Testat-Übung 2: Collections

Semesterwoche 8

Abgabe

Spätestens bis und mit Sonntag, 15. November 2020 23:00 Uhr

Gruppenarbeit bitte nur max. 3 Leute. Abgabe via MS Teams. Bitte Teilnehmende einer Gruppenarbeit in einem zusätzlichen File `group.txt` vermerken. Bei allfälligen Problemen bitte umgehend ein Email an den Übungsbetreuenden senden.

Geben Sie nur `.java`-Files ab! Kein Eclipse-Projekt und kein Zip-Archiv!

Lernziele

1. Konkrete Collections und Maps benutzen.
2. Probleme mittels Collections lösen.

Aufgabe 1: Studienplanung

Eine Studierende/ein Studierender möchte in ihrem/seinem Studium verschiedene Vorlesungsmodule besuchen. Pro Modul können aber vorher besuchte Module als Vorkenntnisse vorausgesetzt werden. Das Problem ist es nun, einen Studienplan zu berechnen, so dass alle definierten Module unter Einhaltung der Voraussetzungen belegt werden.

Zum Beispiel sollen folgende Module besucht werden:

Modul	Voraussetzungen
DB1	OO
DB2	DB1
Math	-
OO	-
AD1	OO
CPI	OO Math
Thesis	DB2 SE2 UI2
SE1	AD1 CPI DB1
SE2	DB1 SE1 UI1
UI1	AD1
UI2	UI1

Es gibt nun die Möglichkeit, die Module nach folgendem Plan zu besuchen:

Semester 1	Semester 2	Semester 3	Semester 4	Semester 5
Math	DB1	DB2	SE2	Thesis
OO	AD1	SE1	UI2	
	CPI	UI1		

Zur Einfachheit wird angenommen, dass jedes Semester alle Module stattfinden und beliebig viele Module pro Semester belegt werden dürfen.

Entwickeln Sie ein Programm, das den Studienplan berechnet.

In der Vorlage steht bereits eine Hilfsklasse bereit, welche die Module und deren Abhängigkeiten von einer Beispiel-Datei `StudyCatalogue.txt` einliest. Beachten Sie, dass je nach IDE ein anderes default Working-Directory gewählt wird. Kann das Testprogramm die Beispiel-Datei also nicht finden, können Sie versuchen

den Pfad zur `StudyCatalogue.txt` anzupassen oder die Datei zu verschieben. In der Beispiel-Datei sind die Module wie folgt codiert: Pro Zeile ist zuerst der Name eines Modules angegeben, danach folgt die Liste der Namen der dazugehörigen vorausgesetzten Module.

```
DB1 OO
DB2 DB1
Math
OO
AD1 OO
CPI OO Math
Thesis DB2 SE2 UI2
SE1 AD1 CPI DB1
SE2 DB1 SE1 UI1
UI1 AD1
UI2 UI1
```

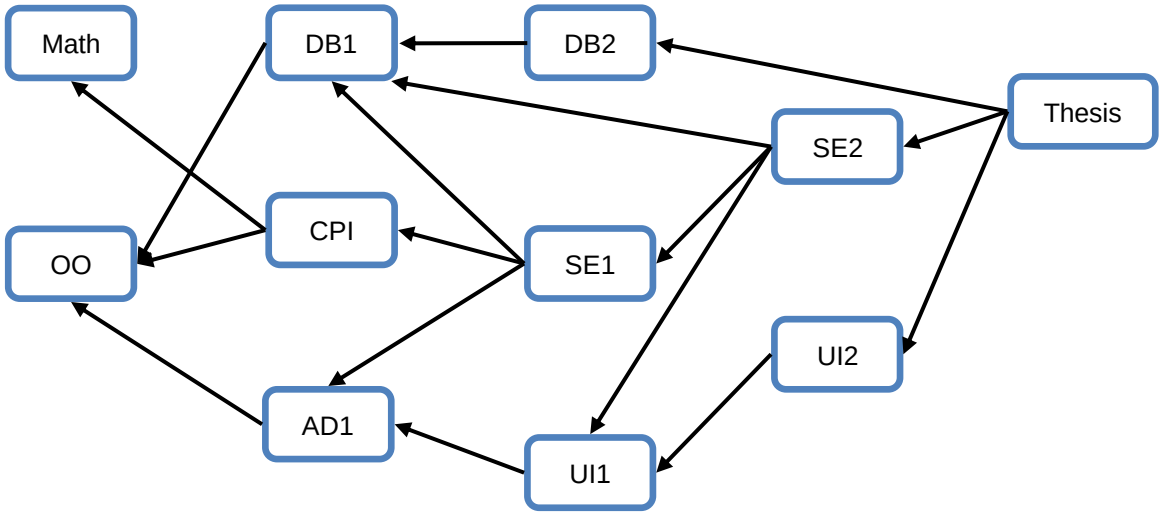
Die Ausgabe können Sie einfach gestalten, zum Beispiel:

```
Semester 1: Math OO
Semester 2: DB1 AD1 CPI
Semester 3: DB2 SE1 UI1
Semester 4: SE2 UI2
Semester 5: Thesis
```

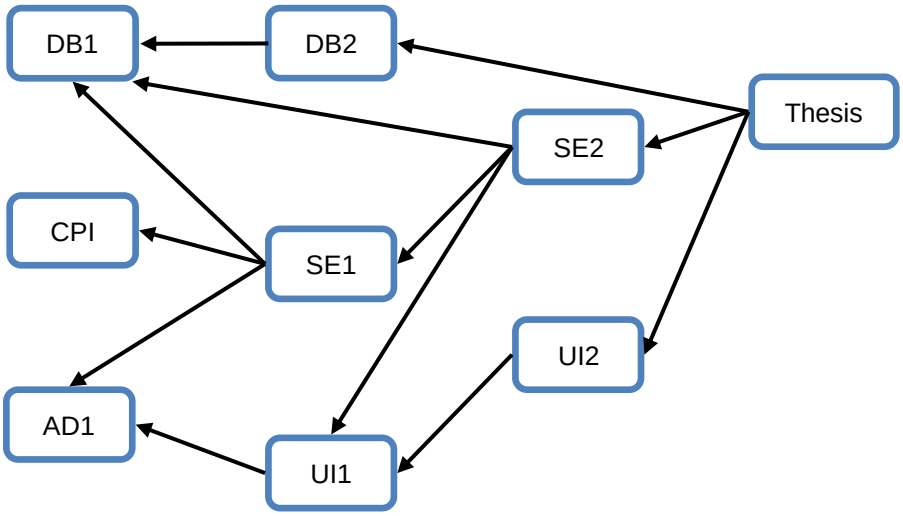
Tipp zur Lösung (sogenanntes topologisches Sortieren):

1. Erstellen Sie für jedes Modul genau ein Objekt. Das Objekt enthält eine Collection seiner vorausgesetzten Module bzw. der Namen seiner vorausgesetzten Module (letzteres ist einfacher).

   Beispiel der Objekt-Struktur:



2. Suchen Sie die Module mit leerer Voraussetzungs-Collection. Diese können im aktuellen Semester besucht werden. Beispiel: Am Anfang haben „Math“ und „OO“ keine Voraussetzungen und können also belegt werden.
3. Speichern Sie die im aktuellen Semester besuchten Module ab bzw. geben Sie diese aus:
   `Semester 1: Math OO`
4. Entfernen Sie nun die besuchten Module als Voraussetzung aus den restlichen Modulen. Beispiel: „Math“ und „OO“ werden nun aus der Struktur entfernt. Danach sieht die Struktur so aus:



5. Gehen Sie zum nächsten Semester und wiederholen Sie den Schritt 2, solange es noch Module gibt.

Spezialfall:

- Wenn in Schritt 2 alle Module Voraussetzungen haben, sind die Abhängigkeiten zirkulär und es gibt keinen möglichen Studienplan. Erkennen Sie solche zyklischen Abhängigkeiten in Ihrem Programm und werfen Sie dann eine geeignete Exception.

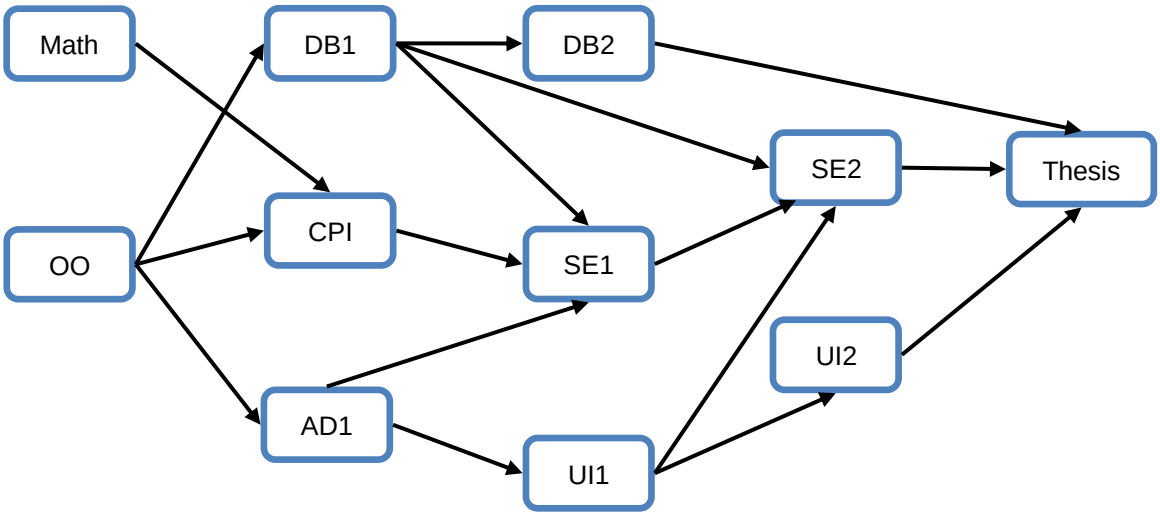
Aufgabe 2: Effiziente topologische Sortierung (fakultativ)

Die Lösung aus Aufgabe 1 lässt sich nicht nur für Studienplanung, sondern grundsätzlich für die Planung von beliebigen Aufgaben mit Abhängigkeiten verwenden (z.B. in der Industrie). Für grössere Eingaben muss der Algorithmus aus Aufgabe 1 jedoch noch effizienter gestaltet werden.

Dazu sind folgende Design-Änderungen hilfreich:

- Ein Objekt speichert nicht mehr die vorausgesetzten Objekte (Vorgänger), sondern verweist nun auf diejenigen Objekte, für die es selbst eine Voraussetzung bildet (Nachfolger).

  Dies entspricht der Umkehrung der Abhängigkeiten (Referenzen) in der Objekt-Struktur:



- Jedes Objekt merkt sich zusätzlich noch die Anzahl seiner vorausgesetzten Objekte, die vorher abgearbeitet werden müssen (Anzahl Vorgänger).

Versuchen Sie nun, eine effizientere Planung zu realisieren. Zur Messung der erreichten Performance-Verbesserung können Sie das Beispiel-File **`LargeCatalogue.txt`** aus der Übungsvorlage verwenden (gleiches Format wie in Aufgabe 1).

Hinweis: Die Laufzeit in Millisekunden können Sie wie folgt messen:

```
long start = System.currentTimeMillis();
/*
   Code, für den die Laufzeit gemessen wird
*/
System.out.println(System.currentTimeMillis() - start);
```