Java-Teil der Praxisphase von Melise

**Ziel**  
Datei generieren, mit Javascript-Code, der eine Map erstellt. Für jedes Attribut wird ein Eintrag in der Map erzeugt, dem die Baumstruktur der Attribute als JSON-String zugewiesen ist.

Das kann per Copy/Paste in das Frontend-Projekt integriert werden. Im Frontend wird zur Eingabe in einem Input-Feld der dazugehörige Baum visualisiert.

Die Projekte befinden sich in Github unter folgenden Adressen:

* Frontend:
  + Ursprung: <https://github.com/olada/praxisphase3-acsd-dependencies-frontend>
  + Fork von Melise: <https://github.com/melakc/praxisphase3-acsd-dependencies-frontend>
* Java
  + Ursprung: <https://github.com/olada/praxisphase3-acsd-dependencies-java>
  + Fork von Melise: <https://github.com/melakc/praxisphase3-acsd-dependencies-java>

Wir achten bei der Implementierung auf:

* Einzelne Arbeitspakete sind klein genug
* Früh in Unit-Tests einsteigen (Test-Driven-Development - TDD)
* Keep it simple and stupid (KISS)
* Wenig Redundanz / Copy-Paste (Don’t Repeat Yourself – DRY)
* Für alle Klassen gilt: Single Responsibility – Eine Klasse soll nicht zu viel machen
* Die Auflistung der Klassen in diesem Leitfaden ist keineswegs final, nach eigenem Ermessen können zusätzliche Klasse eingeführt werden. Insofern bilden die hier erwähnten Klassen nur das absolute Minimum.

**~~Punkt 1~~**  
~~Parametereingabe auf Standard Input (im Dialog)~~

* ~~Eingabe der Attribut-ID (bspw. „co#IX0030“). In der Konsole erscheint die Aufforderung „Bitte eine Attribut-ID eingeben“.~~
* ~~Fertig, wenn die eingegebene Attribut-ID ausgegeben wird (nach Enter-Taste)~~

**~~Punkt 2~~** ~~Text-Datei generieren(mit .js-Dateiendung), in der eine statische Map (die gleiche Map, die bereits im Frontend existiert) erzeugt wird, damit man das bereits ins Frontend importieren kann  
Fertig, wenn die Text-Datei mit JS-Inhalt erstellt wird UND wenn der Inhalt der JS-Datei nach „Rüberkopieren“ ins Frontend nach wie vor ein korrektes Frontend darstellt.~~

**~~Punkt 3~~** ~~Klasse erstellen namens SdAttribute mit Attributen „id“ und „description“. Attribute sind nach Best Practice private und besitzen NUR Getter. Wertzuweisung nur per Konstruktor (damit man es im Nachhinein nicht mehr ändern kann -> Stichwort „immutable“ bzw. „Immutability“).~~

~~Fertig, wenn Unit-Tests zeigen, dass SdAttribute funktioniert (Attribute „id“ und „description“ werden korrekt gesetzt).~~

**~~Punkt 4~~** ~~Einlesen aller statischen Attribute (co#sap.tsv - inkl. Beschreibung)  
Einlesen bspw. mit Files.readAllLines~~

* ~~Für jede Zeile, die eingelesen wird, erstellen wir ein neues Objekt von SdAttribute. Die ID stammt aus Spalte #1, die Beschreibung aus Spalte #2 (Spalte # bezieht sich auf co#sap.tsv; Spaltentrennzeichen = \t)~~
* ~~Alle SdAttribute-Objekte werden in einer HashMap gespeichert (dient dem schnellen Lookup): Key = Attribut-ID, Value = SdAttribute-Objekt.~~
* ~~SdAttribute-Klasse benötigt toString() -> generierbar mit IntelliJ~~
* ~~Fertig, wenn die Anzahl der HashMap-Einträge ausgegeben wird  
  Optional: HashMap in einer eigenen Klasse kapseln (damit der Haupt-Ausführungsstrang nicht zu komplex wird)~~

**~~Punkt 5~~** ~~Ausgabe des SdAttribute-Objekt zur eingegebenen Attribut-ID (siehe Punkt 1)  
Fertig, wenn das SdAttribute-Objekt in der Konsole ausgegeben wird~~

~~Public oder package private Methoden sinnvoll unterteilen, mit Rückgabewerten~~

~~Unit-Tests schreiben für :~~

* ~~splitTsv~~
* ~~fillMap~~
* ~~getAttributeID~~
* ~~getTsvFile~~

~~Scanner von außen-> nicht new Scanner -> dependency injection~~

~~Lies was zu mocks~~

**Punkt 6**  
~~Klasse namens Node erstellen (Baumstruktur).  
Notwendige Attribute:~~  
~~- value (SdAttribute-Objekt)  
- parent (Typ: Node)  
- children (Collection von Nodes, oder auch ein Set oder eine Liste, wie man möchte – am sinnvollsten erscheint Set)~~Notwendige Methoden:  
- ~~Konstruktor: mit einem Parameter von Typ SdAttribute~~  
- ~~add – mit einem Parameter von Typ SdAttribute~~  
- ~~getValue()  
- getChildren()~~  
~~- hasChild mit einem Parameter von Typ SdAttribute~~- ~~hasAnyChild mit einem Parameter von Typ SdAttribute~~  
~~Fertig, wenn Unit-Tests zeigen, dass die Node-Klasse funktioniert.~~  
Wenn fertig, in GENERIC umformulieren (Type = T anstatt SdAttribute)

**Punkt 7**  
~~Node-Objekt instanziieren, mit value = SdAttribute aus Punkt 5 (passend zur Eingabe)  
Fertig, wenn auf der Konsole Node.getValue().toString() ausgegeben wird (String-Repräsentation des SdAttribute-Objekts, welches im Node steckt).~~

~~Erste Zeile der tsv- Datei überspringen~~

**--21.12.2018--**

**Punkt 8**  
~~- toString(): Rekursives Zusammenbauen einer Zeichenketten-Repräsentation des Baums  
Fertig, wenn in einem Unit-Test ein Knoten mit allen Kindknoten ausgegeben wird (ohne Formatierung)  
Fertig, wenn Unit-Test zeigt, dass korrekte String-Repräsentation gebaut wird.~~

**Punkt 9**  
- ~~Einbindung einer neuen Maven Dependency für JSON-Erzeugung (bspw. gson)~~  
~~- Für Klasse „Node“: Neue Methode toJson() schreiben: (rekursiv) Liefert JSON-String, der mit Frontend kompatibel ist  
Fertig, wenn ein Unit-Test zeigt, dass korrektes Json geschrieben wird~~

**--09.01.2019--**

**Punkt 10**  
Einlesen von co#sad.tsv (Attribute mit Abhängigkeiten) und speichern in der Baumstruktur – nur für das eine eingegebene Attribut aus Punkt 5.  
Ähnlich wie bei Punkt 4:  
- Die Zeilen identifizieren, die zu unserem gesuchten Attribut gehören (eingegebene Attribut-ID muss in der ersten Spalten stehen  
- Als Baumeinstiegspunkt wird das Node-Objekt aus Punkt 7 verwendet   
- pro Zeile anhand der Attribut-IDs das dazugehörige Objekt aus der HashMap (oder wie bei „optional“ vermerkt aus einer Klasse, die die HashMap kapselt) auslesen. Das dazugehörige Objekt ist unser bekanntes SdAttribute-Objekt. Tipp: Durch Kombination der Node-Methoden „hasChild“ und „getValue“ kann ein neues Attribut hinzugefügt werden (if hasChild… baum.getValue().add(….), usw..)  
Fertig, wenn zu einem eingegebenen Attribut, ein kompletter Baum per toString() auf der Konsole ausgegeben wird) UND wenn Unit-Tests die Korrektheit belegen.

**--16.01.2019--**

**Punkt 11**  
Fertig, wenn die Ausgabe aus Punkt 9 (Baum als Zeichenkette mittels toJson()) in der Textdatei gespeichert wird (MIT VALIDEM JAVASCRIPT)

**Punkt 12**Alle Punkte von oben für alle Attribute durchführen (Zielbild: Javascript-Datei enthält für jede Attribut-ID eine Zeile mit JSON-Baum). Bedeutet: Es gibt keine interaktive Abfrage nach einer Attribut-ID mittels Standard-Input. Aber es werden Punkte 7 bis 11 für jede einzelne Attribut-ID durchgeführt, sodass für jede Attribut-ID ein JSON-Baum erzeugt wird.

**--21.01.2019--**

**30.01.2019 VORFÜHRUNG!**