Rheinische Friedrich-Wilhelms-Universität Bonn
Institut für Informatik III

Übungen zur Vorlesung Softwaretechnologie

- Wintersemester 2019/20 -

Dr. Günter Kniesel

Übungsblatt 11

Zu bearbeiten bis: 03.01.2020, 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.

Praktische Aufgaben

Aufgabe 1. *Festtagsbedingter Einsatz von Entwurfsmustern* **15 Punkte**

Ein reichlich geschmückter Weihnachtsbaum besteht ausschließlich aus *Baumelementen*, d.h. aus *Zweigen* (an denen weitere immer feiner verästelte Zweige wachsen können) und am Ende eines Zweiges evtl. *Baumschmuck*. Baumschmuck können *Engel*, *Kugeln*, *Sterne* oder *Kerzen* sein.

a) **(7,5 Punkte)** Modellieren Sie den Weihnachtsbaum in Java unter Zuhilfenahme eines geeigneten Entwurfsmusters. Markieren Sie dabei die Rollen des Entwurfsmusters in dem zur entsprechenden Klasse gehörenden *JavaDoc*-Kommentar. Gehen Sie davon aus, dass jedes Baumelement (Zweig und Baumschmuck) eine *draw()*-Methode hat.

Bewertung: Baumelement 2P + Zweig 2P + Baumschmuck 1,5P + Engel, Kugel,… 2 P

```
/**
 * Rolle im Composite Pattern: Component
*/
public interface Baumelement {
      void add(Baumelement e);                // 0,5
      void remove(Baumelement e);             // 0,5
      List<Baumelement> getChildren();        // 0,5
      String draw();                          // 0,5
}
```

```java
/**
 * Rolle im Composite-Pattern: Composite
 *  4x 0,5 Punkte in Aufgabe a) und 1,5 in b) für draw()
 */
public class Zweig implements Baumelement {
	private List<Baumelement> children = new ArrayList<>();

	@Override
	public void add(Baumelement e) {
		children.add(e);
	}
	@Override
	public void remove(Baumelement e) {
		children.remove(e);
	}
	@Override
	public List<Baumelement> getChildren() {
		return children;
	}
	@Override
	public String draw() {
		StringBuffer sb = new StringBuffer("Zweig {");
		for(Baumelement e: children) {
			sb.append("\n\t" + e.draw().replace(„\n“, “\n\t“));
		}
		sb.append("\n}");
		return sb.toString();
	}
}
```

```java
/**
 * Rolle im Composite-Pattern: Leaf
 * 3x 0,5 Punkte in Aufgabe a)
 */
public abstract class Baumschmuck implements Baumelement {
	@Override
	public void add(Baumelement e) {
		throw new NoSuchMethodError();
	}
	@Override
	public void remove(Baumelement e) {
		throw new NoSuchMethodError();
	}
	@Override
	public List<Baumelement> getChildren() {
		throw new NoSuchMethodError();
	}
}
```

```
//  4x 0,5 Punkte in Aufgabe a) für Engel, Kugel, …
public class Engel extends Baumschmuck {
	@Override
	public String draw() {
		return "Engel";
	}
}

public class Kugel extends Baumschmuck {
	@Override
	public String draw() {
		return "Kugel";
	}
}

public class Stern extends Baumschmuck {
	@Override
	public String draw() {
		return "Stern";
	}
}

public class Kerze extends Baumschmuck {
	@Override
	public String draw() {
		return "Kerze";
	}
}
```

b) **(7,5 Punkte)** Entwickeln Sie ein Programm, das – ausgehend vom Stamm – einen zufällig generierten Weihnachtsbaum mit verschiedenem Baumschmuck erzeugt. Achten Sie dabei darauf, dass in unserem Kulturkreis die Zweigrekursionstiefe genau 3 beträgt und jeder Zweig zwischen 1 und 5 Kindelemente haben kann. Geben Sie eine Beschreibung ihres Baumes auf der Konsole aus, indem Sie die *draw()*-Methoden der Baumelemente geeignet implementieren und auf dem Stamm aufrufen.

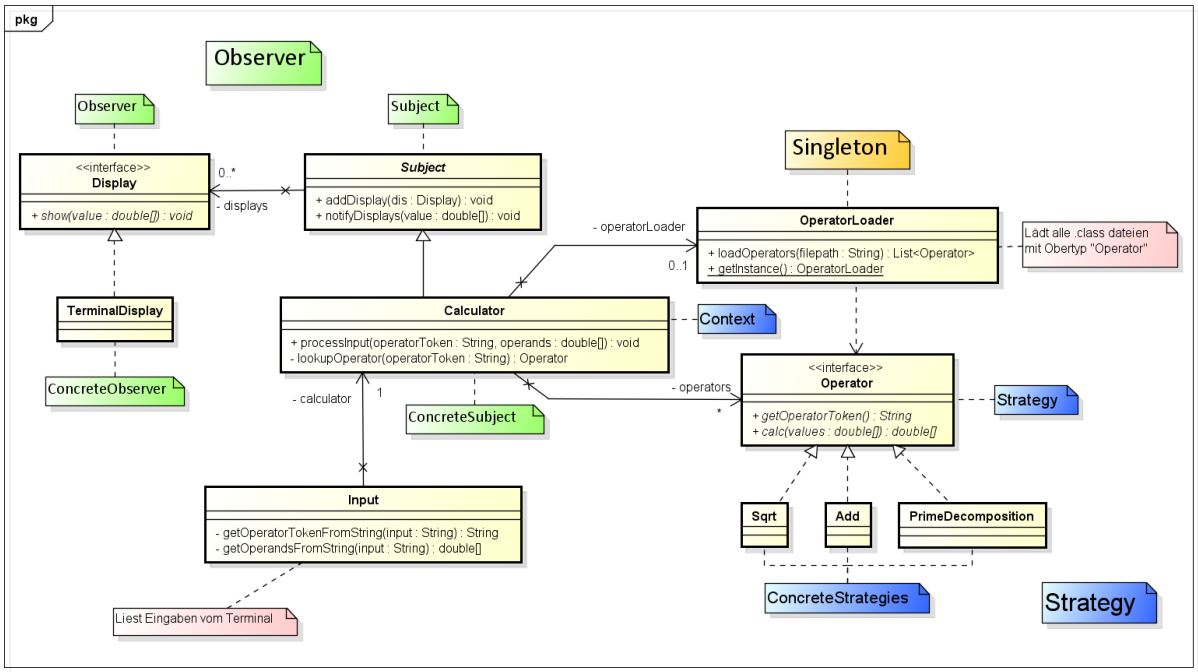
```
public class WeihnachtsbaumSimulator {
	// 0,5 Punkte
	private static Random rand = new Random();

	// 1 Punkt
	private static final int ENGEL = 0;
	private static final int KERZE = 1;
	private static final int KUGEL = 2;
	private static final int STERN = 3;

	// 4 Punkte
	private static Baumelement generiereZweig(int ebene) {
		Zweig z = new Zweig();
		int childrenCount = rand.nextInt(5);
```

```
            for (int i = 0; i <= childrenCount; i++) {
                Baumelement element;
                if (ebene < 3)
                    element = generiereZweig(ebene+1);
                else {
                    switch (rand.nextInt(4)) {
                    case ENGEL:
                        element = new Engel();
                        break;
                    case KERZE:
                        element = new Kerze();
                        break;
                    case KUGEL:
                        element = new Kugel();
                        break;
                    case STERN:
                        element = new Stern();
                        break;
                    default:
                        element = null;
                    }
                }
                z.add(element);
            }
            return z;
        }
        // 0,5 Punkte
        public static void main(String[] args) {
            Baumelement baum = generiereZweig(0);
            System.out.println(baum.draw());
        }
}
```

Aufgabe 2. *Realisierung von Modellen* **10 + 1 Punkte**

Es soll in folgender Aufgabe ein Taschenrechner mit PlugIn-Fähigkeiten modelliert werden.

a) **(2 Punkte)** Analysieren Sie das gegebene Klassendiagramm. Identifizieren Sie, welche Entwurfsmuster in diesem Modell verwendet wurden (die farbige Unterscheidung hat hierfür keinerlei Bedeutung). **Hinweis**: Im Diagramm sind die Attribute der Klassen bewusst weggelassen. Das heißt nicht, dass es keine gibt.



Pro Entwurfsmuster 1P

Für das Strategy-Entwurfsmuster wird kein Punkt vergeben, da dieses unbewusst eingesetzt worden ist und zu spät von den Tutoren erkannt wurde.

b) **(6 Punkte)** Nun soll das Modell in Java umgesetzt werden. Ein Teil ist bereits im Projekt „Taschenrechner“ aus dem readonly-Repository (ssh://gitolite-se-swt@git.iai.uni-bonn.de/swt2019_readonly) vorgegeben. Vervollständigen Sie das Projekt, um dem Klassendiagramm gerecht zu werden und insbesondere die eingesetzten Entwurfsmuster zu realisieren. Ergänzen Sie ggf. fehlende Klassen oder Elemente. Beachten Sie zusätzlich die folgenden Informationen, um Ihre Lösung zu testen:

1. Alle Operationen (*Add*, *PrimeDecomposition*, …) werden als *.class*-Dateien vom vorgegebenen OperatorLoader aus dem *operations* Ordner im *src*-Verzeichnis des Projekts geladen.
2. Führen Sie zum Starten die *main* Methode der Klasse *Input* aus. Der Befehl „exit“ beendet den Test.
3. Verwenden Sie zum Testen die gegebenen *Add* und *PrimeDecomposition* Operationen (Token: Add (*+*), PrimeDecomposition (*prime*)). Diese liegen bereits als *.class* Dateien vor.
4. Der Einfachheit halber wird die Präfix-Notation für Eingaben verwendet (z.B.: „+ 2 3“ oder „prime 18“).

Bewertung: Klassen Input und Calculator korrekt vervollständigt 1,5P + Klasse Subject 2,5P + Display 1P + TerminalDisplay 1P

```
public class Input {
    private static final String EXIT_CMD = "exit";
    private Calculator calculator;
    public static void main(String[] args) {
        new Input().parseInput();
    }
    private Input() {
        calculator = new Calculator();
        calculator.addDisplay(new TerminalDisplay());     // 0,5 P
    }
    private void parseInput() {
        BufferedReader breader;
        String lastLine;
        String operatorToken;
        double[] operands;
        breader = new BufferedReader(new InputStreamReader(System.in));
        try {
            while(!(lastLine = breader.readLine()).equals(EXIT_CMD)) {
                if(!isEmptyString(lastLine)) {
                    try {
                        operatorToken = getOperatorTokenFromString(lastLine);
                        operands = getOperandsFromString(lastLine);
                        calculator.processInput(operatorToken, operands);
                    } catch (NumberFormatException ex) {
                        System.err.println("Invalid input");
                    }
```

```
                }
            }
        } catch (IOException e) {
            System.err.println("IO Exception while reading");
        }
    }

    private boolean isEmptyString(String input) {
        return input.trim().equals("");
    }

    private String getOperatorTokenFromString(String input) {
        return input.split("\\s+")[0];
    }

    private double[] getOperandsFromString(String input) throws NumberFormatException {
        String[] splitInput =  input.split("\\s+");
        double[] values = new double[splitInput.length-1];

        for(int i = 0; i < values.length; i++)
            values[i] = Double.parseDouble(splitInput[i+1]);

        return values;
    }
}
```

```
// 5x 0,5 P
public abstract class Subject {

    private List<Display> displays = new ArrayList<>();

    public void addDisplay(Display dis) {
        this.displays.add(dis);
    }

    public void removeDisplay(Display dis) {
        this.displays.remove(dis);
    }

    public void notifyDisplays(double[] value) {
        for(Display dis : displays) {
            dis.show(value);
        }
    }
}
```

```
public class Calculator extends Subject { // 0,5 P

    // Path where to look for the operator class files
    private static final String CLASS_FILEPATH = System.getProperty("user.dir") + "/bin";

    private OperatorLoader operatorLoader;
    private List<Operator> operators;

    public Calculator() {
        this.operatorLoader = OperatorLoader.getInstance();
        this.operators = operatorLoader.loadOperators(CLASS_FILEPATH);
    }

    public void processInput(String operatorToken, double[] operands) {
        try {
            Operator operator = lookupOperator(operatorToken);
```

```
            super.notifyDisplays(operator.calc(operands));  // 0,5 P
        }
        catch(IllegalArgumentException ex) {
            System.err.println(ex.getMessage());
        }
    }

    private Operator lookupOperator(String operatorToken) {
        for(Operator op : this.operators)
            if(op.getOperatorToken().equals(operatorToken))
                return op;

        throw new IllegalArgumentException("Unknown operator");
    }
}
```

```
public interface Display {      // 0,5 P

    void show(double[] value);  // 0,5 P

}

public class TerminalDisplay implements Display {     // 0,5 P

    @Override
    public void show(double[] value) {
        System.out.println(Arrays.toString(value));   // 0,5 P
    }
}
```

c) **(2 + 1 Punkte)** Ergänzen Sie eine neue Operation *sqrt* (Wurzel ziehen), die ebenfalls als *class*-Datei vom *OperatorLoader* geladen werden kann.

Bewertung: 1P Token und Berechnung korrekt + 1P Prüfen der Argumentenanzahl + negatives Ergebnis berücksichtigt 1P BONUS

```
public class Sqrt implements Operator {

    @Override
    public String getOperatorToken() {
        return "sqrt";
    }

    @Override
    public double[] calc(double[] values) {

        if(values.length != 1)
            throw new IllegalArgumentException("Invalid amount of operands");

        return new double[]{Math.sqrt(values[0]), -Math.sqrt(values[0])};
    }
}
```

Aufgabe 3. *Entwurfsmuster zur Länderfinanzierung* **14 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 Life-CrisisHotline).

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-Repository, welches die Basis-Studiengänge modelliert. Die Klassen enthalten Operationen zur Abfrage der Kostenbeiträge und der Beschreibung eines Studiengangs.

a) **(9 Punkte)** 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 Verantwortlichen für die Umsetzung ausgedacht haben:

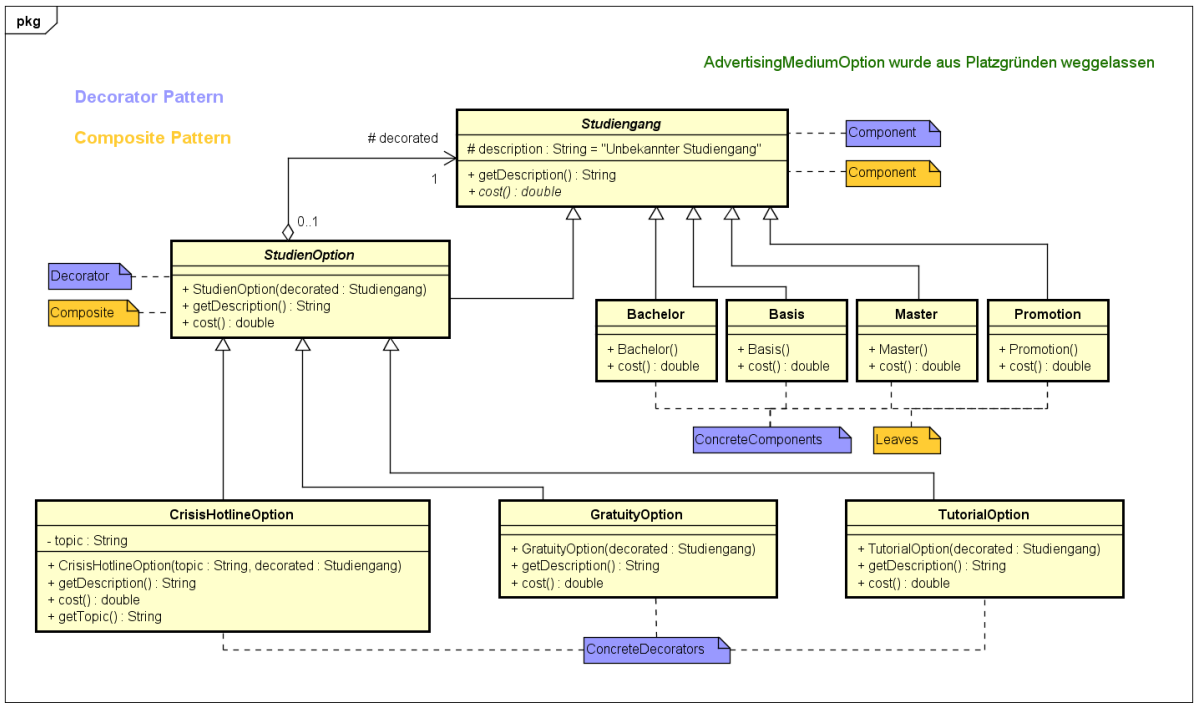
   a. Ein Basis-Studiengang und eine Option bilden selbst wieder einen Studiengang.

   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.

   Beschreiben Sie in Ihrer Implementierung die Rollen, die Ihre Klassen und Methoden im Entwurfsmuster spielen (als Kommentar des jeweiligen Elementes).



Pro Rolle 0,5P = 1,5 P

Bewertung: StudienOption 2,5P + Option 2P + 3x1 Rest

Codebestandteile mit dieser Markierung dienen der Implementation der Vergleichsmethode und wurden nicht gefordert. Solch eine Methode ist beim Decorator Pattern jedoch sehr sinnvoll.

```
public abstract class Studiengang {

	protected String description = "Unbekannter Studiengang";

	public String getDescription() {
		return description;
	}

	public abstract double cost();

	public boolean istGleicherStudiengang(Studiengang s) {
		if (!StudienOption.class.isInstance(s))
			return this.equals(s);

		return s.istGleicherStudiengang(this);
	}
}
```

```
// 5x 0,5 P
public abstract class StudienOption extends Studiengang {

	protected Studiengang decorated;

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

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

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

	public boolean istGleicherStudiengang(Studiengang s) {
		return decorated.istGleicherStudiengang(s);
	}
}
```

```
// 4x 0,5 P für ein richtiges Beispiel, + 1 P für jede weitere Option
public class TutorialOption extends StudienOption {
	public TutorialOption(Studiengang decorated) {
		super(decorated);
	}
	@Override
	public String getDescription() {
		return decorated.getDescription() + ", Tutorial Option";
	}
	@Override
	public double cost() {
		return decorated.cost() + 700.00;
	}
}
```

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

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

```
        }
}
```

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

b) **(3 Punkte)** 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) {
            Studiengang bachelor = new Bachelor();
            // 2 P
            Studiengang a = new GratuityOption(
                                    new CrisisHotlineOption("Mathe",
                                        new CrisisHotlineOption("SWT",
                                            new TutorialOption(
                                                bachelor
                                            )
                                        )
                                    )
                                );
            // 0,5 P
            System.out.println("Studiengang: " + a.getDescription());
            // 0,5 P
            System.out.println(
                  String.format("Studienbeitrag: %.2f €", a.cost()));
```

```
            Studiengang b = new CrisisHotlineOption("LudS",bachelor);
            Studiengang c = new CrisisHotlineOption("LudS",new Bachelor());

            System.out.println("Es folgt hoffentlich ein true:  " +
                                    a.istGleicherStudiengang(b));
            System.out.println("Es folgt hoffentlich ein false: " +
                                    c.istGleicherStudiengang(b));
        }
}
```

c) **(2 Punkte)** Gehen Sie nun davon aus, dass jeder Basis-Studiengang zu jedem Zeitpunkt mit nur einer <u>einzigen</u> (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.

∑ 39+1 Punkte