Programmieren Abschlussaufgaben Tipps

Namenskonventionen

lowerCamelCase für Attribut- und Methodennamen, UpperCamelCase für Klassen und Interfaces, UPPER_CASE_WITH_UNDERSCORES für Konstanten.
Aussagekräftige Namen!!! **`i`** oder **`n`** sind nur als Schleifenzähler in Ordnung!

Do.

```
private int turnCount;
public static final int BOARD_WIDTH = 16;
public void addToken(Token newToken) {
```

Don't.

```
private int n;
public static final int Width = 16;
public void add(Token t) {
```

JavaDocs

Alle nicht-privaten Methoden, Konstanten und Klassenköpfe ausführlich dokumentieren! Besonders auf "Seiteneffekte" in Methoden hinweisen, z.B. wenn zusätzlich zur eigentlichen Methodenfunktion (z.B. "`add...`") noch mitgezählt wird oder irgendwas aktualisiert etc.

Do.

```
/**
 * Method to add a new data point. Updates
 * the count and tells connected
 * representation objects to update.
 * @param data  the data point to be added
 */
void addDataPoint(DataPoint data)
```

Don't.

```
/**
 * add-Method.
 */
void addDataPoint(DataPoint data)
```

Ein letztes Mal sage ich auch das noch:

AUFGABENSTELLUNG AUFMERKSAM LESEN!

Man ärgert sich einfach unglaublich, wenn man irgendein Detail überlesen hat. Ich weiß, dass die Abschlussaufgaben im Umfang etwas enorm sind, aber man kann ja z.B. wichtige Zeilen markieren.

Magic Numbers

Zahlen-Literale außer `-1, 0, 1, 2` sollten in Variablen oder Konstanten verpackt sein, damit man deren Bedeutung gleich erkennt. Das bietet sich auch für Fehlermeldung-`Strings` an.

Do.

```
public static final int DECK_SIZE = 52;
for (int i = 0; i < DECK_SIZE; i++) {

}
```

Don't.

```
for (int i = 0; i < 52; i++) {

}
```

Datenkapselung

Interne Implementierungsdetails sollen eben das sein: intern. Das heißt: Attribute **`private`** machen und getter & setter anbieten. Das gilt auch für Listen! Dann eben Methoden wie `add` und `contains` anbieten.

Do.

```
private int speed; // km/h

public int getSpeedKmh() { return speed; }
public int getSpeedMph() { ... }
```

Don't.

```
public int speed;
```

Objektorientierung

Supertypen / Interfaces als Datentyp verwenden, wo möglich, vor allem bei Collections!

Do.

```
List<String> list = new
       ArrayList<String>();
```

Don't.

```
ArrayList list = new ArrayList();
```

Wenn mehrere Objekte Attribute oder Verhaltensweisen gemeinsam haben überlegen, ob man Teile in eine gemeinsame Oberklasse (→ Polymorphismus) oder ein Interface (z.B. bei gemeinsamen Verhaltensmustern) rausziehen kann!

Separation Of Concerns

Pro Klasse nur passende Verantwortlichkeiten / stark zusammenhängende Aktionen: z.B. Spielfeld kennt nur die eigene String-Repräsentation (`toString()` !), hat aber keine Methoden zur Ausgabe, die liegen außerhalb, da das Ausgeben nicht Aufgabe des Spielfeldes ist! Es stößt auch höchstens die Spielfigurenbewegung an, aber die ganze Logik der Bewegung gehört natütlich zur Klasse der Spielfigur!

Konstanten

Konstanten immer **`public`**, **`static`** und **`final`** machen!
`static` macht sonst eigentlich nur bei allgemein brauchbaren Methoden wie `Math.random()` oder Zahlenstrings mit Nullen auffüllen Sinn und sollte im eigenen Code die absolute Ausnahme sein! Es lässt sich so gut wie immer ein Objekt finden, dass die **`static`** Sachen sinnvoll in Instanzen kapselt.
Ein Beispiel: Ich hatte ursprünglich in meiner Abschlussaufgabe eine Klasse Graph, dessen Kantenliste **`static`** war, weil ich dachte, es soll eh nur einen geben und dann kann ich von überall aus die Liste ausgeben, ohne irgendwie eine Graph-Instanz herzaubern zu müssen. Das ist aber unnötigerweise unflexibel (soll es doch einen zweiten Graphen geben steht man blöd da) und führt garantiert zu Punktabzug!
Die Kantenliste kann stattdessen einfach zu einer Instanzvariable gemacht werden und die womöglich wild verstreuten `print`-Methoden sind sowieso schöner gesammelt in einer Klasse aufgehoben, der man dann einmal beim instanziieren eine Referenz auf das Graph-Objekt mitgibt.

Schachtelung

Nicht zu stark verschachteln, ab Einrückungstiefen > 3 (ungefähr!) sollte man schauen, ob man nicht was auseinanderziehen kann. Das gilt insbesondere für mehr als zwei ineinander geschachtelte Schleifen!
Es kann auch öfters sehr hilfreich sein, Teilaufgaben einer Methode in einzelne Hilfsmethoden auszulagern, besonders, wenn man merkt, dass im Code drei Mal das gleiche steht mit nur je einem unterschiedlichen Wert → private Hilfsmethode, die den variablen Wert als Parameter nimmt!
Hilfsmethoden haben auch den positiven Nebeneffekt, dass sie unübersichtliche Stellen (z.B. zwei ineinander geschachtelte Schleifen) an eine andere Stelle „verstecken“, und dort, wo der Code benutzt wird dann nur noch ein Aufruf der Methode steht, deren Namen erklärt, was überhaupt passiert.

if-Abfragen lassen sich manchmal reduzieren, indem man sie „invertiert“:

Do.

```
if (!outerCheck) {
    return;
}
if (!innerCheck) {
    return;
}
...
```

Don’t.

```
if (outerCheck) {
    if (innerCheck) {
        ...
    }
}
```

Exceptions

Sollten nicht den Konstrollfluss steuern, sondern nur zur Ausnahmenbehandlung benutzt werden! (Ausnahme: `NumberFormatException`)
Generell gilt: `RuntimeException`-Untertypen nie behandeln, sondern den Fehler, der zu ihrem Auftreten führt beheben!
Wenn man eine Exception werfen will und keine vorgegebene passt richtig im Zweifelsfall eine eigene `Exception`-Unterklasse schreiben!

Methoden & Interfaces

Wo sinnvoll `toString()` und `equals()` (dann am besten auch `hashCode()`, auf jeden Fall wenn das Objekt in einer `HashMap` o.ä. benutzt werden soll) überschreiben. Eventuell sind auch `Comparable/Comparator` und `Iterator/Iterable` praktisch zu implementieren, allerdings nur wenn es auch Sinn macht, nicht gezwungenermaßen Interfaces implementieren, weil ich das hier erwähne :)

Erweiterbarkeit & Generics

Meiner Erfahrung nach muss man es mit Erweiterbarkeit und Generics nicht übertreiben. Wenn man eine nicht-generische Lösung hat, die trotzdem schön objektorientiert ist, sollte das auch kein Problem sein, man kann ja auch argumentieren, dass die Spezifikationen der Abschlussaufgabe final sind und man deshalb weniger Wert auf Erweiterbarkeit gelegt hat.
Insgesamt gilt: Man sollte sich von all den Tips hier nicht verrückt lassen machen, man hat oft selbst ein gutes Bauchgefühl, was gut gelöst ist und was nochmal überarbeitet werden muss.

Programmierstil

Ausufernde `if-else-` oder `switch-case`-statements sind tendenziell schlechte Zeichen, davon sollte man so wenig wie möglich (aber halt manchmal doch so viel wie nötig) haben (~~zum googlen: Stichwort "if-less programming"~~). Ein Beispiel, das mir jemand bei meiner Abschlussaufgabe erzählt hat: Eine "`collect`"-Methode sollte Objekte anhand von gewissen Eigenschaften zusammensammeln. Die eine Möglichkeit wäre es, dass `collect` abhängig davon, ob die gewünschte Eigenschaft erfüllt ist oder nicht entweder das Objekt zurückgibt, oder **`null`**. Dann braucht man aber da, wo `collect` aufgerufen wird noch ein **`if`** `(returnedObject == null)` **`...`** **`else`** **`...`**.
Die andere Möglichkeit wäre es, statt direkt ein Objekt oder **`null`**, immer eine Liste zurückzugeben, die dann eben leer ist, oder genau ein Objekt enthält. Dann kann man sich in der Methode, wo `collect` benutzt wird die Sonderbehandlung von **`null`** sparen und benutzt z.B. `results.addAll(collect())`.
Das Beispiel ist jetzt keine Richtlinie für perfekten Code, man könnte z.B. argumentieren, dass das Anlegen einer Liste für null bis ein Element Overkill ist, aber, wenn man obige Argumentation bringt, kann das schon Sinn machen, es ist auch flexibler, falls in Zukunft doch mal mehr als ein Objekt zurückgegeben werden soll.

Das Ganze soll zeigen, dass es oft kein richtig oder falsch gibt, sondern unterschiedliche persönliche Stile, die jeder anders bewertet. Deshalb sollte man auch unbedingt in die Einsicht gehen, wo man mit solchen Argumenten wie hier im Zweifelsfall relativ gut Punkte rausholen kann.

Benutzt man übrigens öfters **`instanceOf`** oder `getClass()`, ist das auch ein gutes Zeichen dafür, dass man mit geschickter Objektorientierung (Stichwort: Polymorphismus) besser bedient wäre.

Ein letzter Punkt noch zum Stil: viele, wild über die Länge einer Methode verteilte **`return`**s sind auch nicht gerne gesehen. Die Anzahl sollte in solchen Fällen nach Möglichkeit reduziert werden oder zumindest nur einmal unten ein **`return`** irgendeiner Variable stehen, die dann an den anderen Stellen nur gesetzt wird, statt ihren Wert direkt dort überall zurückzugeben.

Testen

Viele und “kleine” Testfälle überlegen, d.h. nicht einen komplizierten Fall testen, der die Reaktion auf mehrere außergewöhnliche Situationen gleichzeitig testet, sondern mehrere Tests und jeder überprüft genau eine Situation. Nach jedem Bugfix möglichst alle Tests nochmal laufen lassen (am besten automatisiert mit JUnit)!
Genug Zeit zum Testen einplanen!

Ein paar Hinweise zu den Testfällen:

- im Forum stehen oft ein paar Sachen, die offiziell nicht getestet werden!
- Die häufigste Exception, die bei den Übungsblattabgaben zu fehlschlagenden Error-Handling-Tests geführt hat, war die [Array]IndexOutOfBoundsException! Das heißt: Werte, die in irgendeiner Weise vom Benutzer manipuliert werden können und als Indizes benutzt werden sollen, immer überprüfen!
- Auch testen, ob spezifizierte Ausnahmesituationen auch die vorgegebene Reaktion auslösen (Fehlermeldung und je nach Vorgabe Programm schließen)

Weil ich nichts zur Lösung von den aktuellen Abschlussaufgaben sagen darf, schreibe ich die folgenden Sachen bezogen auf meine Abschlussaufgaben vom vorletzten Wintersemester, das sollte aber alles allgemein genug und übertragbar sein.

"Atomarität" aus den Aufgabestellungen rauslesen

Bei meiner Abschlussaufgabe stand in einem Absatz z.B.

> "*Wird dem Graphen [...] die Beziehung hinzugefügt, soll zusätzlich die Umkehrbeziehung [...] hergestellt werden*".

Sowas sollte dann in der eigenen Implementierung auch beides (Beziehung UND Umkehrbeziehung hinzufügen) mit einem Methodenaufruf `enterRelation(...)` passieren, wobei unbedingt im JavaDoc stehen sollte, dass die Methode neben dem vom Name suggerierten einfügen der Beziehung auch implizit gleich die Umkehrbeziehung einfügt.
(Deshalb hier übrigens auch "enter"Relation statt "add...", weil man bei "add..." normalerweise immer an das Einfügen von genau einem Element in eine Menge denkt)

Weiter mit dem Beispiel Relation-und-Gegenrelation-Einfügen: Man müsste ja in diesem Fall zur gegebenen Relation, die eingefügt werden soll die passende Gegenrelation finden. Das mit einem großen `if-else` / `switch-case` zu machen ist unschön und nicht objektorientiert. Stattdessen sollte die Relation (Objekt oder vielleicht besser Enum!) "wissen", was die zu ihr passende Gegenrelation ist, also eine Methode anbieten, die deren Typ zurückliefert. Das passt dann auch zu Separation of Concerns: nicht die `enterRelation`-Methode sucht die passende Gegenrelation raus, sondern die Relation selbst, es ist ihre Aufgabe.

Generell sollte man sich mal die einzelnen Klassen, die man hat anschauen und überlegen, ob alle nur das machen, was sie sollen, oder ob viele ganz unterschiedliche Aufgaben drinstecken.

Ein weiterer Hinweis: Eigene Objekte zur Datenrepräsentation "so weit" wie möglich benutzen: Soll z.B. für einen `print`-Befehl eine Menge von Objekten in vorgegebenem Format geprintet werden, ist es sinnvoll, erstmal die Objekte als solche zu sammeln und dann erst bei der Ausgabe `String`s daraus zu machen, anstatt gleich nur `String`s der einzelnen Objekte zu sammeln. Bei der Umwandlung zu `String`s hat man zwar noch eine Repräsentation des ursprünglichen Objektes, aber eben eine mit deutlich weniger Informationen, die vielleicht noch brauchbar wären.

Hier noch die Korrekturinformationen von vorletztem Jahr zu Methodik und Stil:

- sinnvolle Verwendung von Klassen
- Wahl von geeigneten Datentypen
- Les- und Wartbarkeit (z.B. Benennung)
- Strukturierung des Kontrollflusses (Länge der Anweisungen, Einrückung)
- Dokumentation (nicht nur JavaDoc, sondern auch Inline-Kommentare)
- sinnvolle Sichtbarkeiten
- abstrakte Interfaces verwenden (`List<…> list = new ArrayList<…>()`)
- korrekte Verwendung von `static`

Zu guter Letzt noch einen Link zur Seite eines ehemaligen Tutors, da steht auch noch mal ein bisschen was zu seinen Abschlussaufgaben:
https://martin-thoma.com/abschlussaufgaben-programmieren/

So. Jetzt ist doch einiges zusammengekommen. Deshalb hier noch mal zwei sehr wichtige Punkte:
1.) Alles was ich geschrieben habe sind keine offiziellen Anforderungen, sondern zum Teil allgemein anerkannte Best Practices und zum Teil Erfahrungen von mir und anderen Tutoren und Programmieren-Absolventen. Deshalb kann dieses Dokument auch Fehler enthalten!
2.) Man sollte jetzt auf keinen Fall Panik bekommen, dass man vielleicht ein paar der Punkte nicht umgesetzt hat, bis auf ein paar Sachen, die aber auch bei den Übungsblättern immer wieder wichtig waren, sind das teils Sachen, die für sehr gute Noten wichtig sind und keinesfalls über Bestehen und Nicht-Bestehen entscheiden. Man sollte auch nicht verkrampft versuchen, irgendwas Abgefahrenes zu implementieren, wenn man es sich nicht ganz zutraut, das wichtigste ist funktionierender Code und sinnvolle Programmierung, die Objektorientierung nicht gerade mit Füßen tritt ;). Man bedenke, dass 15/20 Punkten auf die Funktionalität abfallen! An dieser Stelle kann ich nur noch mal sagen, dass man sich meist auch auf das Bauchgefühl verlassen kann, wie gut die Implementierung ist.
Schaut viel ins Forum und geht am Ende zur Einsicht!
Ich hoffe, die Abschlussaufgaben machen dann zwischendurch auch ein bisschen Spaß und sind lehrreich, sie sind immerhin mit das praktischste, was man in Informatik / Informationswirtschaft so macht (außer PSE und ein, zwei anderen Ausnahmen).

Deshalb:

Viel Spaß und vor allem viel Erfolg!

- Dominik