Beurteilung und Style Guide

Einleitung

Ein Style Guide (ST) gibt Formatierungsregeln vor, die darauf abzielen, die Lesbarkeit des Source Codes zu verbessern. Verschiedene Firmen/Teams verwenden verschiedene Style Guides. Prominente Beispiele findest du etwa unter Google C++ Style Guide, Python Style Guide oder GNU Coding Standards. Wenn du dir diese Regelsammlungen ansiehst, wirst du merken, dass sie sehr detailliert sind. Der hohe Detailgrad führt zu sehr einheitlichem Code, was die Kollaboration in großen Teams vereinfacht.

Auch bei uns in Java wird ein ST verwendet. Wir möchten nämlich, ...

- ... dass du lernst, einen ST genau zu befolgen
- ... dass du den Sinn eines ST und seiner Einhaltung verstehst
- ... dass die Trainer*innen deine Abgaben möglichst einfach korrigieren können.

Zusammenfassung der Regeln

Hier findest du einen kurzen Überblick über die Regeln. Weiter unten werden die Regeln dann im Detail erklärt.

Allgemeines

- Die Sprache des Source Codes und der Kommentare ist Englisch.
- Der Code soll einheitlich sein
- Die Namenskonvention ist einzuhalten

Kommentare

- Die Quelldatei enthält einen Kommentarheader
- Jede Methode hat einen Methodenheader

Codelayout

- Leerzeichen zwischen Operanden und Parametern
- Jeder Befehl steht in einer eigenen Zeile
- Blockklammern richtig setzen
- Jeder Block muss mit zwei oder vier Leerzeichen eingerückt werden
- Die Länge einer Codezeile soll 120 Zeichen nicht überschreiten
- Die Anzahl an Zeilen pro Funktion beschränkt sich auf 80.

PLAGIATE

- Richtige Deklarierung von nicht eigenem Code

Erklärungen der Regeln

Der Code soll einheitlich sein

Der ST soll dir helfen, leserlichen Code zu schreiben, ohne dich mit zu vielen Regeln und Details zu belasten. Daher ist klar, dass der ST bei Weitem nicht alle Eventualitäten abdeckt. In Situationen, in denen die Formatierung nicht durch den ST vorgegeben ist, vertraue ich darauf, dass dein Urteilsvermögen dich zu einer schönen Lösung führt. Bedenke dabei immer, dass Einheitlichkeit wichtig ist!

Ein Beispiel ist die Reihenfolge von Methodenparametern.

```
1 usage  new *
int insert(ArrayList<String> list, Element element) {
    // Code
    return 0;
}
1 usage  new *
int remove(Element element, ArrayList<String> list) {
    return 0;
}
```

Bei insert() ist der erste Parameter eine Liste und der zweite ein Element was eingefügt werden soll. Bei remove() ist es genau umgekehrt obwohl die Methoden die gleichen Parameter bekommen? Seltsam, nicht wahr?

Namenskonventionen

Namenskonventionen beschreiben die Formatierungsregeln für die verschiedenen Arten von Befehlen/Typen, die in einem Programm vorkommen können.

Quelle: https://www.oracle.com/java/technologies/javase/codeconventions-namingconventions.html

Die für den Anfang wichtigsten sind hier zusammengefasst

Typ	Regeln für den Namen	Beispiele
Klassen	Klassennamen sind Nomen, in Fällen von Wörtern, die sich aus mehreren Nomen zusammensetzen, werden die Anfangsbuchstaben der einzelnen Nomen groß geschrieben. Klassennamen sollten kurz und ausdrucksvoll sein.	class.File; class.WindowBuilder;

Methoden	Methoden sind Verbe, in fällen wo sich mehrere aneinanderreihen wird der erste Anfangsbuchstabe klein und die anderen groß geschrieben.	run();
		runFast();
		getBackground();

Variablen	Variablennamen sollten so gewählt werden, dass ein Anderer auf ersten Blick erkennt worum es sich handelt. Ausgenommen hiervon sind „Wegwerf-Variablen“ die nur temporär verwendet werden. Häufig sind solche Variablen: • i, j, k, m, n für Integer • c, d, e für Character Für Variablenamen gelten die selben Regeln für Methoden, allerdings dürfen die Namen wie in 1) ODER 2) verwendet werden (siehe Einheitlicher Code)	1) int i; char c; float myWidth; String myFathersName; 2) int i; char c; float my_width; String my_fathers_name;

Konstanten	Variablen die als Konstanten deklariert werden müssen in CAPS geschrieben werden, außerdem muss jedes Wort mit einem Underscore „_“ getrennt werden.	static final int MIN_WIDTH = 4;
		static final int MAX_WIDTH = 4;
		static final int MAX_NUM_OF_PLAYERS = 4;

Jede Methode hat einen Methodenheader

Methoden sind Herz und Seele deines Java Programms uns es ist (insbesondere für die Korrektur) unabdingbar, dass man leicht erkennt, welche Methode was macht, ohne den Code erst lesen und verstehen zu müssen.

Diese Art von „Headerkommentaren“ sind so weiter verbreitet, dass es Tools gibt, die mit Hlfe dieser Header gleich eine Dokumentation des Programms generieren können. Damit das funktioniert muss natürlich festgelegt sein wie der Headerkommentar aussieht. Ein beleibtes Beispiel ist Doxygen (http://www.doxygen.org/), und daran ist auch der Headerkommentarstil dieses ST angelehnt.

Ein Beispiel dazu findest du weiter unten im Beispielprogramm.

Leerzeichen zwischen Operanden und Parametern

Zu Gunsten besserer Lesbarkeit sind bei binären Operanden (wie +, -, =, etc…) und Parametern Leerzeichen einzufügen.

```
int wrong=0; // wrong!
int correct = wrong + 5; // ok
wrong = correct++; // ok as well
doSomething(wrong, correct); // ok (spaces after ","
```

Jeder Befehl steht in einer eigenen Zeile

Jedes Statement beginnt in einer neuen Zeile, ausgenommen sind natürlich verschachtelte Befehle innerhalb eines „größeren“ Befehls.

Auch Variablendeklarationen sollen für den Anfang einzeln vorgenommen werden, und nicht „aneinandergekettet“. (Lesbarkeit)

Statt:

```
int my_number = 1; char whatever = 'a'; if(my_number == 1) System.out.println("test"); int a = 0; b = 1; c = 2;
```

ist also:

```
int my_number = 1;
char whatever = 'a';
if(my_number == 1)
    System.out.println("test");
int a = 0;
int b = 1;
int c = 2;
```

zu schreiben.

Blockklammern richtig setzen und Einrückung

```
while(condition)
{
    // Code
}
```

```
if(condition)
{
    // Code
}
else if (other_condition)
{
    // Other Code
}
else
{
    // You get it...
}
```

```
while(condition) {
    // Code
}
```

```
if(condition) {
    // Code
} else if (other_condition) {
    // Other Code
} else {
    // You get it...
}
```

```
while(condition){
    // Code
}
```

```
if(condition){
    // Code
} else if (other_condition){
    // Other Code
} else{
    // You get it...
}
```

In den obigen Beispiele wird für die Codeblöcke eine Einrückung von 4 Leerzeichen verwendet (Standard TAB in IntelliJ)

Maximal 120 Zeichen pro Codezeile

Es ist

klar,

dass zu kurze

Zeilen der

Leserlichkeit eines

Codes Schaden

zufügen.

Es ist aber genau so schlecht, zu lange Zeilen zuzulassen. Nicht nur, dass manche den Code zum Korrigieren ausdrucken könnten (hoffentlich drucken sie keine Abgaben aus, um kein Papier zu verschwenden), es ist auch nicht unbedingt angenehm, am Monitor lange horizontal scrollen zu müssen, um Anfang und Ende einer Zeile lesen zu können. Wenn dieser Text bis hierher gelesen wurde, ist sicherlich einleuchtend, was gemeint ist.

Kurzes Beispiel:

```java
package bbrz.StyleGuide;

/**
 * ************************************************************************<p>
 * (MANDATORY!!!)<p>
 * ClassName.java<p>
 * <p>
 * Explanation of the program ...<p>
 * May have multiple lines ...<p>
 * <p>
 * Author: Your name<p>
 * Version: version number (example 0.1 or 1.1.2)<p>
 * Date: date of creation<p>
 * ************************************************************************
 */

no usages  new *
public class StyleGuide {
    1 usage
    static final String input_prompt = "Please enter a number: ";
    /**
     * Descriptive text here...
     * @param prompt_to_print is a given String that is printed before the input is read in
     * @return the input of the user
     */
    1 usage  new *
    public static int getUserInput(String prompt_to_print){
        System.out.println(prompt_to_print);
        // Usually an input is read here, to simplify the integer 5 is returned.
        return 5;
    }

    new *
    public static void main(String[] args) {
        int num_of_repetitions;
        num_of_repetitions = getUserInput(input_prompt);
        System.out.println("Number of repetitions equals: " + num_of_repetitions);
    }
}
```

PLAGIATE

In Bezug auf Abschreiben vertrete ich eine Null-Toleranz-Politik.

Die Arbeiten, die in diesem Unterricht bewertet wird, soll deine eigene sein!

Wird ein Plagiat festgestellt oder der Verdacht geschöpft, dass die abgegebene Arbeit nicht die eigene ist (äußert sich meist durch nicht-erklären-Können des abgegebenen Codes), so fällt das Feedback für das abgegebene Projekt vollständig aus und wird in der Gesamtwertung als 0% bewertet.

Die Noten und das Feedback, das ihr bekommt, sind zwar nicht wie in der Schule ein direktes Aufstiegskriterium, aber eine wichtige Information über das Können und wie gut ihr vorankommt.

Wenn dieses Feedback aufgrund eines Plagiats ausbleibt, gibt es erst beim nächsten Projekt wieder Rückmeldung zu eurer Arbeit.

Nun aber zu erfreulicheren Regeln: Es ist erlaubt, zu einem gewissen Teil auf Algorithmen oder Code-Fragmente zurückzugreifen, die man bei Recherchen in Büchern, im Internet oder durch Erklärungen von Kolleg*innen erhält. In jedem Fall müssen hierbei folgende **Voraussetzungen** erfüllt sein:

- Die Quelle wird genannt und der übernommene Code / Algorithmus ist eindeutig gekennzeichnet.
- Der verwendete Code / Algorithmus muss verstanden worden sein und kann bei einem Abgabegespräch detailliert erklärt werden.
- **Die Länge aller aus externen Quellen stammenden Teile darf 1/5 der Gesamtlänge deines Programms nicht übersteigen.**

Beispiele für Kennzeichnung:

```
// Book Name: , Author: , Page:
// from: https://www.geeksforgeeks.org/try-catch-throw-and-throws-in-java/?ref=lbp
// collaboratively created with fellow colleague X Y / created with the help of trainer X Y
// begin
complexMethod();
doSomething(a, b);
// end
```

Beurteilung

Die Benotung setzt sich aus den benoteten Projekten zusammen.

Die Note ergibt sich aus den aufsummierten Prozenten der Projekte. Diese werden wie folgt umgesetzt:

12,5% - Abstände		**Note**	**Ziffer**
50,00%	0,00%	Nicht genügend	5
62,50%	50,00%	Genügend	4
75,00%	62,50%	Befriedigend	3
87,50%	75,00%	Gut	2
100,00%	87,50%	Sehr gut	1