Übungen zur Vorlesung Softwaretechnologie

- Wintersemester 2018/19 -

Dr. Günter Kniesel

Übungsblatt 11

Dies ist das letzte Übungsblatt!

Zu bearbeiten bis: 18.01.2019, 16 Uhr

Bitte fangen Sie **frühzeitig** mit der Bearbeitung an, damit wir Ihnen bei Bedarf helfen können. Checken Sie die Lösungen zu den Aufgaben bitte in Ihr Repository ein, „Erklärungen" bitte als Textdatei. Fragen zu Übungsaufgaben respektive zur Vorlesung können Sie auf der Mailingliste swt-tutoren@lists.iai.uni-bonn.de, bzw. swt-vorlesung@lists.iai.uni-bonn.de stellen.

Aufgabe 1. *Split Objects* (8 Punkte)

Zur weiteren Flexibilisierung des Studiums (sprich: Sanierung der maroden Länderhaushalte) hat die Kultusministerkonferenz vorgeschlagen zu jedem in Zukunft mit Studiengebühren belegten Studiengang (Bachelor, Master, Diplom) zusätzliche kostenpflichtige Studien-Optionen anzubieten:

i.	*Advertising Medium Option*	mit Studienbeitragszuschlag von	75.00 €
ii.	*Tutorial Option*	mit Studienbeitragszuschlag von	700.00 €
iii.	*Crisis Hotline Option*	mit Studienbeitragszuschlag von	200.00 €
iv.	*Gratuity Option*	mit Studienbeitragszuschlag von	16.000.00 €

Dabei sollen folgende Regeln gelten:

a. Jeder Studiengang soll mit beliebig vielen der obigen Optionen kombinierbar sein.
b. Optionen können auch mehrfach belegt werden (z.B eine Mathe-CrisisHotline, eine SWT-CrisisHotline und eine LifeCrisisHotline).
c. Der Beitrag für einen Studiengang mit Optionen ist die Summe der Beitragszuschläge aller (auch mehrfach) gewählten Optionen und des Basis-Beitrags für den Studiengang.

Als ersten Schritt zur Umsetzung dieser Pläne existiert ein Eclipse-Projekt *StudienPlan* im readonly-Ordner des Repositories, welches die Basis-Studiengänge modelliert. Die Klassen enthalten Operationen zur Abfrage der Kostenbeiträge und der Beschreibung eines Studiengangs.

a) Erweitern Sie das Modell (mit Hilfe eines Entwurfsmusters) um Klassen für die Studien-Optionen, die Methoden zur Berechnung des Kostenbeitrages und einer Beschreibung enthalten. Zu obigen fachlichen Anforderungen soll das erweiterte Modell auch folgende Anforderungen erfüllen, die sich die für die Umsetzung Verantwortlichen ausgedacht haben:

   a. Ein Basis-Studiengang und eine Option bilden selbst wieder einen Studiengang.
   b. Das spätere Hinzufügen von Basis-Studiengängen und Studien-Optionen soll möglich sein, ohne bereits existierenden Code anpassen zu müssen.

c. Die Beschreibung ist die Verkettung der Beschreibungen aller gewählten Optionen z. B.: `Bachelor-Studium, Tutorial Option, Advertising Medium Option`

d. Die *Gratuity Option* soll diskret behandelt werden, das heißt, nicht in der Beschreibung auftauchen. Auch ihre Kosten sollen nicht sichtbar werden.

b) Schreiben Sie ein Programm, das einen Bachelor-Studiengang mit einer Tutorial-, zwei CrisisHotline- und einer Gratuity-Option kombiniert und geben Sie dann die Beschreibung des resultierenden Studiengangs und die Gesamtkosten auf der Konsole aus.

c) Gehen Sie nun davon aus, dass jeder Studiengang zu jedem Zeitpunkt mit nur einer einzigen (aber änderbaren) Studien-Option kombiniert werden darf (ansonsten gelten die Anforderungen aus Teilaufgabe a). Würde sich dadurch ein anderes Entwurfsmuster anbieten, um die Aufgabenstellung zu lösen? Begründen Sie stichpunktartig Ihre Antwort.

Aufgabe 2. *Whitebox Test* (10 Punkte)

Gegeben sei die folgende Methode `sortiere`, welche mittels Bubblesort ein Feld von Variablen des Typs `int` sortiert.

```
public int[] sortiere(int[] bestand) {                    // Anweisungsnr.
      boolean change = true;                              //  1
      if (bestand.length > 1) {                           //  2
            while (change) {                              //  3
                  change = false;                         //  4
                  for (int i = bestand.length - 1;        //  5
                              i > 0;                      //  6
                              i--) {                      //  7
                        int i1 = bestand[i];              //  8
                        int i2 = bestand[i - 1];          //  9
                        if (i1 < i2) {                    // 10
                              bestand[i] = i2;            // 11
                              bestand[i - 1] = i1;        // 12
                              change = true;              // 13
                        }
                  }
            }
      }
      return bestand;                                     // 14
}
```

a) Entwerfen Sie für die Methode `sortiere` einen Kontrollflussgraphen.

b) Geben Sie ein Feld mit Eingabewerten an, das nötig ist, um eine Anweisungsüberdeckung zu erreichen. Schreiben Sie die Reihenfolge auf, in der die Anweisungen getestet werden.

c) Erreichen Sie mit diesem Feld an Eingabewerten auch eine Verzweigungsabdeckung? Begründen Sie kurz Ihre Antwort. Falls ja, geben Sie eine Codemodifikation an, mit der die Verzweigungsabdeckung nicht mehr gegeben wäre. Falls nicht, geben Sie ein weiteres Eingabewerte-Feld an, sodass auch die übrigen Zweige überdeckt werden. Notieren Sie zu diesem neuen Testfall wieder die Reihenfolge der durchgeführten Anweisungen.

d) Wie viele Pfade gibt es? Kurze Begründung.

e) Formulieren Sie ein minimales Programm, für das mindestens zwei verschiedene Testfälle notwendig sind, um eine Anweisungsüberdeckung zu erreichen.

Aufgabe 3. *Blackbox Test* (12 Punkte)

Gegeben sei folgende Schnittstelle

```
interface DateParser {
     java.util.Date parseDate(String input);
}
```

und folgende Dokumentation der Methode `parseDate(String input)`:

```
/**
 * Parses gracefully a date string of the form <i>day-month-year</i><br />
 * <br />
 * <i>day</i> and <i>month</i> can contain one or two digits<br />
 * <i>year</i> can contain two or four digits, two digit years are in 21st
     century<br />
 * <br />
 * If <i>month</i> is below <i>1</i>, January is assumed, if it is above
     <i>12</i>, December is assumed.<br />
 * If <i>day</i> is below <i>1</i>, the first day of the month is assumed,
     if it is greater than the last valid day of month, the last day of
     the month is assumed.
 *
 * @param input The date string to parse
 * @return the parsed date (with time set to 12:00:00) or <i>null</i>, if
     the input string is invalid
 */
Date parseDate(String input)
```

a) Überlegen Sie, welche Äquivalenzklassen auftreten können (s. Skript, Kapitel 10a. Testen, Seite 20). Geben Sie für fünf Äquivalenzklassen (von korrekten oder falschen Eingaben) einen Eingabestring und das erwartete Methoden-Ergebnis an.

b) Im Repository `ssh://gitolite-se@git.iai.uni-bonn.de/swt2018_readonly` finden Sie das Archiv „DateParser.zip“. Darin ist in der Datei `lib/GemaltoDateParser.jar` der ByteCode einer Klasse enthalten, die das Interface `DateParser` implementiert. Im Ordner `tests` finden Sie die noch unvollständige Klasse `GemaltoDateParserTest` die die Methode `parseDate` aus der Klasse `GemaltoParser` mit *JUnit 4* testet. Vervollständigen Sie diese Tests indem Sie jede in (a) identifizierte Fehlerart in einen Testfall umsetzen.
✍ ***Hinweis:*** Nutzen Sie in den Tests die Hilfs-Methode `makeDate(…)`.
→ ***Bonus*** *(2 Punkte):* Welchen Fehler hat die getestete Klasse?