar

```
/* …
PRE
Der Stack darf nicht leer sein (ggf. zuvor mittels
isEmpty() prüfen)

POST
Nimmt das oberste Element vom Stack und gibt
dieses zurück

Erste Version von Stefan Joos, 17.07.96
Letzte Änderung von Stefan Berner, 17.07.96*/
Object pop();
     …
     {…
     }
```

5.5.3 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.

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

```
/* …
PRE
-

POST
Erzeugt/Öffnet die Datei <fileName> mit wahlfreiem
Zugriff zum Lesen oder zum Lesen und Schreiben.

Das Argument <accessMode> gibt die Art des gewünschten
Zugriffs an und muss entweder:
     "r" für schreibenden Zugriff oder
     "rw" für schreibenden und lesenden Zugriff sein.

Ausnahmebehandlung
     Bei Ein-/Ausgabefehler:
          throws IOException
     Bei fehlerhaftem Argument <accessMode>
          throws IllegalArgumentException
     …

OBLIGATION
Der Aufrufer muss dafür Sorge tragen, dass die Datei auch
wieder geschlossen wird.

Erste Version von Martin Glinz, 23.07.96
Letzte Änderung von Stefan Berner, 29.07.96*/
public RandomAccessFile(String fileName, String acessMode)

     throws IOException
     …
     {
     …
     }
```

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 Änderungshistorie

Version 1

SB, 08.Jul.1996
– ungeprüft –

Erste Version des Java-Styleguides. Diese Version des Styleguides für Java-Entwicklungen ist eine Anpassung sowie Auf- und Überarbeitung des Styleguides für Smalltalk-Entwicklungen der Abteilung Software Engineering der Universität Stuttgart (Schneider 1994).

Version 1.1

SB, 15.Jul.1996
– ungeprüft –

- Einige Änderungs- und Verbesserungsvorschläge von Martin Arnold und Stefan Joos eingearbeitet.
- Gliederung geändert.

Version 1.2

SB, 29.Jul.1996
geprüft von MA, MG, SJ, MR am 23.Juli 1996

Befunde des Reviews korrigiert:

- Übersicht hinzugefügt
- Gliederung geändert
- Kapitel 2 erweitert
- Änderung/Erweiterung von Systemklassen nun verboten
- Paketdokumentation nun obligatorisch
- Bedinungs- und Schleifenrümpfe müssen immer "geblockt" werden
- Richtlinien für die Bezeichnerwahl ergänzt
- Vorbedingung/Nachbedingung im Methodenkommentar nun oblingatorsich
- … weitere kleinere Fehlerkorrekturen …

Version 1.3

SB, 09.Okt.1996; SB, 15.Okt.1996

- teilweiser Abgleich mit den Richtlinien von Doug Lea
- einige "typos" korrigiert

Version 1.4

SB, 12.Dez.1996

- Korrekturen und Verbesserungsvorschläge von Martin Glinz eingearbeitet

- ➢ Ausführungen zur Dokumentation von Nachbedingungen ergänzt
- ➢ "typos" korrigiert

Version 1.4.1 SB, 16.Dez.96

- ➢ 1 Tippfehler korrigiert

Version 1.4.2 SB, 29.Mar.1997

- ➢ einige Änderungs- und Korrekturvorschläge von Markus Pilz eingearbeitet
- ➢ weitere "typos" korrigiert

NS, 20. Feb.98 Version 2.0
vollzogene Änderungen geprüft von MG, SJ, JR am 19./ 23.Jan.1998

Abgleich mit anderen Richtlinien siehe Anhang "Literatur" mit folgenden Erweiterungen, Korrekturen:

- ➢ Einbezug Kapitel: Codierregelungen
  Aufnahme von Codierregelungen, die gewonnene Erkenntnisse aus Erfahrungen mit der Anwendung von potentiell fehlerträchtigen Anweisungen weitergeben
- ➢ Einbezug neuer Subkapitel mit u.a. folgenden thematischen Schwerpunkten
  Synchronisation, Exceptions, Interfaces
- ➢ Einflechtung von vor allem ***Sun*** Standards:
  Aufnahme der Präfixe **get / set** für die Bezeichner von Methoden; diese sind nun erlaubt und nebst den bestehenden Richtlinien aufgeführt
  Bezeichner für lokale Variablen (**i**, **g**, **e**,....)
  Javadoc-Konventionen neben bewährter Kommentierung erlaubt
- ➢ Abgleichen der Kommentierung
  Festlegen der Deklarationsreihenfolge der einzelnen Klassenelemente
  Ändern des Einrückens und der Positionierung speziell bei Kopf-Kommentaren
  Java-Syntax für Dokumentationskommentare (**Javadoc**) ist nebst der bestehenden Kommentierungsart gültig. Die vorliegenden Entwicklungsrichtlinien wurden **nicht** auf Javadoc-Konventionen überarbeitet.

Anhang C Glossar

API-Klasse: Eine Klasse des Application Programming Interfaces (API). Das API ist die standardisierte Klassenbibliothek, die auf allen Java-Plattformen verfügbar ist und dank einheitlich definierter Schnittstellen auch gleich angesprochen wird.

Class-Datei: In Dateien mit der Endung ".class" ist übersetzter Java-Quellcode abgelegt. Der Begriff Class-Datei steht hier für eine beliebige Datei mit der Endung ".class", in welcher Übersetzter Java-Quellcode abgelegt ist.

Java-Datei: Dateien mit der Endung ".java" enthalten i.d.R. Java-Quellcode. Der Begriff Java-Datei steht hier für eine beliebige Datei mit der Endung ".java", in welcher sich Java-Quellcode befindet.

Javadoc: ***Der Java-Dokumentationsgenerator***
Javadoc generiert die API-Dokumentation für das in der Kommandozeile angegebene Paket bzw. für die übergebenen Java-Quelldateien. Die Ausgabe der Dokumentation erfolgt im HTML-Format. Die von javadoc generierten Klassendokumentationsdateien beschreiben die Klasse (oder das Interface) sowie ihre Vererbungshierarchie. Ausserdem wird jedes public- und protected-Element der Klasse indexiert und beschrieben. Die generierte Datei enthält auch alle **Doc-Kommentare**, die mit der Klasse und ihren Methoden, Konstruktoren und Variablen verknüpft sind. Ein **Doc-Kommentar** bzw. Dokumentationskommentar ist ein Java-Kommentar, der mit `/**` beginnt und mit `*/` endet. Ein solcher Doc-Kommentar kann **HTML-Tags** enthalten.

Friend: Die Sichtbarkeit eines Bezeichners, wenn dieser nicht **`private`**, **`protected`** oder **`public`** deklariert wurde, d.h. keines dieser Schlüsselworte vorangestellt wurde. Der Bezeichner ist dann ausschliesslich in dem Packet sichtbar, in dem er deklariert wurde.

ReadMe-Datei: Dateien mit dem Suffix ".readMe" beinhalten wichtige Zusatzinformationen. ReadMe Dateien beginnen mit dem Gleichen Präfix wie die Datei, für welche sie Zusatzinformationen enthalten.

Symbolische Konstante: Eine symbolische Konstante ist eine Konstante, für welche ein symbolischer Bezeichner deklariert wurde, so dass der eigentliche Wert der Konstante nicht direkt im Code zu finden ist; z.B.:

```
/* mit symbolischer
Konstante */
…
final real TAX = 0.15;

real tax( real amount)
     /* … */
     {
     return (amount * TAX);
     }
```

```
/* ohne symbolische */
Konstante */

…

real tax(real amount)
     /* … */
     {
     return (amount * 0.15);
     }
```

Anhang D Index

Symbole

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

S

T

U

V

W

Z