RICHTLINIEN JAVA - INFORMATIK 1

Einführung

Dieses Dokument enthält die meisten in der Vorlesung Informatik 1 an unterschiedlichen Orten angegebenen Richtlinien in folgender übersichtlicher Form.

Regel: Beschreibung der Regel
Regel: Beschreibung einer dazugehörigen Regel (optional)
Begründung: Kurze Begründung

Optionale Beispiele
Optionale ausführliche Begründung

Es handelt sich dabei mehrheitlich um Programmierrichtlinien aber auch um einige Entwurfsrichtlinien. Die Regeln sind so gruppiert, dass Regeln, welche die gleichen Programmiersprachelemente betreffen meist beieinander stehen.

Die hier aufgeführten Regeln sind im wesentlichen übereinstimmend mit den von Sun herausgegebenen Java Code Conventions [1]. Sie entsprechend auch meiner Erfahrung nach dem, woran sich die meisten Java-Programmierer in der Praxis halten – zumindest sofern überhaupt Programmierregeln und ein Stil in Quelltexten erkennbar sind. Beachte, dass für weiterführende Java Technologien wie Java Server Pages, Servlets und Enterprise Java Beans noch zusätzliche Richtlinien existieren (siehe dazu java.sun.com).

Viele sind auch gar nicht Java-spezifisch oder betreffen nur objekt-orientierte Programmierung sondern gelten für alle Programmiersprachen. Manche der im Folgenden aufgeführten Regeln sind nicht gleich wichtig – auf eine entsprechende Kennzeichnung habe ich verzichtet. Allerdings ist in der Reihenfolge der Regeln innerhalb eines Abschnitts oft eine entsprechende Priorisierung zu erkennen: die wichtigste Regel zuerst, weniger wichtige am Ende.

Normalerweise sollten Richtlinien sehr kurz sein: etwas 2-3 Seiten. Sind sie länger, dann liest sie keiner und die Regeln werden dann überhaupt nicht angewendet. Diese Kürze ist später in den Programmierrichtlinien einer Firma auch bedenkenlos machbar: die Regeln brauchen dort gar nicht begründet und viele auch nicht mehr erwähnt zu werden, da sie bis zum Ende der Informatikausbildung teilweise in Fleisch und Blut übergehen (sollen). Da diese Richtlinien aber für die Ausbildung von Informatikstudenten im ersten Semester dienen, enthält dieses Dokument entsprechend viele Regeln und detaillierte Begründungen. In der Projektpraxis empfiehlt es sich diese Regeln mit entsprechenden Funktionen der verwendeten Entwicklungsumgebung durchzusetzen (Warnung bei Nichteinhaltung, automatische Formatierung).

Die Regeln zusammengenommen beschreiben letztendlich einen definierten Programmierstil. Da in der Praxis oft bestehende Quelltexte gewartet werden müssen, ist die folgende Regel immer vor allen anderen zu berücksichtigen:

Regel: Mische in einer Klasse nie verschiedene Programmierstile.
Regel: Bei bestehenden Quelltexten gilt insbesondere, dass der bestehende Programmierstil übernommen oder überall in der Klasse geändert wird.
Begründung: Quelltexte werden dadurch für alle lesbarer.

Beispiel:

```
if (a > 0)
  {
  a = a + 1;
  // Klammer sind immer auf neue Zeile
  // und mit den Anweisungen eingerückt
  }
```

Dieser Klammerungsstil mit Einrückung der geschweiften Klammern ist eher selten anzutreffen. Falls eine derartige Klasse im Quelltexte geändert oder ergänzt wird, sollte man die Klammer in dieser Klasse immer in der gleichen Art und Weise setzen.

Werden in einer Klasse verschiedene Stile verwendet, so ist der resultierende Quelltext schwerer verständlich. Man stelle sich einen deutschen Text vor, In Dem Die gross- Und Kleinschreibung bunt durcheinander gewürfelt wird. OderaufLeerzeichen zwischen Wörternmalverzichtetwird und dAnn wIeder mAl nIcht. Der Leser muss sich dadurch andauernd mühsam umstellen.

Diese Regel gilt natürlich nur für die Codeteile, in der ein Stil erkennbar ist: nur weil zum Beispiel mal Leerzeichen vor und nach den Operatoren gesetzt werden und dann mal wieder nicht – es also keinen erkennbaren Programmierstil gibt – heißt das natürlich nicht, dass man das genauso machen soll. In diesem Fall sollte man seinen eigenen Stil für Operatoren folgen und besser noch den Rest des Quelltextes entsprechend verbessern, auch wenn dieser von jemand anderem implementiert wurde.

Namenskonventionen

Bezeichner

Regel: Verwende für Bezeichner immer aussagekräftige und selbsterklärende Namen.
Regel: Verwende keine Abkürzungen in Bezeichner – schreiben diese Namen immer aus.
Begründung: Quelltexte werden dadurch für alle lesbarer.

Beispiel: Statt KFZ, BA, HS, DB immer:
`Kraftfahrzeug, BundesagenturFuerArbeit, HochschuleKarlsruhe, DeutscheBahn`

Da Quelltexte von vielen unterschiedlichen Programmierern (oft aus unterschiedlichen Kulturbereichen) gelesen und geändert werden, darf man nie davon ausgehen, dass die verwendeten Abkürzungen jedem der beteiligten Personen geläufig sind: BA könnte ja auch Berufsakademie bedeuten oder Bundesangestellter.

Regel: Vermische keine verschiedenen Sprachen (Deutsch, Englisch, …) in einer Klasse
Begründung: Quelltexte werden dadurch lesbarer.

Diese Regel gilt nicht nur für die Wahl der Sprache der Bezeichner, sondern auch für die Kommentare: Die Bezeichner und Kommentare sollten pro Klasse in genau einer Sprache geschrieben sein. Dadurch muss der Leser nicht immer wieder zwischen den Sprachen wechseln und als Programmierer muss man sich nicht die Mühe machen die Wörter zu übersetzten. Insbesondere letzteres ist bei speziellen Begriffen aus einer Branche schwierig und meist einfach falsch: was heißt zum Beispiel Leistungsreglement in Englisch? oder Warenrechnung?. Normalerweise ist es eine gute Wahl die Sprache zu nehmen, die der Kunde spricht und in der die Anforderungen geschrieben sind. Bei Software, die von Personen aus verschiedenen Ländern programmiert wird, ist allerdings oft Englisch die lingua franca. Ausnahmen von dieser Sprachregel sind die englischen Präfixe für Getter- und Settermethoden (get, set, is) und auch add, remove zum Hinzufügen von Elementen.

Regel: Verwende ausschließlich die alphanumerischen Zeichen A-Z, a-z, 0-9 und den Unterstrich _ für Bezeichner.
Begründung: Damit die Quelltexte international portabel werden.

Beispiel: `Person, zugFahren, MAXIMALE_ANZAHL, ueberpruefen`

In Java sind zwar alle Unicodealphabete für die Wahl von Bezeichnern zugelassen, aber nicht alle Texteditoren unterstützen jedes Unicode-Alphabet und nicht jeder Java-Quelltext ist in einem Unicodeformat geschrieben. Lediglich der durch den Compiler erzeugte Bytecode ist immer portabel. Des Weiteren werden Paketnamen und Klassennamen mit Dateiordner und Dateien identifiziert: je nach Betriebssystem sind internationale Sonderzeichen dort gar nicht erlaubt. Darüber hinaus gibt es auch kuriose Effekte, wenn z.B. ein Bezeichner mit arabischen Buchstaben geschrieben wird: Arabisch wird von rechts nach links gelesen, die Java Schlüsselwörter sind aber englisch und werden von rechts nach links gelesen. Viele Editoren berücksichtigen dies und wenn man kontinuierlich den Cursor mit der rechten Pfeiltaste bewegt, dann wandert der Cursor bei den englischen Schlüsselwörtern wie erwartet nach rechts, beim Beginn eines arabischen Wortes springt er dann – eigentlich auch wie zu erwarten – an den Anfang des arabischen Wortes. Dies ist aber dann ein Sprung nach Rechts über das ganze Wort. Danach wandert der Cursor bei jedem Druck der rechten Taste nach

links bis zum Ende des arabischen Wortes. Versuchen Sie es einfach mal mit dem folgenden Java Bezeichner in Eclipse: رمضان (der Fastenmonat im Islam).[1]

Wie das bei Sprachen aussieht, bei denen die Seiten von hinten nach vorne geblättert werden oder die Sätze von unten nach oben geschrieben werden, mag sich jeder selbst ausdenken.

Regel (Upper case camel style): Bei Bezeichner, die aus mehreren Teilwörtern bestehen, wird der erste Buchstabe jedes Teilworts (ab dem zweiten Teilwort) groß geschrieben, wenn die entsprechenden Wörter normal auseinander geschrieben würden.
Begründung: Damit lange Bezeichner lesbarer sind.

Beispiel: `zugFahren, Donaudampfschifffahrt, HochschuleKarlsruhe`

Diese Regel ist analog zur Grammatik natürlicher Sprachen, die vorschreibt die Wörter eines Satzes mit einem Leerzeichen zu trenne, anstattsiedirekthintereinanderzuschreibenumplatzzusparenoderauswelchengründenauchimmer.

Im Deutschen kann man im Gegensatz zu vielen anderen Sprachen – wie den romanischen – beliebig lange Substantive bilden (Donaudampfschifffahrtsgesellschaft). Spätestens bei drei Substantiven empfiehlt der Duden einen Bindestrich zu verwenden: Donau-Dampf-Schifffahrts-Gesellschaft würde dann also zu folgendem Java Bezeichner: `DonauDampfSchifffahrtsGesellschaft`. Das Wort Schifffahrtsgesellschaft wird aber zu `Schifffahrtsgesellschaft`. Diese Unterscheidung nach Duden für Javabezeichner anzuwenden ist allerdings nicht immer so genau zu nehmen.

Klassen

Regel: Verwende (mindestens) die Einzahl eines Substantiv für den Namen einer Klasse
Begründung: Damit Quelltexte lesbarer werden.

Beispiel: `Person, Adresse, Student, AdressenUeberpruefer`

Klassen beschreiben Mengen von Objekten, die Abstraktionen realer Objekte sind. In der Realität sind sie immer durch ein Substantiv selbsterklärend beschrieben. Es sollte immer die Einzahl verwendet werden, so dass suggestiv die Klasse als „Stellvertreter" eines Objekts dieser Klasse herhalten kann.

Variablen, Attribute und Parameter

Die meisten der folgenden Regeln gelten auch für die Bezeichnung von Attribute und Parameter. Es ist allgemein aber immer von Variablen die Rede.

Regel: Schreibe Variablen klein (Ausnahme sind Konstanten). Besteht eine Variable aus mehren Teilwörtern, so wird bei jedem folgenden Teilwort der erste Buchstabe groß geschrieben.
Begründung: Damit Quelltexte lesbarer werden.

Beispiel: `manfredMueller, anzahlPersonen, quersumme`

Der erste Buchstabe einer Variablen wird immer klein geschrieben, um sofort Variablen von Klassen unterscheiden zu können: analog zur Gross-/Kleinschreibung von Substantiven und den anderen Wortarten im Deutschen.

[1] In Word springt der Cursor übrigens nicht an den Anfang des arabischen Wortes. Selbst wenn ich für das Wort die Sprache Arabisch festlege.

Regel: Konstanten (static final) werden immer mit Grossbuchstaben bezeichnet und Teilwörter mit dem Unterstrich _ abgetrennt.
Begründung: Damit Quelltexte lesbarer werden.

Beispiel: `ERD_GRAVITATION, LICHTGESCHWINDIGKEIT`

Symbolische Konstanten sind dadurch sofort im Quelltext als solche zu erkennen. Diese Regel gilt aber nur für primitive Datentypen oder Klassen, deren Objekte *unveränderlich* sind (z.B. String). Da bei Java Klassen immer Referenztypen sind, wird ist nur die Referenz konstant nicht aber der Wert (das Objekt). Bei veränderlichen Objekten sollte man besser nicht final verwenden und dann auch nicht die Großschreibung: der Leser nimmt sonst fälschlicherweise an, dass auch das Objekt und nicht nur dessen Referenz unveränderlich ist.

Regel: Verwende (mindestens ein) Substantiv für Variablen.
Begründung: Damit Quelltexte lesbarer werden.

Beispiel: `name, alter, studentImErstenStudiensemster, MAXIMALE_ANZAHL`

Variablen sind Platzhalter für Daten. Diese Daten haben einen Bezug zu Objekten oder Objekteigenschaften in der Wirklichkeit: die Bedeutung dieser Daten. Diese ist immer sehr gut durch ein Substantiv beschreibbar, welche die *Bedeutung* der Daten in der Realität ausdrücken. Wenn die Variable wie bei einem Feld mehrer Werte des Typs enthalten kann, so sollte man die Mehrzahl verwenden, ansonsten die Einzahl.

```
Person student;
Person [] studenten;
```

Regel: Verwende die Buchstaben i, j, k, l für rein technische ganzzahlige Schleifenvariablen.
Begründung: Aus historischen Gründe.

Beispiel:

```
for (int i = 0; i < personen.length; i++) {
   …
}
```

Für die Aufzählung aller Elemente eines Feldes oder einer Datenstruktur des Java Collection Frameworks, sollte aber möglichst die ab JDK 5.0 zusätzliche for-Schleife verwendet werden. Die technischen Schleifenvariablen werden dann ganz vermieden:

```
for (Person person : personen) {
  …
}
```

In den frühen Programmiersprachen wie COBOL, FORTRAN und deren Abkömmlinge wie ABAP oder BASIC gab es außer elementaren Datentypen für die Organisation komplexerer Daten eigentlich nur Felder oder etwas Analoges. Strukturen wie ein Record, ein struct oder eine Klasse entstanden erst in ALGOL und dessen Abkömmlingen. Integers als Indizes waren deswegen sehr häufig in Quelltexten anzutreffen und hatten oft wenig Bezug zu realen Daten. Suggestiv wurde dann meist i(nteger) als Name für die erste Indexvariable verwendet. Bei Verschachtlung von Schleifen hat man einfach den nächsten Buchstaben im Alphabet verwendet.

Regel: Um besser zwischen Objektattribute und lokalen Variablen zu unterscheiden, sollten

Objektattribute mit this referenziert werden.
Begründung: Die Quelltexte werden lesbarer und Fehler werden vermieden.

Beispiele: `this.name = name; // wobei name ein Parameter ist`

Durch diese Regel kann auch oft vermieden werden, sich unnötigerweise für einen Parameter, der das gleiche bedeutet wie ein Attribut, einen neuen Namen auszudenken. Manchmal findet man in Quelltexten eine an die Ungarische Notation angelegte Variante, in der die Präfixe s_ und m_ oder einfach s und m jeweils statische- und member-Variablen (Objektattribute) und -Methoden unterscheiden. Dies sollte vermieden werden, da auch ohne diese nicht jedem geläufigen Abkürzungen derselbe Nutzen mit this bei Objektattributen und vorangestelltem Klassennamen bei Klassenvariablen erreicht wird.

Durch diese Konvention können auch Fehler vermieden werden, die dadurch entstehen, dass eine lokale Variable oder ein Parameter ein Attribut gleichen Namens unbeabsichtigt verdeckt.

Regel: Verwende keine Ungarische Notation.
Begründung: Damit Quelltexte lesbarer und wartbarer werden.

Bei der Ungarischen Notation werden zwei Präfixe dem eigentlichen Bezeichner vorangestellt:

- Der erste für den Verwendungszweck des Bezeichners,
- der zweite für den Typ.

Zum Beispiel p für Pointer, i für Integer: piAnzahl ist ein Zeiger auf einen Integerwert.

Die Begründung für diese Regel scheint widersprüchlich, da die Ungarische Notation ja der Lesbarkeit dienen soll. Dies gilt allerdings eher für nicht typisierte Sprachen. Es gibt allerdings auch verschiedene Varianten der Ungarischen Notation, so dass die Abkürzungen nicht jedem geläufig sind. Viele der Präfixe für den Verwendungszweck, wie p für Pointer, machen in Java auch gar keinen Sinn. Sehr viele Programmierer verwenden auch gar nicht diese von Charles Simonyis entwickelte Notation, sondern eine Vereinfachung und Abwandlung aus der Windowswelt (Visual Basic, Excel). Bei dieser Variante ist der Präfix nur eine Codierung des Typs und nicht des Verwendungszwecks. In Java – aufgrund sehr eingeschränkter Überladung von Operatoren und sehr strengen Typisierung – ist der Typ der Variablen in den meisten Fällen ohnehin aus dem Kontext ersichtlich. Ansonsten bieten alle modernen Entwicklungsumgebungen die direkte Anzeige des Typs und der zugehörigen Javadoc, so dass die Ungarische Notation der Lesbarkeit bei Javaprogrammen eher schadet als nutzt. Neben der Lesbarkeit leidet aber auch die Wartbarkeit der Quelltexte bei Verwendung der Ungarischen Notation in typisierten Sprachen: Da Typinformationen an zwei Orten – sowohl in der Deklaration als auch im Bezeichnernamen – enthalten sind, werden diese Informationen bei häufigen Änderungen des Quelltextes erfahrungsgemäß inkonsistent: wird der Datentyp von double nach int geändert und im Präfix diese Änderung nicht nachgeführt, so enthält der Quelltext nun irreführende Informationen.

Methoden

Regel: Verwende mindestens ein Verb in Präsensform für eine Methode. Das Verb soll möglichst genau beschreiben, was die Methode macht.
Begründung: Damit Quelltexte verständlicher werden.

Beispiel: `personSuchen(), loeschen(), produktVerkaufen()`

Methoden definieren Funktionen eines Programms, in denen etwas getan wird. Diese Aktionen lassen sich sprachlich am besten mit Verben beschreiben.

Es gibt noch folgende Spezialisierung dieser Regel, bei der auch eine Ausnahme gemacht wird, das z.B. Deutsch und Englisch nicht in einer Klasse vermischt werden soll.

Regel: Verwende als Verb das englische Präfix get mit einem nachfolgenden Substantiv, wenn eine Methode (ohne Parameter) einen Wert, der durch das Substantiv beschrieben ist, zurückgibt. Bei getter-Methoden mit booleschen Rückgabewert, wird statt get meist is verwendet.
Regel: Verwende als Verb das englische Präfix set mit einem nachfolgenden Substantiv, wenn eine Methode ohne Rückgabewert und mit einem Parameter einen Zustandswert (insbesondere ein Attribut) eines Objekts ändert.
Begründung: Zur Wahrung des Geheimnisprinzips bei der Programmierung.

Beispiel: `String getVorname(), void setVorname(String vorname) , boolean isSchaltjahr(), void setSchaltjahr(boolean schaltjahr)`

Objekt-Attribute sollten nie öffentlich sein, da die resultierenden Programme sonst zu viele direkte Abhängigkeiten haben und damit sehr schnell zu komplex werden sowie Objekt-orientierte Merkmale wie Überschreiben von Methoden nur schwer oder gar nicht anwendbar sind. Stattdessen dürfen Objekt-Attribute immer nur über Methoden geändert werden, um die Komplexität der Programme zu reduzieren. Und diese Regel wird auch gleich wieder eingeschränkt, da das mit dem Geheimnisprinzip sich nicht einfach durch derartige, mechanisch erzeugbare Zugriffsmethoden wahren lässt, sondern oft das Gegenteil erreicht wird. Diese „Java-Beans“ Namenskonventionen sind im Detail noch etwas weitreichender.

Oft wird auch noch add und remove als Präfix für den Zugriff auf 1-n Beziehungen hinzugenommen.

Static

Regel: Referenziere statische Methoden oder Variablen in der Klasse, in der sie definiert sind, immer mit vorangestelltem Klassennamen.
Begründung: Damit Quelltexte lesbarer werden.

Beispiel: `Math.abs(-16), Color.red, System.out`

Dadurch sind statische Variablen und -Methoden immer klar von lokalen Variablen sowie Objekt-attributen oder –Methoden unterschieden.

Packages

Regel: Verwende ausschließlich Kleinbuchstaben und Zahlen sowie Substantive für Paketnamen. Geläufige Abkürzungen sind erlaubt, um die Paketnamen kurz zu halten.
Regel: Um Pakete international eindeutig zu halten, sollte der Name mit dem umgekehrten Domainnamen der Organisation begonnen werden, dem dieser Quelltext zugehörig ist. Als weitere Abstufung können interne Abteilungsnamen und Produkt/Projektname verwendet werden.
Begründung: Portabilität der Javaprogramme.

Beispiele: `de.sap.r3.internetshop, de.hska.info1`

Paketnamen werden immer mit Verzeichnissen identifiziert. Je nach Betriebssystem sind die dazu verfügbaren Zeichen beschränkt. Ebenso unterscheiden Betriebssysteme wie Unix Groß- und Kleinbuchstaben. Die konsequente Kleinschreibung dient der Portabilität zwischen verschiedenen Betriebsystemen. Wird beispielsweise unter Windows entwickelt, dann wird bei `import meinPaket.A` die Klasse `A` im Verzeichnis `meinpaket` gefunden. Aber nicht mehr unter Unix,

den dort wird sie in `meinPaket` gesucht. Dieser Order existiert aber gar nicht, wenn der übersetze Bytecode auf das Unixsystem übertragen wird.

Der Domainname ist international registriert und damit eindeutig. Somit ist mit dieser Regel auch jedes Paket und deren Klassen einer Firma zuzuordnen. Bei zugekaufter Javasoftare entstehen so keine unauflösbare Konflikte bei Namensgleichheiten.

Regel: Die Paketnamen java und javax sind ausschließlich für offizielle Java-Erweiterungen erlaubt.
Begründung: Sicherheit der Javaprogramme.

Neben der Portabilität steht hier vor allem im Vordergrund, dass niemand elementare Javastandards umgehen kann: zum Beispiel durch Angabe einer eigenen String Klasse java.lang.String oder gar java.lang.Object. Die darauf basierende Software würde zwangsläufig ein Desaster werden, da diese Implementierungen dann auch von den Klassen der Laufzeitumgebung verwendet werden. Wegen derartigen Sicherheitsgründen sind auch viele vorhandene Klassen wie String als final deklariert.

Quelltextformatierung

Allgemein

Regel: Schreibe nie Zeilen mit mehr als 80 Zeichen.
Regel: Stelle den Editor so ein, dass er diese Grenze als vertikale Linie klar darstellt.
Begründung: Damit Quelltexte lesbarer werden.

Die 80 Zeichengrenze hat historischen Gründe: frühe Terminals hatten max. 80 Zeichen pro Zeile. Heutzutage ist die Grenze je nach Bildschirmauflösung und Font etwas größer. Grundsätzlich sollte man aber nicht von einer zu hohen Anzahl Zeichen pro Zeile ausgehen. Überlange Zeilen sind für Menschen schlecht lesbar, da nur eine begrenzte Anzahl von „Wörtern“ (ca. 5-7) mühelos erfasst werden können ohne dass man das Gefühl bekommt angestrengt zu lesen. Des Weiteren ist das horizontale Scrollen bei überlangen Zeilen im Fenster im Gegensatz zum vertikalen mühsam, da bei Sprung auf die nächste Zeile, insgesamt der Quelltext sich sprunghaft nach links verschiebt (zu starker Kontextwechsel).

Anweisungen

Regel: Verwende bei Kontrollstrukturen wie if, else, while, … immer geschweifte Klammer für die Anweisungen (auch bei Einzelanweisungen)
Regel: Rücke jede Anweisung um 2 bis 4 Zeichen nach rechts ein (immer konsistent die gleiche Anzahl von Zeichen und nicht mal 2, mal 3 und dann mal wieder 4 Zeichen)
Regel: Stelle den Editor so ein, dass bei Drücken der Tabulatortaste, immer Leerzeichen eingefügt werden. Ansonsten geht aufgrund anderer Tabulatoreinstellungen bei einem anderen Editor die Struktur meist verloren.
Begründung: Bessere Lesbarkeit von Programmen und vermeiden von Programmierfehlern.

Beispiel:

```
if (student.hatZulassung () && student.hatZugesagt()) {
  student.immatrikulieren();
}
```

Falls die Klammern bei einzelnen Anweisungen weggelassen werden, kann bei nachträglichem Hinzufügen von zusätzlichen Anweisungen oder anderen Änderungen vergessen werden, die Klammern zu setzten, da durch die Einrückung suggeriert wird, dass die Anweisungen zusammengehören. Man betrachte auch Folgenden Situation (ist in anderer Form wirklich in der Praxis – leider – anzutreffen, es dauert oft Stunden einen derartigen Fehler zu finden):

```
if (student.hatZulassung () && student.hatZugesagt());
  student.immatrikulieren();
```

Obiges Fragment ist höchstwahrscheinlich aus der folgenden ersten Version hervorgegangen, wo gar keine Anweisung im Rumpf existiert. Dann muss aber ein Semikolon gesetzt werden.

```
if (student.hatZulassung () && student.hatZugesagt());
```

Bei nachträglichem Hinzufügen einer Anweisung entsteht dann unbeabsichtigt obige, falsche Version. Mit konsequenten Setzen der Klammern kann dies leicht vermieden werden:

```
if (student.hatZulassung () && student.hatZugesagt()) {
  ; // das Semikolon könnte hier aber auch weggelassen werden
}
```

Eine zusätzliche Anweisungen hinzuzufügen ergibt nun immer problemlos eine korrekte Version:

```
if (student.hatZulassung () && student.hatZugesagt()) {
  student.immatrikulieren();
}
```

Ausnahme obiger Regel: Folgt auf else direkt wieder ein if, so braucht das if nicht in ein Paar geschweifter Klammern eingeschlossen zu werden.
Begründung: Damit Quelltexte lesbarer werden.

Beispiel:

```
if (a < 7) {
  a = a + 1;
} else if (a > 18) {
  a = a - 1;
} else if (a == 10) {
  ...
}
```

Dies vermeidet, dass bei einem kaskadierenden if-else if-else if - … - else durch fortgesetztes Einrücken die Zeilen immer weiter nach rechts wandern und so der Quelltext unnötig tief verschachtelt ist. Man kann sich diese Regel auch so verstellen, dass damit im Gegensatz zu anderen Programmiersprachen ein in Java nicht vorhandenes Schlüsselwort „elsif“ simuliert wird.

Regel: Schreibe entweder a) die geschweifte öffnende Klammer auf die nächste Zeile oder b) hinter der Kontrollanweisung ohne weitere Anweisungen hinter der öffnenden Klammer. Schreibe die zugehörige schließende Klammer immer auf eine neue Zeile ohne weitere Anweisungen oder Bezeichner davor oder dahinter (Ausnahme else).
Begründung: Damit Quelltexte kürzer und lesbarer werden.

Beispiel zu a)

```
if (a > 7)
{
  a = a + 1;
}
```

Beispiel zu b)

```
if (a > 7) {
  a = a + 1;
} else {
  a = a - 1;
}
```

Die zu einer Kontrollanweisung zugehörigen Anweisungen sind dadurch immer auf einer neuen Zeile und durch die Verschachtlung klar ersichtlich. Insbesondere wenn noch folgende Regel beachtet wird:

Regel: Schreibe jede einzelne Anweisung in eine separate Zeile.
Begründung: Damit Quelltexte lesbarer werden.

Dies gilt auch für Variablendeklarationen: pro Variable immer eine Zeile. So lassen diese sich leichter individuell kommentieren und man kann eine Variable weniger leicht übersehen.

Regel: Vermeide die Schlüsselwörter continue und break, um Schleifen fortzuführen oder eine Kontrollstruktur abzubrechen. Ausnahme: break bei case-Anweisungen.
Begründung: Damit Quelltexte lesbarer werden.

Durch break und continue in Kontrollanweisungen wird der sequentielle Kontrollfluss unterbrochen. Dieser ist dadurch für einen Menschen schlechter zu verstehen wie bei der Verwendung von Sprunganweisungen in BASIC. Wie auch Sprunganweisungen, lassen sich continue und break in den meisten Fällen vermeiden, ohne das das Programm unleserlicher wird.

Regel: Vermeide eine zu tiefe Verschachtlung von Kontrollanweisungen (ca. drei Kontrollanweisungen).
Begründung: Damit Quelltexte lesbarer werden.

Beispiel:

```
...
for (int x = 0; x < felder.length; x++) {
  for (int y = 0; y < felder[x].length; y++) {
    verschiebeFeldNachLinks(x,y);
  }
}
...

private void verschiebeFeldNachLinks(int x, int y) {
  if (0 < x && x < felder.length
          && 1 < y && y < felder[x].length) {
     felder[x][y-1] = felder[x][y];
  } else if (x == felder.length - 1
                && y == felder[x].length - 1) {
     felder[x][y] = Feld.LEERES_FELD;
  }
}
```

In diesem Beispiel gehören die beiden for-Schleifen zusammen. Den inneren Teil inklusive der zweiten for-Schleife in eine Methode zu verlagern wäre schlecht (man denke an eine zukünftige Erweiterung an eine dritte Dimension). Jeweils zwei Methoden für die beiden if-Anweisungen zu machen wäre auch schlecht da die Tiefe sich nicht reduziert und zwei logisch zusammengehörige Teile in zwei verschiedene Methoden verschwänden.

Bei einer tiefen Verschachtlung von Schleifen und Bedingungen ist der Kontrollfluss des Programms für einen Menschen oft schwer zu erfassen. Bei zu tiefer Verschachtlung können die inneren zusammenhängenden Teile in eine private Methode verschoben werden (Automatisch mit Refactoring-Funktionen der IDE). Dieses Auslagern innerer Teile in eine Methode mit gut gewählten Namen hilft das Programm schneller und besser zu verstehen, da man sich auf weniger Details und eine weniger komplexe Struktur konzentrieren muss.

Ausdrücke

Regel: Setzte vor und nach jedem binären Operator ein Leerzeichen.
Begründung: Damit Quelltexte lesbarer werden.

Beispiel:

```
! isVolljaehrig
a = 1 + 7 * (5 / a);
flaecheninhalt = kreisradius * kreisradius * 3.14159265;
```

Operatoren sind wichtig, um zusammengehörige Teilausdrücke zu identifizieren und zu verstehen. Da sie im Gegensatz zu Bezeichnern oder Zahlen sehr kurz sind „verschwinden" sie für menschlichen Leser sehr schnell. Durch Hinzufügen der Leerzeichen werden die Operatoren wieder hervorgehoben.

Regel: Breche überlange Zeilen wie folgt um:

- bei arithmetischen Operatoren vor einem Operator mit der schwächsten Bindung (Operatoren so auswählen, dass nicht zu viele Zeilenumbrüche entstehen)
- bei Methodenaufruf oder Parameterdeklaration nach einem Komma
- rücke den umgebrochenen Teil soweit nach rechts ein, dass er sich unterhalb des zugehörigen linken Teilausdrucks des Operators befindet mit zwei bis vier zusätzlichen Leerzeichen

Begründung: Damit Quelltexte lesbarer werden.

Beispiel:

```
a * a * a + 3 * a * a * b + 3 * a * b * b + b * b *b
```

nicht bei *, sondern bei + umbrechen (+ bindet schwächer als *)

```
   a * a * a + 3 * a * a * b
     + 3 * a * b * b + b * b *b
```

```
x1 * 3 + (a + b + 6) * ((x2 + 8) + x3 / 11)
```

Bei + umbrechen im zweiten Faktor, wäre schlecht, da durch Einrücken, dann x3 / 11 über die Grenze hinausragt. Deswegen besser bei * umbrechen.

```
x1 * 3 + (a + b + 6)
           * ((x2 + 8) + x3 / 11)
```

```
public Address(String strasse,
               String hausnummer,
               int postleitzahl,
               String ortsname) {
  ...
}
```

Damit die Operatoren bei komplexeren Ausdrücken mit längeren Bezeichner oder Zahlen nicht so leicht überlesen werden und die Struktur eines Ausdrucks besser erkennbar ist.

Regel: Vergleiche Boolesche Werte in einem Booleschen Ausdruck nicht mit == auf true oder false. Verwende stattdessen bei true, den Booleschen Ausdruck selbst und bei false dessen Negation (bei != entsprechend umgekehrt)

Begründung: Damit Quelltexte lesbarer werden.

Beispiel:
Anstatt folgenden Booleschen Ausdruck

```
schaltjahr == true && volljaehrig == false
```

verwende besser

```
schaltjahr && ! volljaehrig
```

Bei Verwendung der Getter-Methoden mit vorangestellten is und Aussprache von ! als „nicht“ oder „not“, werden Boolesche Ausdrücke so kürzer und verständlicher.

```
isSchaltjahr() && ! isVolljaehrig()
```

Regel: Bei mathematischen Vergleichen wie 0 < i < j < n behalte die Reihenfolge der Operatoren und Variablen in der Implementierung bei.
Begründung: Quelltexte werden verständlicher.

Beispiel:

```
if (0 < i && i < j && j < n) {

}
```

Oder für i = j = 7 < n

```
if (i == j && j == 7 && 7 < n) {

}
```

Der resultierende Ausdruck ist dann sehr nah an der mathematischen Schreibweise orientiert und leichter verständlich, da er die ursprüngliche zu implementierende Form beibehält.

Methoden

Regel: Verwende nur so viele Zeilen für die Implementierung einer Methode, so dass die Methode komplett auf den Bildschirm passt (ca. 20-30 Zeilen), wenn möglich auch noch die zugehörige Javadokumentation.
Regel: Verlagere bei zu langen Zeilen zusammengehörende Teile in private Methoden.
Begründung: Damit Quelltexte verständlicher werden.

Um die Struktur und Funktionsweise einer Methode zu verstehen, sollte der Programmier nicht gezwungen sein, vertikal zu scrollen: zugehörige Programmteile if-else sind dann oft nicht mehr auf einen Blick zu erfassen. Mit den Refactor-Funktionen der Entwicklungsumgebung geht eine derartige nachträgliche Formatierung des Quelltextes sehr schnell.

Manchmal findet man Quelltexte, bei denen die schließende Klammer mit einem einzeiligen Kommentar versehen wird, in dem das Schlüsselwort der zugehörigen Kontrollanweisung meist mit vorangestelltem end versehen wird.

```
if ( nichtNachmachen ) {
  while (auchNichtNachmachen) {

  } // end while
} // end if
```

Dies ist leider manchmal auch in einigen Dokumentationsrichtlinien zu finden. Es handelt sich bei dieser Richtlinie aber lediglich um Sympton- und nicht Ursachenbekämpfung, da die obige Regel diese unsinnige Dokumentationsrichtlinie unnötig macht.[2] Die zusätzlichen Kommentare bei der schließenden Klammern haben auch den Nachteil oft bei Änderungen nicht nachgeführt zu werden:

[2] Natürlich wäre es besser im Sprachentwurf für jede Kontrollanweisung eine spezielle schließende Klammer vorzusehen, wie zum Beispiel für if ein endif. Da der Ursprung von Java leider in C++ und weniger in einem direkten ALGOL Abkömmling liegt, ist dies aber unterlassen worden.

zum Beispiel wird sehr leicht vergessen aus `// end while` ein `// end for` zu machen, wenn eine while zu einer for-Schleife abgeändert wird.

Regel: Deklariere alle lokalen Variablen (außer Schleifenvariablen) am Anfang der Methode gefolgt von einer Leerzeile.
Regel: Bei einer Funktion sollte insgesamt nur ein return in der Methode existieren (notwendigerweise am Ende der Methode). Setzte vor diesem return eine Leerzeile.
Begründung: Damit Methodenimplementierung verständlicher wird.

Beispiel:

```
public void berechneQuersumme(int zahl) {
  int quersumme = 0; // die zu berechenden Quersumme, wird
                     // am Ende der Methode zurückgegeben
  while (zahl != 0) {
    quersumme = zahl % 10;
    zahl = zahl / 10;
  }

  return Math.abs(quersumme);
}
```

In vielen Methoden werden Berechnungen angestellt und die Teilergebnisse meist zwischengespeichert. Für die Verständlichkeit ist es hilfreich, wenn alle diese Variablen am Anfang aufgeführt und deren Verwendungszweck dokumentiert wird bevor sie verwendet werden. Es sollte nur ein return vorhanden sein, da return die Methode abbricht: für einen Menschen ist der Programmverlauf der Methode bei vielen einzelnen return-Anweisungen oft nur schwer ersichtlich – insbesondere, wenn return in verschachtelten Schleifen und Bedingungen auftritt.

Diverses

Primitive Datentypen

Regel: Verwende möglichst int statt byte, short oder long
Regel: Verwende möglichst double statt float
Regel: Mische möglichst keine verschiedenen Zahlentypen in einem Ausdruck.
Regel: Verwende wissenschaftliche Notation nur bei sehr kleinen oder sehr großen Zahlen.
Begründung: Vermeiden von Fehlern und bessere Lesbarkeit

Beispiel

Statt 1.52E+5 besser 15200 schreiben. Aber lieber 1E-12 statt 0.000000000001.

Byte und short haben im Gegensatz zu int und long einen relativ kleinen Wertebereich. Da ein Überlauf in Java anderes als in C# nicht durch eine Ausnahme angezeigt wird, sollte möglichst mit Datentypen höherer Genauigkeit gerechnet werden. int ist meist ausreichend vom Wertebereich und schneller als long. Es ist dabei normalerweise auch nicht langsamer als byte oder short, da die interne Prozessorregister heutzutage mindestens 32bit breit sind. Es entfällt auch das bei long lästige Schreiben von l oder L hinter jeder Zahl.

Float hat gegenüber double eigentlich nur den Vorteil halb so viel Speicher zu verbrauchen – Speicherverbrauch ist heutzutage aber kein Problem mehr. Die Berechnungen mit float sind normalerweise nicht schneller als mit double, da Mikroprozessoren intern mindestens mit der Genauigkeit von double rechnen. Teilweise kann float sogar langsamer sein, da die Konvertierung vom internen Gleitkommaformat zu float mehr Zeit kosten kann als zu double: insbesondere, falls die CPU intern immer mit double rechnet. Analog wie bei long entfällt auch das lästige Schreiben von f oder F hinter jeder Floatzahl.

Verschiedene Zahlentypen in einem Ausdruck zu mischen sollte vermieden werden, da dadurch unbeabsichtigt Fehler entstehen. Zum Beispiel ist der folgende Ausdruck immer 0, auch wenn x ein double Wert ist: `5 / 9 * x` .

Die wissenschaftliche Notation sollte nur bei sehr großen oder sehr kleinen Zahlen (ab ca. 7-8 Ziffern) angewendet werden, da diese Notation für den Menschen sonst schwerer zu verstehen ist.

Lokale Variablen

Regel: Vermeide lokale Variablen, deren Wert nur genau einmal verwendet wird. Verwende stattdessen die Berechnung des Wertes anstatt der lokalen Variablen.
Begründung: Programme werden kürzer und lesbarer

Beispiel:

Statt folgende Methodenimplementierung

```
public void getFahrenheit() {
  int fahrenheit;

  fahrenheit = 1.8 * celsius + 32.0;

  return fahrenheit;
}
```

Besser

```
public void getFahrenheit() {
  return 1.8 * celsius + 32.0;
}
```

Felder

Regel: Vermische bei mehrdimensionalen Felder nie den C- mit dem Java Deklarationsstil.
Begründung: Programme werden lesbarer.

Beispiel:

```
int [][] matrix; // Java-Stil bevorzugen
int matrix [] []; // C-Stil vermeiden
```

Unglücklicherweise kann man in Java Felder wie im obigen Bespiel gezeigt auf zwei Arten deklarieren. Beides bedeutet dasselbe. Da [ ] letztendlich eine Typinformation ist, ist es am sinnvollsten diese im Javastil beim Datentyp aufzuführen und nicht hinter dem Bezeichner. Leider habe die Sprachentwickler von Java das Unglück aber noch weiter getrieben und es ist erlaubt beide Stile in einer Deklaration zu vermischen: `int [] matrix [].` Dies sollte aus Gründen der besseren Lesbarkeit der Programme auf jeden Fall unterlassen werden.

Javadoc

Die durch Javadoc erstellte HTML Dokumentation dient Entwicklern dazu Klassen verwenden zu können ohne die Quelltexte lesen zu müssen oder wenn nur der Bytecode aber nicht die Quelltexte verfügbar sind. Die Javadoc-Kommentare müssen deswegen kurz und spezifisch für alle nicht private Member einer Klasse beschreiben

- was eine Klasse für eine Menge von Objekten beschreibt
- was ein Attribute bedeutet und welche Werte es annehmen kann (bzw. nicht darf)
- was eine Methode macht, was für ein Wert zurückgegeben wird und was die Parameter bedeuten und welche Werte sie annehmen dürfen (bzw. nicht dürfen)

In einem Javadoc-Kommentar sollte nie beschrieben werden wie etwas implementiert ist, es sei denn es ist für die Verwendung der Klasse und deren Objekte notwendig.

Ein Javadoc-Kommentar ist ein mehrzeilliger Kommentare der mit /** eingeleitet und mit */ beendet wird. Ob man im Javadoc-Kommentare in jeder Zeile auch noch einen zusätzlichen * hinzufügt oder nicht, ist Geschmackssache. Der erste * in jeder Zeile innerhalb eines Javadoc-Kommentars wird ignoriert. Folgende Varianten sind in der resultierenden HTML-Dokumentation identisch. Man sollte sich generell für eine entscheiden. Am besten die Voreinstellung der Entwicklungsumgebung verwenden.

Grundsätzlich gilt für jede Art von Kommentar folgende Regel:

Regel: Ein Kommentar muss so kurz wie möglich und spezifisch wie nötig sein.

Spezifisch bedeutet mehr Information anzugeben, als im Klassennamen oder der Methodendeklaration enthalten ist: dadurch wird der Kommentar länger.
Kurz bedeutet nicht relevante und vor allem unspezifische Informationen wegzulassen.

Klasse

Regel: Drücke im Klassenkommentar in einem Satz beginnende mit dem Klassennamen die wesentlichen Entwurfseinscheidung (Attribute und Beziehungen) und/oder die Verantwortung (abstrakte Beschreibung des Verhaltens der Objekte) aus.
Regel: Um das Geheimnisprinzip zu waren, vermeide möglichst jeden rein Implementierungstechnischen Bezug (z.B. Nennung von Datentypen)
Begründung:

Beispiel:

Nehmen wir an es existieren die Klassen Hochschule, Student, Dozent und Studiengang.

```
/**
  Eine Hochschule mit Studenten, Dozenten und Studiengängen.
  */
public class Hochschule {

}
```

Der obige Kommentar drückt den Entwurf der Beziehungen von Hochschule zu anderen Klassen aus: die Klasse Hochschule hat jeweils eine 1-n Beziehung zu Student, Dozent und Studiengang.

```
/**
   Ein Student mit Namen, Matrikelnummer und seine
   eingeschriebene Studiengänge.
  */
public class Student {
  private String name;
  private String matrikelnummer;
  private Studiengang studiengaenge [];
}
```

Dieser Kommentar nennt die wichtigsten Eigenschaften des Studenten und drückt eine 1-n Beziehung zum Studiengang aus.

Um eine Klasse besser verstehen und verwenden zu können, muss deren Zweck mindestens grob beschrieben sein. Ansonsten muss jeder Entwickler sich aus den Detailinformationen (Methoden, Attribute) den Zweck der Klasse selbst zusammenreimen. Um die Klasse zu ändern, muss deren Zweck durch Nennung der wesentlichen Entwurfsentscheidungen klar abgegrenzt sein. Ansonsten ist die Gefahr groß, dass die Klasse so erweitert wird, dass sie die ursprünglichen Entwurfszielen nicht mehr erfüllt sind: normalerweise wird dadurch das Programm komplexer, schwerer zu verstehen und fehleranfälliger.

Regel: Beschreiben nie wo (von welchen anderen Klassen) die Klasse verwendet wird.
Begründung: Dies hilft normalerweise nicht zu verstehen wofür die Objekte der Klasse verantwortlich sind und ist daher in einem Javadoc Kommentar meist nutzlos.

Attribute

Regel: Beschreibe die Bedeutung des Attributs (bzw. dessen Werte).
Begründung: Attribute enthalten Daten. Um ein Attribut zu verstehen, muss deutlich werden, was dessen Daten bedeuten.

Beispiel:

```
/**
  Eine deutsche 5-stellige Postleitzahl.
  Es sind nur gültige Postleitzahl aus dem offiziellen
  Verzeichnis der Post erlaubt.
 */
 private int postleitzahl;

 /**
   Der Verkaufspreis in Euro. Er darf nicht negativ sein.
  */
 private double verkaufspreis;

 /**
  Der obligatorische Hauptwohnsitz. Er darf nicht null
  sein.
  */
 public Adresse hauptwohnsitz;
```

Grundsätzlich ist es bei primitiven Datentypen immer auch sinnvoll zu beschreiben, ob alle Werte aus dem Wertebereich angenommen werden dürfen. Bei Referenztypen sollte immer auch beschrieben sein, ob null ein erlaubter Wert ist: in diesem Fall aber auch die Bedeutung von null beschreiben.

Methoden

Regel: Beschreibe immer was die Methode macht, nicht wie sie implementiert ist.
Regel: Fange den Kommentar mit dem Verb des Methodennamens in Präsensform an.
Regel: Führe alle Parameter mit dem @param-Tag auf und beschreiben sie.
Regel: Beschreibe die Bedeutung des Rückgabewerts bei @return mit einer kurzen Phrase

```
public Hochschule {

  /**
   Immatrikuliert den student an dieser
   Hochschule: ordnet dem Studenten eine Matrikelnummer zu
   und richtet ein Studienkonto für Studiengebühren und
   Fachsemester ein. Die Matrikelnummer von student enthält
   danach eine neue Matrikelnummer.

   @param student der zu immatrikulierende student:
                  darf nicht null sein. Er darf noch
                  nicht an dieser Hochschule immatrikuliert
                  sein.
   */
  public immatrikulieren(Student student) {
  }

  /**
   Gibt eine neue, eindeutige Matrikelnummer zurück.
   Sie ist eine fortlaufende 13-stellige Nummer mit
   führenden Nullen.

   @return eindeutige Matrikelnummer mit führenden Nullens
   */
  private String getNeueMatrikelnummer() {
  }
```

Literatur

[1] Code Conventions for the Java Programming Language,
http://java.sun.com/docs/codeconv/ , überarbeitete Version vom April 1999.
[2] How to Write Doc Comments for the Javadoc Tool,
http://java.sun.com/j2se/javadoc/writingdoccomments/ .