Übungen zur Vorlesung Softwaretechnologie

- Wintersemester 2017/18 -

Dr. Günter Kniesel

Übungsblatt 11 – Lösungen

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:

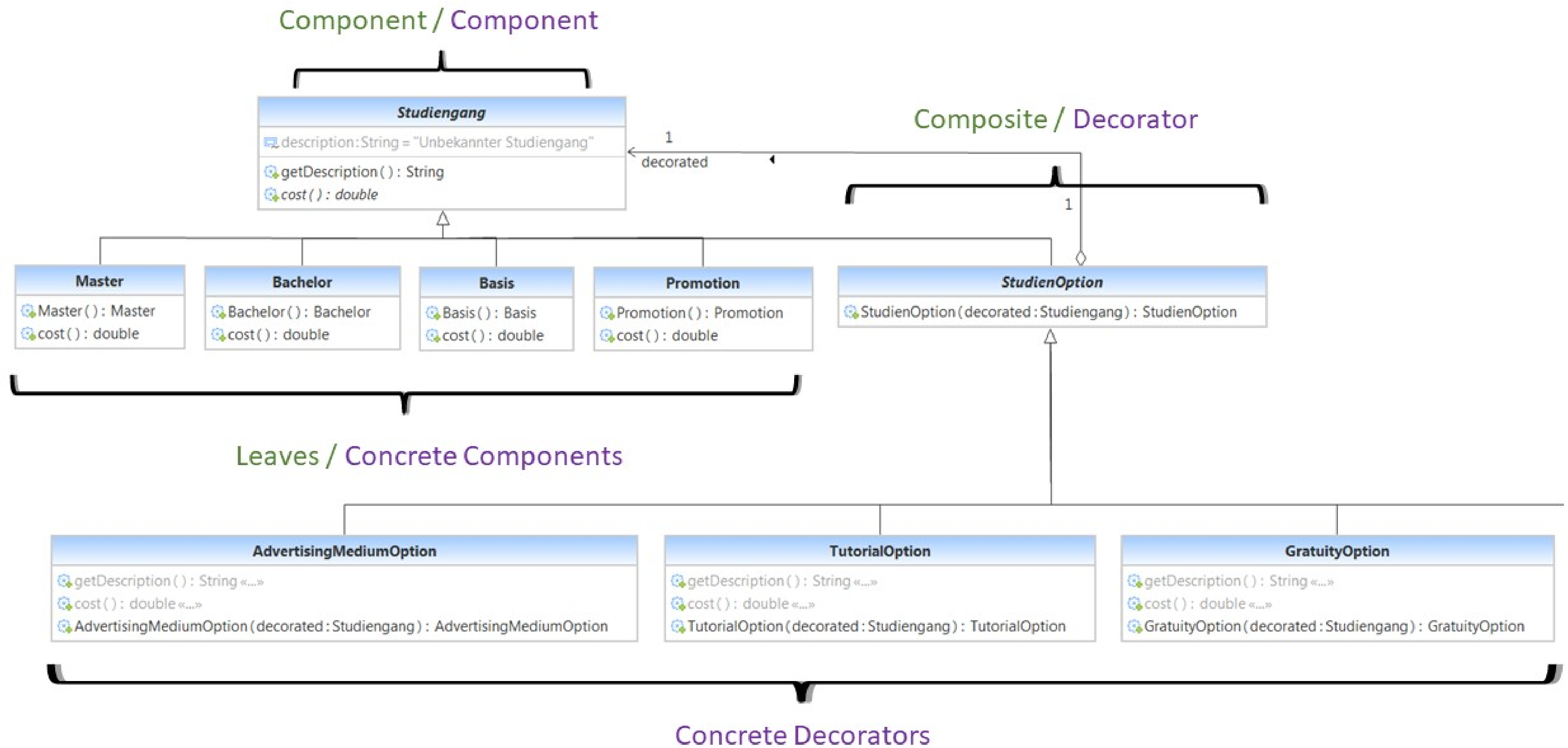
   a. Ein Basis-Studiengang und eine Option bilden selbst wieder einen Studiengang.

   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.

Anwendung des Decorator Entwurfsmusters:



(*CrisisHotlineOption* aus Platzgründen abgeschnitten)

```java
public interface StudiengangI {
	public String getDescription() ;

	public double cost() ;
}


public abstract class StudienOption implements StudiengangI {

	StudiengangI  decorated;

	public StudienOption(StudiengangI  decorated) {
		this.decorated = decorated;
	}

	@Override
	public String getDescription() {
		return decorated.getDescription();
	}

	@Override
	public double cost() {
		return decorated.cost();
	}

	@Override
	public boolean equals(Object o) {
		if (!StudiengangI.class.isInstance(o)) {
			return false;
		}

		if (Studiengang.class.isInstance(o)) {
			return decorated.equals(o);
		} else {
			return o.equals(decorated);
		}
	}
}


public class TutorialOption extends StudienOption {
	public TutorialOption(StudiengangI  decorated) {
		super(decorated);
	}
	@Override
	public String getDescription() {
		return decorated.getDescription()+ ", Tutorial Option";
	}
	@Override
	public double cost() {
		return decorated.cost() + 700.00;
	}
}
```

```java
public class CrisisHotlineOption extends StudienOption {
	String topic;

	public CrisisHotlineOption(String topic, StudiengangI  decorated) {
		super(decorated);
		this.topic = topic;
	}
	@Override
	public String getDescription() {
		return decorated.getDescription()+ ", Crisis Hotline Option";
	}
	@Override
	public double cost() {
		return decorated.cost() + 200.00;
	}
}


public class AdvertisingMediumOption extends StudienOption {
	public AdvertisingMediumOption(StudiengangI  decorated) {
		super(decorated);
	}
	@Override
	public String getDescription() {
	  return decorated.getDescription()+ ", Advertising Medium Option";
	}
	@Override
	public double cost() {
		return decorated.cost() + 75.00;
	}
}


public class GratuityOption extends StudienOption {
	public GratuityOption(StudiengangI decorated) {
		super(decorated);
	}
	@Override
	public String getDescription() {
		return decorated.getDescription();
	}
	@Override
	public double cost() {
		return decorated.cost(); // + 16000.00;
	}
}
```

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.

```
public class StudienDemo {
  public static void main(String[] args) {
        StudiengangI s =  new GratuityOption(
                            new CrisisHotlineOption("Mathe",
                               new CrisisHotlineOption("SWT",
                                   new TutorialOption(
                                       new Bachelor()
                                   )
                               )
                            )
                          );
        System.out.println(
            String.format("Studiengang: %s", s.getDescription()));
        System.out.println(
            String.format("Studienbeitrag: %.2f €", s.cost()));
  }
}
```

c) Gehen Sie nun davon aus, dass jeder Basis-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.

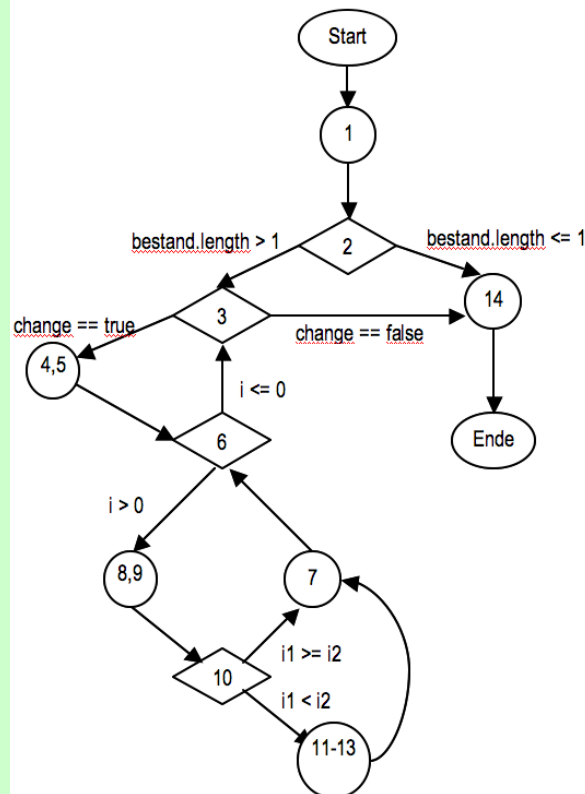
Das Strategie-Entwurfsmuster ist bei Kombination mit nur einer Option sinnvoller, da man auf den Basis-Studiengang und die Option zugreifen kann, ohne Verschachtelungen traversieren zu müssen. Ferner „sehen" dann die Klienten das Studiengang-Objekt direkt (anstatt von Decorator-Objekten), so dass bei Identitätstests (==) keine Verwirrungen auftreten.

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.

[4,3] oder jedes andere Feld mit mindestens zwei Zahlen und für das gilt, dass mindestens zwei Zahlen in der falschen Reihenfolge vorliegen.

Die Anweisungsreihenfolge ist (von 1 bis 14):

1	change wird auf true gesetzt
2	if: bestand.length > 1 wird zu true ausgewertet
3	while: change wird zu true ausgewertet
4	change wird auf false gesetzt
5	for: i wird initialisiert (i = 1)
6	for: i > 0 wird zu true ausgewertet
8	i1 wird auf 3 gesetzt
9	i2 wird auf 4 gesetzt
10	if: i1 < i2 wird zu true ausgewertet
11-13	4 und 3 vertauschen; change wird true
7	for: i wird angepasst (i = 0)
6	for: i > 0 wird zu false ausgewertet
3	while: change wird zu true ausgewertet
4	change wird auf false gesetzt
5	for: i wird initialisiert (i = 1)
6	for: i > 0 wird zu true ausgewertet
8	i1 wird auf 4 gesetzt
9	i2 wird auf 3 gesetzt
10	if: i1 < i2 wird zu false ausgewertet
7	for: i wird angepasst (i = 0)
6	for: i > 0 wird zu false ausgewertet
3	while: change wird zu false ausgewertet
14	bestand wird zurückgegeben

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.

Nein. Verzweigungsabdeckung erfordert zusätzlich ein Feld der Größe 1 als Eingabe.

Begründung: Betrachtet man die Anweisungsreihenfolge aus Aufgabenteil a), so ist ersichtlich, dass die Verzweigungsabdeckung für den if-Block in Zeile 10 bereits erfüllt ist. Für eine vollständige Verzweigungsabdeckung muss schließlich noch der else-Teil für den ersten if-Block in Zeile 2 abgedeckt werden. Dies lässt sich mit einem Feld der Größe 1 als zusätzliche Eingabe erreichen.

d) Wie viele Pfade gibt es (so dass in einem Pfad eine Schleife maximal 1 mal durchlaufen wird)? Kurze Begründung.

- Es gibt im Graphen 5 Pfade:

  1. 1 → 2 → 14
  2. 1 → 2 → ~~3 → 14~~
  3. 1 → 2 → 3 → 4,5 → ~~6 → 3~~ → 14
  4. 1 → 2 → 3 → 4,5 → 6 → 8,9 → 10 → 7 → 6 → 3 → 14
  5. 1 → 2 → 3 → 4,5 → 6 → 8,9 → 10 → 11-13 → 7 → 6 → 3 → 14

- Allerdings wird 2) zur Laufzeit nie durchlaufen: Weil zu diesem Zeitpunkt noch "change == true" gilt, kann die Kante 3 → 14 nicht verfolgt werden. Genauso wird 3) nicht durchlaufen, weil der Schritt 6 → 3 nicht möglich ist, da zu diesem Zeitpunkt das Array die Länge 2 hat und somit die Schleifenvariable i >= 1 ist.

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

```
public void myMath(int a){
      if (a >= 0)
            a--;
      else
            a++;
}
```

Aufgabe 1. *Blackbox Test* (12 Punkte)

Gegeben sei folgende Schnittstelle:

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

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

Erzeugt ein Date-Objekt aus einem Datumsstring der Form *Tag-Monat-Jahr*.

- *Tag* und *Monat* können ein- order zweistellig sein, evtl. mit führender 0.
- *Jahr* kann zwei- oder vierstellig sein, zweistellige Angaben beziehen sich auf das 21.-te Jahrhundert.
- Wenn *Monat* kleiner *1* ist, wird Januar angenommen, bei Werten über *12* Dezember.
- Wenn *Tag* kleiner *1* ist wird der Monatserste angenommen, bei Werten größer dem zuletzt gültigen Tag der Monatsletzte.

Parameter:
Der Datumsstring, der interpretiert werden soll
Liefert zurück:
Das interpretierte Datum (mit dem Uhrzeitwert 12:00:00) oder *null* bei ungültiger Eingabe

Überlegen Sie, welche Äquivalenzklassen auftreten können (s. Skript, Kapitel 9a. Testen, Seite 20). Geben Sie für fünf Äquivalenzklassen (von korrekten oder falschen Eingaben) einen Eingabestring und das erwartete Methoden-Ergebnis an. **(5 Punkte)**

- Korrekte vollständige Eingabe
  - Testet ob richtig geparst wird
  - Voraussetzung: Gültiges Datum der Form dd-mm-yyyy
  - Beispiel: "`24-10-2010`" → *24.10.2010, 12:00:00*
- Korrekte vollständige Eingabe mit führenden Nullen
  - Testet ob richtig geparst wird
  - Voraussetzung: Gültiges Datum der Form 0d-0m-yyyy
  - Beispiel: "`04-01-2010`" → *4.1.2010, 12:00:00*
- Korrekte vollständige Eingabe mit verkürztem Tag/Monat
  - Testet ob richtig geparst wird
  - Voraussetzung: Gültiges Datum der Form d-m-yyyy
  - Beispiel: "`4-1-2010`" → *4.1.2010, 12:00:00*
- Korrekte vollständige Eingabe mit zweistelliger Jahreszahl
  - Testet ob richtig geparst wird
  - Voraussetzung: Gültiges Datum der Form dd-mm-yy
  - Beispiel: "`24-01-10`" → *24.1.2010, 12:00:00*
- Jahr 0 (Extremwert)
  - Testet ob richtig geparst wird
  - Voraussetzung: Gültiges Datum der Form dd-mm-0000
  - Beispiel: "`24-01-0000`" → *24.1.0, 12:00:00*
- Jahr 9999 (Extremwert)
  - Testet ob richtig geparst wird
  - Voraussetzung: Gültiges Datum der Form dd-mm-9999
  - Beispiel: "`24-01-9999`" → *24.1.9999, 12:00:00*

- Monat < 1
    - Testet ob richtig angepasst wird
    - Voraussetzung: Datum der Form dd-mm-yyyy mit mm < 0
    - Beispiel: "`24-00-2010`" → *24.1.2010, 12:00:00*
- Monat > 12
    - Testet ob richtig angepasst wird
    - Voraussetzung: Datum der Form dd-mm-yyyy mit mm > 12
    - Beispiel: "`24-13-2010`" → *24.12.2010, 12:00:00*
- Tag < 1
    - Testet ob richtig angepasst wird
    - Voraussetzung: Datum der Form dd-mm-yyyy mit dd < 0
    - Beispiel: "`00-01-2010`" → *1.1.2010, 12:00:00*
- Tag > 31 (z.B Januar, März etc.)
    - Testet ob richtig angepasst wird
    - Voraussetzung: Datum der Form dd-mm-yyyy mit dd > 31 und mm = 1
    - Beispiel: "`32-01-2010`" → *31.1.2010, 12:00:00*
- Tag > 30 (z.B April, Juni etc.)
    - Testet ob richtig angepasst wird
    - Voraussetzung: Datum der Form dd-mm-yyyy mit dd > 30 und mm = 4
    - Beispiel: "`31-04-2010`" → *30.4.2010, 12:00:00*
- Tag > 28 (Februar, nicht Schaltjahr)
    - Testet ob richtig angepasst wird
    - Voraussetzung: Datum der Form dd-mm-yyyy mit dd > 28 und mm = 2 und yyyy ist kein Schaltjahr
    - Beispiel: "`29-02-2010`" → *29.2.2010, 12:00:00*
- Tag > 29 (Februar, Schaltjahr)
    - Testet ob richtig angepasst wird
    - Voraussetzung: Datum der Form dd-mm-yyyy mit dd > 29 und mm = 2 und yyyy ist Schaltjahr
    - Beispiel: "`30-02-2012`" → *29.2.2012, 12:00:00*
- Tag > 28 (Februar, Säkularjahr)
    - Testet ob richtig angepasst wird
    - Voraussetzung: Datum der Form dd-mm-yyyy mit dd > 28 und mm = 2 und yyyy ist Säkularjahr
    - Beispiel: "`29-02-2100`" → *28.2.2100, 12:00:00*
- Tag > 29 (Februar, Säkularjahr durch 400 teilbar)
    - Testet ob richtig angepasst wird
    - Voraussetzung: Datum der Form dd-mm-yyyy mit dd > 29 und mm = 2 und yyyy ist Säkularjahr und durch 400 teilbar
    - Beispiel: "`30-02-2000`" → *29.2.2000, 12:00:00*
- Kein Tag angegeben
    - Testet ob keine Ausnahme auftritt
    - Voraussetzung: Datum der Form -mm-yyyy
    - Beispiel: "`-01-2010`" → *null*
- Kein Monat angegeben
    - Testet ob keine Ausnahme auftritt
    - Voraussetzung: Datum der Form dd--yyyy
    - Beispiel: "`24--2010`" → *null*
- Kein Jahr angegeben
    - Testet ob keine Ausnahme auftritt

- Voraussetzung: Datum der Form dd-mm-
    - Beispiel: "24-01-" → *null*
- Kein Jahr angegeben (ohne Bindestrich)
    - Testet ob keine Ausnahme auftritt
    - Voraussetzung: Datum der Form dd-mm
    - Beispiel: "24-01" → *null*
- Zuviel Bindestriche
    - Testet ob keine Ausnahme auftritt
    - Voraussetzung: Datum der Form dd-mm-yyyy-
    - Beispiel: "24-01-2010-" → *null*
- Ungültige Zeichen in der Eingabe
    - Testet ob keine Ausnahme auftritt
    - Voraussetzung: Eingabe mit ungültigen Zeichen
    - Beispiel: "24 – 01 – 2010" → *null*
- Übergabe von null
    - Testet ob ein leerer String zurückgegeben wird
    - Voraussetzung: null wird als Eingabe übergeben
    - Beispiel: `null` → *null*
- Übergabe des Leerstrings
    - Testet ob ein leerer String zurückgegeben wird
    - Voraussetzung: der Leerstring wird als Eingabe übergeben
    - Beispiel: "" → *null*

Im Repository `ssh://gitolite-se@git.iai.uni-bonn.de/swt2016_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:*** *Sie können in den Tests die Hilfs-Methode* `makeDate` *(…) verwenden.*
→***Bonus*** *(2 Punkte):* Welchen Fehler hat die getestete Klasse? **(7 Punkte)**

```java
import java.util.Date;
import java.util.GregorianCalendar;

import org.junit.Before;
import org.junit.Test;

import static org.junit.Assert.*;

public class GemaltoDateParserTest {

	protected DateParser systemUnderTest;

	@Before
	public void setUp() {
		systemUnderTest = new GemaltoDateParser();
	}

	/* add your tests here */

	@Test
	public void testStandardInput() {
		assertEquals(makeDate(2010,10,24),
				systemUnderTest.parseDate("24-10-2010"));
	}
```

```java
	@Test
	public void testLeadingZeros() {
		assertEquals(makeDate(2010,1,4),
				systemUnderTest.parseDate("04-01-2010"));
	}

	@Test
	public void testShort() {
		assertEquals(makeDate(2010,1,4),
				systemUnderTest.parseDate("4-1-2010"));
	}

	@Test
	public void testShortYear() {
		assertEquals(makeDate(2010,1,24),
				systemUnderTest.parseDate("24-1-10"));
	}

	@Test
	public void testYearZero() {
		assertEquals(makeDate(0,1,24),
				systemUnderTest.parseDate("24-01-0000"));
	}

	@Test
	public void testYear9999() {
		assertEquals(makeDate(9999,1,24),
				systemUnderTest.parseDate("24-01-9999"));
	}

	@Test
	public void testMonthBelow1() {
		assertEquals(makeDate(2010,1,24),
				systemUnderTest.parseDate("24-00-2010"));
	}

	@Test
	public void testMonthOver12() {
		assertEquals(makeDate(2010,12,24),
				systemUnderTest.parseDate("24-13-2010"));
	}

	@Test
	public void testDayBelow1() {
		assertEquals(makeDate(2010,1,1),
				systemUnderTest.parseDate("00-01-2010"));
	}

	@Test
	public void testDayOver31() {
		for (int i: new int[]{1,3,5,7,8,10,12})
			assertEquals(makeDate(2010,i,31),systemUnderTest.
				parseDate(String.format("32-%02d-2010",i)));
	}

	@Test
	public void testDayOver30() {
		for (int i: new int[]{4,6,9,11})
			assertEquals(makeDate(2010,i,30),systemUnderTest.
				parseDate(String.format("31-%02d-2010",i)));
	}

	@Test
	public void testFebDayOver28() {
```

```java
		assertEquals(makeDate(2010,2,28),
				systemUnderTest.parseDate("29-02-2010"));
	}

	@Test
	public void testFebDayOver29LeapYear() {
		assertEquals(makeDate(2012,2,29),
				systemUnderTest.parseDate("30-02-2012"));
	}

	@Test
	public void testFebDayOver28Secular() {
		assertEquals(makeDate(2100,2,28),
				systemUnderTest.parseDate("29-02-2100"));
	}

	@Test
	public void testFebDayOver29SecularBy400() {
		assertEquals(makeDate(2000,2,29),
				systemUnderTest.parseDate("30-02-2000"));
	}


	@Test
	public void testNoDay() {
		assertNull(systemUnderTest.parseDate("-01-2010"));
	}

	@Test
	public void testNoMonth() {
		assertNull(systemUnderTest.parseDate("24--2010"));
	}

	@Test
	public void testNoYear() {
		assertNull(systemUnderTest.parseDate("24-01-"));
	}

	@Test
	public void testNoYearNoDash() {
		assertNull(systemUnderTest.parseDate("24-01"));
	}

	@Test
	public void testAddDashs() {
		assertNull(systemUnderTest.parseDate("24-01-2010-"));
	}

	@Test
	public void testInvalidSpaces() {
		assertNull(systemUnderTest.parseDate("24 - 01 - 2010"));
	}

	@Test
	public void testNull() {
		assertNull(systemUnderTest.parseDate(null));
	}

	@Test
	public void testEmpty() {
		assertNull(systemUnderTest.parseDate(""));
	}

	/* test helper */
```

```
	/**
	 * Returns a Date object representing 12:00 of a given date
	 *
	 * @param year the year of the date
	 * @param month the month of the date (1-12)
	 * @param day the day of the date
	 * @return the Date object
	 */
	private Date makeDate(int year, int month, int day) {
		return new GregorianCalendar(year,
					month-1,day,12,0,0).getTime();
	}
}
```

Antwort auf die Bonus-Frage: Die gegebene Klasse interpretiert zweistellige Jahreszahlen ab 2010 nicht korrekt.