HSLU Hochschule Luzern

Objektorientierte Programmierung (OOP)

Übung SW04: Schnittstellen und Datenkapselung

Themen: Schnittstellen (O06), Datenkapselung (P01), JavaDoc, Modularisierung.
Zeitbedarf: ca. 240min.

Roland Gisler, Version 1.6.2 (FS23)

1 Javadoc

1.1 Lernziele

- Dokumentieren von verschiedenen Java-Elementen (Interface, Klasse, Methode) direkt im Quellcode mit Hilfe der JavaDoc.
- Einfache Regeln für eine gute JavaDoc kennen.
- Javadoc als effizientes Hilfsmittel für eine gute Dokumentation verwenden.

1.2 Grundlagen

Beachten Sie bitte, dass für die Erstellung der JavaDoc in BlueJ alle vorhandenen Klassen im jeweiligen Projekt fehlerfrei kompilierbar sein müssen. Ausserdem muss unter `Einstellungen`, Tab «`Diverses`» die korrekte URL auf die Dokumentation von Java selber hinterlegt sein. Sie lautet: https://docs.oracle.com/en/java/javase/17/docs/api/

Beachten Sie auch den «Appendix I: JavaDoc» im Buch.

1.3 Aufgaben

a.) Öffnen Sie die API-Dokumentation der aktuellen Version Java 17 im Browser unter https://docs.oracle.com/en/java/javase/17/docs/api/ oder besser, wenn Sie alles gemäss Anleitung installiert haben, direkt aus dem lokalen JDK-Installationsverzeichnis im Unterverzeichnis **`$JAVA_HOME/docs/api/`**. Tipp: Legen Sie einen Link an!

b.) Wählen Sie zuerst aus der Liste das Modul **`java.base`** aus, und danach das Package **`java.lang`** aus. Suchen Sie dann nach der Klasse **`String`**. Studieren Sie die Dokumentation dieser Klasse, und sehen Sie, welche Attribute und Methoden sie anbietet.

c.) Sie finden im Installationsverzeichnis des JDK (Java Development Kit) im Unterverzeichnis **`/lib`** eine ZIP-Datei mit dem Namen **`src.zip`**. Darin sind die Quellen von **allen** Java-Klassen enthalten, denn Java ist eine quelloffene Sprache!
Öffnen Sie darin den Quellcode der Klasse **`java.lang.String`**, welche Sie im Verzeichnis **`/java.base/java/lang/String.java`**[1] finden und vergleichen Sie die im Quellcode enthaltene JavaDoc einer Methode (am Besten mit formalen Parametern und Rückgabewert) mit der daraus generierten HTML-JavaDoc, die Sie im Browser betrachten!

d.) Wählen Sie aus den bisherigen Übungen ein paar Klassen und Methoden aus, welche Sie nun exemplarisch selbst im JavaDoc-Format dokumentieren. Verwenden Sie dazu bei den Methoden im Minimum die JavaDoc-Tags **`@param`** und **`@return`**.

[1] Das etwas «merkwürdige» Verzeichnis rührt von der seit Java Version 9 neu eingeführten «Modularisierung» her, welche wir aber erst in späteren Semestern bzw. Modulen behandeln werden.

e.) Generieren Sie die JavaDoc in BlueJ über **`Werkzeuge→Dokumentation erzeugen`** und kontrollieren (und ggf. verbessern) Sie das Ergebnis bis Sie zufrieden sind.

f.) Gewöhnen Sie sich an, mindestens Interfaces (vollständig) und Klassen (Beschreibung und alle öffentlichen Methoden) **immer** zu dokumentieren. Achten Sie darauf, **kurz** und **prägnant** zu schreiben.

Hinweis: Wirklich gute Dokumentationen zu schreiben ist eine Herausforderung, und schärft gleichzeitig die Abgrenzung einer Klasse oder eine Interfaces.

2 Schnittstellen

2.1 Lernziele

- Differenzierung zwischen Schnittstellen und Klassen.
- Schnittstellen definieren und implementieren können.
- Potential von Schnittstellen (Rollen) erkennen.

2.2 Grundlagen

Im Input finden Sie eine Schnittstelle **`Switchable`**, welche einen einfachen Schalter mit Ein/Aus-Semantik definiert. Sie bildet die Grundlage für diese Aufgabe.

2.3 Aufgaben

a.) Kopieren Sie den Quellcode von **`Switchable`** aus dem Input und definieren Sie die Schnittstelle in einem neuen BlueJ-Projekt. Ergänzen Sie die fehlende JavaDoc und beschreiben Sie die Semantik der vier Methoden gemäss **Ihrem** Verständnis der Schnittstelle. Denken Sie an die «Ein-Satz-und-Punkt.»-Regel für den jeweils ersten Satz eines Dokumentationsteils.

b.) Erstellen Sie die HTML-Dokumentation, und verbessern Sie diese, bis Sie zufrieden sind. Dann vergleichen Sie Ihre Dokumentation (primär inhaltlich) mit anderen Studierenden. Haben Sie das gleiche semantische Verständnis der Schnittstelle?

c.) Entwerfen Sie eine einfache Klasse für einen Motor, der mindestens ein- und ausgeschaltet werden kann. Darum soll er auch die Schnittstelle **`Switchable`** implementieren. Was passiert bei der Kompilation, wenn Sie den Klassenkopf entsprechend erweitert haben?

d.) Implementieren Sie alle Methoden, welche die Schnittstelle verlangt. Da es sich um einen Motor handelt, könnten Sie den Betriebszustand z.B. in **`rpm`** (Umdrehungen pro Minute) festhalten. Testen Sie dann den Motor und stellen Sie sicher, dass er sich korrekt ein- und ausschalten lässt.

e.) Stellen Sie sich ein (sehr) einfaches Fahrzeug vor. Dieses Fahrzeug soll als Ganzes ebenfalls «Switchable» sein, sprich Sie können es ein- und ausschalten. Und natürlich hat das Auto (neben anderen Komponenten wie Licht, Scheibenwischer, Klima etc.) auch einen Motor. Modellieren Sie diese Situation mit einer Skizze. Versuchen Sie diese in UML zu zeichnen (von Hand). Vorlagen dazu finden Sie in verschiedenen Inputs.

f.) Implementieren Sie das skizzierte Modell (im Minimum das Fahrzeug, den Motor und noch **eine** beliebige weitere Komponente). Welche Klassen implementieren alle die Schnittstelle **`Switchable`**?

g.) Überlegen Sie zuerst, und probieren Sie es erst dann aus:
Können Sie für die verschiedenen Komponenten des Fahrzeuges (Motor, Licht etc.) auch den Typ des Interfaces verwenden? Das ist ➔**Polymorphie**!

3 Datenkapselung und Information Hiding

3.1 Lernziele

- Prinzip der Datenkapselung und des Information Hiding verstehen.
- Erkennen, wenn unerwünscht interne Repräsentation nach aussen gelangen.
- Entwerfen und Beurteilen verschiedener Designvarianten.

3.2 Grundlagen

In dieser Aufgabe modellieren wir eine einfache Linie mit Start- und Endpunkt in einem zweidimensionalen Raum. Beachten Sie, dass Sie dafür auf eine Klasse **`Point`** zurückgreifen können, die Sie bereits letzte Woche entworfen haben und ggf. noch leicht verbessern müssen.

Hinweis: Im Rahmen dieser Aufgabe müssen Sie keine Linie «zeichnen», sondern es geht «nur» um die objektorientierte Abstraktion einer Linie! In den vermeintlich einfachen Lösungsvarianten stecken bereits viele spannende Designfragen. Das Ziel ist die Diskussion und Bewertung dieser verschiedenen Varianten. Ausserdem könnten Sie auch eine unerwartete Überraschung erleben!

3.3 Aufgaben

a.) Entwerfen Sie eine Klasse **`Line`**, von welcher Sie mit einem Konstruktor mit zwei Koordinatenpaaren (**`x1`**, **`y1`**, **`x2`**, **`y2`**) als Parameter für den Start- und Endpunkt der Linie Instanzen erzeugt werden können. Entscheiden Sie sich für eine interne Darstellung: Offensichtliche Alternativen sind entweder vier **`int`**-Attribute oder aber zwei **`Point`**-Attribute. Überlegen Sie sich die Vor- und Nachteile. Was ist «objektorientierter»?

b.) Implementieren und testen Sie mindestens eine der Varianten von a), besser aber **Beide** (der Aufwand lohnt sich!).

c.) Sie erhalten eine neue Anforderung: Von einer bereits erzeugten Linie sollen sowohl der Start- als auch der Endpunkt mit Getter-Methoden abgefragt werden können. Wie lösen Sie diese Anforderung in den beiden Varianten von a)?

d.) Vielleicht ahnen Sie es bereits: Jetzt möchte man tatsächlich von bereits existierenden Linien sowohl den Start- als auch den Endpunkt nachträglich **verschieben** (bzw. neu setzen) können. Es werden somit Setter-Methoden benötigt. Auch hier haben Sie wieder zwei bzw. sogar vier Varianten!

e.) Nach all diesen Erweiterungen: Könnte es sein, dass es für die nun vorhandene Funktionalität evt. eine klar bessere interne Darstellung gibt? Scheuen Sie nicht den Aufwand ein Refactoring zu machen, und diese bessere interne Darstellung zu implementieren. ➔ Sie erleben gerade was Datenkapselung und Information Hiding wirklich heisst: Sie können die interne Datenhaltung verändern, **ohne** dass das nach Aussen (auf die Schnittstelle) eine Auswirkung hat!

f.) Und jetzt kommt (vielleicht) die versprochene Überraschung: Es ist sehr wahrscheinlich, dass Sie je eine Getter-Methode für den Start- und Endpunkt implementiert haben, welche jeweils ein **`Point`**-Objekt zurückliefern (Wenn nein: Implementieren Sie diese!).
Was passiert mit der Linie, wenn Sie auf einem Punkt (den Sie mittels der Getter-Methode ausgelesen haben) die Koordinaten (mit **`setX()`** oder **`setY()`**) verändern, sofern Sie das überhaupt zulassen?

g.) Wenn Sie keine Überraschung erlebt haben, kann das zwei Gründe haben: Entweder Sie haben bereits eine richtig gute, sichere, objektorientierte Lösung implementiert. Oder aber Sie haben etwas übersehen! Melden Sie sich in beiden Situationen auf jeden Fall bei Ihrem Coach!

h.) Können Sie erklären was genau passiert ist und was das für Konsequenzen hat? Auch das sollten Sie mit Ihrem Coach, oder mindestens mit anderen Studierenden diskutieren!