Übungen zur Vorlesung Softwaretechnologie

- Wintersemester 2018/19 -
Dr. Günter Kniesel

Übungsblatt 10 – Lösungen

Aufgabe 1. *Jahreszeitbedingte Anwendung von Entwurfsmustern* (16 Punkte)

Ein reichlich geschmückter Weihnachtsbaum besteht ausschließlich aus *Baumbestandteilen*, 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) 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.

```
/**
 * Rolle im Composite Pattern: Component
*/
public interface Baumelement {

	void add(Baumelement e);
	void remove(Baumelement e);
	List<Baumelement> getChildren();
	String toString();
}



/**
 * Rolle im Composite-Pattern: Composite
 */
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 toString() {
		StringBuffer sb = new StringBuffer("Zweig[");
		for(Baumelement e: children) {
			sb.append(e.toString());
		}
		sb.append("] ");
		return sb.toString();
	}

}


/**
 * Rolle im Composite-Pattern: Leaf
 */
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();
	}
}

public class Engel extends Baumschmuck {
	@Override
	public String toString() {
		return getAdjektive(true) + "Engel ";
	}
}

public class Kugel extends Baumschmuck {
	@Override
	public String toString() {
		return getAdjektive(false) + "Kugel ";
	}
}

public class Stern extends Baumschmuck {
	@Override
	public String toString() {
		return getAdjektive(true) + "Stern ";
	}
}

public class Kerze extends Baumschmuck {
	@Override
	public String toString() {
		return getAdjektive(false) + "Kerze ";
	}
}
```

b) 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 maximale Zweigrekursionstiefe 3 beträgt und jeder Zweig maximal 5 Kindelemente haben kann. Geben Sie eine Beschreibung ihres Baumes auf der Konsole aus, indem Sie die *toString()*-Methoden der Baumelemente geeignet implementieren und auf dem Stamm aufrufen.

```
public class WeihnachtsbaumSimulator {

	private static Random rand = new Random();

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

	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;
	}

	public static void main(String[] args) {
		Baumelement baum = generiereZweig(0);
		System.out.println(baum);
	}

}
```

c) Zur Baumpflege werden im Rheinland traditionell Heinzelmännchen eingesetzt, kleine Spezialisten, die den Baum entlang klettern und eine spezifische Aktion auf den konkreten Baumelementen ausführen können. Modellieren Sie in Ihrem Projekt unter Nutzung eines geeigneten Entwurfsmusters ein abstraktes *Heinzelmännchen*. Ändern Sie, wenn nötig auch bestehende Klassen, so dass später beliebige konkrete Heinzelmännchen die Baumpflege übernehmen können. Beachten Sie, dass die Ausprägung einer konkreten Heinzelmännchen-Aktion je nach Baumelement unterschiedlich sein kann! Annotieren Sie die am Entwurfsmuster beteiligten Klassen und Methoden mit ihren Rollen, wie in *Teilaufgabe a*).

```
/**
 * Rolle im Visitor-Pattern: Visitor
 */
public abstract class Heinzelmaennchen {

	public abstract void visitEngel(Engel e);
	public abstract void visitKugel(Kugel k);
	public abstract void visitStern(Stern s);
	public abstract void visitKerze(Kerze k);

	public void visitZweig (Zweig zweig) {
		for (Baumelement b: zweig.getChildren())
			b.accept(this);
	}
}

/**
 * Rolle im Composite Pattern: Component
 * Rolle im Visitor-Pattern: Element
 */
public interface Baumelement {
	…
	void accept(Heinzelmaennchen h);
}

public class Zweig implements Baumelement {
	…
	public void accept(Heinzelmaennchen h) {
		h.visitZweig(this);
	}
}

public class Engel extends Baumschmuck {
	…
	public void accept(Heinzelmaennchen h) {
		h.visitEngel(this);
	}
}

public class Kugel extends Baumschmuck {
	…
	public void accept(Heinzelmaennchen h) {
		h.visitKugel(this);
	}
}

public class Stern extends Baumschmuck {
	…
	public void accept(Heinzelmaennchen h) {
		h.visitStern(this);
	}
}
```

```java
public class Kerze extends Baumschmuck {
        …
        public void accept(Heinzelmaennchen h) {
                h.visitKerze(this);
        }
}
```

d) Implementieren Sie zwei konkrete Heinzelmännchen:

- *Glitzer-Heinzelmännchen*: besprüht jedes besuchte Element mit Glitzerstaub
- *Vergoldungs-Heinzelmännchen*: streicht jedes besuchte Element goldfarbig an

Ergänzen Sie evtl. zusätzlich benötigte Eigenschaften in den Baumelementen.

```java
public abstract class Baumschmuck implements Baumelement {

    private String farbe;
    private boolean glitzernd;
    …
}

public class VergoldungsHeinzelmaennchen extends Heinzelmaennchen {

        @Override
        public void visitEngel(Engel e) {
            e.setFarbe("gold");
        }

        @Override
        public void visitKugel(Kugel k) {
            k.setFarbe("gold");
        }

        @Override
        public void visitStern(Stern s) {
            s.setFarbe("gold");
        }

        @Override
        public void visitKerze(Kerze k) {
            k.setFarbe("gold");
        }

        public String getAdjektive(boolean maennlich) {
            StringBuilder adjektive = new StringBuilder();
            if (this.isGlitzernd()) {
                if (maennlich)
                    adjektive.append("glitzernder ");
                else
                    adjektive.append("glitzernde ");
            }
            if (this.getFarbe() != null) {
                if (maennlich)
                    adjektive.append(this.getFarbe()).append("ener ");
                else
                    adjektive.append(this.getFarbe()).append("ene ");
            }
            return adjektive.toString();
        }
}
```

```java
public class GlitzerHeinzelmaennchen extends Heinzelmaennchen {

	@Override
	public void visitEngel(Engel e) {
	    e.setGlitzernd(true);
	}

	@Override
	public void visitKugel(Kugel k) {
	    k.setGlitzernd(true);
	}

	@Override
	public void visitStern(Stern s) {
	    s.setGlitzernd(true);
	}
	@Override
	public void visitKerze(Kerze k) {
	    k.setGlitzernd(true);
	}
}
```

Aufgabe 2. *Umsetzung von Modellen* (15 Punkte)

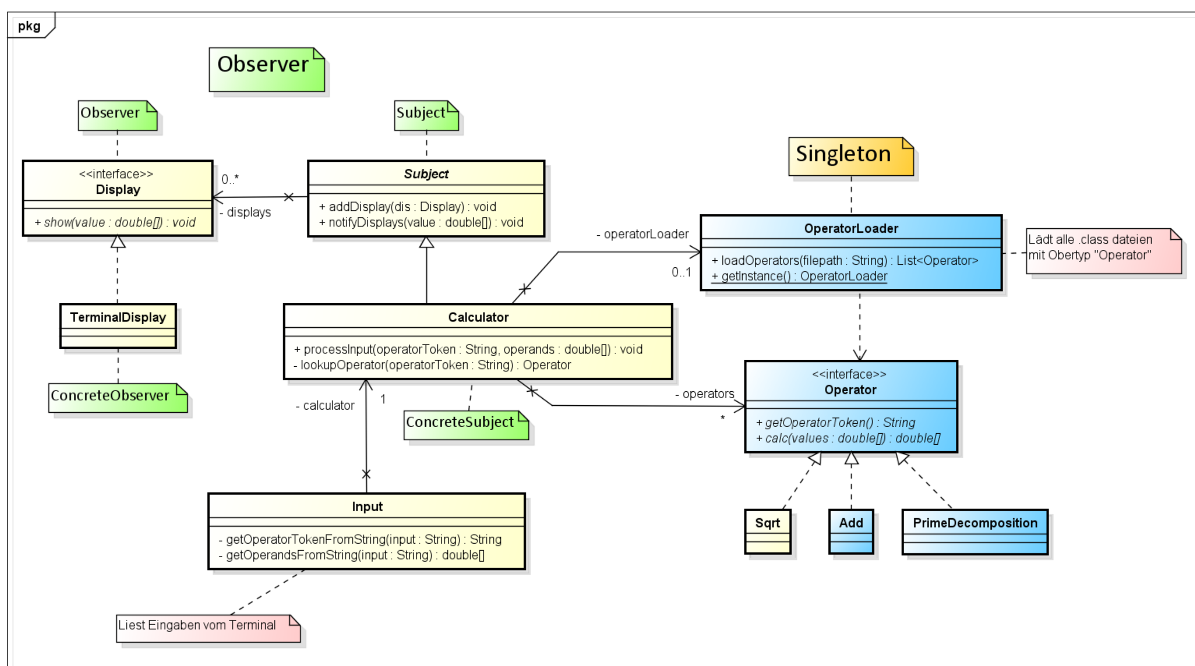
In dieser Aufgabe stellen wir Ihnen ein Klassendiagramm und Sequenzdiagramm zur Verfügung, das einen Taschenrechner mit PlugIn-Fähigkeiten modelliert. Ihre Aufgabe besteht darin, das gegebene Modell in Java umzusetzen.

Die benötigten Diagramme finden Sie unter:

ssh://git-se@git.iai.uni-bonn.de/swt2018_readonly

Die blau eingefärbten Elemente des Klassendiagramms geben wir Ihnen vor. Sie sind als Java-Code (Quell- oder Bytecode / *jar*-Datei) ebenfalls im Repository enthalten.

a) (*2 Punkte*) Analysieren Sie das gegebene Klassendiagramm. Identifizieren Sie, welche Entwurfsmuster in diesem Modell verwendet wurden.



b) (*8 Punkte*) Setzen Sie das Modell in Java um. Verwenden Sie das Klassendiagramm, um die Struktur, und das Sequenzdiagramm, um den Ablauf einer Rechenaktion zu verstehen. Beachten Sie zusätzlich die folgenden Informationen:

1. Alle Operationen (*Add*, *Sqrt*, …) sollen als *.class*-Dateien vom vorgegebenen OperatorLoader geladen werden, und sind nicht Teil des Rechners.
2. Verwenden Sie die gegebenen *Add.class* und *PrimeDecomposition.class* Dateien, um ihre Implementierung zu testen. (Token: Add (*+*), PrimeDecomposition (*prime*))
3. Der Einfachhalt halber können Sie die Präfix-Notation für Eingaben verwenden (z.B.: „+ 2 3“ oder „prime 18“).

Ergänzen Sie eigene Methoden, wenn es Ihnen notwendig erscheint. Achten Sie in diesem Fall darauf, dass Sie Modell und Code konsistent halten.

```java
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());
    }
    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;
    }
}
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) {
```

```java
            dis.show(value);
        }
    }
}

public class Calculator extends Subject {

    // 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));
        }
        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 abstract class Subject {

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

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

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

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

public interface Display {

    void show(double[] value);

}

public class TerminalDisplay implements Display {

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

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

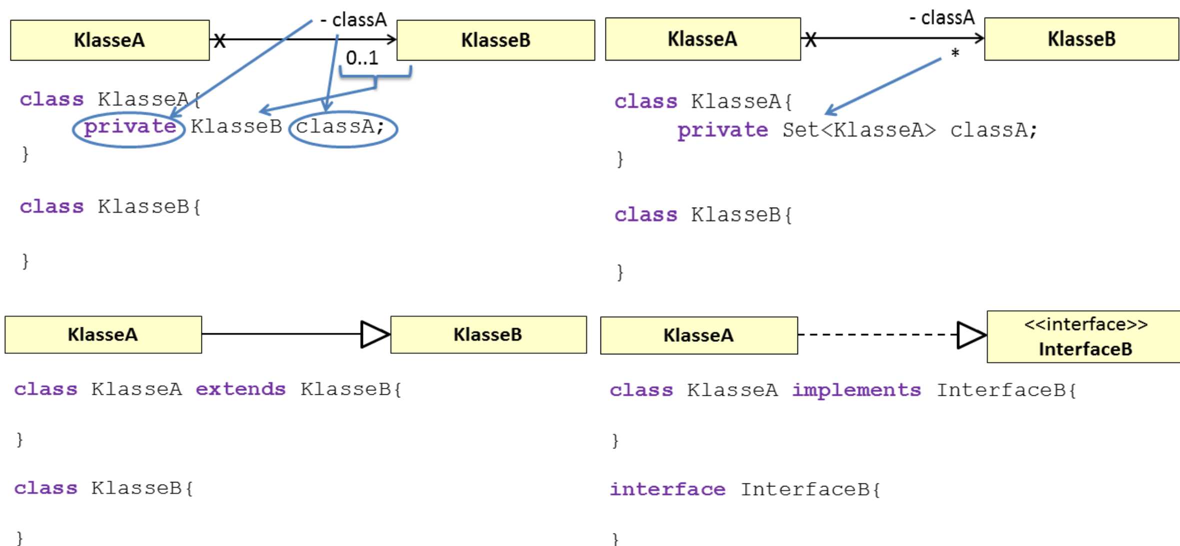
```
public class Sqrt implements Operator {
    @Override
    public String getOperatorToken() {
        return "sqrt";
    }
    @Override
    public double[] calc(double[] values) {
        if(values.length != 1)
            throw new IllegalArgumentException("Invalid amount of operands");
        return new double[]{Math.sqrt(values[0])};
    }
}
```

d) (*3 Punkte*) Erklären Sie, wie Sie die verschiedenen Elemente des Klassendiagramms in Code umgesetzt haben.

Umsetzung verschiedener Modellelemente:

- **Klasse** → Umsetzung als Java-Klasse
- **Attribut** → Umsetzung als Instanzvariable (inkl. Sichtbarkeit, Typ)
- **Operation** → Umsetzung als Methode (inkl. Sichtbarkeit, Rückgabetyp, Parameter)

Umsetzung von Assoziation und Vererbung:



Umsetzung einer unidirektionalen 1:1-Assoziation:

Objektentwurfsmodell mit Assoziation

Objektentwurfsmodell ohne Assoziation

- Unidirektionale 1:1 Assoziationen sind einfach:
  - Durch Instanzvariable in der „referenzierenden" Klasse implementieren
    - ⇨ Der Name der Instanzvariablen ist der Rollenname am Pfeilende
    - ⇨ Der Typ der Instanzvariablen ist die referenzierte Klasse.
  - Die „referenzierte" Klasse hat **keine** solche Instanzvariable

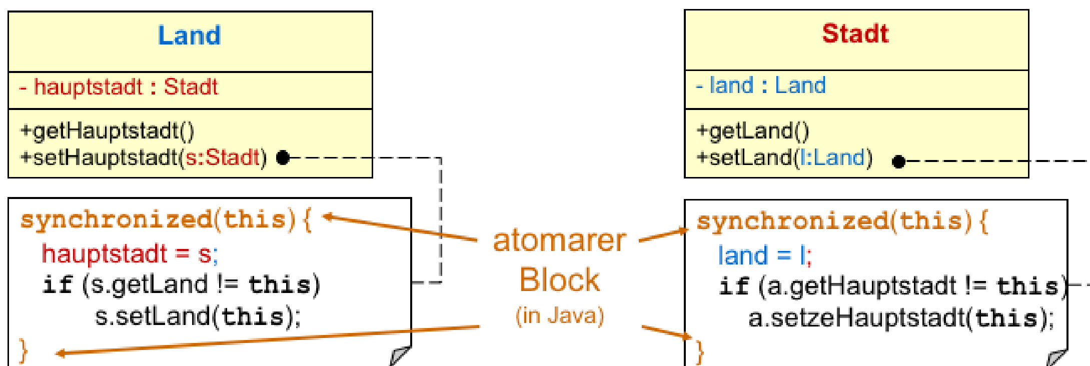
Umsetzung einer unidirektionalen 1:n-Assoziation:

Die **unidirektionale 1:n-Assoziation** wird analog umgesetzt, jedoch als `Set<>` implementiert.

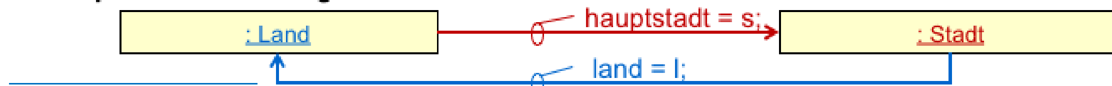
Umsetzung einer bidirektionalen 1:1-Assoziation:

Objektentwurfsmodell mit Assoziation

Objektentwurfsmodell ohne Assoziation



Beispiel-Instanziierung



Umsetzung einer bidirektionalen 1:n-Assoziation:

Die **bidirektionale 1:n-Assoziation** wird analog umgesetzt, jedoch als `Set<>` implementiert. Auch hier muss auf die korrekte Synchronisation geachtet werden!