



<a href="http://ovm-kassel.de">http://ovm-kassel.de</a>   Information	
Information IT-AE-JA-INFO-5.1 Objektorientierte Software-Entwicklung	
Code	IT-AE-JA-INFO-5.1
Autor	André Bauer <a(dot)bauer(at)ovm-kassel(dot)de>
Datum	25. Februar 2018
Links	
Verwandte Literatur	
Lizenz	 Dieses Werk ist lizenziert unter einer Creative Commons Namensnennung - Weitergabe unter gleichen Bedingungen 4.0 International Lizenz.

# Objektorientierte Software-Entwicklung

Wenn eine Software mehr als etwa hundert Zeilen Quellcode umfasst, spätestens ab mehreren hundert, benötigt man Techniken, um der Software eine klare Struktur zu geben. Eine klare Struktur ist wichtig, damit die Software leicht angepasst und erweitert werden kann, sowohl durch bisherige als auch durch andere Entwickler.

## 1. Objektorientierte Analyse (OOA)

### 1.1. User-Stories

Eine sehr häufig angewandte Technik zum Entwurf und zur Strukturierung ist die objektorientierte Software-Entwicklung. Dazu wird in der objektorientierten Analyse anhand von Szenarien ermittelt, wie die zukünftige Nutzung des (Software-)Systems ablaufen soll. Dabei werden sogenannte [User-Stories](#) verfasst:

#### *User-Story 1*

Die Lampen lampe1, lampe2 und lampe3 sind anfangs ausgeschaltet. Der Schalter schalter1 ist ebenfalls in der Position „aus“. Der Benutzer betätigt den Schalter schalter1. Die Lampen lampe1, lampe2 und lampe3 werden eingeschaltet.

Aus den User-Stories können nun die relevanten Objekte abgeleitet werden; im Beispiel sind dies der Benutzer, der Schalter schalter1 sowie die Lampen lampe1, lampe2 und lampe3.

### 1.2. Verfahren von Abbott

Mit dem Verfahren von Abbott werden aus einer Problem- oder Situationsbeschreibung oder

entsprechenden User-Stories mögliche Objekte, Attribute und Methoden ermittelt. Dieses Verfahren wird auf dem Informationsblatt IT-AE-JA-INFO-5.2 erläutert.

### 1.3. Sequenzdiagramme

Stellt man die User-Story in einem **UML-Sequenzdiagramm** dar, so kann man den Programmablauf entlang der Pfeile ablesen. So sendet der Schalter schalter1 die Nachricht `einschalten()` an die Lampe lampe1. Erst wenn diese den Einschaltvorgang abgeschlossen hat, setzt schalter1 seine Arbeit fort und sendet lampe2 die Nachricht `einschalten()` usw.

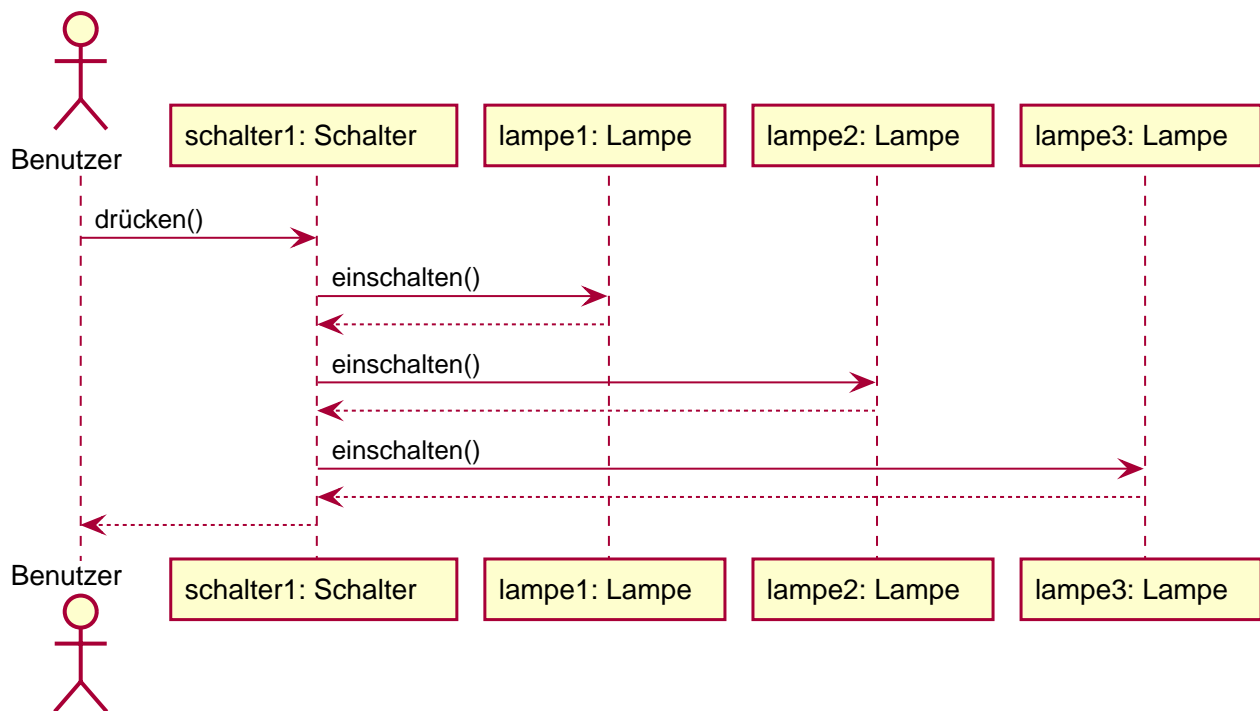


Abbildung 1. Sequenzdiagramm zur *User-Story 1*

Eine zweite User-Story vervollständigt die möglichen Anwendungsfälle der Schaltung:

#### *User-Story 2*

Die Lampen lampe1, lampe2 und lampe3 sind eingeschaltet. Der Schalter schalter1 ist ebenfalls in der Position „ein“. Der Benutzer betätigt den Schalter schalter1. Die Lampen lampe1, lampe2 und lampe3 werden ausgeschaltet.

Daraus ergibt sich das folgende Sequenzdiagramm:

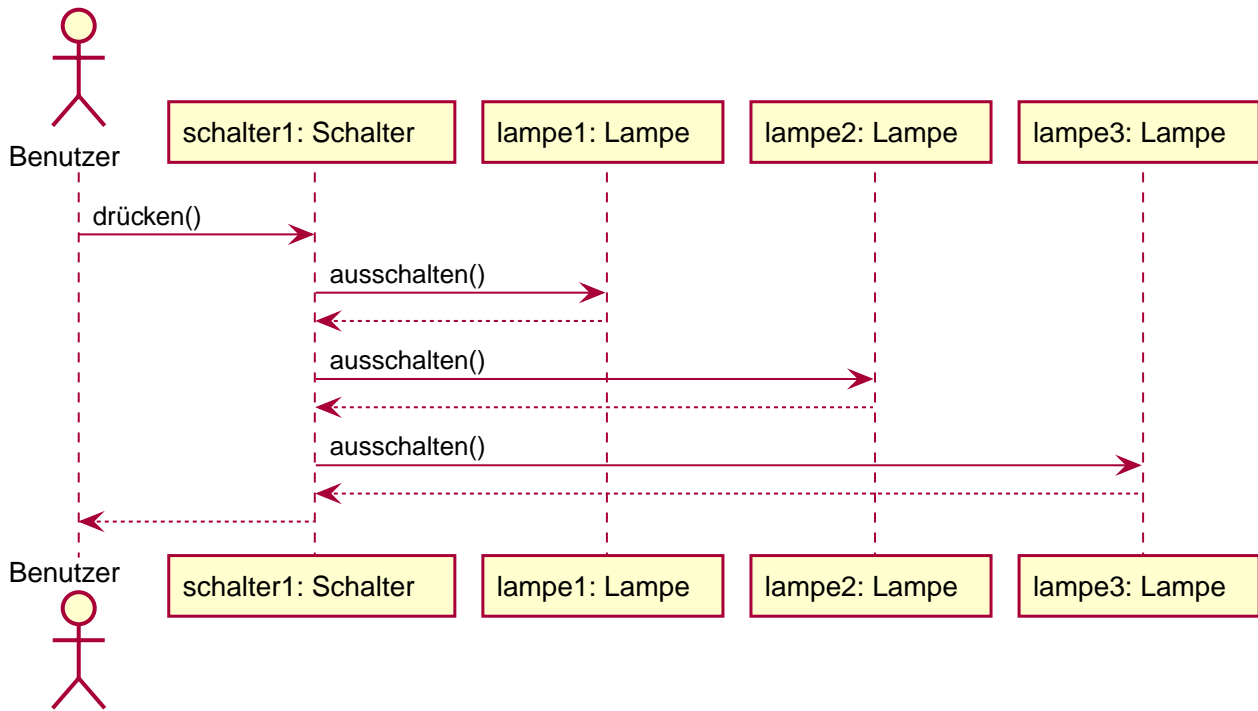


Abbildung 2. Sequenzdiagramm zur User-Story 2

## 1.4. Objektdiagramme

Aus den User-Stories bzw. den zugehörigen Sequenzdiagrammen kann man ablesen, wie die beteiligten Objekte miteinander verknüpft sein müssen, um Nachrichten austauschen zu können:

Der Schalter schalter1 verwendet die Lampen lampe1, lampe2 und lampe3, um ihnen jeweils die Nachricht `einschalten()` bzw. `ausschalten()` senden zu können. Diese Beziehung wird in einem Objektdiagramm durch Pfeile dargestellt.

Objekte, die Nachrichten austauschen, werden im Diagramm mit Linien verbunden. Dieser Beziehungstyp wird *Assoziation* genannt (auch Kennt-Beziehung). Die Beziehungen werden mit einem Namen versehen, im Beispiel lautet dieser „schaltet“. Wenn die Nachrichten nur in eine Richtung ausgetauscht werden, wird dies durch eine Pfeilspitze kenntlich gemacht.

Daraus ergibt sich das folgende Objektdiagramm:

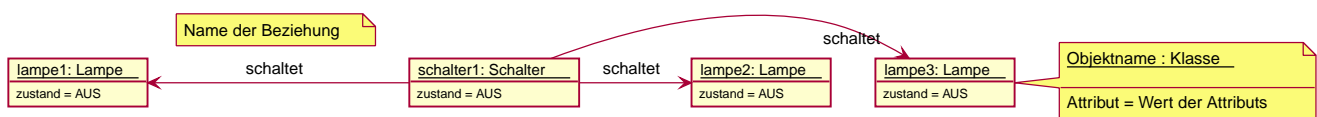


Abbildung 3. Objektdiagramm für einen Aufbau mit einem Schalter und drei Lampen zu Beginn der User-Story 1

Ein Objektdiagramm stellt den Zustand eines Software-Systems zu einem bestimmten Zeitpunkt dar. Das Diagramm in **Abbildung 3** stellt den Zustand zu Beginn der User-Story 1 bzw. zum Ende der User-Story 2 dar, da der Schalter und die Lampen den Zustand `AUS` haben.

Der Zustand zum Ende der User-Story 1 bzw. zu Beginn der User-Story 2 sieht so aus:

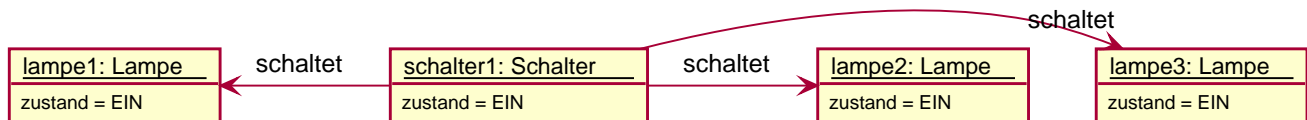


Abbildung 4. Objektdiagramm für einen Aufbau mit einem Schalter und drei Lampen zum Ende der User-Story 1

Man könnte jetzt den Schalter und jede der drei Lampen direkt programmieren, aber dies hätte Nachteile: für die Lampen würde man dreimal dasselbe Programm schreiben. Es wäre daher eine unflexible Lösung, man möchte mit derselben Software auch Schaltungen mit vier oder mehr Lampen mit einem Schalter verbinden können.

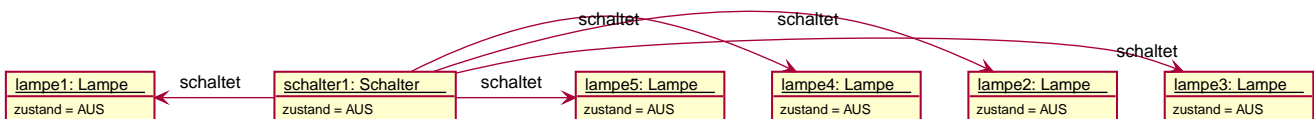


Abbildung 5. Objektdiagramm für einen Aufbau mit einem Schalter und fünf Lampen

## 1.5. Klassendiagramme

Daher verallgemeinert man die möglichen Objektbeziehungen in Form eines UML-Klassendiagramms:

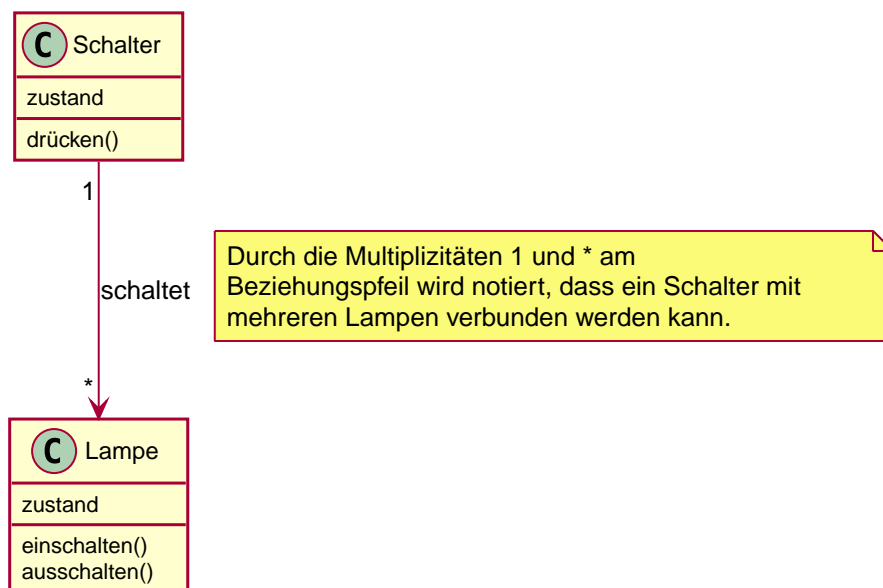


Abbildung 6. Klassendiagramm als Ergebnis der OOA

In einem Klassendiagramm werden Objekte mit derselben Struktur zu einer Klasse zusammengefasst. Die Lampen lampe1, lampe2 und lampe3 werden also zu der Klasse **Lampe** zusammengefasst. Die Nachricht **drücken()** wird dem Schalter und **ausschalten()** bei der Lampe notiert, also jeweils bei der Klasse, der die entsprechende Nachricht gesendet wird. Statt Nachricht wird im Zusammenhang mit der Programmiersprache Java der Begriff *Methode* verwendet. Die Daten, die auch Attribute der Objekte bzw. Klassen genannt werden, können im Klassendiagramm mit ihren initialen Werten angegeben werden, zudem kann auch der entsprechende Datentyp benannt werden wie in [Abbildung 7](#) und [Abbildung 10](#).

Das Klassendiagramm stellt im Gegensatz zu einem Objektdiagramm keinen Systemzustand dar, sondern die Struktur der zugrundeliegenden Software bzw. des Quellcodes.

Die Darstellung einer Klasse im Klassendiagramm folgt dem folgenden Schema:

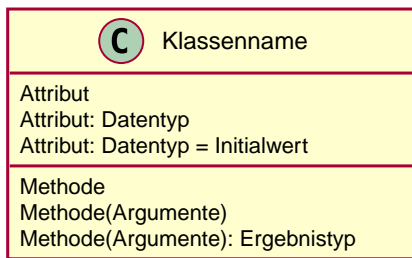


Abbildung 7. Schema der graphischen Darstellung einer Klasse

## 2. Objektorientierte Modellierung (OOM)

In der objektorientierten Modellierung werden u. a. die bereits vorgenommenen Modellierungen in den Klassendiagrammen verfeinert und z. B. Datentypen und initiale Werte eingefügt. Gegebenenfalls werden die Diagramme um weitere Klassen ergänzt, die sich nicht unmittelbar aus der Analyse ergeben haben, aber für die Implementierung in der gewählten Programmiersprache benötigt werden, wie die Definition einer Enumeration.

### 2.1. Enumerations

In den Objektdiagrammen in [Abbildung 3](#) und [Abbildung 4](#) haben die Objekte das Attribut **zustand** mit den möglichen Werten **ON** und **OFF**. Anstatt eines vorhandenen Java-Datentyps wie **int**, **boolean** oder **string** erstellen wir mithilfe einer Aufzählung (engl. enumeration, in Java abgekürzt zu **enum**) einen passenden neuen und aussagekräftigen Datentyp.

Quellcode 1. Enumumertation State in Java

```
enum State {  
    ON,  
    OFF  
}
```

Die Aufzählung **State** ergänzen wir in dem Klassendiagramm und verwenden zudem ab jetzt englische Begriffe. Die Verwendung englischer Sprache im Quellcode ist üblich, da bei den meisten Programmiersprachen für die Sprachelemente sowie die **Programmbibliotheken** ohnehin englische Sprache verwendet wird, wie auch bei Java.

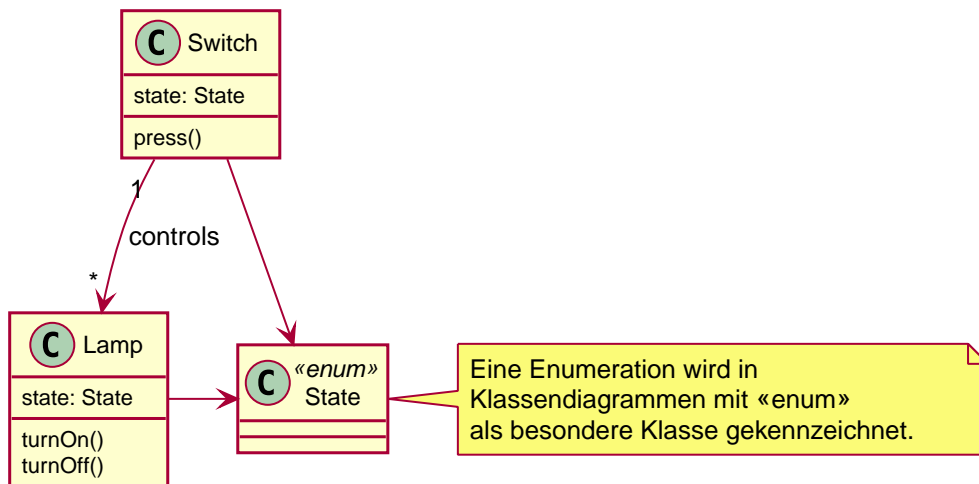


Abbildung 8. Klassendiagramm mit Enumeration und englischen Begriffen

Sowohl **Switch** als auch **Lamp** verwenden **State**, was durch Pfeile in **Abbildung 8** gekennzeichnet ist.

Damit haben wir aus den User-Stories eine objektorientierte Struktur für die Software entworfen. Man nennt diesen Schritt **objektorientierte Modellierung**.

## 2.2. Die Aggregation (Hat-Beziehung)

Neben der Assoziation (Kennt-Beziehung) gibt es noch stärkere Beziehungstypen, u. a. die **Aggregation (Hat-Beziehung)** zwischen Objekten bzw. Klassen. So *hat* ein Motorrad ein Vorderrad.

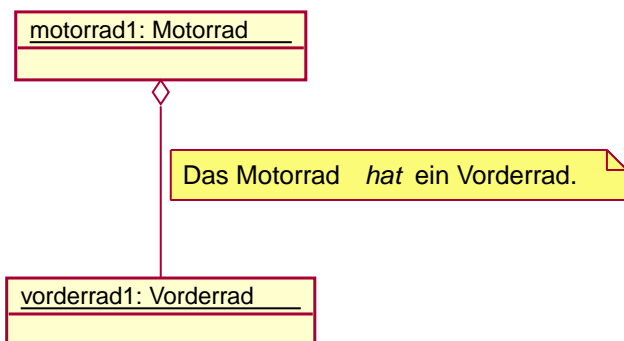


Abbildung 9. Objektdiagramm mit Aggregation

Die Raute (auch Diamant genannt), die diesen Beziehungstyp kennzeichnet, darf *nicht* wie die Pfeilspitze der Assoziation interpretiert werden, denn die Beziehung geht vom Motorrad zum Vorderrad. Auf der Ebene der Programmierung ist bei einer Aggregation die Klasse Motorrad für die Verwaltung der Objekte der Klasse Vorderrad zuständig.

## 3. Objektorientierte Programmierung (OOP) mit Java

Dem Klassendiagramm entspricht das folgende Java-Programm:

## Quellcode 2. Java-Programm

```
// Datei State.java
public enum State {
    ON,
    OFF
}

// Datei Switch.java
public class Switch {
    private State state;

    public void press() {
    }
}

// Datei Lamp.java
public class Lamp {
    private State state;

    public void turnOn() {
    }

    public void turnOff() {
    }
}
```

Ist damit die Entwicklung der Software abgeschlossen? Nein, noch nicht, wir haben bislang erst eine objektorientierte Struktur erstellt.

### 3.1. Was wird noch benötigt, damit der Schalter die Lampen steuert?

*Wir benötigen ...*

- Konstruktoren, die wie in den User-Stories den Anfangszustand herstellen und Objekte einer Klasse kreieren.
- eine Ausgabe, wenn die Lampe ihren Zustand wechselt, damit wir das Programm testen können.
- Quellcode, der die Beziehungen zwischen den Objekten erzeugt. Dazu betrachten wir zunächst eine Vereinfachung.
- Quellcode für die Aktionen der Objekte, wenn sie eine Nachricht erhalten, das heißt die Methoden `press()` in `Switch` sowie `turnOn()` sowie `turnOff()` in `Lamp` müssen implementiert werden.

Wir vereinfachen zunächst die Kennt-Beziehung zwischen `Switch` und `Lamp` so, dass nur eine `Lamp` mit einem `Switch` verbunden werden kann, dies nennt man eine *1-zu-1-Beziehung*.

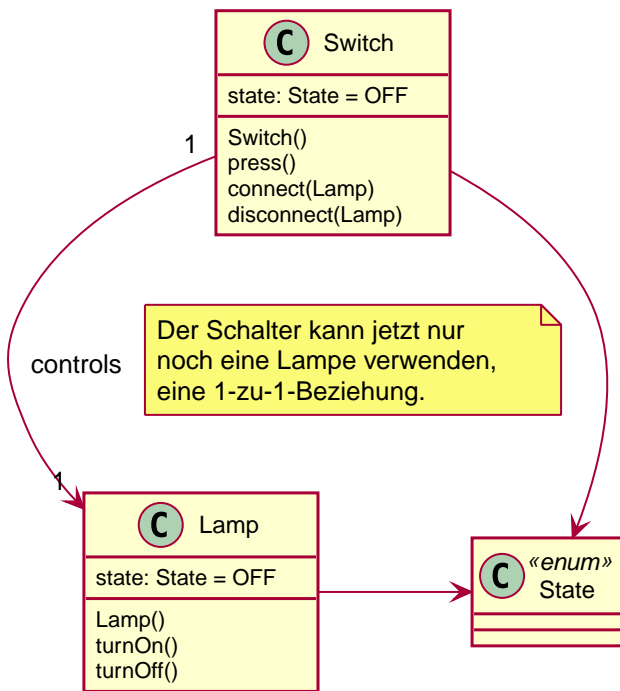


Abbildung 10. Klassendiagramm mit einer 1-zu-1-Beziehung zwischen **Switch** und **Lamp**.

In Java muss jede Klassen mit dem Zugriffsmodifikator **public** jeweils in einer eigenen **java**-Datei, die den Namen der Klasse trägt, gespeichert werden.

#### Quellcode 3. Java-Programm

```

// State.java
public enum State {
    ON,
    OFF
}

// Datei Lamp.java
public class Lamp {
    private State state;

    private Color color;

    public Lamp(Color color) {
        state = State.OFF;
        this.color = color;
    }

    public void turnOn() {
        state = State.ON;
        System.out.println(this + " ON"); ①
    }

    public void turnOff() {
        state = State.OFF;
        System.out.println(this + " OFF");
    }
}
  
```



```
}  
}  
  
// Switch.java  
public class Switch {  
    private State state;  
    private Lamp lamp; ②  
  
    public Switch() {  
        state = State.OFF;  
    }  
  
    public void press() {  
        if (lamp != null) { ③  
            switch (state) {  
                case OFF:  
                    this.state = State.ON;  
                    lamp.turnOn();  
                    break;  
  
                case ON:  
                    this.state = State.OFF;  
                    lamp.turnOff();  
            }  
        }  
    }  
  
    public void connect(Lamp lampToConnect) { ④  
        if (lamp == null) { ⑤  
            lamp = lampToConnect;  
        }  
    }  
  
    public void disconnect(Lamp lampToDisconnect) { ⑥  
        if (lamp == lampToDisconnect) { ⑦  
            lamp = null;  
        }  
    }  
}
```

- ① Hier wird eine Text auf der Konsole ausgegeben. Diese setzt sich aus dem aktuellen Objekt `this` sowie dem Text " ON" zusammen.
- ② Um eine Beziehung von `Switch` zu `Lamp` herstellen zu können, wird ein Attribut der Klasse `Lamp` benötigt, dieses hat den initialen Wert `null`, d. h. es ist anfangs kein Objekt verknüpft.
- ③ Die Methoden `turnOn()` und `turnOff()` können nur aufgerufen werden, wenn `lamp` mit einem Objekt der Klasse `Lamp` verknüpft ist.
- ④ Die Methode `connect()` stellt die Verknüpfung einer Lampe mit einem Schalter her.
- ⑤ Eine neue Verknüpfung wird nur dann hergestellt, wenn der Schalter noch nicht verbunden ist.

- ⑥ Die Methode `disconnect()` löst eine Verknüpfung zwischen einer Lampe und einem Schalter.
- ⑦ Die Verknüpfung wird nur gelöst, wenn die im Parameter angegebene `lamp` mit der bereits verknüpften identisch ist.

Mit diesen Klassen können jetzt die Objekte `lamp1` und `switch1` erzeugt werden und die Lampe `lamp1` mit dem Schalter `switch1` verbunden werden.

#### Quellcode 4. Java-Programm

```
Lamp lamp1 = new Lamp(); ①  
Switch switch1 = new Switch(); ②  
switch1.connect(lamp1); ③  
switch1.press(); ④
```

- ① Ein neues Objekt der Klasse `Lamp` mit dem Bezeichner `lamp1` wird erzeugt.
- ② Ein neues Objekt der Klasse `Switch` mit dem Bezeichner `switch1` wird erzeugt.
- ③ Die Lampe `lamp1` wird mit dem Schalter `switch1` verbunden.
- ④ „Betätigen“ des Schalters.

Diese Befehle können in `BlueJ` auch ohne `main`-Methode direkt über die Oberfläche oder im integrierten CodePad ausgeführt werden. Für ein vollständiges Java-Programm wird hingegen in einer Klasse eine `main`-Methode benötigt wie in der Klasse `Controller` im Quellcode 6.

## 3.2. Wie kann man mehrere Lampen mit einem Schalter verbinden?

Wir benötigen dazu eine Objekt-Sammlung im `Switch`. In Java gibt es dazu vorgefertigte Komponenten, z.B. eine `ArrayList`. Diese hat u.a. die Methoden

`add(E e)`

Fügt am Ende der Liste ein Objekt hinzu.

`get(int index): E`

Gibt das Objekt an der angegebenen Position `index` aus.

`remove(Object o)`

Entfernt ein Objekt aus der Sammlung.

Zudem gibt es in Java eine besondere `for`-Schleife für Collections wie u. a. die `ArrayList`:

#### Quellcode 5. Java-Programm

```
for (Lamp lamp: lamps) { ①  
    lamp.turnOn(); ②  
}
```

- ① In dem Schleifenkopf wird eine Collection ausgewählt, hier `lamps` sowie ein Bezeichner `lamp` der Klasse `Lamp` definiert, der nur in dem Schleifenrumpf gültig ist.
- ② Im Schleifenrumpf werden die Aktionen festgelegt, die mit jedem Objekt der Sammlung

durchgeführt werden, dazu verwendet man den oben definierten Bezeichner **lamp**.

Hier der vollständige Quellcode, der **User-Story 1** entspricht.:

*Quellcode 6. Vollständiges Java-Programm mit Javadoc-Kommentaren*

```
// State.java
/**
 * The representation of a state with the values ON and OFF.
 *
 * @author André Bauer
 * @version 1.0
 */
public enum State {
    ON,
    OFF
}

// Lamp.java
/**
 * A lamp with the states ON and OFF. It is not dimable.
 *
 * @author André Bauer
 * @version 1.0
 */
public class Lamp {
    /**
     * The current state of the lamp.
     */
    private State state = State.OFF;

    /**
     * The color of the lamp.
     */
    private Color color;

    /**
     * Creates a new lamp, with the initial state OFF.
     */
    public Lamp(Color color) {
        state = State.OFF;
        this.color = color;
    }

    /**
     * Turns the state of the lamp to ON.
     */
    public void turnOn() {
        state = State.ON;
        System.out.println(this + " ON");
    }
}
```

```
}

/**
 * Turns the state of the lamp to OFF.
 */
public void turnOff() {
    state = State.OFF;
    System.out.println(this + " OFF");
}
}

// Switch.java
import java.util.*;

/**
 * A simple switch with the states ON and OFF.
 *
 * @author André Bauer
 * @version 1.0
 */
public class Switch {
    /**
     * The current state of the switch.
     */
    private State state = State.OFF;

    /**
     * The list of connected lamps.
     */
    private List<Lamp> lamps;

    /**
     * Creates a new switch witch has an initial state OFF
     * and an empty list of connected lamps.
     */
    public Switch() {
        lamps = new ArrayList<Lamp>();
    }

    /**
     * Turns the state of the switch and
     * signals every connected lamp to turn
     * its state to the state of the switch.
     */
    public void press() {
        switch (state) {
            case OFF:
                this.state = State.ON;
                for (Lamp lamp: lamps) {
```

```
        lamp.turnOn();
    }
    break;

    case ON:
        this.state = State.OFF;
        for (Lamp lamp: lamps) {
            lamp.turnOff();
        }
    }
}

/**
 * Connects a lamp with the switch.
 *
 * @param lamp The lamp, which should become connected with the switch.
 */
public void connect(Lamp lamp) {
    lamps.add(lamp);
}

/**
 * Disconnects a lamp from the switch, if connected
 *
 * @param lamp The lamp, which should become disconnected from the switch.
 */
public void disconnect(Lamp lamp) {
    lamps.remove(lamp);
}
}

// Controller.java
/**
 * The main class which runs User-Story 1.
 *
 * @author André Bauer
 * @version 1.0
 */
public class Controller {
    public static void main(String args[]) {
        // User-Story 1
        Lamp lamp1 = new Lamp(Color.BLUE);
        Lamp lamp2 = new Lamp(Color.RED);
        Lamp lamp3 = new Lamp(Color.YELLOW);
        Switch switch1 = new Switch();
        switch1.connect(lamp1);
        switch1.connect(lamp2);
        switch1.connect(lamp3);
        switch1.press();
    }
}
```

```
}
```

Der Quellcode wird mit dem Java-Compiler `javac` übersetzt. Dabei genügt es, die Datei `Controller.java` anzugeben. Der Java-Compiler übersetzt dann automatisch alle weiteren benötigten Dateien.

```
$ javac Controller.java
```

Das Programm wird mit `java` ausgeführt:

```
$ java Controller  
Lamp@6d06d69c ON ①  
Lamp@7852e922 ON  
Lamp@4e25154f ON
```

① `Lamp@6d06d69c` wird durch `this` in der Ausgabe mit `System.out.println(this + " ON");` erzeugt und setzt sich aus dem Namen der Klasse und der ID des Objektes zusammen.