

3 Das Projekt Stromkreis

Bemerkung:

In der 11. Jahrgangsstufe wird mit Hilfe des Programms *Automat* (entwickelt von Albert Wiedemann) die zustandsorientierte Modellierung eingeführt. Anschließend wird zur Implementierung das Konzept der Zustandstabellen und Zustandsübergangstabellen in BlueJ übernommen und eine elektrische Schaltung simuliert. Das hier vorgestellte Projekt *Stromkreis* ist eine Weiterentwicklung, in der das *State-Muster* zur Beschreibung der Zustände verwendet wird.

Die Entwurfsmuster werden beschrieben im Buch *Entwurfsmuster von Kopf bis Fuß* von Eric Freeman und Elisabeth Freeman.

Die Gestaltungsmöglichkeiten einer grafischen Oberfläche mit Swing werden in *The JFC Swing Tutorial, Second Edition* von Kathy Walrath u.a. dargestellt.

Im Projekt *Stromkreis* wird eine einfache Schaltung simuliert. Die grafische Oberfläche, also die Sicht auf die Schaltung, besteht aus den beiden Teilen Darstellung und Einstellung. Dahinter steckt natürlich ein Schaltungsmodell, das die beiden Sichten steuert, also auf Benutzereingaben reagiert.

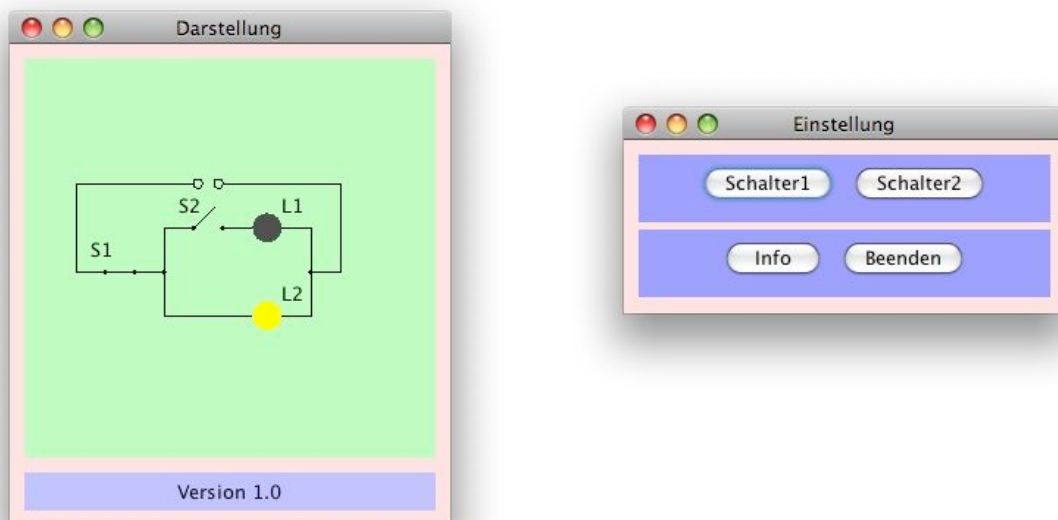


Abbildung 3.1: Elemente des Stromkreises

Drückt der Benutzer auf den Button *Schalter1* bzw. *Schalter2*, so werden in der Schaltskizze die entsprechenden Schalter geschlossen oder geöffnet und die jeweiligen Lampen leuchten.

Übung 3.1:

In der unten dargestellten Tabelle gilt:

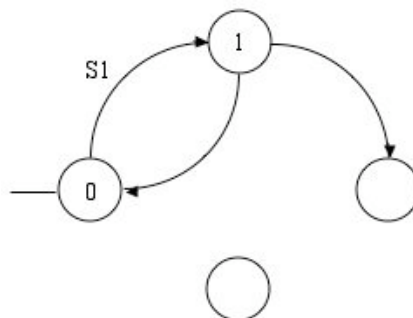
0: Schalter geöffnet 1: Schalter geschlossen

0: Lampe aus 1: Lampe an

Vervollständige die Kombination aus Zustandstabelle und Zustandsübergangstabelle.

Zustand	S1	S2	L1	L2	Folgezustand bei	
					S1	S2
0	0	0	0	0	1	
1	1	0				
2	1	1				
3	0	1				

Vervollständige das Zustandsübergangsdiagramm.



Nun hast du dich mit dieser elektrischen Schaltung soweit beschäftigt, so dass du diese programmieren kannst. Hierzu wirst du ein weiteres Entwurfsmuster anwenden, das so genannte **State-Muster**.

3.1 Das State-Muster

3.1.1 Einführung in das State-Muster

Bei diesem Beispiel der Schaltung handelt es sich um einen Automaten. Es existieren vier verschiedenen Zustände (zustand 0, zustand1, zustand2 und zustand3) sowie zwei verschiedenen Ereignisse (s1Druecken, s2Druecken). Das Verhalten der Schaltung auf die jeweiligen Ereignisse hängt ab von seinem momentanen Zustand. Wie in Abbildung 3.2 dargestellt, bewirkt ein Drücken auf den Schalter S1 einen Übergang

in den zustand1, wenn sich die Schaltung in zustand0 befindet ,
 in den zustand0, wenn sich die Schaltung in zustand1 befindet,
 in den zustand3, wenn sich die Schaltung in zustand2 befindet,
 in den zustand2, wenn sich die Schaltung in zustand3 befindet.

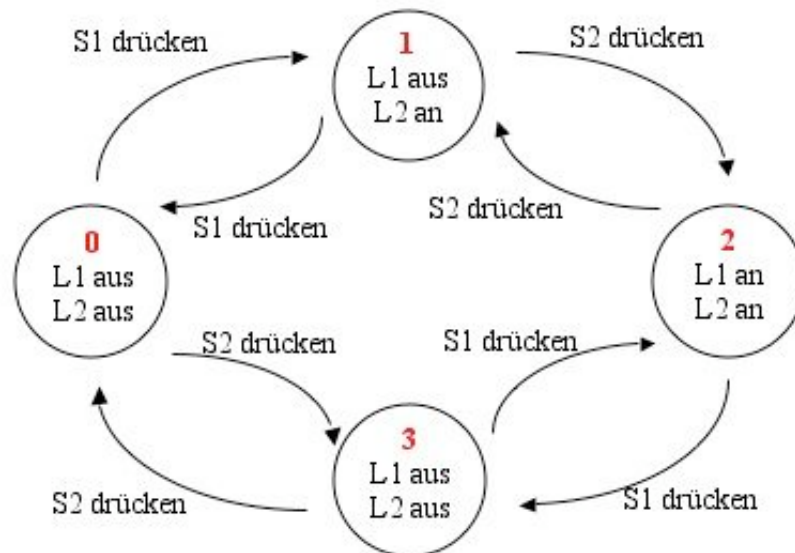


Abbildung 3.2: Zustandsübergangsdiagramm des Automaten Schaltung

Grundsätzlich gilt also, dass das Verhalten abhängig von seinem Zustand ist. Die einfachste Möglichkeit ist, in einer Methode `s1Druecken()` mit Hilfe von einer Reihe von `if`-Anweisungen oder mit Hilfe einer `switch`-Anweisung die Zustände und ihre zugehörigen Aktionen abzufragen.

In meinem Skript der 11.Klasse im Kapitel *ZOM_BlueJ* habe ich ein anderes Modellierungskonzept verwendet. Hier wurde die Zustandsübergangstabelle aus der Übung 3.1 implementiert und hieraus die entsprechenden Folgezustände entnommen.

Diese beiden Möglichkeiten führen bei einer Veränderung oder Erweiterung des Automaten zu einem enormen Aufwand der Programmierung von neuem Quelltext und somit wahrscheinlich auch zu neuen Fehlern. Deswegen soll wieder oberstes Prinzip gelten:

Kapseln, was variiert

Du musst in deinem Quelltext alle Teile nehmen, die sich ändern können, und sie kapseln, damit du diese veränderlichen Teile später ändern oder erweitern kannst, ohne dass das Auswirkungen auf diejenigen Teile hat, die sich nicht ändern. Somit hat eine Quelltextveränderung weniger unvorhergesehene Folgen und die Flexibilität bei Erweiterungen wird erhöht.

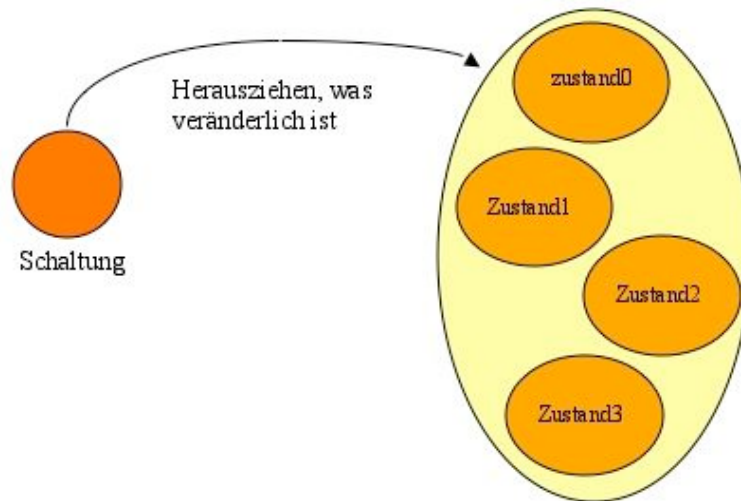


Abbildung 3.3: Das, was veränderlich ist, von dem trennen, was gleich bleibt

Für jeden Zustand des Automaten wird also eine separate Zustandsklasse implementiert. Diese Zustandsklassen sind dann für das Verhalten des Automaten verantwortlich. Somit muss jeder Zustand seine eigenen Aktionen besitzen.



Abbildung 3.4: Modellierung der Zustandsklassen

Übung 3.2:

Um diese Zustände zu implementieren, musst du das Verhalten der Zustandsklassen bei Aufruf der einzelnen Aktionen beschreiben.

Beispielsweise steht in der Klasse *Zustand0* in der Methode

```
s1Druecken()   Setze Schaltung in den Zustand1
s2Druecken()   Setze Schaltung in den Zustand3.
```

Formuliere die entsprechenden Aktionen für die restlichen drei Zustände. Verwende hierzu das Zustandsübergangsdiagramm aus Abbildung 3.2.

Du weißt nun also, wie jeder Zustand sein entsprechendes Verhalten festlegt. Alle diese Zustände werden nun von der Klasse *Schaltung* erzeugt und verwaltet.

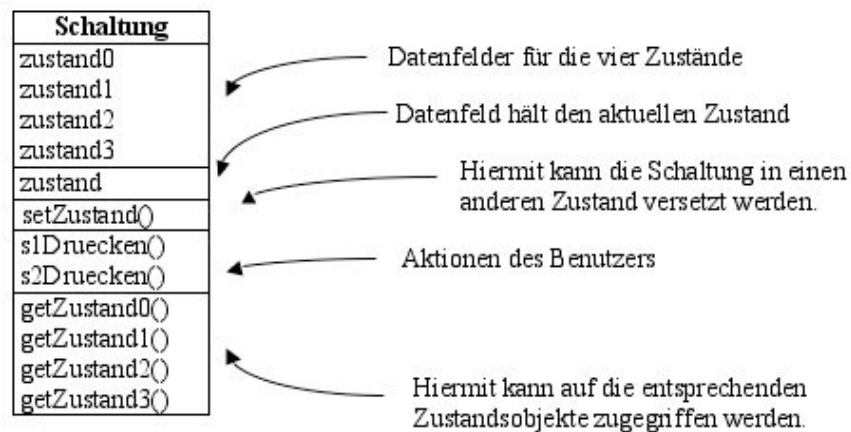


Abbildung 3.5: Modellierung der Klasse *Schaltung*

Die Klasse *Schaltung* benötigt also vier Datenfelder *zustand0*, ... *zustand3*, um die vier Zustände zu halten, und ein weiteres Datenfeld *zustand* für den aktuellen Zustand, in dem sich die Schaltung momentan befindet. Der Konstruktor erzeugt die vier Zustandsobjekt und belegt das Datenfeld *zustand* mit dem Startzustand.

Die Methode *setZustand()* versetzt die Schaltung während der Laufzeit in einen anderen Zustand. Anschließend folgen die beiden Methoden *s1Druecken()* und *s2Druecken()* für die Aktionen des Benutzers, wenn er auf die Schalter S1 oder S2 drückt. Zum Schluss benötigst du noch Methoden *getZustandx()*, um auf die einzelnen Zustandsobjekte zugreifen zu können.

Hast du dir überlegt, welche Objekte das Datenfeld *zustand* halten soll? Wie Abbildung 3.5 darstellt, soll hier der momentane Zustand gespeichert werden. Dieser momentane Zustand ist aber vom Datentyp *Zustand0* oder *Zustand1* usw. Wie kann eine Variable verschiedene Datentypen speichern?

Dieses Problem löst du mit Hilfe eines Interfaces *Zustand*. Hier wird also der Polymorphismus verwendet wird. Die Schaltung verwendet den Supertyp *Zustand*, die tatsächlichen Objekttypen *Zustand0*, ... , *Zustand3* muss sie nicht kennen. Somit können später an die Schaltung alle möglichen Zustände angeknüpft werden, wichtig ist nur, dass diese das Interface *Zustand* implementiert haben. Es ist also immer gewährleistet, dass all diese Klassen auch die Methoden *s1Druecken()* und *s2Druecken()* besitzen. Wird nun später vielleicht ein neuer Zustand eingeführt, so muss der Quelltext der Klasse *Schaltung* nur noch minimal verändert werden.

Schaltung und alle Zustände sind locker gebunden, sie können miteinander agieren, müssen aber nur wenige Kenntnisse voneinander besitzen.

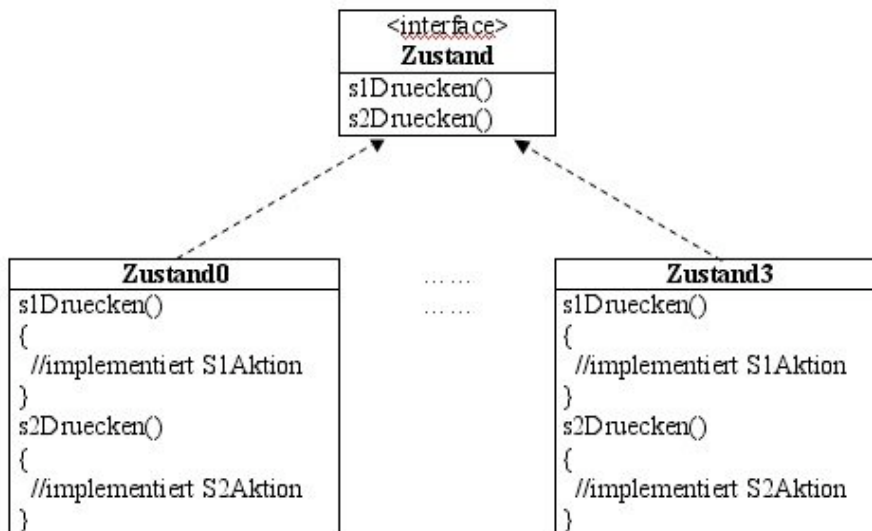
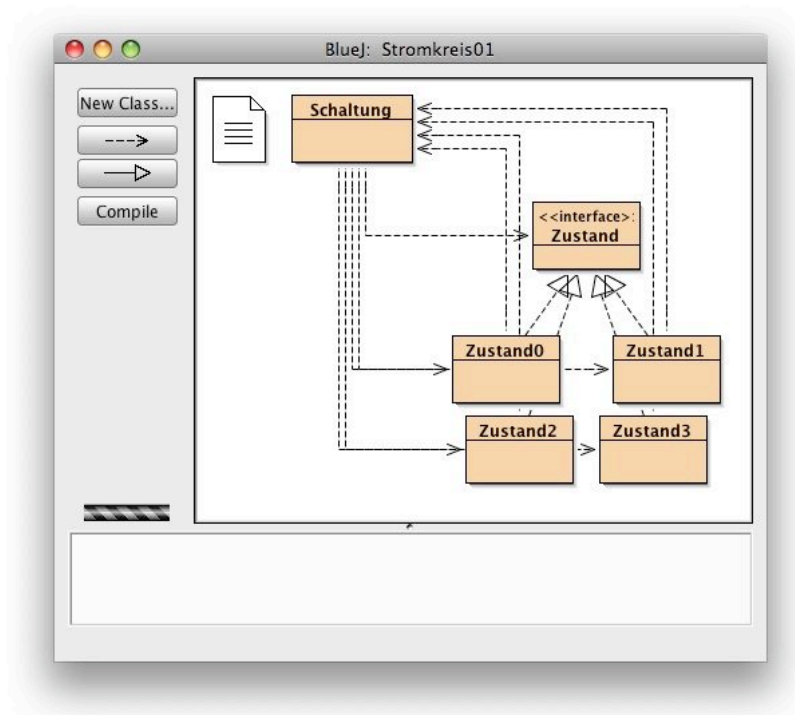


Abbildung 3.6: Interface und Zustandsklassen

Bevor du jedoch dieses *State*-Muster implementierst, solltest du dir die Zusammenhänge an Hand des Klassendiagramms in Abbildung 3.7 verdeutlichen.

Abbildung 3.7: Klassendiagramm des Projekts *Stromkreis01*

Du erkennst die vier Zustände `Zustand0`, ..., `Zustand3`. Da sie alle das Interface `Zustand` implementieren, kann die `Schaltung`

- a) jeden einzelnen Zustand als ihren momentanen Zustand im Datenfeld

zustand halten,

- b) auf die Methoden *s1Druecken()* und *s2Druecken()* des jeweils momentanen Zustands zurückgreifen, ohne die vom Zustand abhängigen Details dieser Methoden zu kennen.

Nun wird es Zeit, dass du eine solche Zustandsklasse implementierst.

3.1.2 Implementierung des Interfaces *Zustand*

Du wirst mit dem Interface *Zustand* beginnen.

```
public interface Zustand
{
    void s1Druecken();
    void s2Druecken();
}
```

Abbildung 3.8: Quelltext des Interfaces *Zustand*

3.1.3 Implementierung der Zustandsklasse *Zustand0*

Nun zur Klasse *Zustand0*. Die Abbildung 3.9 zeigt den Quelltext dieser Klasse:

```
public class Zustand0 implements Zustand
{
    private Schaltung schaltung;

    public Zustand0(Schaltung schaltung)
    {
        this.schaltung = schaltung;
    }

    public void s1Druecken()
    {
        schaltung.setZustand(schaltung.getZustand1());
    }

    public void s2Druecken()
    {
        schaltung.setZustand(schaltung.getZustand3());
    }
}
```

Abbildung 3.9: Quelltext der Klasse *Zustand0*

Der Quelltext der Klasse *Zustand0* sieht jetzt nicht sehr schwierig aus. Als erstes musst du natürlich das Interface *Zustand* implementieren. Mit Hilfe des

Konstruktors erhältst du eine Referenz auf die Schaltung, die du im Datenfeld *schaltung* festhältst. Wenn der Benutzer auf den Schalter S1 drückt, wird der Zustand der Schaltung in *Zustand1* geändert. Falls der Benutzer jedoch auf den Schalter S2 drückt, geht die Schaltung in den *Zustand3* über. Dies wurde in Abbildung 3.2 dargestellt. Wie das genau funktioniert, wirst du gleich sehen.

3.1.4 Implementierung der Klasse Schaltung

Zuerst wird jedoch noch die Klasse *Schaltung* implementiert. Die Abbildung 3.10 zeigt den zugehörigen Quelltext:

```
public class Schaltung
{
    private Zustand zustand0;
    private Zustand zustand1;
    private Zustand zustand2;
    private Zustand zustand3;

    Zustand zustand;

    public Schaltung()
    {
        zustand0 = new Zustand0(this);
        zustand1 = new Zustand1(this);
        zustand2 = new Zustand2(this);
        zustand3 = new Zustand3(this);

        zustand = zustand0;
    }

    public void setZustand(Zustand zustand)
    {
        this.zustand = zustand;
    }

    public void s1Druecken()
    {
        zustand.s1Druecken();
    }

    public void s2Druecken()
    {
        zustand.s2Druecken();
    }

    public Zustand getZustand0()
    {
        return zustand0;
    }
}
```

```
public Zustand getZustand1()
{
    return zustand1;
}

public Zustand getZustand2()
{
    return zustand2;
}

public Zustand getZustand3()
{
    return zustand3;
}
}
```

Abbildung 3.10: Quelltext der Klasse *Schaltung*

Im Konstruktor wird für jeden Zustand ein Zustandsobjekt erzeugt. Beachte, dass die Schaltung den Supertyp *Zustand* verwendet, deswegen wurden die Datenfelder *zustand0* ... *zustand3* als Datentyp *Zustand* deklariert, aber als Objekte des entsprechenden Datentyps *Zustand0*, ... , *Zustand3* erzeugt. Zum Schluss benötigst du noch ein Datenfeld *zustand*, das den aktuellen Zustand hält.

Bemerkung:

Die Verwendung des Interfaces *Zustand* erlaubt polymorphe Variablen und Methodenaufrufe. Somit können Objekte von Subtypen (also speziellere Objekte) einheitlich behandelt werden. Lediglich bei der Erzeugung dieser Subtypen-Objekte muss tatsächlich der Subtyp genannt werden:

```
private Zustand zustand0 = new Zustand0(this);
```

Hiermit ist gewährleistet, dass die Variable *zustand0* vom Typ *Zustand* ist. Die Objekte aller Zustandsklassen können somit von der Schaltung einheitlich verwendet werden.

Die Methode *setZustand()* ermöglicht es den vier Zustandsobjekten, die Schaltung in den neu gewünschten Zustand zu setzen.

Die beiden Methoden *s1Druecken()* und *s2Druecken()* reagieren auf die Aktionen des Benutzers. Je nach aktuellem Zustand muss die Schaltung auf verschiedene Weise auf diese Aktionen reagieren. Genaue Kenntnisse über diese Reaktion haben allerdings nur die Zustandsobjekte. Deswegen wird einfach weiter an den jeweiligen Zustand delegiert.

Zum Abschluss benötigst du noch Methoden, mit denen du auf die einzelnen Zustandsobjekte zugreifen kannst.

3.1.5 Zusammenspiel zwischen der Klasse *Schaltung* und den Zustandsklassen

Betrachte jetzt einmal zusammenfassend, was du bereits gemacht hast. Die *Schaltung* besitzt also jetzt ein Objekt von jeder Zustandsklasse. Zu Beginn befindet sich die *Schaltung* im Startzustand *zustand0*.

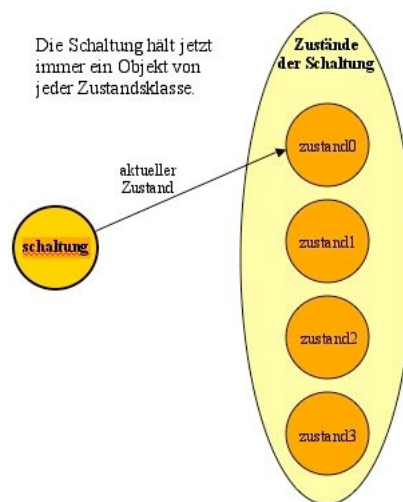


Abbildung 3.11: Startzustand der Schaltung

Nun drückt der Benutzer auf den Schalter S1. Deshalb wird in der Klasse *Schaltung* die Methode *s1Druecken()* aufgerufen.

```
public void s1Druecken()
{
    zustand.s1Druecken();
}
```

Abbildung 3.12: Die Methode *s1Druecken()* in der Klasse *Schaltung*

Da aber die *Schaltung* selbst keine Kenntnis darüber besitzt, was nun zu tun ist, delegiert sie diese Aktion weiter an den im Datenfeld *zustand* enthaltenen aktuellen Zustand, in diesem Fall also an *zustand0*.

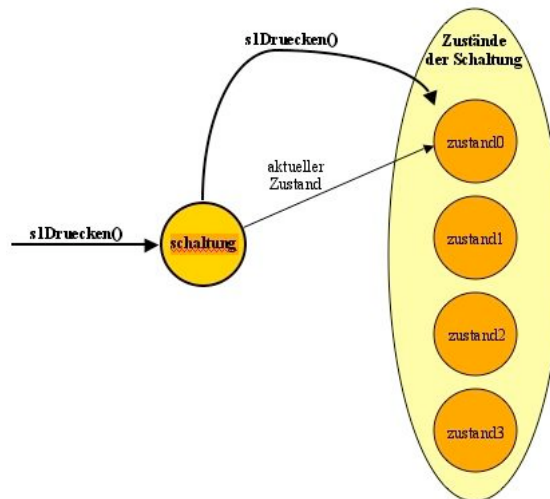


Abbildung 3.13: Die Aktion wird an den aktuellen Zustand delegiert

Nur der aktuelle Zustand *zustand0* weiß nun genau, was zu tun ist, nämlich die Schaltung in den Zustand *zustand1* zu versetzen.

```
public void s1Druecken()
{
    schaltung.setZustand(schaltung.getZustand1());
}
```

Abbildung 3.14: Die Methode *s1Druecken()* in der Klasse *Zustand0*

Da die Schaltung von jeder Zustandsklasse ein Objekt besitzt, kann *zustand0* von der Schaltung mit *getZustand1()* das Objekt *zustand1* anfordern und die Schaltung mit *setZustand()* in den *zustand1* versetzen.

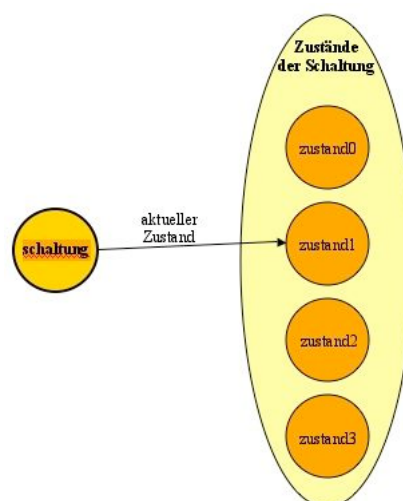


Abbildung 3.15: Schaltung wechselt in den Zustand *zustand1*

Das war jetzt sehr viel Theorie. Nun solltest du jedoch in der Lage sein, die weiteren Zustandsübergänge selbst zu untersuchen.

Übung 3.3:

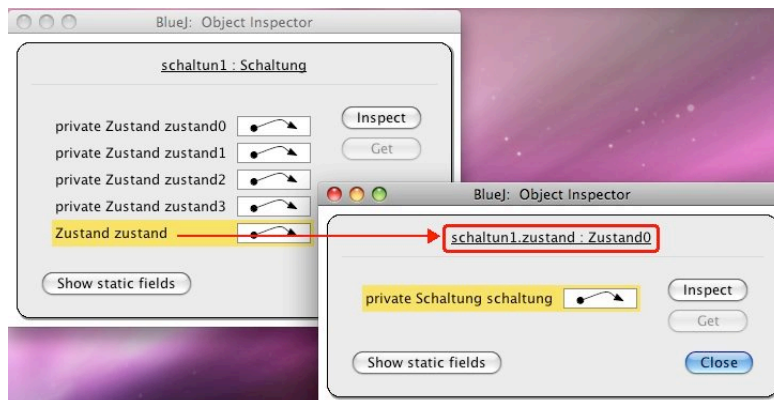
Skizziere analog wie die Abbildungen 3.10, 3.12 und 3.14 dargestellt ausgehend vom Zustand1 die restlichen Zustandsübergänge bei Drücken auf die Schalter S1 bzw. S2. Verwende dazu auch die Abbildung 3.2.

Übung 3.4:

Implementiere das Interface *Zustand* sowie die Klassen *Schaltung* und *Zustand0*. Schreibe zu jeder Methode in einem Kommentar, welche Aufgaben diese zu erledigen hat und wozu die Parameter verwendet werden.

Übung 3.5:

Implementiere die restlichen drei Zustandsklassen, die du in Übung 3.3 skizziert hast. Zum Schluss sollte dein Klassendiagramm wie in Abbildung 3.7 aussehen.



Der Inspektor zeigt, dass zu Beginn die Schaltung sich im Startzustand *zustand0* befindet.

Nach Aufruf der Methode *s1Druecken()* wechselt die Schaltung in den Zustand *zustand1*.

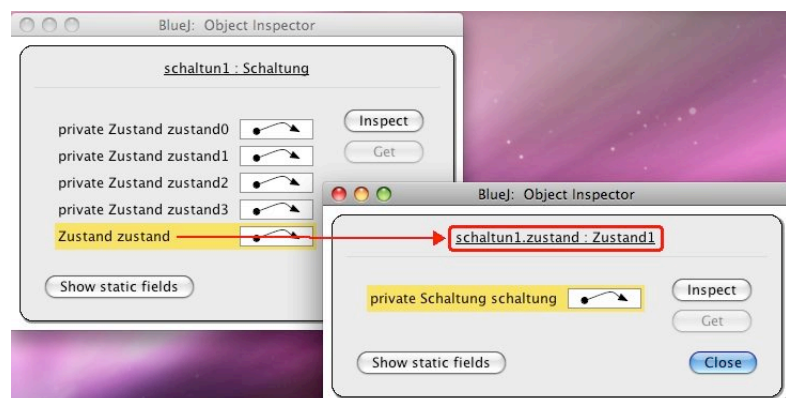


Abbildung 3.16: Ereignis *s1Druecken()* führt zu einem Zustandsübergang

Übung 3.6:

Überprüfe mit Hilfe des Inspektors alle Zustandsübergänge beim Aufruf der Methoden `s1Druecken()` und `s2Druecken()`. Vergleiche diese mit der Abbildung 3.16.

Alternativ kannst du auch in den entsprechenden Methoden `System.out.println()`-Anweisungen verwenden.

Bemerkung:

Im Projekt *Stromkreis01b* wurden in den entsprechenden Methoden `System.out.println()`-Anweisungen implementiert. Zusätzlich besitzt es die Klasse *Testlauf*, die einige Aktionen des Kunden simuliert.

3.1.6 Definition des State-Musters

Nun zu Definition des *State-Musters*:

Das *State-Muster* ermöglicht einem Objekt, sein Verhalten zu verändern, wenn sein interner Zustand sich ändert.

Der typische Anwendungsfall für das *State-Muster* ist folgende Situation:

In einer Klasse haben viele Methoden in Abhängigkeit vom Objektzustand ein völlig unterschiedliches Verhalten, so dass die Implementierung dieser Methoden jeweils im Wesentlichen wie folgt aussieht:

```
wenn Zustand1, dann Aktion1  
wenn Zustand2, dann Aktion2  
...  
wenn ZustandN, dann AktionN
```

Die Idee beim *State-Muster* ist es, die zustandsabhängigen Methoden und Daten in ein eigenes dafür erzeugtes Zustandsobjekt auszulagern. Das ursprüngliche Objekt delegiert dann die betreffenden Funktionen an das Zustandsobjekt. Ein Zustandswechsel wird nicht mehr durch das Ändern der Instanzvariablen erreicht, sondern die Zustandsreferenz auf das entsprechende Zustandsobjekt mit Hilfe der Methode `setZustand()` gesetzt

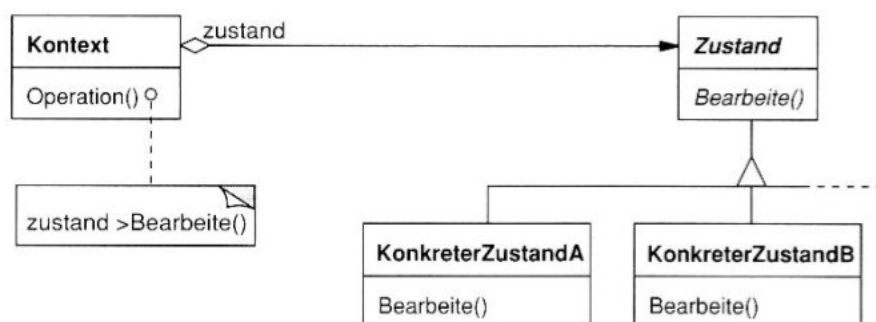


Abbildung 3.17: Das State-Musters

In diesem Beispiel ist der Kontext die Klasse *Schaltung*, die mehrere innere Zustände besitzt. Das Interface *Zustand* (manchmal ist es auch sinnvoll, hierzu eine abstrakte Superklasse zu verwenden) definiert eine gemeinsame Schnittstelle für alle konkreten Zustände. Da alle Zustände dieses Interface implementieren, sind diese Zustände untereinander austauschbar. Immer wenn eine Anfrage für den Kontext eingeht (beispielsweise der Schalter S1 wurde gedrückt), dann wird diese an den aktuellen Zustand übergeben. Jeder konkrete Zustand stellt für eine Anfrage seine eigene Implementierung bereit. Und wenn der Kontext seinen Zustand ändert, so verändert sich auch gleichzeitig sein Verhalten.

3.1.7 Ein kleines Feature

Zum Abschluss kannst du noch ein kleines Feature einbauen:

```
public String gibVersion()
{
    return "Version 1.0";
}

public String gibAutor()
{
    return "Ralph Henne";
}
```

Abbildung 3.18: Quelltext der Methoden *gibVersion()* und *gibAutor()*

Diese beiden Methoden werden abwechselnd verwendet, wenn der Benutzer auf die *info*-Taste drückt.

Somit ist die Schaltung nun modelliert. Jetzt kannst du dich der grafischen Oberfläche widmen. Hierzu wirst du ein weiteres Programmiermuster kennen lernen, das so genannte MVC-Muster.

3.2 Einführung in das MVC-Muster

Model-View-Controller (MVC, wörtlich etwa „Modell-Präsentation-Steuerung“) bezeichnet ein Architekturmuster zur Aufteilung von Softwaresystemen in die drei Einheiten: Datenmodell (engl. *Model*), Präsentation (engl. *View*) und Programmsteuerung (engl. *Controller*).

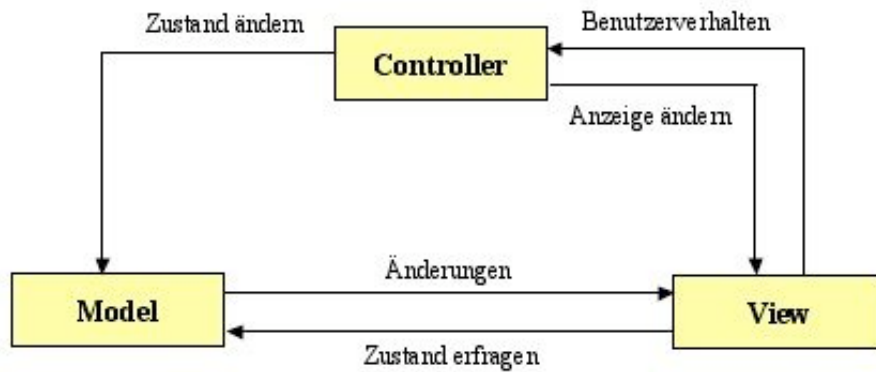


Abbildung 3.19: Das MVC-Muster

Ziel des Musters ist ein flexibles Programmdesign, das eine spätere Änderung oder Erweiterung erleichtern und eine Wiederverwendbarkeit der einzelnen Komponenten ermöglichen soll.

Als **Model** wird die Komponente bezeichnet, die die Datenstruktur der Anwendung definiert. Das Model speichert die Daten und somit den Zustand der Anwendung und stellt die Methoden zur Änderung der Daten zur Verfügung. Das Model kennt weder den View noch den Controller.

Als **View** wird die Komponente bezeichnet, die die Daten des Models auf dem Bildschirm präsentiert, jedoch keine Programmlogik besitzt. Der Benutzer führt auf dem View die Aktionen aus, die durch den Controller an das Model weitergeleitet werden.

Der View kennt das Model, ist dort registriert und kann dessen Zustand erfragen.

Als **Controller** wird die Komponente bezeichnet, die die Ablaufsteuerung darstellt. Der Controller kennt die beiden anderen Komponenten, den View und das Model. Er reagiert auf die Interaktionen der Anwender in dem View und überprüft diese. Anschließend ruft er die jeweilige Methode des Models auf und verändert dessen Zustand.

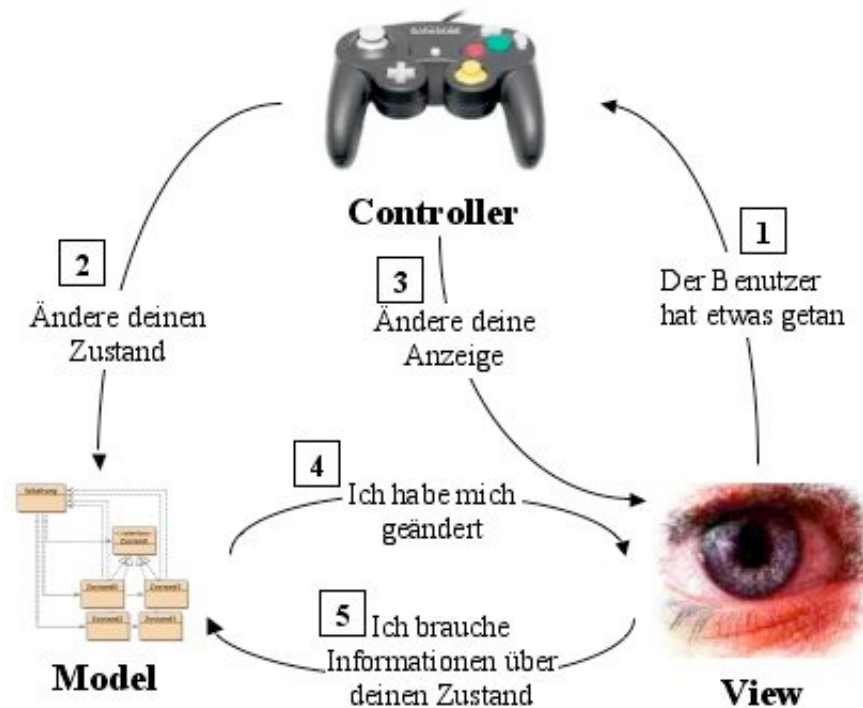


Abbildung 3.20: Beziehungen zwischen Model, View und Controller

1. ***Du als Beobachter interagierst mit dem View.***
Der View ist deine Sicht auf das Model. Wenn du also irgendetwas mit dem View machst, beispielsweise auf die *Plus*-Taste klickst, teilt der View dem Controller mit, was du getan hast. Es ist dann Aufgabe des Controllers, entsprechende Steuerungsmaßnahmen zu ergreifen.
2. ***Der Controller fordert das Model auf, seinen Zustand zu ändern.***
Der Controller nimmt deine Aktionen an und interpretiert sie. Wenn du auf einen Button klickst, muss der Controller herausfinden, was das bedeutet und wie das Model auf Grund dieser Aktion beeinflusst werden muss.
3. ***Der Controller kann auch den View auffordern, seinen Zustand zu ändern.***
Wenn der Controller eine Aktion vom View erhält, muss er den View auf Grund dessen eventuell auffordern, sich zu ändern. Beispielsweise könnte der Controller bestimmte Buttons oder Menüpunkte in der grafischen Benutzeroberfläche aktivieren oder deaktivieren.
4. ***Das Model benachrichtigt den View, wenn sich sein Zustand geändert hat.***
Wenn sich am Model etwas ändert – entweder auf Grund einer Aktion von dir (beispielsweise Eingeben einer Ziffer) oder einer internen Veränderung (beispielsweise Aufbau einer Zahl) -, meldet das Model dem View, dass sich sein Zustand geändert hat.
5. ***Der View fragt das Model nach seinem Zustand.***

Der View erhält den Zustand, den er anzeigt, direkt vom Modell. Wird er beispielsweise vom Modell benachrichtigt, dass er eine Rechenoperation durchgeführt hat, fragt der View das Modell nach dem aktuellen Ergebnis und zeigt dieses als Anzeigewert an. Der View kann das Modell auch nach dessen Zustand fragen, wenn er vom Controller aufgefordert wurde, die Ansicht zu verändern.

Nun wirst du die drei Komponenten im Projekt *Stromkreis* zusammenbauen.

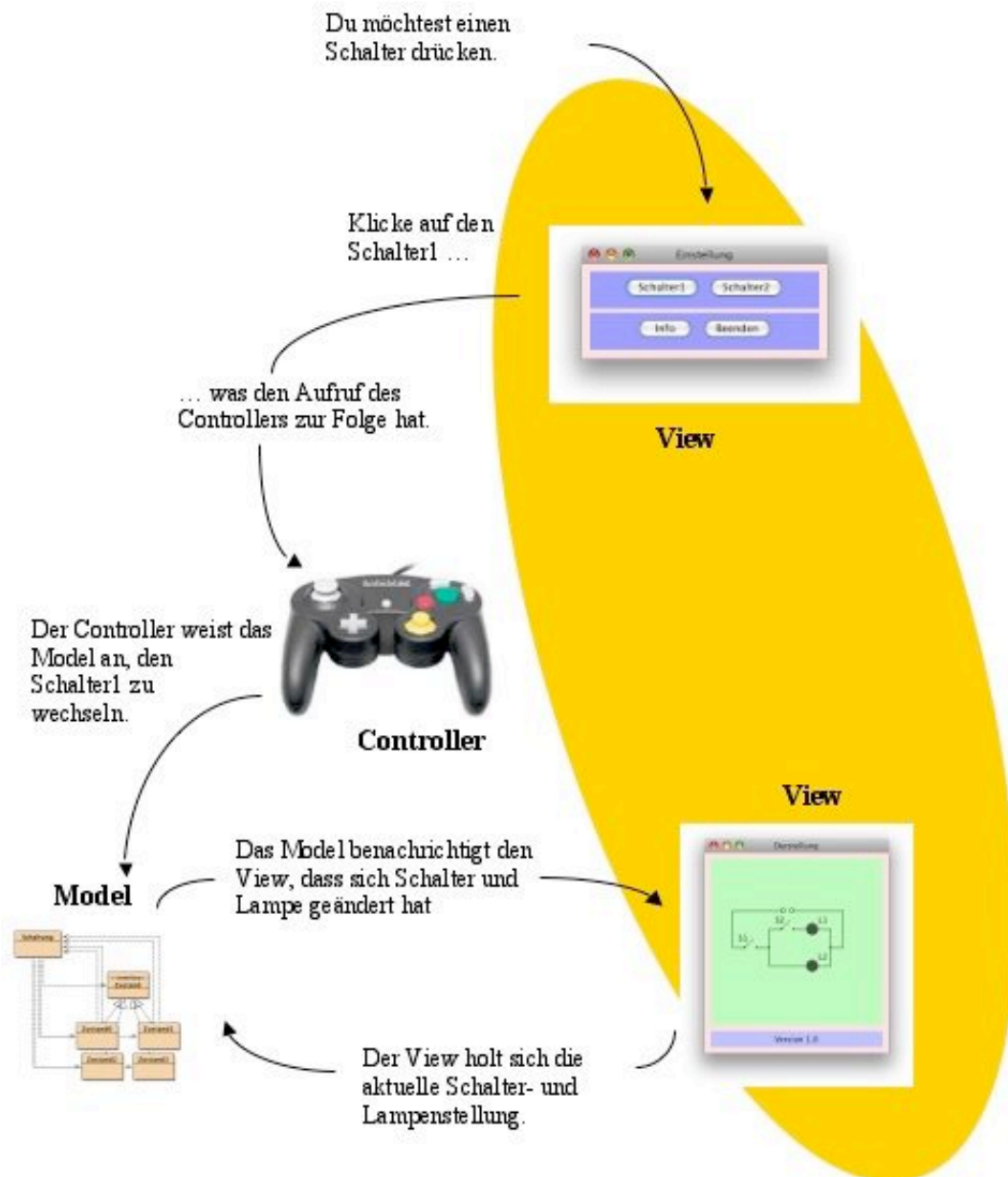


Abbildung 3.21: Der Stromkreis im MVC-Muster

Wenn du den Text aufmerksam gelesen hast, wirst du bemerkt haben, dass das Modell den View benachrichtigt, dass sich der Anzeigewert geändert hat, obwohl

das Model weder den View noch den Controller kennt. Trotzdem zeigen alle drei Abbildungen einen Pfeil vom Model zum View.

Der View besteht aus mehreren Fenstern. In Abbildung 3.21 sind die beiden Fenster *StromView* und *SchalterView* dargestellt. Du kannst dir noch weitere Darstellungsmöglichkeiten ausdenken.

Um all diese verschiedenen Views zu verwalten, verwendet das Model das Observer-Muster. Es benötigt daher Methoden zur Registrierung von Objekten als Schaltungbeobachter und Methoden zum Verschicken von Nachrichten an diese Schaltungbeobachter. Das Model kennt also nur diese Schaltungbeobachter. Das Observer-Muster verlangt, dass *Schaltungbeobachter* ein Interface ist und dass alle Views, die die Anzeige benötigen, das Interface *Schaltungbeobachter* implementieren. Somit kennt das Model nicht die Komponenten des Views, sondern die Schaltungbeobachter.

3.3 MVC-Muster konkret

3.3.1 Die Klasse Schaltung

Die Schaltung muss nun mehrere Views überwachen, in diesem Beispiel den SchalterView, den StromView und vielleicht später noch weitere Viewkomponenten, beispielsweise Kaffeeautomat oder ein Bankkonto oder was sich die Hersteller von Automaten sonst noch so einfallen lassen.

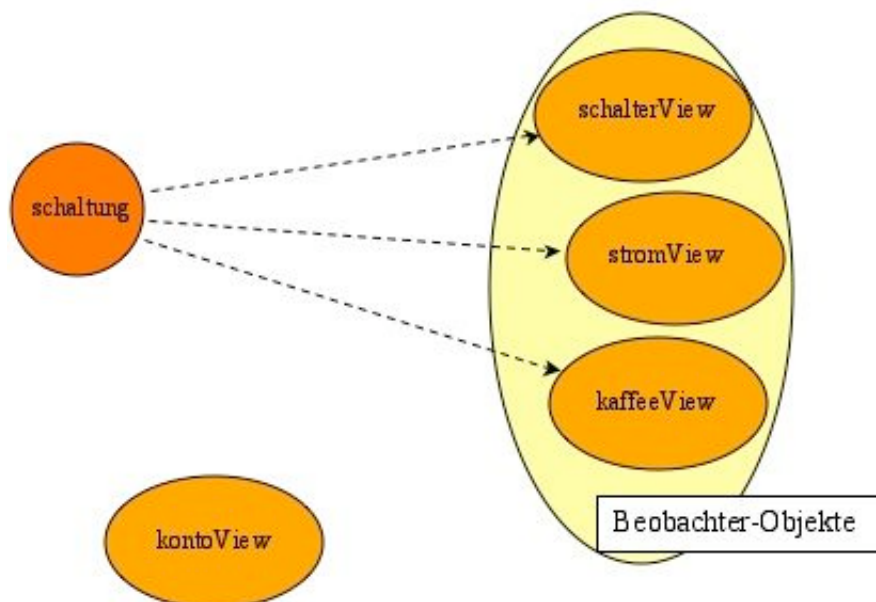


Abbildung 3.22: Das Observer-Muster

Nur die Schaltung allein weiß genau, in welchem Zustand sie sich befindet. Die Beobachter-Objekte, also die *schalterView*, *stromView* und *kaffeeView*, haben ein Abonnement bei der Schaltung. Das bedeutet, sie haben sich bei der Schaltung registriert, um Aktualisierungen zu erhalten, wenn also die Schaltung sich in einem neuen Zustand befindet..

Das Objekt *kontoView* hat leider Pech gehabt. Da es kein Beobachter ist, wird es auch nicht informiert, wenn sich der Zustand der Schaltung ändert. Aber *kontoView* kann der Schaltung sagen, dass es Beobachter werden möchte. Dazu lässt es sich bei der Schaltung registrieren und wird somit in die Liste der Beobachter-Objekte aufgenommen. Nun wartet *kontoView* zusammen mit den anderen Views auf die nächste Benachrichtigung von der Schaltung, dass sie wieder einen neuen Zustand besitzt.

Genauso gut kann nun *kaffeeView* darum bitten, als Beobachter entfernt zu werden. Die Schaltung berücksichtigt diese Anfrage und entfernt *kaffeeView* aus seiner Liste der Beobachter. Alle Beobachter, außer *kaffeeView*, der nun nicht mehr der Beobachtermenge angehört, erhalten weitere Benachrichtigungen von der Schaltung.

Das Prinzip des Observer-Muster ist nun erklärt. Jetzt wirst du dieses Muster modellieren. Die Abbildung 3.23 zeigt die beiden Klassen *Schaltung* und *StromView* mit ihren für das Observer-Muster wichtigen Datenfeldern und Methoden.

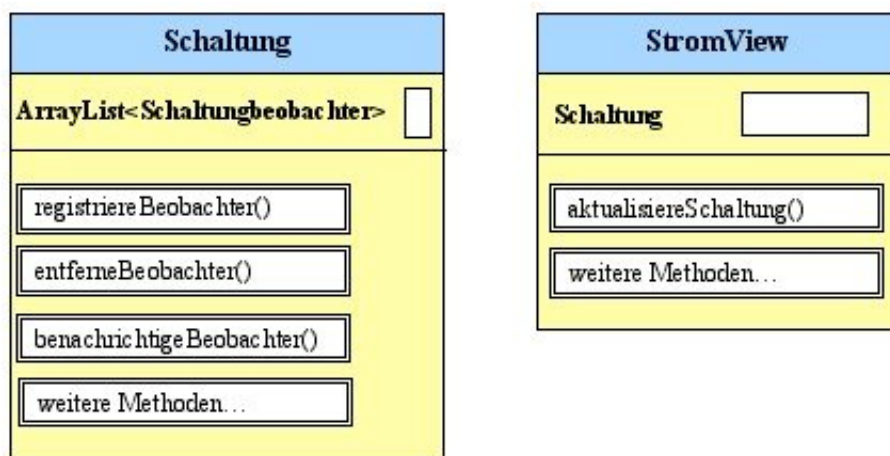


Abbildung 3.23: Die Klassen *Schaltung* und *StromView*

In der Klasse *Schaltung* muss du also eine *schalterbeobachterliste* erzeugen und die Methoden `registriereBeobachter()`, `entferneBeobachter()` und `benachrichtigeBeobachter()` implementieren.

```
import java.awt.*;
```

```
import java.util.ArrayList;

public class Schaltung
{
    //Datenfelder fuer alle Zustaende
    //bereits vorhanden

    //beobachtende Objekte, die von der Schaltung benachrichtigt werden
    private ArrayList<Schaltungbeobachter> schaltungbeobachterliste;

    public Schaltung()
    {
        schaltungbeobachterliste = new ArrayList<Schaltungbeobachter>();

        //bereits vorhanden
    }

    public void setZustand(Zustand zustand)
    {
        //bereits vorhanden
    }

    public String gibVersion()
    {
        //bereits vorhanden
    }

    public String gibAutor()
    {
        //bereits vorhanden
    }

    public void s1Druecken()
    {
        //bereits vorhanden
    }

    public void s2Druecken()
    {
        //bereits vorhanden
    }

    public Zustand getZustand0()
    {
        //bereits vorhanden
    }

    public Zustand getZustand1()
    {
        //bereits vorhanden
    }
}
```

```
public Zustand getZustand2()
{
    //bereits vorhanden
}

public Zustand getZustand3()
{
    //bereits vorhanden
}

/** ===== Methoden des Observer-Musters ===== */

public void registriereBeobachter(Schaltungbeobachter b)
{
    schaltungbeobachterliste.add(b);
}

public void entferneBeobachter(Schaltungbeobachter b)
{
    int i = schaltungbeobachterliste.indexOf(b);
    if (i >= 0) {
        schaltungbeobachterliste.remove(i);
    }
}

public void benachrichtigeSchaltungbeobachter()
{
    for (Schaltungbeobachter beobachter : schaltungbeobachterliste) {
        beobachter.aktualisiereSchaltung();
    }
}
}
```

Abbildung 3.24: Quelltext der Klasse *Schaltung* erweitert um das Observer-Muster

Übung 3.7:

Sichere dein bisheriges Projekt als *Stromkreis02a*. Implementiere die Klasse *Schaltung* aus Abbildung 3.24.

Übung 3.8:

Überlege, in welchen Methoden der Klasse *Schaltung* die Methode *benachrichtigeSchaltungbeobachter()* aufgerufen werden muss. Implementiere nun diese Aufrufe.

Die Methoden *s1Druecken()* und *s2Druecken()* benötigen die Methode *benachrichtigeSchaltungbeobachter()*.

3.3.2 Das Interface Schaltungbeobachter

Ganz so einfach ist diese Sache jedoch nicht. Wie in Abbildung 3.23 dargestellt gibt es verschiedene Views. Alle diese Views sehen ein bisschen anders aus, haben somit verschiedene Methoden und alle diese verschiedenen Views müssen von einer einzigen Beobachterliste verwaltet werden. Das Problem ist nun, dass diese Beobachterliste nur einen einzigen Typ sammeln kann.

Dieses Problem wird mit Hilfe eines Interfaces gelöst. Jeder View, der irgendwie den aktuellen Zustand der Schaltung benötigt, muss dieses Interface implementieren. Somit verfügen all diese Views über die Methode *aktualisiereSchaltung()*, und können nun vom Rechner davon unterrichtet werden, dass ein neues Rechenergebnis vorliegt.

```
public interface Schaltungbeobachter
{
    public void aktualisiereSchaltung();
}
```

Abbildung 3.25: Quelltext des Interfaces *Schaltungbeobachter*

Betrachte ein Interface zunächst einmal als eine Vereinbarung, dass alle Klassen, die dieses Interface implementieren, auch die Methode *aktualisiereSchaltung()* besitzen. Die Schaltung kann nun also sicher sein, dass alle ihre Beobachter auch ihre Benachrichtigung erhalten und damit etwas anfangen können. Sie muss nichts darüber wissen, was die einzelnen Beobachter nun damit anfangen. Mal kann der Zustand in einem gewöhnlichen Schaltbild dargestellt werden, mal kann sie als Blackbox dargestellt werden, damit Schüler auf logische Schaltungen schließen können.

Weiterhin wird hier auch der Polymorphismus genutzt, indem auf ein Supertyp programmiert wird, damit das tatsächliche Laufzeitobjekt nicht im Quelltext festgeschrieben werden muss. Die Schaltung sammelt also in seiner Beobachterliste nur noch Supertypen *Schaltungbeobachter*, sie muss also die tatsächlichen Objekttypen (Display, Abakus) nicht kennen. Somit können an die Schaltung alle möglichen Geräte zur Darstellung angeschlossen werden, wichtig ist nur, dass diese das Interface *Schaltungbeobachter* implementiert haben. Wird nun ein neuer Beobachter hinzugefügt, so muss die Schaltung nicht mehr verändert werden.

Schaltung und Beobachter sind locker gebunden, sie können miteinander agieren, müssen aber nur wenige Kenntnisse voneinander besitzen.

Übung 3.9:

Implementiere das Interface *Schaltungbeobachter* im Projekt *Stromkreis02a*.

3.3.3 Die Klasse *SchaltbildView*

Bemerkung:

Das Programmieren einer grafischen Benutzeroberfläche ist hier nicht das Thema. Die Gestaltungsmöglichkeiten mit Swing werden in *The JFC Swing Tutorial, Second Edition* von Kathy Walrath, in *Handbuch der Java-Programmierung* von Guido Krüger, u. a. oder in ähnlichen Büchern beschrieben.

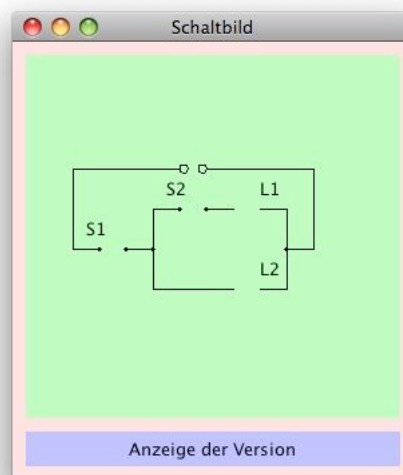


Abbildung 3.26: Grafische Oberfläche zur Darstellung des Schaltbilds

Übung 3.10:

Hole vom Schulserver (oder von meiner Homepage) das Projekt *SchaltbildGUI*. Füge mit Hilfe von *Add Class from File...* die Klasse *SchaltbildView* und *Buehne* in dein Projekt *Stromkreis02a* ein. Studiere den Quelltext der Klassen *SchaltbildView* und *Buehne*, auch wenn du nicht alles verstehst. Erzeuge ein Objekt der Klasse *SchaltbildView*, es wird eine Schaltskizze ohne Schalter und Lampen angezeigt.

Wie in Abbildung 3.23 bereits dargestellt, besitzt die Klasse *SchaltbildView* also ein Schaltung-Objekt. Somit kann das *SchaltbildView*-Objekt nun die Schaltung bitten, dass Schaltung dieses *SchaltbildView*-Objekt registriert und somit in seine Beobachterliste aufnimmt. Dies erfolgt im Konstruktor.

```
public SchaltbildView(Schaltung schaltung)
{
```

```
this.schaltung = schaltung;
schaltung.registriereBeobachter((Schaltungbeobachter) this);

versionAnzeigen = true;

erzeugeSchaltbildView();
}
```

Abbildung 3.27: Quelltext des Konstruktors *SchaltbildView*

Falls sich der Zustand der Schaltung geändert hat, so durchläuft die Schaltung nun ihre Beobachterliste, um alle registrierten Beobachter zu benachrichtigen, dass beispielsweise ein Schalter gedrückt wurde. Allen Beobachter wird also gesagt, dass sie ihre Anzeige aktualisieren müssen. Und diese Beobachter können nun mit dieser Nachricht machen, was sie wollen, hier beispielsweise ein Lämpchen leuchten lassen..

Übung 3.11:

Die Klasse *SchaltbildView* soll nun das Interface *Schaltungbeobachter* implementieren und ein Datenfeld *schaltung* erhalten. Im Konstruktor soll *SchaltbildView* sich bei der Schaltung als *Schaltungbeobachter* registrieren. Zuletzt muss *SchaltbildView* noch die im Interface *Schaltungbeobachter* vertraglich festgelegte (noch leere) Methode *aktualisiereSchaltung()* erhalten. Vergleiche hierzu auch die Abbildung 3.26.

Hat sich also nun der Zustand der Schaltung (des Models) geändert, so durchläuft die Schaltung ihre Liste aller *Schaltungbeobachter* (und somit auch alle Views) und benachrichtigt diese, dass irgendein Schalter gedrückt wurde. Die Views fragen nun beim Model nach seinem aktuellen Zustand..

Übung 3.11:

Betrachte die Abbildungen 3.20 und 3.21 und suche die oben beschriebenen Datenflüsse.

Nach dem Drücken des Benutzers auf einen der beiden Schalter hat sich der Zustand der Schaltung verändert. Im Schaltbild muss also der Schalter seine Stellung wechseln und das Lämpchen gelb bzw. grau erscheinen.

```
public void aktualisiereSchaltung()
{
    buehne.holeSchaltung(schaltung);
    buehne.repaint();
}
```

Abbildung 3.28: Methode *aktualisiereSchaltung()* in der Klasse *SchaltbildView*

Übung 3.12:

Implementiere die Methode *aktualisiereSchaltung()* in der Klasse *SchaltbildView*.

Teste dein Projekt *Stromkreis02a*. Erzeuge zuerst eine Schaltung und dann den SchaltbildView.

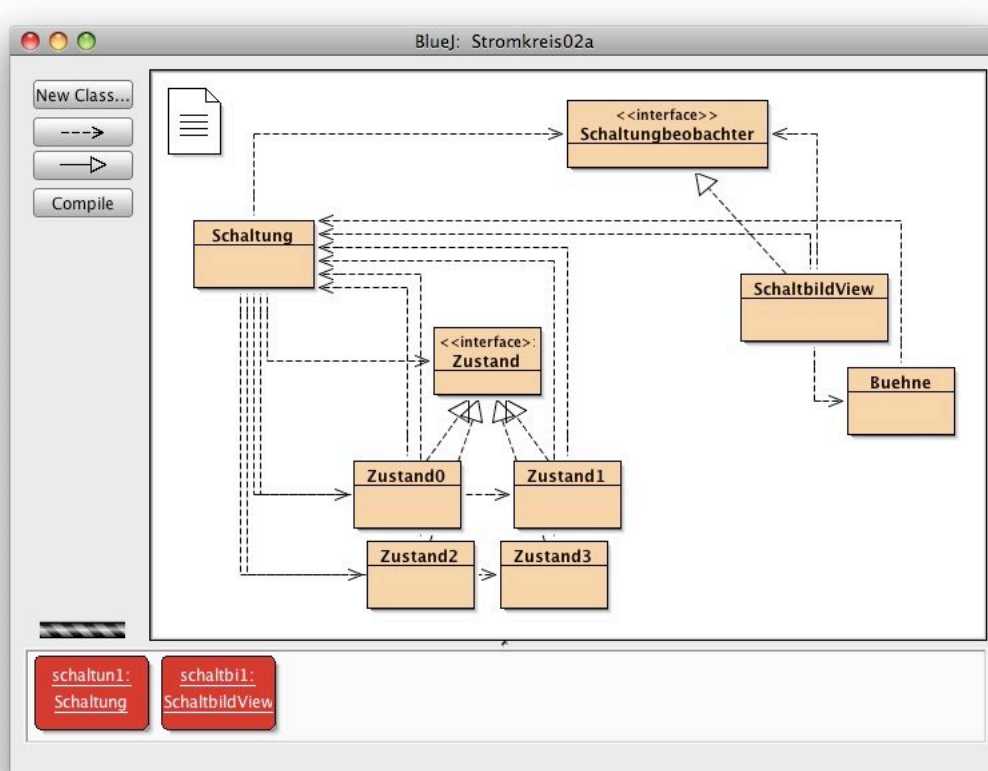


Abbildung 3.29: Klassendiagramm des Projekts *Stromkreis02a*

In der Übung 3.12 wirst du festgestellt haben, dass du zuerst ein Schaltung-Objekt erzeugen musst. Damit sich der SchaltbildView als Beobachter bei der Schaltung registrieren lassen kann, muss das SchaltbildView-Objekt der Schaltung als Parameter übergeben werden.

3.3.4 Die Klasse SteuerungView

Die grafische Oberfläche zeigt zwar nun das Schaltbild des Stromkreises, aber ist noch nicht für Benutzereingaben vorbereitet. Hierzu wird der View erweitert durch ein neues Fenster, dem *SteuerungView*.

Mit Hilfe der beiden Buttons *Schalter1* und *Schalter2* kann der Benutzer die entsprechenden Schalter im Schaltbild schließen oder wieder öffnen. Die Versionsnummer bzw. der Autor wird mit der *Info*-Taste gezeigt.



Abbildung 3.30: Die grafische Oberfläche zur Steuerung des Stromkreises

Übung 3.13:

Sichere dein bisheriges Projekt als *Stromkreis02b*.

Hole vom Schulserver (oder von meiner Homepage) das Projekt *SteuerungGUI*. Füge mit Hilfe von *Add Class from File...* die Klasse *TastaturView* in dein Projekt *Stromkreis02b* ein. Studiere den Quelltext der Klasse *SteuerungView*.

- Wie werden die Buttons erzeugt erzeugt?
- Ist es sinnvoll, die vier Schalter auf zwei verschiedene Panels zu setzen? Experimentiere mit verschiedenen Layout-Managern!

Der Benutzer hat also nun eine Schaltertaste gedrückt. Der *SteuerungView* delegiert diese Benutzeraktion an den Controller. Der Controller ist das einzige Objekt, das weiß, wie man mit den Aktionen des Benutzers umgeht. Der Controller entscheidet,

- ob er dem Model mitteilen soll, dass es seinen Zustand ändern soll,
- ob er dem View mitteilen soll, dass er seine Anzeige ändern soll.

Übung 3.14:

Betrachte die Abbildungen 3.20 und 3.21 und suche die oben beschriebenen Datenflüsse.

Die Benutzeraktionen werden vom *SteuerungView* direkt an den Controller weitergeleitet. Die Klasse *SteuerungView* muss deshalb ein Datenfeld *controller* erhalten. Der Konstruktor erhält als Parameter somit ein Controller-Objekt, das in dem Datenfeld *controller* gehalten wird

```
private Controller controller;

public SteuerungView(Controller controller)
{
    this.controller = controller;

    erzeugeSteuerungView();
}
```

Abbildung 3.31: Datenfeld und Konstruktor der Klasse *SteuerungView*

Klickt der Benutzer nun auf einen beliebigen Schalter, so wird dies dem Controller mitgeteilt.

```
private void schaltAction(int schalter)
{
    if (schalter == 1) {
        controller.schalter1Druecken();
    }
    if (schalter == 2) {
        controller.schalter2Druecken();
    }
}
```

Abbildung 3.32: Mitteilungen von Benutzereingaben an den Controller in der Klasse *SteuerungView*

Übung 3.15:

Implementiere den in den Abbildungen 3.31 und 3.32 dargestellten Quelltext. Suche in den Abbildungen 3.20 und 3.21 den hier programmierten Schritt.

3.3.5 Die Klasse Controller

Jetzt musst du dich noch um den Controller kümmern.

Eine der Aufgabe des Controllers ist, auf Grund der Benutzeraktion im View zu entscheiden, welche Daten im Model geändert werden müssen. (Später wird der Controller auch noch den View ändern.) Der Controller darf jedoch nicht die Daten selbst manipulieren, dies muss das Model machen.

```
import java.awt.*;
import java.awt.event.*;

public class Controller
```

```
{
    private Schaltung schaltung;
    private SchaltbildView schaltbildView;
    private SteuerungView steuerungView;

    public Controller(Schaltung schaltung)
    {
        this.schaltung = schaltung;

        schaltbildView = new SchaltbildView(schaltung);
        steuerungView = new SteuerungView(this);
    }

    public void schalter1Druecken()
    {
        //leer
    }

    public void schalter2Druecken()
    {
        //leer
    }

    public void informiere()
    {
        //leer
    }
}
```

Abbildung 3.33: Quelltext der Klasse *Controller*

Wie bereits in der Einführung in das MVC-Muster beschrieben, kennt der Controller die beiden anderen Komponenten, den View und das Model. Er ist also dasjenige Objekt, das alles zusammenhält. Deswegen erhält der Konstruktor das Model (die Schaltung) als Argument und erzeugt den View. Der View besteht jetzt bereits schon aus zwei Teilen,

1. dem SteuerungView, der den Controller benötigt, der ja die Benutzereingaben delegieren muss,
2. dem SchaltbildView, der das Model (die Schaltung) benötigt, um deren Zustand abzufragen.

Hat der Anwender in der grafischen Benutzeroberfläche beispielsweise auf den Schalter1 gedrückt, meldet der *SteuerungView* dies dem *Controller* mit der Methode *controller.schalter1Druecken()*. Der *Controller* teilt nun der Schaltung mit, dass Schalter1 gedrückt wurde. Die *Schaltung* weiß dann schon, was sie mit dieser Benutzeraktion anfangen soll und benachrichtigt anschließend den

SchaltbildView (eigentlich alle Schaltungbeobachter), dass sich ihr Zustand geändert hat. Der *SchaltbildView* fragt nun bei *Schaltung* nach dem aktuellen Zustand, erstellt daraus das Schaltbild und zeichnet dieses auf die Zeichenfläche der Bühne.

Übung 3.16:

Versuche die oben beschriebenen Datenflüsse im Klassendiagramm des Projekts *Stromkreis02b* nachzuvollziehen. Vergleiche deine Gedanken auch mit den Beziehungen zwischen Model, View und Controller in den Abbildungen 3.20 und 3.21.

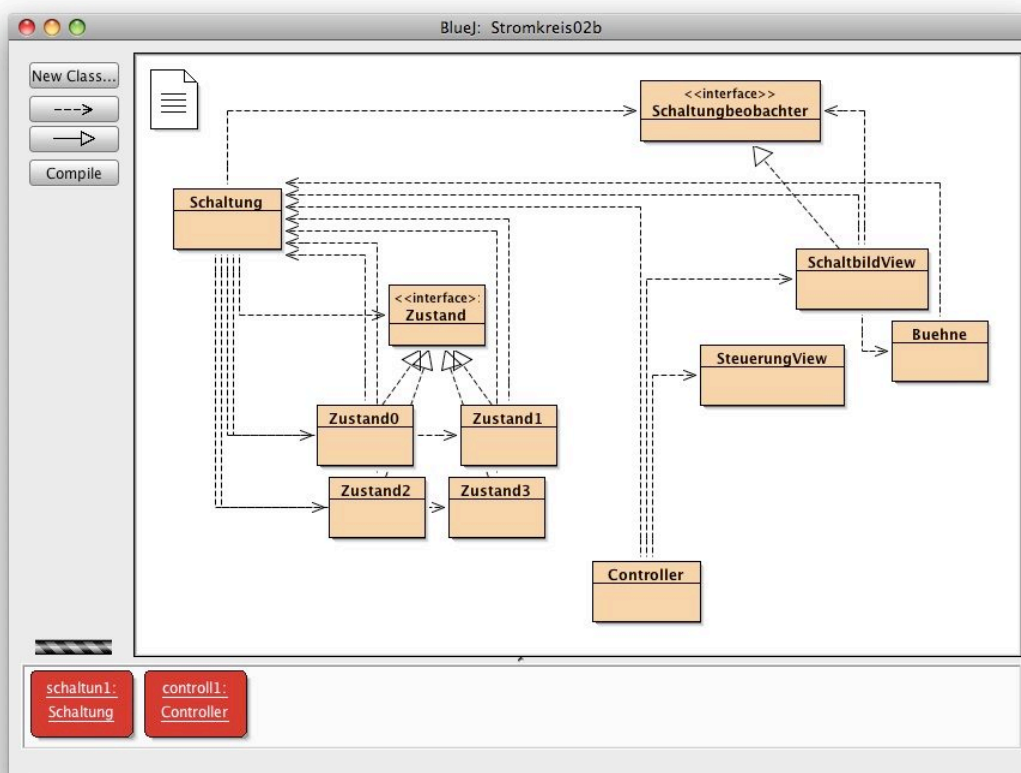


Abbildung 3.34: Klassendiagramm des Projekts *Stromkreis02b*

Übung 3.17:

Implementiere die Klasse *Controller* in deinem Projekt *Stromkreis02b*. Vervollständige nun auch die Methoden *schalter1Druecken()* und *schalterS2Druecken()*. Teste dein Projekt *Stromkreis02b*. Erzeuge dazu ein *Schaltung*-Objekt und anschließend ein *Controller*-Objekt.

Übung 3.18:

Beim Testen deines Projekts *Stromkreis02b* wirst du bemerkt haben, dass weder die Schalter noch die Lampen, sondern nur die Leitungen und die Beschriftung gezeichnet werden.

- a) Formuliere eine Begründung für dieses Verhalten.
- b) In welcher Klasse ist es sinnvoll, den Quelltext für die Schalter und Lampen zu implementieren?

```
public void zeichne(Graphics g)
{
    Color farbe;

    if (zustand == zustand0) {
        farbe = Color.black;
        g.setColor(farbe);
        g.drawLine(50, 140, 64, 126); //S1
        g.drawLine(110, 110, 124, 96); //S2
        farbe = Color.darkGray;
        g.setColor(farbe);
        g.fillOval(150,100,20,20); //L1
        g.fillOval(150,160,20,20); //L2
    }
    if (zustand == zustand1) {
        ..... ähnliche Anweisungen .....
    }
}
```

Abbildung 3.35: Die Methode *zeichne()* in der Klasse *Schaltung*

Übung 3.19:

Implementiere die Methode *zeichne()* und vervollständige den Quelltext. Teste dein Projekt *Stromkreis02b*.

3.3.6 Der Info-Button

Übung 3.20:

Betrachte das MVC-Muster in Abbildung 3.20. Überlege, welche der dort dargestellten Datenflüsse im bisherigen Projekt *Stromkreis02b* noch nicht berücksichtigt worden ist?

Klickt der Benutzer auf die *Info*-Taste, so erhält der *Controller* vom *SteuerungView* die Aufforderung, den *SchaltbildView* zu ändern. Im *SchaltbildView* muss das *infoL* wechseln zwischen den Texten *Version 1.0* und *Ralph Henne*. Der Controller erhält eine Aktion vom View (hier vom

SteuerungView) und fordert anschließend nun direkt den View (hier den *SchaltbildView*) auf, sich zu verändern. Der Zustand des Models ändert sich dabei. Es wird also die in der Abbildung 3.36 rot dargestellte Aktion durchgeführt.

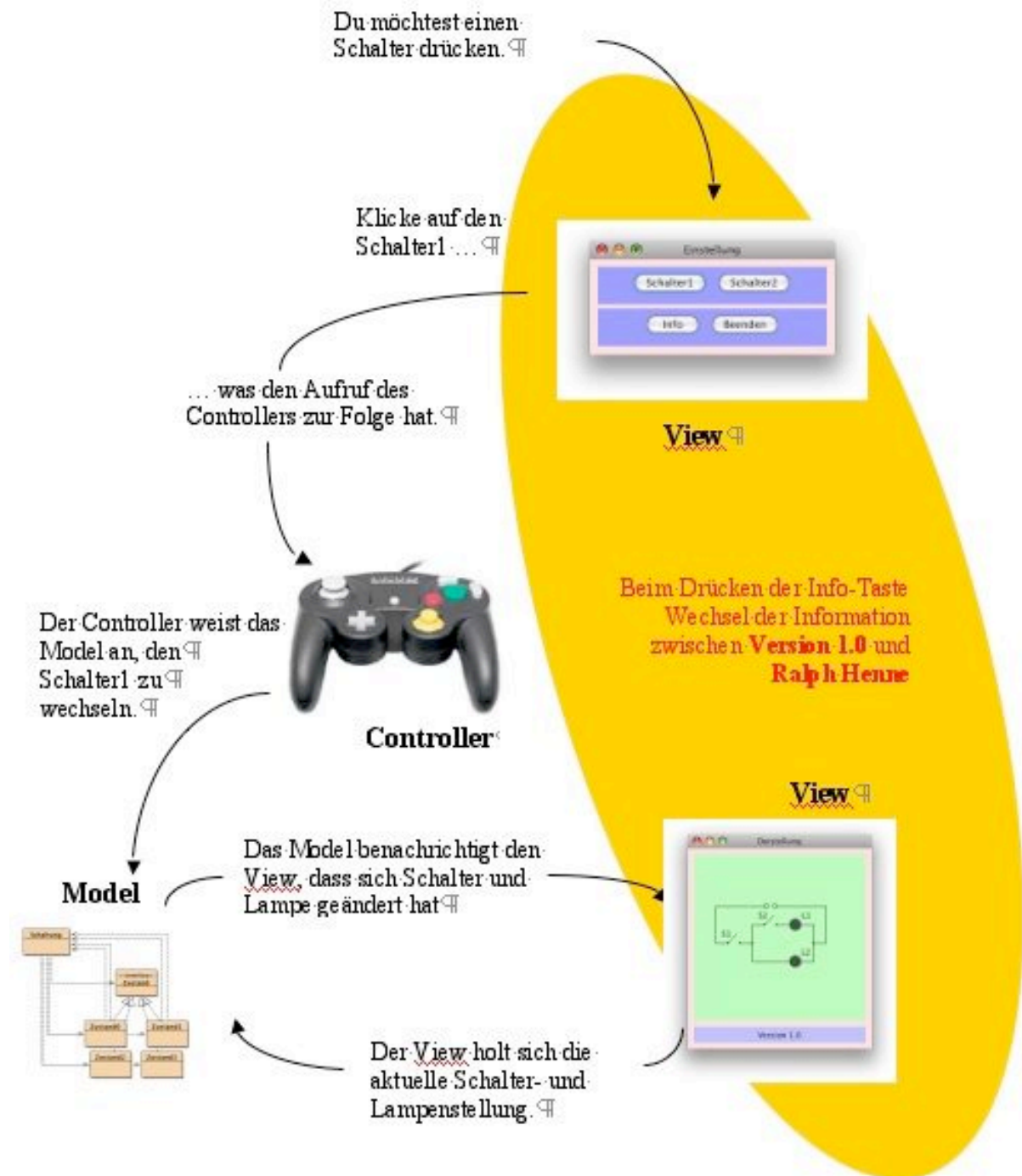


Abbildung 3.36: Der Controller weist den View an, sich zu ändern

Je nach Wert des Boolean *versionAnzeigen* holt sich der *DisplayView* vom *Rechner* den Autor bzw. die Version.

```
public void infoZeigen()
{
    if (versionAnzeigen) {
        infoL.setText(rechner.gibAutor());
    }
    else {
        infoL.setText(rechner.gibVersion());
    }
    versionAnzeigen = !versionAnzeigen;
}
```

Abbildung 3.37: Methode *infoZeigen()* in der Klasse *SchaltbildView*

Drückt also der Benutzer im *SteuerungsView* die *Info*-Taste, so wird die Methode *controller.informiere()* aufgerufen. Der Controller befiehlt mit der Methode *schaltbildView.infoZeigen()* direkt dem *SchaltbildView*, seinen Schriftzug zu wechseln

Übung 3.21:

Implementiere die Methode *infoZeigen()* in der Klasse *SchaltbildView* und vervollständige die Methode *informiere()* in der Klasse *Controller*.

Sorge auch dafür, dass gleich zu Beginn im *infoL* die Versionsnummer angezeigt wird. Teste dein Projekt durch mehrmaliges Drücken der *Info*-Taste.

Bemerkung:

Der View und der Controller stellen das Strategy-Muster dar. Der View ist nur für die Sicht zuständig, die Entscheidungen über sein Verhalten delegiert er an den Controller. Der Controller ist somit das Verhalten (die Strategie) des Views. Wird für den View ein anderes Verhalten gewünscht, muss der Controller ausgetauscht werden.

Außerdem bleibt durch die Verwendung des Strategy-Musters der View entkoppelt vom Model. Nur der Controller weiß, wie man mit Benutzeraktionen umgeht.

Streng genommen verlangt das Strategy-Muster ein Interface, z.B.

ControllerInterface, an das alle möglichen Controller angeschlossen werden können. So ähnlich wie beim Rechenverhalten oder bei den Beobachtern müssten alle Controller dieses Interface implementieren. Ich glaube, da in diesem Beispiel nur ein einziger Controller verwendet wird, dass man auf ein *ControllerInterface* verzichten kann.

Genauer kann man jedoch im eingangs erwähnten Buch *Entwurfsmuster von Kopf bis Fuß* nachlesen.

3.4 Zusammenfassung des Views

Die beiden Teile des Views – die Sicht auf das Model (*SchaltbildView*) und die Steuerungselemente für die Benutzerschnittstelle (*SteuerungView*) werden in zwei getrennten Fenstern angezeigt. Deswegen wurden diese beiden Teile in zwei verschiedene Klassen gelegt. Das Klassendiagramm vereinfacht sich jedoch erheblich, wenn diese in einer einzigen Klasse *SchaltungView* stecken. Dies wird nun aber kein großes Problem mehr darstellen, weil die beiden Klassen *SchaltbidView* und *SteuerungView* für die geplante Zusammenfassung schon weit gehend vorbereitet sind.

Bemerkung:

Im Vergleich zum Projekt *Uhr* aus Kapitel 1 werden hier die beiden Klassen für die grafische Benutzeroberfläche nicht direkt von *JFrame* abgeleitet. Vielmehr wird im Konstruktor der entsprechenden View-Klassen ein *JFrame*-Objekt erzeugt. Diese Aufgabe wird dann später die Klasse *Controller* übernehmen, die u. A. für die Erzeugung des Views zuständig ist.

Die Klasse *SchaltungView* implementiert das Interface *Schaltungbeobachter* und übernimmt alle Datenfelder, die in den beiden Klassen *SchaltbildView* und *SteuerungView* bereits eingeführt wurden. Der Konstruktor erhält als Parameter sowohl ein Controller-Objekt als auch ein Schaltung-Objekt.

```
public SchaltungView(Controller controller, Schaltung schaltung)
{
    this.controller = controller;
    this.schaltung = schaltung;
    schaltung.registriereBeobachter((Schaltungbeobachter) this);

    versionAnzeigen = true;
}
```

Abbildung 3.38: Konstruktor der Klasse *SchaltungView*

Die Erzeugung der entsprechenden Frames geschieht nicht mehr hier im Konstruktor, diese Aufgabe übernimmt jetzt direkt der Controller.

Übung 3.22:

Sichere dein bisheriges Projekt als *Stromkreis04a*. Du wirst nun aus den beiden Klassen *SchaltbildView* und *SteuerungView* eine neue Klasse *SchaltungView* zusammenstellen.

- a) Erzeuge eine neue Klasse *SchaltungView*. Implementiere alle Datenfelder, die auch in den Klassen *SchaltbildView* und *SteuerungView* verwendet wurden.
- b) Implementiere den Konstruktor aus Abbildung 3.38 in der Klasse *SchaltungView*.
- c) Implementiere (aus der Klasse *SchaltungView*) die Methoden *erzeugeSchaltbildView()*, *anzeigePERstellen()* und *informationPERstellen()* in der Klasse *SchaltbildView*.
- d) Implementiere die sonstige wichtige Methode im SchaltbildFrame *infoZeigen()* in der Klasse *SchaltbildView*.
- e) Implementiere die Beobachter-Methode im SchaltbildFrame *aktualisiereSchaltung()* in der Klasse *SchaltungView*.
- e) Implementiere (aus der Klasse *SteuerungView*) die Methoden *erzeugeSteuerungView()*, *schalterPERstellen()* und *buttonPERstellen()* in der Klasse *SchaltungView*.
- f) Implementiere die Aktions- Methode im SteuerungFrame *schaltAction()*, *versionAction()* und *beendenAction()* in der Klasse *SchaltungView*.
- g) Entferne nun die beiden Klassen *SchaltbildView* und *SteuerungView* aus dem Projekt *Stromkreis04a*.

Nun musst du, wie bereits oben angedeutet, einige Veränderungen in der Klasse *Controller* durchführen.

```
public Controller(Schaltung schaltung)
{
    this.schaltung = schaltung;
    schaltungView = new SchaltungView(schaltung);

    schaltungView.erzeugeSchaltbildView();
    schaltungView.erzeugeSteuerungView();
}
```

Abbildung 3.39: Neue Datenfelder und Konstruktor in der Klasse *Controller*

Der Konstruktor der Klasse *Controller* erzeugt nun ein *SchaltungView*-Objekt, das seinerseits den *SchaltbildView* und *SteuerungView* erzeugt.

Übung 3.23:

- a) Ersetze die beiden Datenfelder *schaltbildView* und *steuerungView* durch das Datenfeld *schaltungView* in der Klasse *Controller*.

- b) Implementiere den Konstruktor aus Abbildung 3.39 in der Klasse *Controller*.
- c) Verbessere die Methode *informiere()* in der Klasse *Controller*.

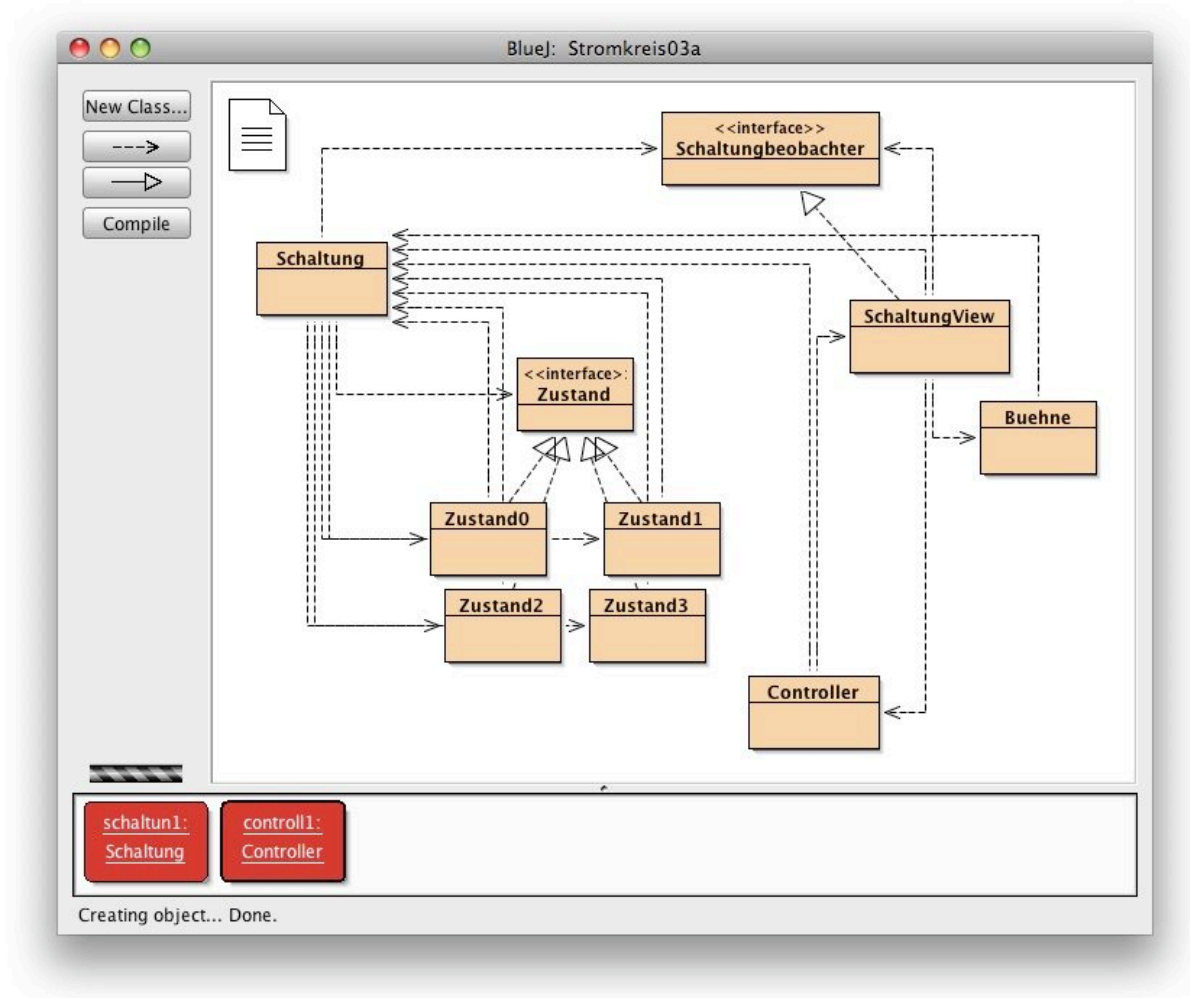


Abbildung 3.40: Klassendiagramm des Projekts *Stromkreis03a*

3.5 Abschlussarbeiten

Du hast nun alles, was du brauchst: ein Model, einen View und einen Controller. Und all dies hast du in **BlueJ** manuell zu einem MVC-Muster zusammengesetzt, so dass sie gut miteinander arbeiten.

Zum Schluss fehlt noch eine Klasse *Simulation*, die dies automatisch für dich erledigt. Für eine stand-alone-Applikation benötigst du zudem noch die Methode *main()*.

```
public class Simulation
{
```

```
public static void main (String[] args)
{
    Schaltung schaltung = new Schaltung();
    Controller controller = new Controller(schaltung);
}
}
```

Abbildung 3.41: Die Klasse *Simulation*

Übung 2.24:

Sichere dein bisheriges Projekt als *Stromkreis03b*. Implementiere die Klasse *Simulation*.