Rheinische Friedrich-Wilhelms-Universität Bonn
Institut für Informatik III

Übungen zur Vorlesung Softwaretechnologie

- Wintersemester 2019/20 -

Dr. Günter Kniesel

Übungsblatt 10

Zu bearbeiten bis: 13.12.2019, 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.

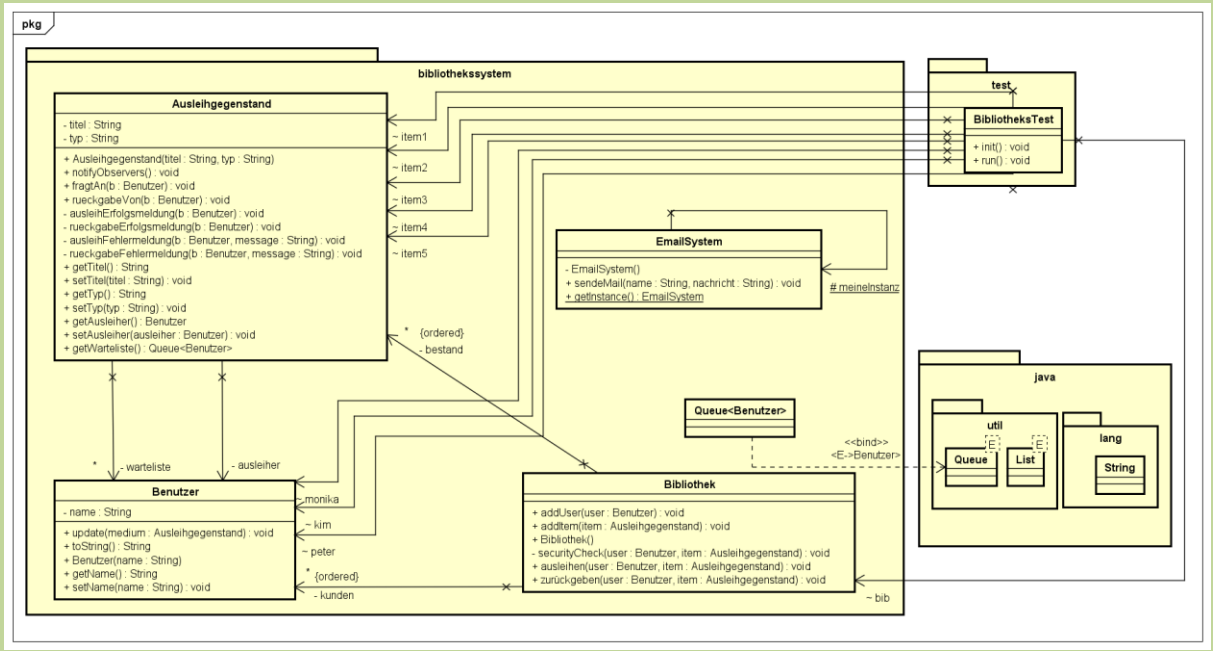
Praktische Aufgaben

Nicht lauffähiger Code wird mit 0 Punkten gewertet!

Aufgabe 1. *Entwurfsmuster im Ausleihsystem* **17,5 Punkte**

Gegeben sei das Bibliotheksverwaltungs-Projekt, das bereits im letzten Übungsblatt vorgestellt wurde. Der Programmcode steht im readonly-Repository (ssh://gitolite-se-swt@git.iai.uni-bonn.de/swt2019_readonly), im Ordner „Bibliothek“ zur Verfügung.

a) **(5 Punkte)** Reverse Engineering:
Erzeugen Sie ein Klassendiagramm des Quellcodes des **kompletten** Bibliotheks-verwaltungs-Projekts mit Hilfe von Astah (http://astah.net/student-license-request). Nutzen Sie „Tools > Java > Import Java…“. Exportieren Sie das generierte Modell. Machen Sie sich Gedanken, ob das Ergebnis Ihrer Intuition entspricht bzw. ob Sie es anders gemacht hätten. Markieren Sie die „Problemstellen“ im Diagramm (z.B. durch farbige Kringel) und geben Sie das um die Markierungen erweiterte Bild ab, zusammen mit einem Text, der beschreibt, was Ihnen aufgefallen ist.



1 Punkt für das Erzeugen des Diagramms

- (1P) Implizite 1-en auf der nicht-Pfeil-Seite (z.B. sollte ein Benutzer nicht nur einen Ausleihgegenstand haben dürfen)
- (1P) An manchen Pfeil-Enden keine explizite Kardinalität, was eigentlich eine 1 impliziert, wo auch 0..1 Sinn machen würde (Ausleihgenstand hat nicht immer einen Benutzer)
- (1P) Navigierbarkeit (und nicht-Navigierbarkeit) ist genauso markiert, wie im Code implementiert, da das Tool nicht raten kann, ob wir auf konzeptueller Ebene gern in die Gegenrichtung navigieren wollen oder nicht.
- (1P) Aus der Tatsache, dass die Implementierung der Referenzen der Bibliothek auf Benutzer eine Liste nutzt wurde der {ordered} constraint eingefügt, obwohl das evtl. konzeptionell nicht sein muss, dass Benutzer sortiert werden können.

Das existierende System sieht für Ausleihgegenstände nur eine Klasse vor. Diese hat ein Feld, welches den Typ des Mediums angibt. Diese Entscheidung ist sinnvoll, wenn man davon ausgeht, dass die Medien sich nur in ihrem Typ unterscheiden, aber keine ansonsten andere Struktur oder anderes Verhalten besitzen.

Unser System soll aber in Zukunft an Dritte verkauft werden und ihnen die Möglichkeit bieten, in ihre Variante der Bibliothek neue Medienarten mit eigenen Verhalten zu integrieren, ohne die von uns erstellten Klassen ändern zu müssen. Beispielsweise sollen Videos innerhalb einer Ausleihe dem Benutzer online abgespielt werden können.

Das bisherige Design soll daher nun modifiziert werden.

b) **(1 Punkt)** Entwurfsmuster:
Welches Entwurfsmuster bietet sich an, um beliebige Medien-Objekte mit dem statischen Typ *Ausleihgegenstand* erzeugen zu können, ohne dass der konkrete Typ des jeweiligen Objektes schon bei der Entwicklung der Bibliothek bekannt ist?

Factory Method

c) **(1,5 Punkte)** Implementierungsvarianten:
Welche Implementierungsvarianten dieses Entwurfsmuster kennen Sie?
„Fester Produkt-Typ“: Jede Unterklasse kreiert festgelegte Produkt-Art

„Codierter Produkt-Typ“: Mehrere Produkte, Fallunterscheidung anhand Parameter

„Klassenobjekt als Produkt-Typ“: Produkt-Typ als Parameter, reflektiver Aufruf. Nur als Idiom möglich.

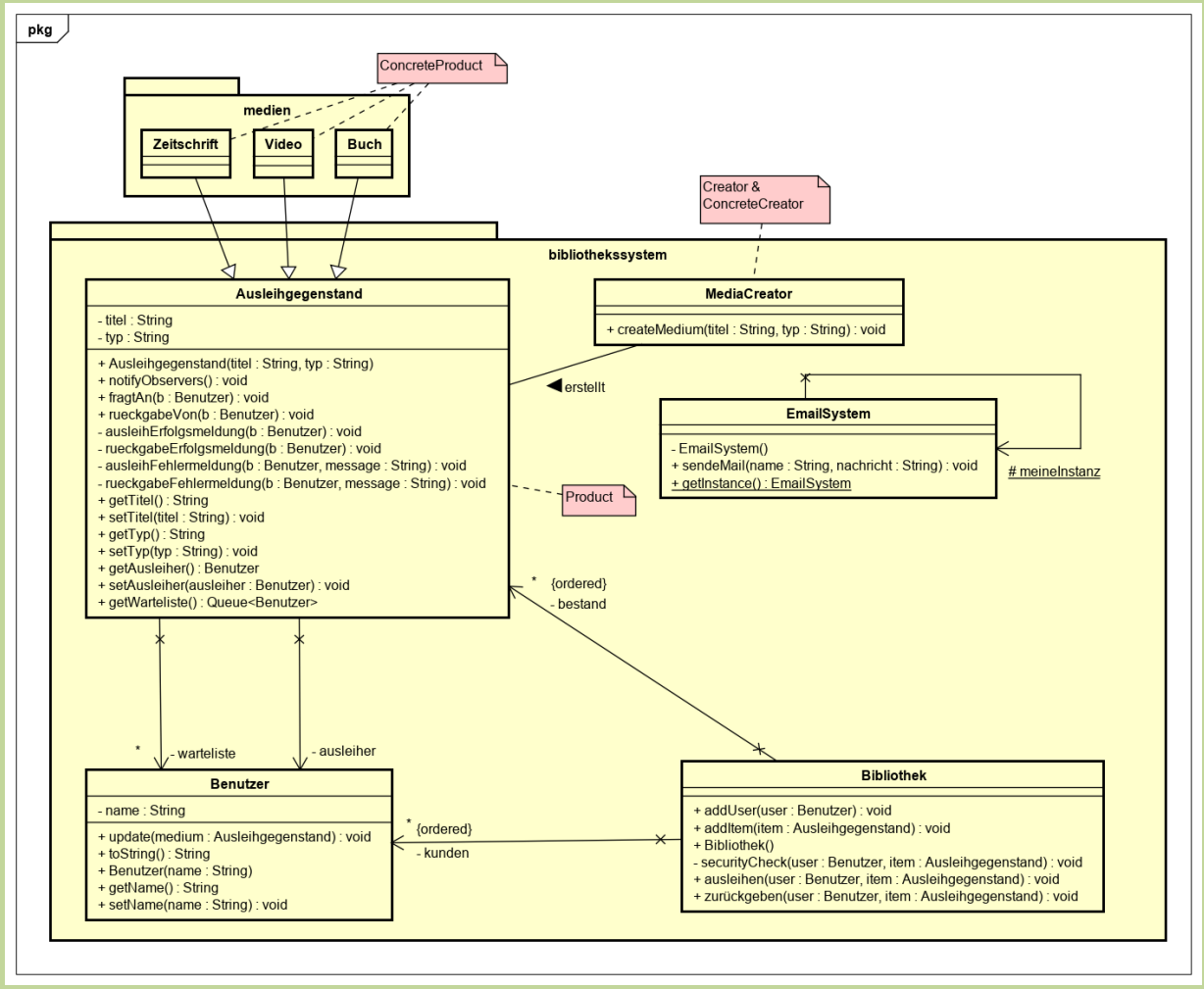
Siehe Skript Kap. 8, Folie „Factory Method: Implementierung”

d) **(1 Punkt)** Welche würden Sie in diesem Fall nutzen? Begründen Sie Ihre Entscheidung.
Variante 2 („Codierter Produkt-Typ“), da wartungsfreundlicher als Variante 1 und da die Codes ja schon in dem ursprünglichen Design des Programms enthalten waren (als Werte der Instanzvariable in der Klasse Ausleihgegenstad), also somit die wenigsten Änderungen zum bestehenden Programm anfallen.

e) **(1,5 Punkte)** Rollen:
Welche Rollen gibt es in der von Ihnen gewählten Implementierungsvariante? Wie können Sie diese Rolle bestehenden oder noch zu entwickelnden Klassen des Projektes zuordnen?

Creator & ConcreteCreator → neue Klasse „MediaCreator“
Product (Abstrakt) → vorhandene Klasse „Ausleihgegenstand“
ConcreteProduct → neue Klassen „Video“, „Buch“, „Zeitschrift“, etc.

f) **(2 Punkte)** Anwendung des Entwurfsmusters:
Erweitern Sie das Klassendiagramm des Bibliotheksverwaltungs-Projektes um Typen und Methoden, die Sie zur Anwendung des Entwurfsmusters zusätzlich brauchen. Entwerfen Sie dabei Klassen für *Video*, *Buch* und *Zeitschrift*, die von der Klasse *Ausleihgegenstand* erben. Markieren Sie in dem erweiterten Diagramm alle Typen und Methoden, die eine Rolle in dem Entwurfsmuster spielen mit dem entsprechenden Rollennamen.



0,5 pro Klasse mit Rolle = 2 Punkte

(Testklasse hier zur Übersichtlichkeit ausgelassen)

g) **(3 Punkte)** Implementierung des Entwurfsmusters:
Implementieren Sie das Entwurfsmuster im Projekt (ohne Änderungen an *Ausleihgegenstand* oder *Bibliothek*).

```
public class Buch extends Ausleihgegenstand { // 1 Punkt
        public Buch(String titel) {
                super(titel, "Buch");
        }
}

public class Video extends Ausleihgegenstand {
        public Video(String titel) {
                super(titel, "Video");
        }
}

public class Zeitschrift extends Ausleihgegenstand {
        public Zeitschrift(String titel) {
                super(titel, "Zeitschrift");
        }
```

```
}

public class MediaCreator { // 1 Punkt
	public Ausleihgegenstand createMedium(String title, String typ) {
		switch (typ) {
			case "Buch":
				return new Buch(title);
			case "Zeitschrift":
				return new Zeitschrift(title);
			case "Video":
				return new Video(title);
			default:
				return new Ausleihgegenstand(title, typ);
		}
	}
}
```

BibliotheksTest: 1 Punkt

```
MediaCreator fac = new MediaCreator();
item1 = fac.createMedium("Leben des Brian", "Video");
item2 = fac.createMedium("Herr der Ringe", "Video");
item3 = fac.createMedium("How to write unmaintainable Code", "Buch");
item4 = fac.createMedium("Per Anhalter durch die Galaxis", "Buch");
item5 = fac.createMedium("Schöner Wohnen Mai 2006", "Zeitschrift");
```

h) **(2,5 Punkte)** Nutzung des Entwurfsmusters:
Üben Sie nun, wie Sie auf Basis eines statischen vorgegeben Projektes/Frameworks ein eigenes Projekt aufbauen können. Hier wird nun angenommen, dass der Inhalt des packages „Bibliothek“ als von Ihnen nicht veränderbarer Binärcode, z.B. als jar-File, vorgegeben ist (es gibt aber eine Dokumentation der Schnittstellen). Bauen Sie darauf ein „Uni-Bibliothek“-Package auf, das normale Bibliotheken zu Universitäts- Bibliotheken erweitert. In Uni-Bibliotheken gibt es die zusätzlichen Typen ‚Seminararbeit‘ und ‚Uralter Wälzer‘. Passen Sie abschließend die Testklasse an, so dass sie auch ‚Seminararbeiten‘ und ‚Uralter Wälzer‘ erzeugt.

```
public class Seminararbeit extends Ausleihgegenstand { // 0,5 Punkte
	public Seminararbeit(String titel) {
		super(titel, "Seminararbeit");
	}
}

public class UralterWälzer extends Ausleihgegenstand {
	public UralterWälzer(String titel) {
		super(titel, "Uralter Wälzer");
	}
}
```

```
public class UniMediaCreator extends MediaCreator { // 1,5 Punkte
	@Override
	public Ausleihgegenstand createMedium(String title, String typ) {
		switch (typ) {
			case "Seminararbeit":
				return new Seminararbeit(title);
```

```
                    case "Uralter Wälzer":
                          return new UralterWälzer(title);
                    default:
                          break;
              }
              return super.createMedium(title, typ);
        }
}
```

Bibliothekstest: 1 Punkt

```
MediaCreator fac = new UniMediaCreator();
item1 = fac.createMedium("Leben des Brian", "Video");
item2 = fac.createMedium("Herr der Ringe", "Video");
item3 = fac.createMedium("How to write unmaintainable Code", "Buch");
item4 = fac.createMedium("Per Anhalter durch die Galaxis", "Buch");
item5 = fac.createMedium("Schöner Wohnen Mai 2006", "Zeitschrift");
item6 = fac.createMedium("Entwicklung eines UML Tools", "Seminararbeit");
item7 = fac.createMedium("Bibel", "Uralter Wälzer");
```

Aufgabe 2. *Entwurfsmuster für Back-Up-System* **9 Punkte**

a) **(4,5 Punkte)** Im Studiensekretariat wird ein Bandlaufwerk für die langfristige Speicherung von Daten genutzt. Jeden Tag wird eine Sicherungskopie dieser für das Prüfungsbüro wichtigen Daten gemacht. Ein Schreibvorgang der Daten dauert hierbei über 60 Minuten, da das Gerät relativ alt ist. Das Studiensekretariat hat aber keine Mittel, um ein neues Gerät zu kaufen und stellt Ihnen die Aufgabe, eine Möglichkeit zu finden, das Gerät ohne Änderung der verwendeten Schnittstelle weiter zu verwenden und

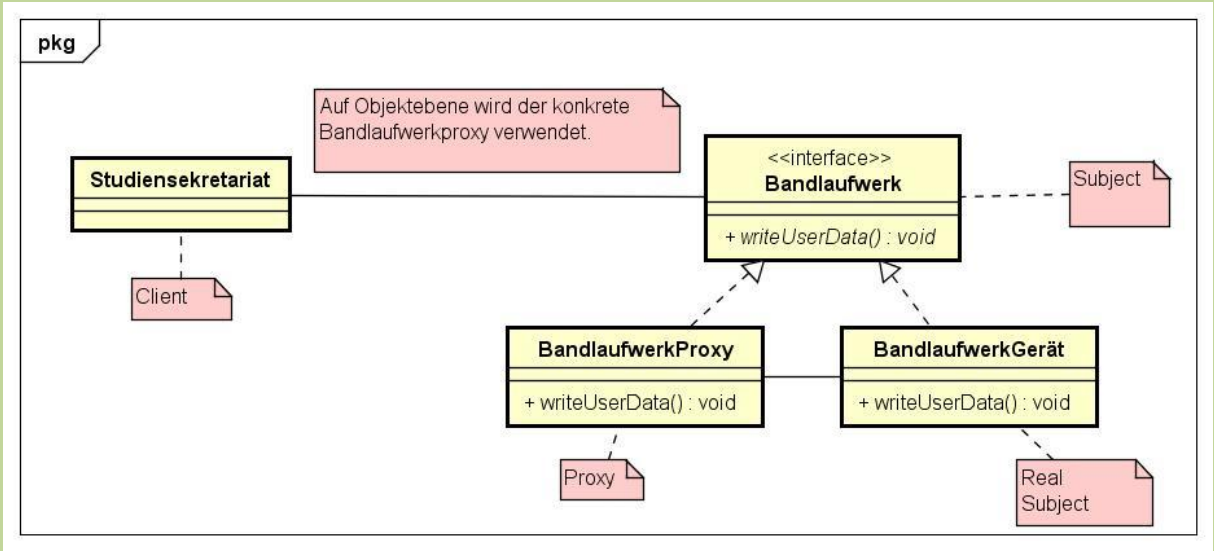
   a. Schreibvorgänge nur zwischen 22:25 und 05:45 Uhr zu erlauben bzw.

   b. Schreibvorgänge außerhalb dieses Zeitfensters oder während eines laufenden Schreibvorgangs in einer Warteschlange zwischen zu speichern und verzögert auszuführen.

   Überlegen Sie mit welchem Entwurfsmuster man die obigen Anforderungen realisieren kann, wie die Grundstruktur der Anwendung aussehen könnte, und wie das Muster auf diese Grundstruktur angewendet werden kann.

   Zeichnen Sie für Ihre Lösung ein Klassendiagramm. Für die Klassen und Methoden, die eine Rolle im Rahmen des Entwurfsmusters spielen, erläutern Sie die jeweilige Rolle in Form einer Notiz an dem zugehörigen Element.

Proxy-Pattern

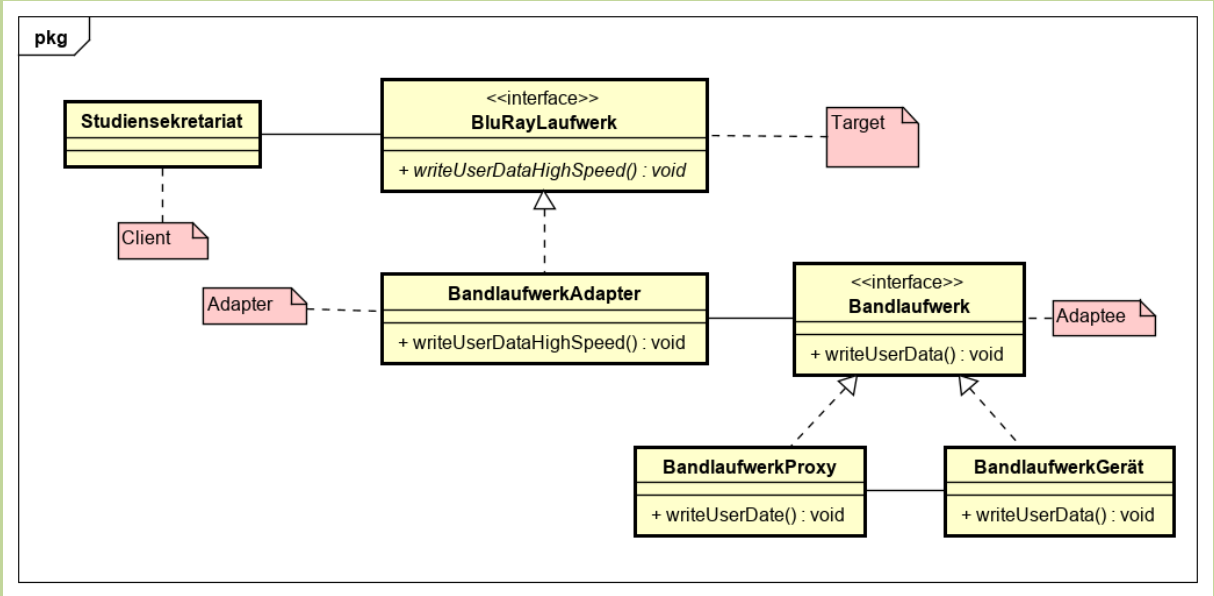


Pro Rolle 1 P + 0,5P Bezeichner = 4,5 Punkte

b) **(4,5 Punkte)** Eine neue Version der Sekretariats-Software unterstützt keine Bandlaufwerke mehr, sondern nur noch die Hochgeschwindigkeits-Sicherung auf Bluray Discs, die eine andere Schnittstelle haben als Bandlaufwerke. Wie müssen Sie den Entwurf aus a) anpassen, um dennoch das Bandlaufwerk weiter verwenden zu können? Welches Entwurfsmuster setzen Sie ein?

Zeichnen Sie wieder ein Diagramm Ihrer Lösung und erläutern Sie die Rollen in Form von Notizen.

Adapter-Pattern



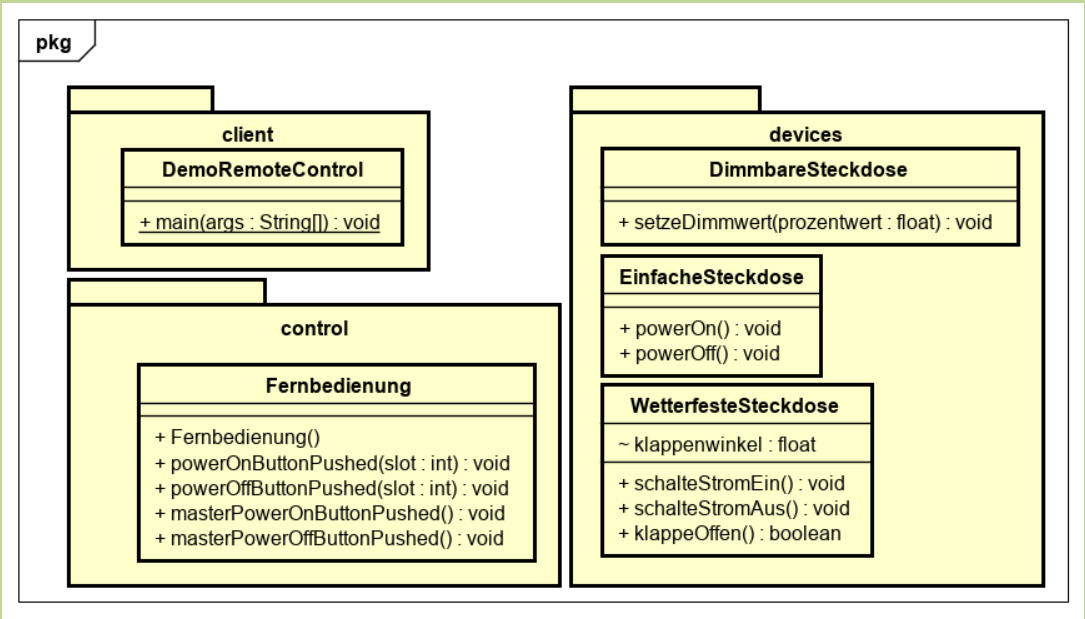
Pro Rolle 1P + 0,5P Bezeichner = 4,5 Punkte

Aufgabe 3. *Entwurfsmuster zur festlichen Erleuchtung* **27 Punkte**

Passend zur Vorweihnachtszeit hatte ein großer Discounter in der vergangenen Woche verschiedene Funksteckdosen verschiedener Hersteller im Angebot: Einfache Dosen, dimmbare Dosen und wetterfeste Dosen. Alle Steckdosen können mit einer Fernbedienung gesteuert werden.

Die Fernbedienung enthält u.a. vier frei belegbare Reihen von An-/Aus-Knöpfen, sowie eine Reihe „Master An / Master Aus“, mit der alle angeschlossenen Steckdosen gleichzeitig ein- oder ausgeschaltet werden.
Leider konnte die Steuerungs-Entwicklung nicht rechtzeitig abgeschlossen werden. Den bisherigen Code finden Sie im *readonly*-Repository als *Fernbedienung.zip*.

a) **(0,5 Punkte)** Klassendiagramm:
Erzeugen Sie ein Klassendiagramm des Projekts. Sie dürfen es manuell oder mit einem Software-Werkzeug machen.



b) **(2 Punkte)** Entwurfsmuster wählen:
Mit welchem Entwurfsmuster lassen sich die folgenden, noch fehlenden Funktionalitäten der Fernbedienung realisieren:

a. Jedes Paar An-/Aus-Knöpfe sollen einem beliebigen Gerät zugeordnet werden können.
b. Alle Dosenarten sollen unterstützt werden, ohne die Klassen zu verändern, welche die einzelnen Dosenarten darstellen.
c. Neue Geräte sollen später hinzugefügt werden können, ohne existierende Klassen verändern zu müssen.

Begründen Sie Ihre Antwort.

Das einzig relevante Verhalten der Geräte, die man integrieren soll, ist, sie an und aus zu schalten. Das lässt sich auf verschiedene Arten realisieren:

1. Eine Schnittstelle mit einer an() und einer aus() Methode und einer Implementierung je nach Geräte-Art,
2. zwei verschiedene Schnittstellen (eine nur mit an() und eine nur mit aus()), von denen pro Geräteart beide implementiert werden oder
3. eine Schnittstelle mit einer einzigen Methode run(), die pro Geräteart mal so implementiert ist, dass sie einschaltet und mal so, dass sie ausschaltet.

Lösung 1 entspricht der Anwendung des *Adapter* Patterns. Jede Implementierung der Schnittstelle ist ein Adapter. Um ein Knöpfe-Paar auf der Fernbedienung zu konfigurieren, wird dem *Paar* genau eine Adapter-Instanz zugeordnet.

Lösung 3 entspricht der Anwendung des *Command* Patterns und ist am flexibelsten, da beliebiges sonstiges Verhalten (außer An- und Ausschalten) ohne weitere Änderungen modelliert werden kann. Der Preis für die Flexibilität ist, dass man bei der Konfiguration der Fernbedienung Knöpfe *einzeln* konfigurieren muss und darauf achten muss, dass man einem Knöpfe-Paar ein Paar zueinander *passender* Commands zuordnet, also nicht versehentlich

- einem an-Knopf ein aus-Kommando zuordnet oder
- einem ein-Knopf das ein- Kommando für eine einfache Steckdose und dem dazugehörigen aus-Knopf das aus-Kommando einer dimmbaren Steckdose, oder
- einem ein-Knopf das ein- Kommando einer einfachen Steckdose und dem dazugehörigen aus-Knopf das aus-Kommando einer anderen einfachen Steckdose.

Lösung 2 macht keinen Sinn, denn sie hat die gleichen Nachteile wie 3, ist aber weniger flexibel, so dass es keinen Grund gibt, warum man die Nachteile in Kauf nehmen sollte.

Im Folgenden haben wir Ihnen daher jeweils den Code für die *Adapter* oder die *Command*-Lösung beigefügt.

Es ist auch hilfreich zu sehen, was bei obiger Aufgabenstellung **nicht** in Frage kommt: Da es nur um die Konfigurierbarkeit der Fernbedienung geht, steht der Aspekt der dynamischen Verhaltensänderung nicht im Vordergrund. Es geht also nicht um *State* oder *Strategy*. Das hätte besser gepasst, wenn man pro Steckdose nur einen Knopf gehabt hätte, dessen Verhalten sich zustandsspezifisch geändert hätte (wenn die Dose an ist, schaltet der Knopf aus, wen sie aus ist, schaltet der gleiche Knopf sie an). Hier hätten Sie dann ein degeneriertes State-Pattern gehabt, das wie bei der *Strategy* nur noch ein einziges Verhalten variiert. Es wäre trotzdem erkennbar ein *State* gewesen, denn die Logik der Zustandsänderung ist dem Gerät inhärent, es wird nicht von außen bestimmt wann welche „Strategie“ zu nutzen ist. Es wäre ziemlich sinnlos, einem Knopf, der gerade eine Dose ausgeschaltet hat, explizit eine aus-Strategie zu setzen, entgegen der Erwartung, dass der nächste Knopfdruck die Dose wieder einschaltet.

c) **(3 Punkte)** Anwendung des Entwurfsmusters:
Erweitern Sie das Klassendiagramm um die benötigten Klassen, um das Entwurfsmuster umzusetzen. Markieren Sie die Rollen der Klassen des Entwurfsmusters im Diagramm.

➔ *Siehe Diagramm am Ende der Aufgabe.*

d) **(12 Punkte)** Implementierung des Entwurfsmusters:
Implementieren Sie die benötigten Klassen und realisieren sie das Entwurfsmuster in der Steuerung der Fernbedienung.

Bewertung: Command 0,5P;

3 = 3 Punkte für ein komplett korrekte Umsetzung (0,5 implements, 0,5 Instanzvariable, 1 Konstruktor, 1 execute()-Methode)

2,5 = Je 0,5 Punkte für die Duplikate der Klassen

6 Punkte für die Fernbedienung (alle Attribute zusammen 1 Punkt, und je 1 Punkt pro Methode)

Implementierung als Adapter:

```
public interface SteckdosenAdapter {
      public void an();
      public void aus();
}

public class AdapterFuerEinfacheSteckdosen implements SteckdosenAdapter {

      EinfacheSteckdose dose;

      public AdapterFuerEinfacheSteckdosen(EinfacheSteckdose dose) {
            super();
            this.dose = dose;
      }

      @Override
      public void an() { dose.powerOn(); }
      @Override
      public void aus() { dose.powerOff(); }
}

public class AdapterFuerDimmbareSteckdosen implements SteckdosenAdapter {

      DimmbareSteckdose dose;

      public AdapterFuerDimmbareSteckdosen(DimmbareSteckdose dose) {
            super();
            this.dose = dose;
      }

      @Override
      public void an() { dose.setzeDimmwert(100); }
      @Override
      public void aus() { dose.setzeDimmwert(0); }
}

public class WetterfesteSteckdoseOnCommand implements SteckdosenAdapter {
```

```
	WetterfesteSteckdose dose;

	public WetterfesteSteckdoseOnCommand(WetterfesteSteckdose dose) {
		super();
		this.dose = dose;
	}

	@Override
	public void an() { dose.schalteStromEin(); }
	@Override
	public void aus() { dose.schalteStromAus(); }

}


public class Fernbedienung {

	private static final int NUMBER_OF_BUTTONROWS = 4;

	SteckdosenAdapter[] adapter =
		new SteckdosenAdapter [NUMBER_OF_BUTTONROWS];

	public void configure(int slot, SteckdosenAdapter adapter) {
		if ((slot < 0) || (slot >= NUMBER_OF_BUTTONROWS))
				return;
		this.adapter[slot] = adapter;
	}

	public void powerOnButtonPushed(int slot){
		SteckdosenAdapter a = adapter[slot];
		if (a != null) a.an();
	}

	public void powerOffButtonPushed(int slot){
		SteckdosenAdapter a = adapter[slot];
		if (a != null) a.aus();
	}

	public void masterPowerOnButtonPushed(){
		for (int i = 0; i < NUMBER_OF_BUTTONROWS; i++)
			powerOnButtonPushed(i);
	}

	public void masterPowerOffButtonPushed(){
		for (int i = 0; i < NUMBER_OF_BUTTONROWS; i++)
			powerOffButtonPushed(i);
	}

}
```

Implementierung als Command:

```
public interface Command {
	public void execute();
}


public class EinfacheSteckdoseOnCommand implements Command {

	EinfacheSteckdose dose;
```

```
	public EinfacheSteckdoseOnCommand(EinfacheSteckdose dose) {
		super();
		this.dose = dose;
	}

	@Override
	public void execute() {
		dose.powerOn();
	}

}

public class EinfacheSteckdoseOffCommand implements Command {

	EinfacheSteckdose dose;

	public EinfacheSteckdoseOffCommand(EinfacheSteckdose dose) {
		super();
		this.dose = dose;
	}

	@Override
	public void execute() {
		dose.powerOff();
	}
}

public class DimmbareSteckdoseOnCommand implements Command {

	DimmbareSteckdose dose;

	public DimmbareSteckdoseOnCommand(DimmbareSteckdose dose) {
		super();
		this.dose = dose;
	}

	@Override
	public void execute() {
		dose.setzeDimmwert(100);
	}
}

public class DimmbareSteckdoseOffCommand implements Command {

	DimmbareSteckdose dose;

	public DimmbareSteckdoseOffCommand(DimmbareSteckdose dose) {
		super();
		this.dose = dose;
	}

	@Override
	public void execute() {
		dose.setzeDimmwert(0);
	}
}

public class WetterfesteSteckdoseOnCommand implements Command {
```

```
        WetterfesteSteckdose dose;

        public WetterfesteSteckdoseOnCommand(WetterfesteSteckdose dose) {
                super();
                this.dose = dose;
        }

        @Override
        public void execute() {
                dose.schalteStromEin();
        }
}

public class WetterfesteSteckdoseOffCommand implements Command {

        WetterfesteSteckdose dose;

        public WetterfesteSteckdoseOffCommand(WetterfesteSteckdose dose) {
                super();
                this.dose = dose;
        }

        @Override
        public void execute() {
                dose.schalteStromAus();
        }

}

public class Fernbedienung {

        private static final int NUMBER_OF_BUTTONROWS = 4;

        Command[] onCommands = new Command[NUMBER_OF_BUTTONROWS];
        Command[] offCommands = new Command[NUMBER_OF_BUTTONROWS];

        public void powerOnButtonPushed(int slot){
                Command command = onCommands[slot];
                if (command != null)
                        command.execute();
        }

        public void powerOffButtonPushed(int slot){
                Command command = offCommands[slot];
                if (command != null)
                        command.execute();
        }

        public void masterPowerOnButtonPushed(){
                for (int i = 0; i < NUMBER_OF_BUTTONROWS; i++)
                        powerOnButtonPushed(i);
        }

        public void masterPowerOffButtonPushed(){
                for (int i = 0; i < NUMBER_OF_BUTTONROWS; i++)
                        powerOffButtonPushed(i);
        }

        public void setCommands(int slot, Command onCommand,
                                                Command offCommand) {
                if ((slot < 0) || (slot >= NUMBER_OF_BUTTONROWS))
```

```
                        return;
            onCommands[slot] = onCommand;
            offCommands[slot] = offCommand;
        }
}
```

e) **(2 Punkte)** Nutzung des Entwurfsmusters:
Schreiben Sie ein Programm, das jeder Tasten-Reihe ein Gerät zuordnet und anschließend alle Steckdosen nacheinander einmal ein und ausschaltet und dann die Master-Funktion zum Ein- und Ausschalten verwendet. Komplettieren Sie dazu die Klasse in dem Paket *client*.

```
public class DemoRemoteControl { // mit Adapter

  public static void main(String[] args) {
    EinfacheSteckdose dose1 = new EinfacheSteckdose();
    EinfacheSteckdose dose2 = new EinfacheSteckdose();
    DimmbareSteckdose dose3 = new DimmbareSteckdose();
    WetterfesteSteckdose dose4 = new WetterfesteSteckdose();

    Fernbedienung fernbedienung = new Fernbedienung();

    fernbedienung.configureButtonPair(0, new
           AdapterFuerEinfacheSteckdose(dose1));
    fernbedienung.configureButtonPair(1, new
           AdapterFuerEinfacheSteckdose(dose2));
    fernbedienung.configureButtonPair(2, new
           AdapterFuerDimmbareSteckdose(dose3));
    fernbedienung.configureButtonPair(3, new
           AdapterFuerWetterfesteSteckdose(dose4));

    for (int i = 0; i < 4; i++) {
     fernbedienung.powerOnButtonPushed(i);
     fernbedienung.powerOffButtonPushed(i);
    }

    fernbedienung.masterPowerOnButtonPushed();
    fernbedienung.masterPowerOffButtonPushed();
  }
}
```

```
public class DemoRemoteControl { // mit Command

      public static void main(String[] args) {
            EinfacheSteckdose dose1 = new EinfacheSteckdose();
            EinfacheSteckdose dose2 = new EinfacheSteckdose();
            DimmbareSteckdose dose3 = new DimmbareSteckdose();
            WetterfesteSteckdose dose4 = new WetterfesteSteckdose();
            Fernbedienung fernbedienung = new Fernbedienung();

            fernbedienung.setCommands(0,
                    new EinfacheSteckdoseOnCommand(dose1),
                    new EinfacheSteckdoseOffCommand(dose1));
```

```
            fernbedienung.setCommands(1,
                    new EinfacheSteckdoseOnCommand(dose2),
                    new EinfacheSteckdoseOffCommand(dose2));
            fernbedienung.setCommands(2,
                    new DimmbareSteckdoseOnCommand(dose3),
                    new DimmbareSteckdoseOffCommand(dose3));
            fernbedienung.setCommands(3,
                    new WetterfesteSteckdoseOnCommand(dose4),
                    new WetterfesteSteckdoseOffCommand(dose4));

            for (int i = 0; i < 4; i++) {
                  fernbedienung.powerOnButtonPushed(i);
                  fernbedienung.powerOffButtonPushed(i);
            }

            fernbedienung.masterPowerOnButtonPushed();
            fernbedienung.masterPowerOffButtonPushed();
      }
}
```

f) **(2,5 Punkte)** Kombination von Mustern*:*
Überlegen Sie (erst mal als UML-Diagramm) wie sie den Entwurf aus Aufgabenteil (c) so anpassen können, dass Sie das Abstract Factory Entwurfsmuster bei der Belegung der Knöpfe einsetzen. Markieren sie jede Klasse, die eine Rolle im Abstract Factory Entwurfsmuster spielt mit der jeweiligen Rolle.

siehe Klassendiagramm am Ende des Dokuments

g) **(5 Punkte)** Implementierung des Entwurfsmusters*:*
Kopieren Sie das Programm aus *Aufgabenteil d) + e)* und implementieren Sie dessen laut *Aufgabenteil f)* um die Nutzung des *Abstract Factory* Entwurfsmusters erweiterte Version.

Bewertung:

Steckdose Interface 0,5P

Eine korrekte Steckdose 1P, beide weiteren je 0,5P = 2P

Anpassung Fernbedienung 1P

Anpassung DemoRemote 1,5P

Implementierung mit Adapter:

```
public interface AbstractSteckdosenAdapterFactory { //Adapter-Factory

      SteckdosenAdapter getAdapter ();
}

public class EinfacheSteckdoseAdapterFactory implements SteckdosenAdapter {
      @Override
      public Adapter getAdapter() {
            return new AdapterFuerEinfacheSteckdose(new
                  EinfacheSteckdose());
      }
}
```

```
public class DimmbareSteckdoseAdapterFactory implements SteckdosenAdapter {
	@Override
	public Adapter getAdapter() {
		return new AdapterFuerDimmbareSteckdose(new
			DimmbareSteckdose());
	}
}

public class WetterfesteSteckdoseAdapterFactory implements
SteckdosenAdapter {
	@Override
	public Adapter getAdapter() {
		return new AdapterFuerWetterfesteSteckdose(new
			WetterfesteSteckdose());
	}
}

public class Fernbedienung {

	[…]

	public void setDevice(int slot, AbstractSteckdosenAdapterFactory
		adapter){
		if ((slot < 0) || (slot >= NUMBER_OF_BUTTONROWS))
			return;

		this.adapter[slot]  = adapter.getAdapter();
	}
}



public class DemoRemoteControlWithFactory {

	public static void main(String[] args) {

		Fernbedienung fb = new Fernbedienung();

		fb.setDevice(0, new EinfacheSteckdoseAdapterFactory());
		fb.setDevice(1, new EinfacheSteckdoseAdapterFactory());
		fb.setDevice(2, new DimmbareSteckdoseAdapterFactory());
		fb.setDevice(3, new WetterfesteSteckdoseAdapterFactory());

		for (int i = 0; i < 4; i++) {
			fb.powerOnButtonPushed(i);
			fb.powerOffButtonPushed(i);
		}

		fb.masterPowerOnButtonPushed();
		fb.masterPowerOffButtonPushed();
	}
}
```

Implementierung mit Command:

```
public interface SteckdoseCommand { //Command-Factory

	Command getOnCommand();
	Command getOffCommand();
}
public class EinfacheSteckdoseCommandFactory implements SteckdoseCommand {
```

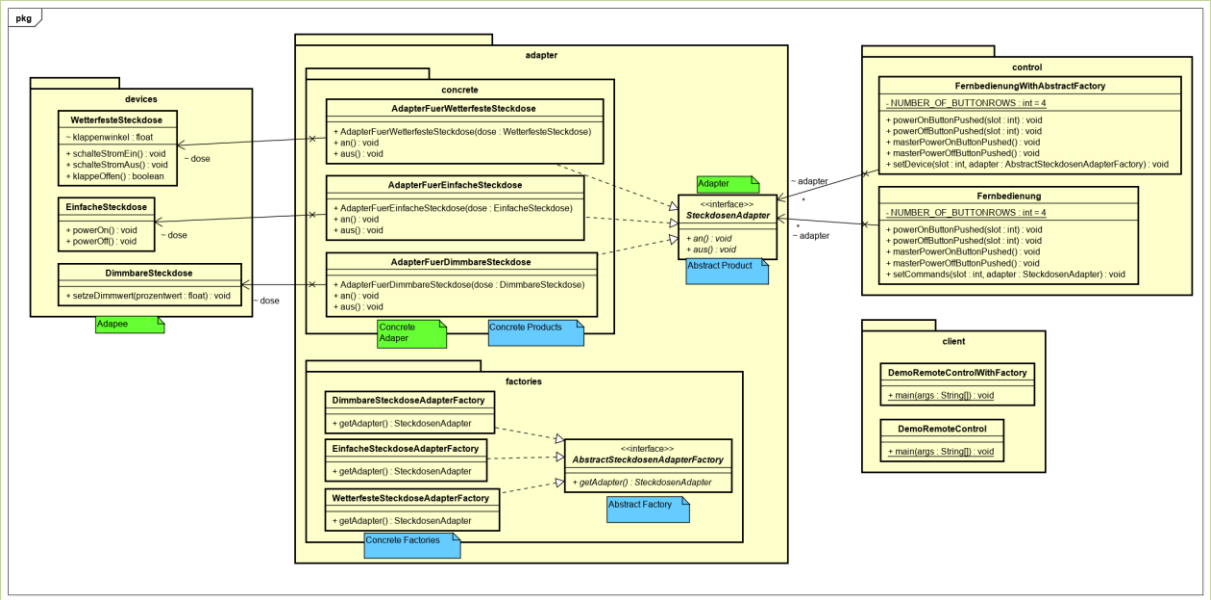
```
        EinfacheSteckdose dose;

        public EinfacheSteckdose() {

                this.dose = new EinfacheSteckdose();

        }

        @Override
        public Command getOnCommand() {
                return new EinfacheSteckdoseOnCommand(this.dose);
        }

        @Override
        public Command getOffCommand() {
                return new EinfacheSteckdoseOffCommand(this.dose);
        }
}

public class DimmbareSteckdoseCommandFactory implements SteckdoseCommand {

        DimmbareSteckdose dose;

        public DimmbareSteckdose() {

                this.dose = new DimmbareSteckdose();

        }

        @Override
        public Command getOnCommand() {
                return new DimmbareSteckdoseOnCommand(this.dose);
        }

        @Override
        public Command getOffCommand() {
                return new DimmbareSteckdoseOffCommand(this.dose);
        }
}

public class WetterfesteSteckdoseCommandFactory implements SteckdoseCommand
{

        WetterfesteSteckdose dose;

        public WetterfesteSteckdose() {

                this.dose = new WetterfesteSteckdose();

        }

        @Override
        public Command getOnCommand() {
                return new WetterfesteSteckdoseOnCommand(this.dose);
        }

        @Override
        public Command getOffCommand() {
                return new WetterfesteSteckdoseOffCommand(this.dose);
        }
}

public class Fernbedienung {

        […]

        public void setDevice(int slot, SteckdoseCommand
        dose){
                if ((slot < 0) || (slot >= NUMBER_OF_BUTTONROWS))
                        return;
```

```
            onCommands[slot]  = dose.getOnCommand();
            offCommands[slot] = dose.getOffCommand();
      }
}


public class DemoRemoteControlWithFactory {
      public static void main(String[] args) {
            Fernbedienung fb = new Fernbedienung();

            fb.setDevice(0, new EinfacheSteckdoseCommandFactory());
            fb.setDevice(1, new EinfacheSteckdoseCommandFactory());
            fb.setDevice(2, new DimmbareSteckdoseCommandFactory());
            fb.setDevice(3, new WetterfesteSteckdoseCommandFactory());

            for (int i = 0; i < 4; i++) {
                  fb.powerOnButtonPushed(i);
                  fb.powerOffButtonPushed(i);
            }

            fb.masterPowerOnButtonPushed();
            fb.masterPowerOffButtonPushed();
      }
}
```

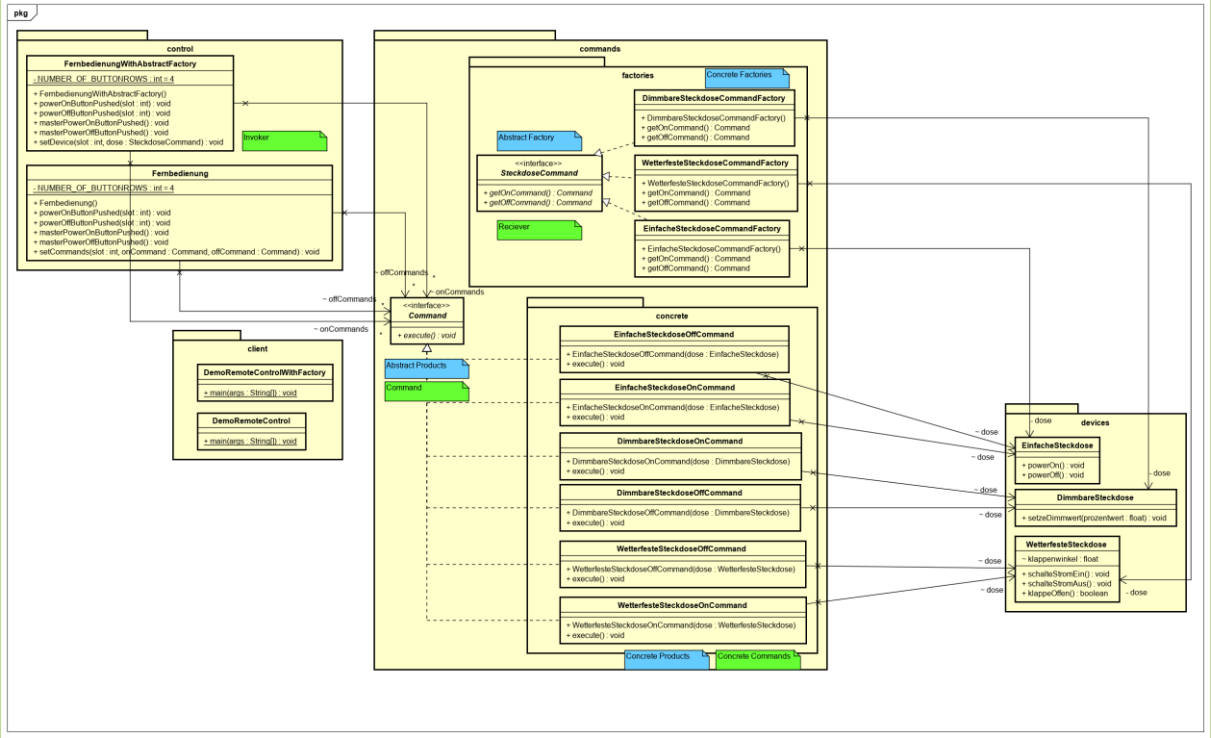
Frage: *Ist der Aufruf der super()-Methode bei der Implementation eines Interfaces (*z.B. in *WetterfesteSteckdoseOnCommand*) *im Konstruktor nicht falsch oder überflüssig?*

Da jede Klasse grundsätzlich von der Klasse *Object* erbt, ist dies nicht falsch. Der Aufruf der super()-Methode im Konstruktor ist sogar empfehlenswert und wird andernfalls durch den Compiler umgesetzt.

Implementierung mit Adapter



Implementierung mit Command



Für Aufgabe c:

Adapter: Adapter 0,5P + ConcreteAdapter 1P + Adaptee 1P + 0,5P Bezeichner = 3 Punkte

Command: Reciever 0,5P + Invoker 0,5P + Command 0,5P + Concrete Commands (On + Off) 1P = 2,5 Punkte + 0,5P Bezeichner = 3 Punkte

Für Aufgabe f: Abstract Factory 0,5P + Concrete Factories 0,5P + Abstract Product 0,5P + Concrete Products (On + Off) 1P = 2,5 Punkte

Theoretische Aufgaben

Aufgabe 4. *Proxy-Entwurfsmuster* **6 Punkte**

a) **(1 Punkt)** Erläutern Sie kurz die Motivation und die Funktionsweise des Proxy-Entwurfmusters.

- Absicht
  - Stellvertreter für ein anderes Objekt
  - bietet Kontrolle über Objekt-Erzeugung und -Zugriff
- Motivation
  - kostspielige Objekt-Erzeugung verzögern (z.B.: große Bilder)
  - verzögerte Objekterzeugung soll Programmstruktur nicht global verändern
  - Beim Zugriff gewisse Zusatzoperationen ausführen (checks, Buchhaltung, Benachrichtigung, Pufferung, …)

b) **(2 Punkte)** Welches sind die teilnehmenden Rollen (Klassen) des Proxy-Entwurfsmusters? Welche Aufgaben übernehmen diese? Welche Operationen sind notwendig?

- Proxy
  - bietet gleiches Interface wie "RealSubject"
  - referenziert eine "RealSubject"-Instanz
  - kontrolliert alle Aktionen auf dem "RealSubject"
- Subject
  - definiert das gemeinsame Interface
- RealSubject
  - das Objekt das der Proxy vertritt
  - eigentliche Funktionalität
- Zusammenspiel
  - selektives Forwarding

c) **(3 Punkte)** Nennen Sie mindestens drei Beispiele für die Anwendung eines Proxy-Entwurfsmusters und erläutern Sie kurz die Motivation für den jeweiligen Pattern-Einsatz.

- Virtueller Proxy
  - verzögerte Erzeugung "teurer" Objekte bei Bedarf
  - Beispiel: Bilder in Dokument, persistente Objekte
- Remote Proxy
  - Zugriff auf entferntes Objekt
  - Objekt-Migration
  - Beispiele: CORBA, RMI, mobile Agenten
- Concurrency Proxy
  - nur eine direkte Referenz auf RealSubject
  - locking des RealSubjects vor Zugriff (threads)
- Copy-on-Write
  - kopieren erhöht nur internen "CopyCounter"
  - wirkliche Kopie bei Schreibzugriff und "CopyCounter">0
  - Verzögerung teurer Kopier-Operationen

∑ 59,5 Punkte