

6 Zustandsorientierte Modellierung in BlueJ

Bemerkung:

Bei komplexen Automaten halte ich es jedoch sinnvoller, bei der zustandsorientierten Programmierung keine *if-else*-Anweisungen zu verwenden, sondern die Zustandsübergangstabelle direkt zu implementieren. In diesem Kapitel zeige ich einige Beispiele.

6.1 Ampelanlage

Nun werden wir eine Ampelanlage in Java mit **BlueJ** programmieren.

6.1.1 Autoampel

Abbildung 6.1 zeigt eine mögliche Darstellung eine Autoampel.

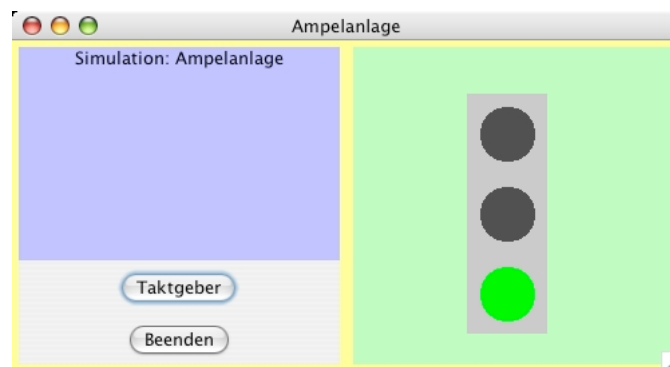


Abbildung 6.1: Die Autoampel

Durch Drücken auf den Button *Taktgeber* springt die Ampel von einer Phase in die nächste.

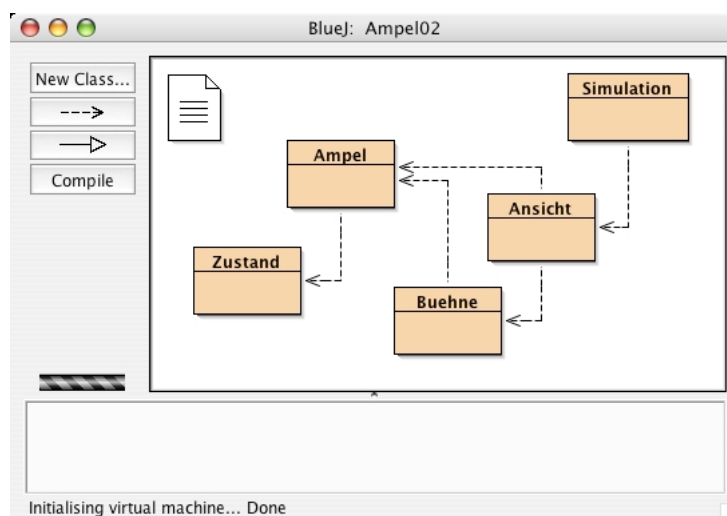


Abbildung 6.2: Klassendiagramm von *Ampel02*

Lass dich von diesem Klassendiagramm des fertigen Projekts *Ampel02* nicht erschrecken, sondern versuche die Aufgaben jeder Klasse zu verstehen. Du wirst nun schrittweise das Projekt entwickeln.

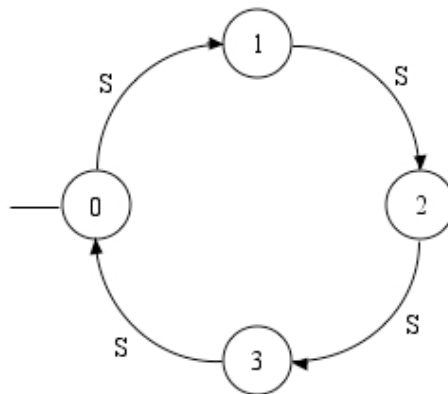
Die Ampel hat die drei Lichter „Rot“, „Gelb“ und „Grün“, die je nach Ampelphase an bzw. aus sind. Ein Ampeldurchlauf besteht aus vier verschiedenen Zustände, die in einer Zustandstabelle zusammenfasst werden:

Zustand	aRo	aGe	aGr
0	0	0	1
1	0	1	0
2	1	0	0
3	1	1	0

Abbildung 6.3: Zustandstabelle

Du benötigst also vier Zustände *zustand0*, ..., *zustand3*, die mit den entsprechenden Werten belegt werden. In *Abbildung 6.3* zeigt der *zustand0* die Grünphase, d. h. das grüne Licht ist an und das gelbe und rote Licht ist jeweils aus. Der *zustand1* ist dann die entsprechende Gelbphase, usw.

In einem Zustandsübergangdiagramm kannst du diese Phasenübergänge übersichtlich darstellen.

**Abbildung 6.4:** Zustandsübergangdiagramm

In einem solchen Zustandsübergangdiagramm wird dargestellt, in welcher Reihenfolge die Zustände durchlaufen werden.

Für die Implementierung dieser Ampel ist nun eine Kombination aus Zustandstabelle und Zustandsübergangdiagramm ideal. Du ergänzt die

Zustandstabelle um eine weitere Spalte. Diese gibt den jeweiligen Folgezustand bei Betätigen des Schalters an.

Zustand	aRo	aGe	aGr	Folgezustand
0	0	0	1	1
1	0	1	0	2
2	1	0	0	3
3	1	1	0	0

Abbildung 6.5: Zustandstabelle und Zustandsübergangstabelle der Ampel

Nun hast du dich mit dieser Ampel soweit beschäftigt, dass du diese programmieren kannst.

Die Datenfelder, die für die in Abbildung 6.5 dargestellte benötigt werden, werden in der Klasse *Zustand* implementiert.

```
public class Zustand
{
    private int aRo, aGe, aGr, fz;

    public Zustand(int aRoH, int aGeH, int aGrH, int fzH)
    {
        aRo = aRoH;
        aGe = aGeH;
        aGr = aGrH;
        fz = fzH;
    }

    /** Liefert die Werte aller Datenfelder als Array. */
    public int[] gibWerte()
    {
        int[] werte = {aRo, aGe, aGr, fz};
        return werte;
    }
}
```

Abbildung 6.6: Quelltext der Klasse *Zustand*

In dieser Klasse besitzen die Datenfelder die Zugriffsstufe *private* und ihre Werte können von anderen Klassen nur mit Hilfe von sondierenden Methoden wie z.B. *gibRot()* gelesen werden. Das bedeutet, du benötigst bereits bei diesem einfachen Automaten vier sondierende Methoden in der Klasse *Zustand*. Bei komplizierten Automaten erhöht sich der Programmieraufwand deshalb enorm.

Alternativ kannst du auch alle vier Werte in das Array *werte* einlesen. Du benötigst nun also nur noch eine einzige Methode *gibWerte()*, die dieses Array komplett, und somit die Werte aller vier Datenfelder auf einmal liefern kann.

Übung 6.1.1:

Erstelle in **BlueJ** ein Projekt *Ampel01*. Implementiere nun die Klasse *Zustand* wie in Abbildung 6.6 dargestellt.

Erzeuge anschließend die vier Objekte *zustand0*, ... , *zustand3* und untersuche diese mit Hilfe des Inspektors.

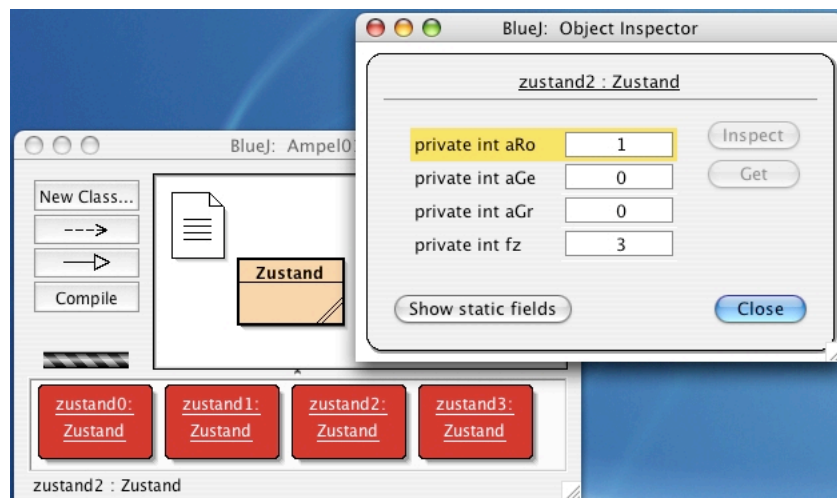


Abbildung 6.7: Inspektor zeigt die Werte des Objekts *zustand2*

Nun kannst du also die vier Objekte *zustand0*, ... , *zustand3* erzeugen, in der die jeweiligen Zeilen der obigen Tabelle gespeichert werden. Abbildung 6.7 zeigt den *zustand2*, in dem also das rote Licht leuchtet. Wird anschließend auf den Schalter gedrückt, geht die Schaltung in den *zustand3* über.

Die Klasse *Ampel* muss nun diese vier Zustände kennen. Dazu stellt sie ein Array *tabelle* zur Verfügung, in der diese Zustände gespeichert werden. Das Füllen der Tabelle mit entsprechenden Werten wird von der Methode *fülleTabelle()* durchgeführt. Hier werden die vier Zustände als Array *testdaten* erzeugt und anschließend in *tabelle* eingelesen.

```
public class Ampel
{
    private Zustand[] tabelle;
    private int aRot, aGelb, aGruen, foZu;

    /**Konstruktor */
    public Ampel()
    {
        tabelle = new Zustand[4];
    }
}
```

```
        fuelleTabelle(); //Fuellt die Tabelle mit den Zustaenden
    }

    /** Fuellt die Tabelle mit den Zustaenden. */
    private void fuelleTabelle()
    {
        //Kombination aus Zustands- und Zustandsuebergangstabelle
        Zustand[] testdaten = {
            new Zustand(0, 0, 1, 1),
            new Zustand(0, 1, 0, 2),
            new Zustand(1, 0, 0, 3),
            new Zustand(1, 1, 0, 0)
        };
        //Einlesen in das Datenfeld tabelle
        tabelle = testdaten;
    }
}
```

Abbildung 6.8: Quelltext der Klasse *Ampel*

Bemerkung:

Du hättest die Tabelle bereits im Konstruktor direkt füllen können. Somit wäre die Methode *fuelleTabelle()* überflüssig. Andererseits besteht nun mit Hilfe der Methode *fuelleTabelle()* die Möglichkeit, die Werte von einer anderen Klasse als Parameter zu übernehmen.

Übung 6.1.2:

Implementiere im Projekt *Ampel01* die Klasse *Ampel* wie in Abbildung 6.8 dargestellt.

Erzeuge anschließend ein Objekt der Klasse *Ampel* und untersuche dieses mit Hilfe des Inspektors.

Vergleiche dazu die Abbildung 6.9 mit der Abbildung 6.7!

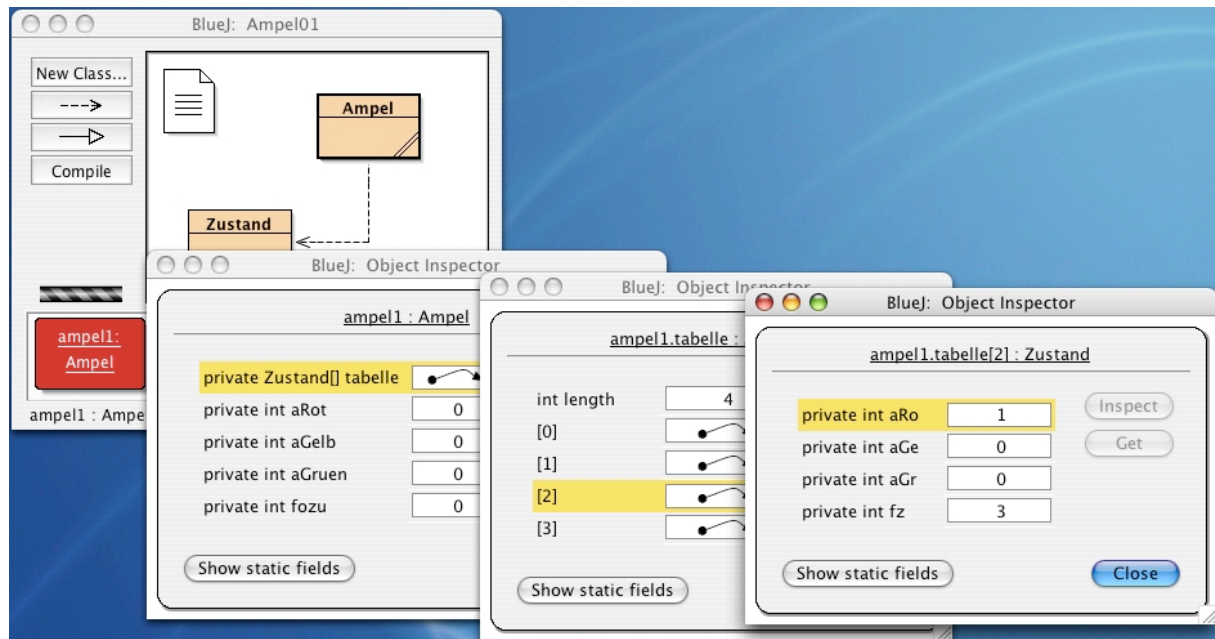


Abbildung 6.9: Inspektor zeigt die Werte des Zustands2

Abbildung 6.9 zeigt, dass nun alle Zustände von der Klasse *Ampel* korrekt gespeichert wurden, jedoch befindet sich die Ampel selbst noch nicht im richtigen Zustand. In den Datenfelder *aRot*, *aGelb*, *aGruen* und *fozu* ist noch jeweils der Standardwert 0 gespeichert. Hierzu musst du der Methode *geheInZustand()* als Parameter die Nummer des gewünschten Zustands übergeben und somit die Werte aus der entsprechenden Werte aus der Tabelle in die Datenfelder *aRot*, *aGelb*, *aGruen* und *fozu* einliest.

```
/** Setzt die Ampel in den entsprechenden Folgezustand. */
public void geheInZustand(int nummer)
{
    Zustand z = tabelle[nummer];
    aRot = z.gibWerte()[0];
    aGelb = z.gibWerte()[1];
    aGruen = z.gibWerte()[2];
    fozu = z.gibWerte()[3];
}
```

Abbildung 6.10: Die Methode *geheInZustand()* in der Klasse *Ampel*

Die lokale Variable *z* übernimmt nun den gewünschten Zustand aus dem Array *tabelle* und übergibt die entsprechenden Werte an die Datenfelder.

Übung 6.1.3:

Implementiere in der Klasse *Ampel* die Methode *geheInZustand()* wie in Abbildung 6.10 dargestellt. Teste anschließend diese Methode mit Hilfe des Inspektors. Vergleiche dazu die Abbildungen 6.11 und 6.12!

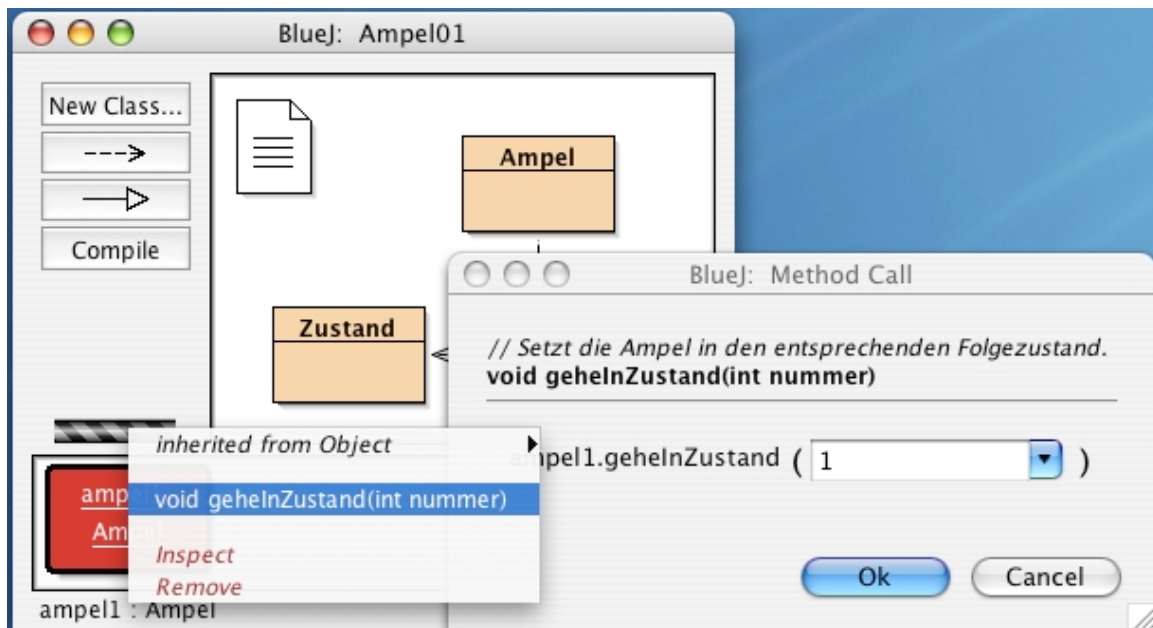


Abbildung 6.11: Aufruf der Methode *geheInZustand(1)*

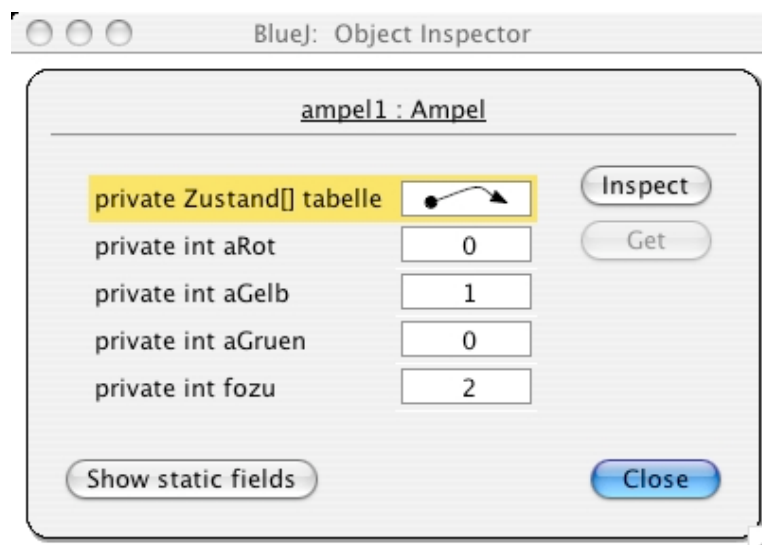


Abbildung 6.12: Inspektor zeigt den gewünschten *zustand1*

Die Abbildungen zeigen, dass sich das Objekt *ampel1* im *zustand1* befinden, d.h. nur das Gelblicht leuchtet und der Folgezustand ist *zustand2*.

Übung 6.1.4:

Ergänze den Konstruktor der Klasse *Ampel*, so dass sich ein Objekt dieser Klasse bei der Erzeugung gleich im Anfangszustand *zutsand0* befindet.

Drückt der Anwender nun auf den Schalter, so geht das Objekt aus dem *zustand1* in den *zustand2* über. Das bedeutet, die Datenfelder *aRot*, *aGelb*, *aGruen* und *fozu* müssen neue Werte des jeweiligen Folgezustands erhalten. Die Nummer des Folgezustands ist im Datenfeld *fozu* gespeichert. Die Methode *holeFolgezustand()* übergibt diesen Wert der Methode *geheInZustand()* als Parameter.

```
/** Holt sich den Folgezustand. */  
public void holeFolgezustand()  
{  
    geheInZustand(fozu);  
}
```

Abbildung 6.13: Die Methode *holeFolgezustand()* in der Klasse *Ampel*

Übung 6.1.5:

- Implementiere die Methode *holeFolgezustand()* in der Klasse *Ampel*. Teste diese Methode mit Hilfe des Inspektors.
- Betrachte die grafische Benutzeroberfläche in der Abbildung 4.1. Welche der Methoden in der Klasse *Ampel* sollten nun als *private*, welche müssen als *public* deklariert werden

Im Prinzip bist du nun fertig, das Projekt funktioniert. Der Rest ist noch das Gestalten einer grafischen Benutzeroberfläche. Dies erfolgt in den Klassen *Ansicht* und *Buehne*. Diese beiden Klassen haben also folgende Aufgaben:

Ansicht: erstellt die Oberfläche. Auf der linken Seite befindet sich der Einstellungsbereich mit Buttons *Taktgeber* und *Beenden*, auf der rechten Seite im Darstellungsbereich das Bild der Ampel.

Buehne: erstellt eine Komponente, auf der das Bild der Ampel gezeichnet wird.

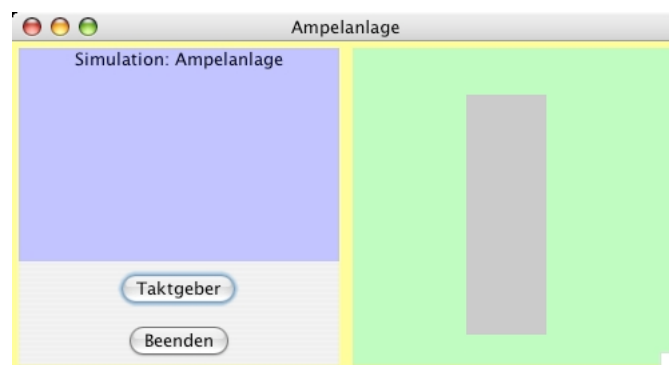


Abbildung 6.14: Grafische Benutzeroberfläche des Projekts *Ampel***Bemerkung:**

Ich habe versucht, die weiteren Projekte *Schaltung* und *Apfelwein* mit einer ähnlichen grafischen Benutzeroberfläche zu entwickeln. Somit muss diese GUI nur einmal im Unterricht besprochen werden und kann mit kleinen Abweichungen für die anderen Projekte übernommen werden.

Ich erkläre jetzt nur die wichtigsten Teile des Quelltexts dieser beiden Klassen. In den folgenden Übungen wirst du diesen Quelltext verändern müssen.

Übung 6.1.6:

Hole dir vom Schulserver das Projekt *AmpelGUI* und speichere dieses unter dem Namen *Ampel02*. Versuche den Quelltext der Klassen *Ansicht* und *Buehne* zu verstehen.

In der Klasse *Buehne* wird mit *fillRect()* an der Position (80/30) das hellgraue Rechteck mit der Länge 60 und der Höhe 180 gezeichnet. In dieses Rechteck muss die Klasse *Ampel* dann die Lichter in den entsprechenden Farben zeichnen.

Übung 6.1.7:

Füge mit *Edit > Add Class from File...* die Klassen *Ampel* und *Zustand* in das Projekt *Ampel02* ein. Entferne in der Klasse *Ansicht* und in der Klasse *Buehne* die entsprechenden Auskommentierungen.

Übung 6.1.8:

Beim Testen deines Projekts *Ampel02* wirst du bemerkt haben, dass nicht die Lampen, sondern nur der graue Kasten gezeichnet werden.

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

```
/** Zeichnet die Lichter der Ampel. */
public void zeichne(Graphics g)
{
    Color farbe;

    //Auto-Rotlicht
    if (aRot == 1) farbe = Color.red;
    else farbe = Color.darkGray;
    g.setColor(farbe);
    g.fillOval(90,40,40,40);
}
```

```
//Auto-Gelblicht  
..... ähnliche Anweisungen .....  
  
//Auto-GruenLicht  
..... ähnliche Anweisungen .....  
  
}
```

Abbildung 6.15: Die Methode *zeichne()* in der Klasse *Ampel*

Übung 6.1.9:

- Implementiere die Methode *zeichne()* und vervollständige den Quelltext.
- Hole dir die Klasse *Simulation* aus dem Projekt *Digitaluhr* (*Edit > Add Class from File ...*) und vervollständige das Projekt *Ampel02*. Vergleiche dazu die Abbildung 6.16! Teste dein Projekt! Anschließend erstelle aus deinem Projekt eine stand-alone-Applikation *Ampel.jar*.

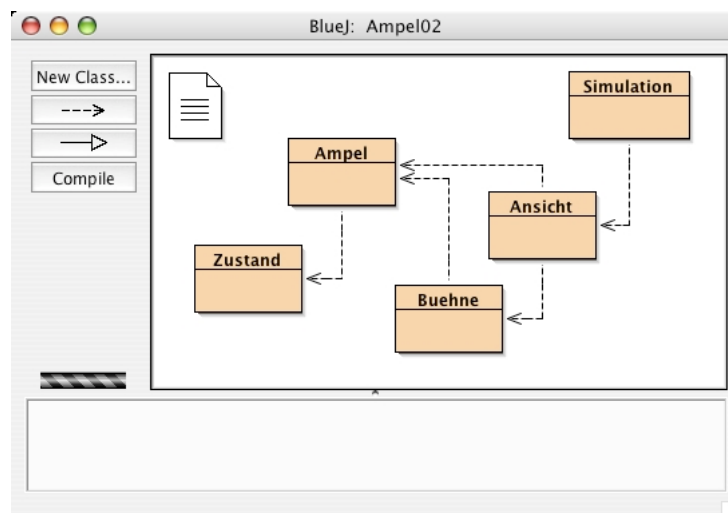


Abbildung 6.16: Klassendiagramm von *Ampel02*

6.1.2 Ampelanlage

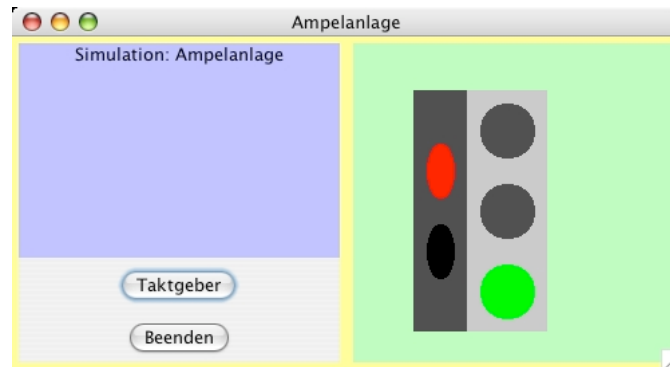


Abbildung 6.17: Die Ampelanlage *Ampel03*

Übung 6.1.10:

Erstelle in **BlueJ** das Projekt *Ampel03*. Übernehme die Klassen von *Ampel02* und verändere den Quellcode so, dass die in Abbildung 6.17 dargestellte Ampelanlage simuliert wird.

Übung 6.1.11: (für Experten)

In der Wirklichkeit drückt ein Fußgänger nur einmal auf den Druckknopf. Die Ampel durchläuft dann selbstständig ihre Phasen.

Versuche dieses Modell mit einem Timer zu realisieren. In Kapitel 4 *Digitaluhr* hast du den Timer bereits kennen gelernt.

6.2 Stromkreis

Du wirst nun die in Abbildung 6.18 dargestellte elektrische Schaltung programmieren.

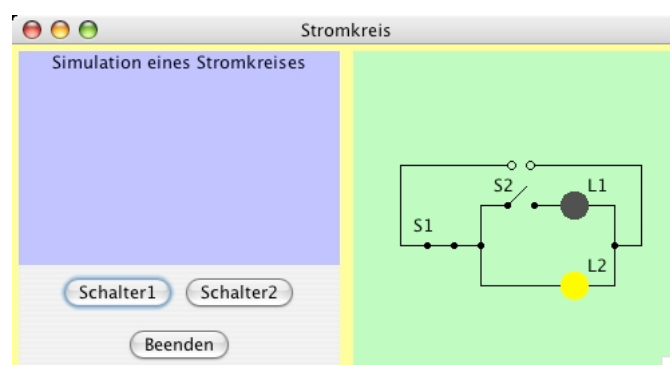


Abbildung 6.18: Die elektrische Schaltung

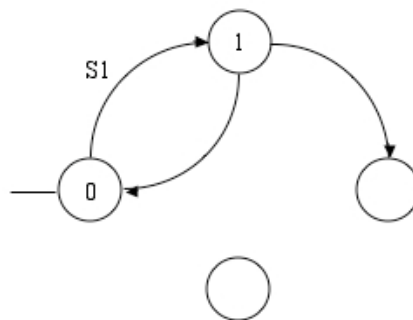
Drückt der Benutzer auf die Button *Schalter1* bzw. *Schalter2*, so werden in der Schaltskizze die entsprechenden Schalter geschlossen oder geöffnet und die jeweiligen Lampen leuchten.

Übung 6.2.1:

Vervollständige die Zustandstabelle. Dabei gilt:
 0: Schalter geöffnet 1: Schalter geschlossen
 0: Lampe aus 1: Lampe an

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

Vervollständige das Zustandsübergangsdiagramm.



Vervollständige die Kombination aus Zustandstabelle und Zustandsübergangstabelle.

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

Nun hast du dich mit dieser elektrischen Schaltung soweit beschäftigt, dass du diese programmieren kannst.

Die in Übung 6.2.1 erstellte Tabelle der Zustände und Zustandsübergänge zeigt bereits die Implementierung der Klasse *Zustand*.

```
public class Zustand
{
    private int s1, s2, l1, l2, fz1, fz2;

    /** Konstruktor */
    public Zustand(int s1H, int s2H, int l1H, int l2H, int fz1H, int fz2H)
    {
        s1 = s1H;
        s2 = s2H;
        l1 = l1H;
        l2 = l2H;
        fz1 = fz1H;
        fz2 = fz2H;
    }

    /** Liefert die Werte aller Datenfelder als Array. */
    public int[] gibWerte()
    {
        int[] werte = {s1, s2, l1, l2, fz1, fz2};
        return werte;
    }
}
```

Abbildung 6.19: Quelltext der Klasse *Zustand*

Um die Datenkapselung einzuhalten, werden sämtliche Datenfelder als *private* eingestuft. Nun können jedoch nur noch sondierende Methoden die Werte dieser Datenfelder liefern. Für die sechs Datenfelder benötigst Du also jeweils eine Methode, was unnötigen Programmieraufwand erfordert.

Alternativ werden hier die Werte in das Array *werte* gelesen. Du benötigst also nur noch eine einzige Methode *gibWerte()*, die dieses Array komplett, und somit die Werte aller sechs Datenfelder auf einmal liefert.

Nun kannst du also vier Objekte *zustand0*, ... , *zustand3* erzeugen, in der die jeweiligen Zeilen der obigen Tabelle gespeichert werden. Im *zustand0* sind also laut dieser Tabelle beide Schalter offen und keine Lampe brennt. Wird anschließend auf den Schalter S1 gedrückt, geht die Schaltung in den *zustand1* über.

Du musst dafür sorgen, dass die Schaltung

1. die vier Zustände kennt,
2. beim Drücken eines Schalters in den entsprechenden Folgezustand geht.

Hierzu implementierst du die Klasse *Schaltung*.

```
public class Schaltung
{
```

```
private Zustand[] tabelle;
private int sch1, sch2, lam1, lam2, fozu1, fozu2;

public Schaltung()
{
    tabelle = new Zustand[4];
}
}
```

Abbildung 6.20: Datenfelder und Konstruktor der Klasse *Schaltung*

Die Klasse *Schaltung* besitzt als Datenfeld das Array *tabelle*, in der die vier Zustände verwaltet werden. Der Quelltext wird übersichtlicher, wenn noch zusätzlich Datenfelder *sch1*, *sch2*, *lam1*, *lam2*, *fozu1*, *fozu2* verwendet werden, die den momentanen Zustand speichern.

Das Füllen der Tabelle mit entsprechenden Werten wird von der Methode *fuelleTabelle()* durchgeführt. Hier werden die vier Zustände als Array *testdaten* erzeugt und anschließend in *tabelle* eingelesen.

```
/** Fuelle die Tabelle mit den Zustaenden. */
public void fuelleTabelle()
{
    //Kombination aus Zustands- und Zustandsuebergangstabelle
    Zustand[] testdaten = {
        new Zustand(0, 0, 0, 0, 1, 3),
        new Zustand(1, 0, 0, 1, 0, 2),
        new Zustand(1, 1, 1, 1, 3, 1),
        new Zustand(0, 1, 0, 0, 2, 0)
    };
    tabelle= testdaten;
}
```

Abbildung 6.21: Die Methode *fuelleTabelle()* in der Klasse *Schaltung*

Bemerkung:

Du hättest die Tabelle bereits im Konstruktor direkt füllen können. Somit wäre die Methode *fuelleTabelle()* überflüssig. Andererseits besteht nun mit Hilfe der Methode *fuelleTabelle()* die Möglichkeit, die Werte von einer anderen Klasse als Parameter zu übernehmen.

Die Methode *geheInZustand()* sorgt dafür, dass in die Datenfelder *sch1*, *sch2*, *lam1*, *lam2*, *fozu1* und *fozu2* die Werte des momentanen Zustands eingelesen werden. Hierzu übergibst du als Parameter die *nummer* des gewünschten Zustands.

```
/** Setzt die Schaltung in den entsprechenden Folgezustand. */
```

```

public void geheInZustand(int nummer)
{
    Zustand z = tabelle[nummer];
    sch1 = z.gibWerte()[0];
    sch2 = z.gibWerte()[1];
    lam1 = z.gibWerte()[2];
    lam2 = z.gibWerte()[3];
    fozu1 = z.gibWerte()[4];
    fozu2 = z.gibWerte()[5];
}

```

Abbildung 6.22: Die Methode *geheInZustand()* in der Klasse *Schaltung*

Übung 6.2.2:

- Erstelle in **BlueJ** das Projekt *Stromkreis01*. Implementiere die Klassen *Zustand* und *Schaltung*.
- Untersuche mit Hilfe des Inspektors die Funktionsweisen der Methoden *fuelleTabelle()* und *geheInZustand()*.
- Was passiert, wenn du vergessen hast, zuerst die Methode *fuelleTabelle()* aufzurufen? Verbessere den Quelltext!
- Untersuche mit Hilfe des Inspektors den Anfangszustand deiner Schaltung und vergleiche diesen mit der Tabelle aus Übung 6.2.1. Verbessere den Quelltext!

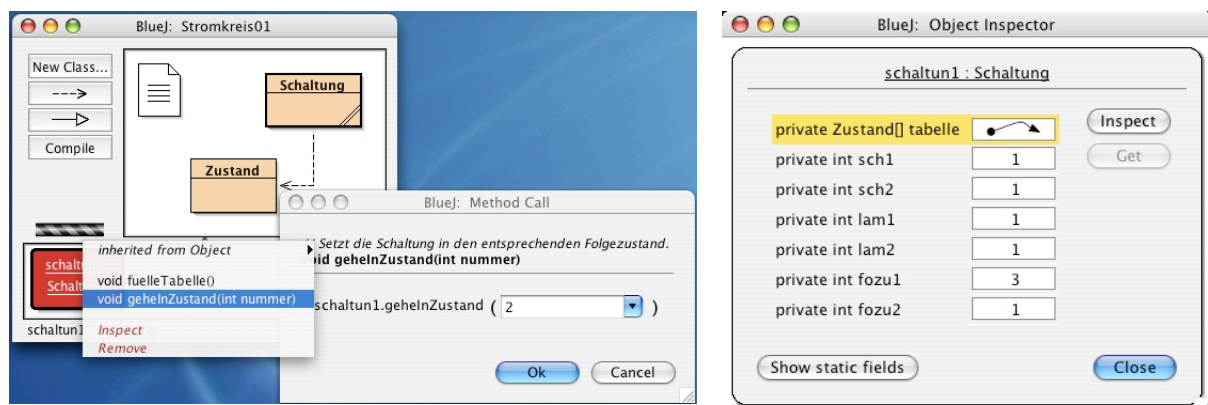


Abbildung 6.23: Inspektor zeigt den Zustand nach Aufruf der Methode *geheInZustand(2)*

Abbildung 6.23 zeigt, dass die Schaltung in den *zustand2* übergeführt wurde. Laut Tabelle aus Übung 6.2.1 sind nun die beiden Schalter geschlossen und die beiden Lampen leuchten. Wird Schalter S1 bzw. S2 gedrückt, geht die Schaltung in den *zustand3* bzw. *zustand1*. Allerdings musst du nun eine Methode *holeFolgezustand()* implementieren, die sich die entsprechenden Werte der Folgezustände aus der Tabelle holt.

```

/** Holt sich den Folgezustand fuer den entsprechenden Schalter. */
public void holeFolgezustand(int schalter)
{
    int nummer;

    switch(schalter) {
        case 1: nummer = fozu1;
                break;
        case 2: nummer = fozu2;
                break;
        default: nummer = 0;
    }
    geheInZustand(nummer);
}

```

Abbildung 6.24: Die Methode *holeFolgezustand()* in der Klasse *Schaltung*

Drückt der Benutzer den Schalter S1, so wird der Methode *holeFolgezustand()* der Parameter 1 übergeben. Nun wird also der Folgezustand *fozu1* für S1 in *nummer* gespeichert und der Methode *geheInZustand()* übergeben. Entsprechendes erfolgt auch beim Drücken auf Schalter S2.

Übung 6.2.3:

- Implementiere die Methode *holeFolgezustand()* in der Klasse *Schaltung*. Teste diese Methode mit Hilfe des Inspektors.
- Betrachte die grafische Benutzeroberfläche in der Abbildung 6.18. Welche der Methoden in der Klasse *Schaltung* sollten nun als *private*, welche müssen als *public* deklariert werden

Im Prinzip bist du nun fertig, das Projekt funktioniert. Der Rest ist wiederum das Gestalten einer grafischen Benutzeroberfläche. Dies erfolgt in den Klassen *Ansicht* und *Buehne*.

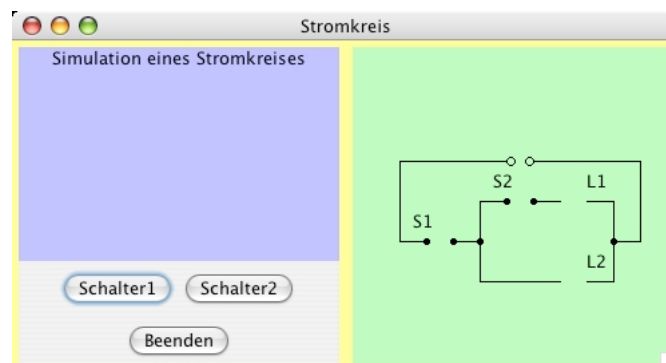


Abbildung 6.25: Grafische Benutzeroberfläche des Projekts *StromkreisGUI*

Ansicht: erstellt die Oberfläche. Auf der linken Seite befindet sich der Einstellungsbereich mit den Buttons, auf der rechten Seite der Darstellungsbereich, der das noch unvollständige Bild der Schaltskizze zeigt.

Buehne: erstellt eine Komponente, auf der das Bild der Schaltskizze gezeichnet wird.

Übung 6.2.4:

Hole dir vom Schulserver das Projekt *StromkreisGUI* und speichere dieses unter dem Namen *Stromkreis02*. Versuche den Quelltext der Klassen *Ansicht* und *Buehne* zu verstehen.

Übung 6.2.5:

Füge mit *Edit > Add Class from File...* die Klassen *Schaltung* und *Zustand* in das Projekt *Stromkreis02* ein. Entferne in der Klasse *Ansicht* und in der Klasse *Buehne* die entsprechenden Auskommentierungen.

Übung 4.2.6:

Beim Testen deines Projekts *Stromkreis02* wirst du bemerkt haben, dass weder die Schalter noch die Lampen, sondern nur die Leitungen und die Beschriftung gezeichnet werden.

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

```
/** Zeichnet die Schalterstellungen und Lampen. */
public void zeichne(Graphics g)
{
    Color farbe;
    //Linie des Schalters 1
    if (sch1 == 0) g.drawLine(50, 140, 64, 126);
    else g.drawLine(50, 140, 70, 140);
    //Linie des Schalters 2

    ..... ähnliche Anweisungen .....

    //Lampe 1
    if (lam1 == 1) farbe = Color.yellow;
    else farbe = Color.darkGray;
    g.setColor(farbe);
    g.fillOval(150,100,20,20);
    //Lampe 2
```

..... ähnliche Anweisungen

}

Abbildung 6.26: Die Methode *zeichne()* in der Klasse *Schaltung*

Übung 6.2.7:

- Implementiere die Methode *zeichne()* und vervollständige den Quelltext.
- Hole dir die Klasse *Simulation* und vervollständige das Projekt *Stromkreis02*. Vergleiche dazu die Abbildung 6.27! Teste dein Projekt! Anschließend erstelle aus deinem Projekt eine stand-alone-Applikation *Stromkreis02.jar*.

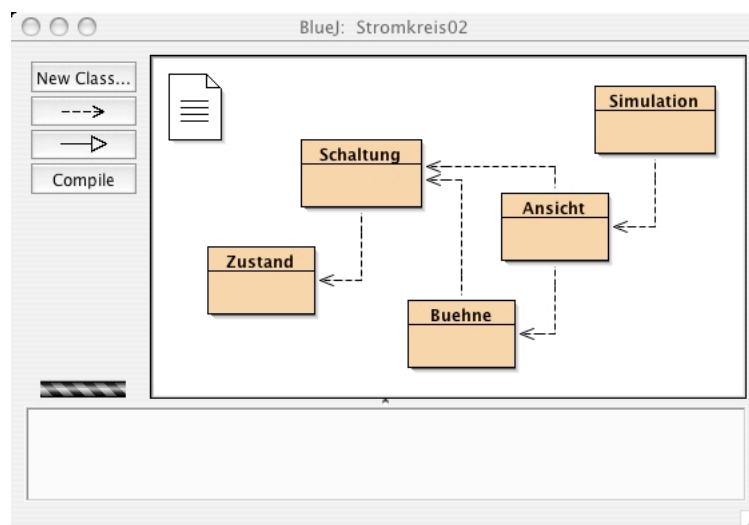
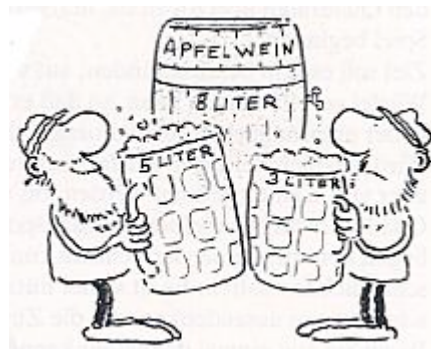


Abbildung 6.27: Klassendiagramm von *Stromkreis02*

Herzlichen Glückwunsch! Dieses Projekt war schon rechts anspruchsvoll.

6.3 Apfelwein

Ein Bauer und sein Freund kauften ein 8-Liter-Fass, gefüllt mit besten Apfelwein. Sie wollten den Wein gerecht aufteilen, besaßen aber nur einen 5-Liter-Krug und einen 3-Liter-Krug. Wie gelang es ihnen dennoch?



Diese bekannte Knobelaufgabe wirst Du nun programmieren. Abbildung 6.28 zeigt eine mögliche Benutzeroberfläche für dieses Spiel.

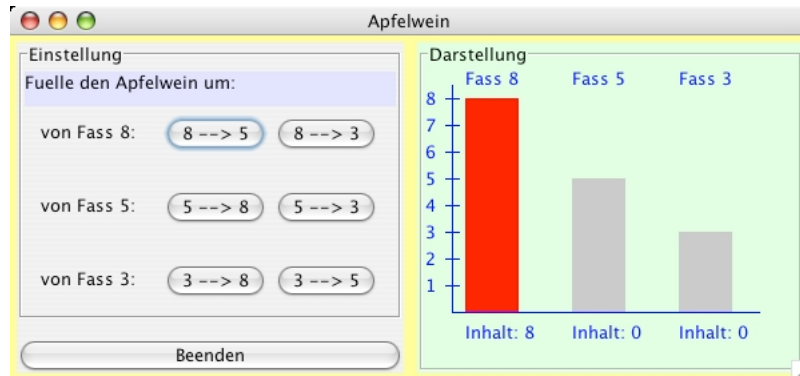


Abbildung 6.28: Apfelwein-Problem

Mit Hilfe der sechs Buttons im Einstellungsbereich kann der Benutzer den Apfelwein von einem Fass in ein anderes umfüllen. Der Darstellungsbereich zeigt den jeweiligen momentanen Inhalt der drei Fässer.

Übung 6.3.1:

Vervollständige die unten abgebildete Kombination aus Zustandstabelle und Zustandsübergangstabelle. Erstelle ein Zustandsübergangdiagramm und suche alle möglichen Lösungswege.

Zustand	F8	F5	F3	Folgezustand bei					
				8/5	8/3	5/8	5/3	3/8	3/5
0	8	0	0	1	2	-	-	-	-
1	3	5	0	-	3	0	4	-	-
2	5	0	3	5	-	-	-	0	6
3									
4									
5									
6									
7									
8									
9									
10									
11									
12									
13									
14									
15									
16									

Bemerkung:

Das Zeichen „-“ bedeutet, dass hier kein Zustandsübergang existiert ist und somit auch kein Folgezustand existiert. Bei der Implementierung muss hier die Nummer des jeweiligen Zustands eingetragen werden.

Übung 6.3.2:

Erstelle in **BlueJ** das Projekt *Apfelwein01*. Implementiere die Klassen *Zustand* und *Wein*. Orientiere dich bei der Implementierung an die bisherigen Projekte. Die Abbildungen 6.29 und 6.30 geben dir ein Skelett der Quelltexte für diese Klassen. Teste dein Projekt ausführlich mit Hilfe des Inspektors.

```
public class Zustand
{
    int f8, f5, f3; //Fass
    int fz85, fz83, fz58, fz53, fz38, fz35; //Folgezustand

    /** Konstruktor*/
    public Zustand(int f8H, int f5H, int f3H, int fz85H,
                   int fz83H, int fz58H, int fz53H, int fz38H, int fz35H)
    {
        ..... weitere Anweisungen .....
    }

    /** Liefert die Werte aller Datenfelder als Array. */
    public int[] gibWerte()
    {
        ..... weitere Anweisungen .....
    }
}
```

Abbildung 6.29: Unvollständiger Quelltext der Klasse *Zustand*

```
public class Wein
{
    private Zustand[] tabelle;
    private int fass8, fass5, fass3,
               fozu85, fozu83, fozu58, fozu53, fozu38, fozu35;

    /**Konstruktor */
    public Wein()
    {
        ..... weitere Anweisungen .....
    }
}
```

```
/** Fuelle die Tabelle mit den Zustaenden. */
private void fuelleTabelle()
{
    ..... weitere Anweisungen .....
}

/** Setzt die Füuellstand in den entsprechenden Folgezustand. */
private void geheInZustand(int nummer)
{
    ..... weitere Anweisungen .....
}

/** Holt sich den Folgezustand fuer den entsprechenden Schalter. */
public void holeFolgezustand(int uebergang)
{
    ..... weitere Anweisungen .....
}
}
```

Abbildung 6.30: Unvollständiger Quelltext der Klasse *Wein*

Anschließend musst du dich noch um die grafische Benutzeroberfläche kümmern.

Übung 6.3.3:

Hole dir vom Schulserver das Projekt *ApfelweinGUI* und speichere dieses unter dem Namen *Apfelwein02*. Versuche den Quelltext der Klassen *Ansicht* und *Buehne* zu verstehen.

Übung 6.3.4:

Füge mit *Edit > Add Class from File...* die Klassen *Wein* und *Zustand* in das Projekt *Apfelwein02* ein. Entferne in der Klasse *Ansicht* und in der Klasse *Buehne* die entsprechenden Auskommentierungen.

Übung 6.3.5:

Beim Testen deines Projekts *Apfelwein02* wirst du bemerkt haben, dass noch keine Fässer, sondern nur die Koordinatenachsen und die Beschriftung gezeichnet werden.

- a) Formuliere eine Begründung für dieses erhalten.
- b) Implementiere die Methoden *zeichne()* und *zeichneFass()* in der Klasse *Wein* wie in Abbildung 6.31 dargestellt.

```

/** Berechnet die Positionen und Groesse der Faesser. */
public void zeichne(Graphics g)
{
    int boden = 180; //Bodenlinie aller Faesser
    int faktor = 20; //Vergroesserungsfaktor
    int xPos, yPos, hoehe, inhalt;

    //Fass8
    xPos = 30;
    yPos = boden - 8 * faktor; //somit steht Fass8 auf Bodenlinie
    hoehe = 8 * faktor; //Hoehe des Fasses
    inhalt = fass8 * faktor; //momentaner Inhalt des Fasses
    zeichneFass(g, xPos, yPos, hoehe, inhalt);
    g.setColor(Color.blue);
    g.drawString("Inhalt: " + fass8, xPos, 200); //schreibt momentanen
                                                    Inhalt

    //Fass5

    ..... ähnliche Anweisungen fuer Fass5 .....

    //Fass3

    ..... ähnliche Anweisungen fuer Fass3 .....
}

/** Zeichnet die Rechtecke der Faesser. */
private void zeichneFass(Graphics g, int xPos, int yPos,
                        int hoehe, int inhalt)
{
    int breite = 40; //Breite aller Faesser

    g.setColor(Color.lightGray);
    g.fillRect(xPos, yPos, breite, hoehe - inhalt);
    g.setColor(Color.red);
    g.fillRect(xPos, yPos + (hoehe - inhalt), breite, inhalt);
}

```

Abbildung 6.31: Die Methoden *zeichne()* und *zeichneFass()* in der Klasse *Wein*

Übung 6.3.6: (für Experten)

Abbildung 6.32 zeigt eine grafische Benutzeroberfläche mit den Registerkarten *manuell* und *automtisch*. Mit Hilfe der letzteren Registerkarte kann der

Umfüllvorgang automatisch durchgeführt werden und die Zustände werden in einem Textarea aufgelistet.

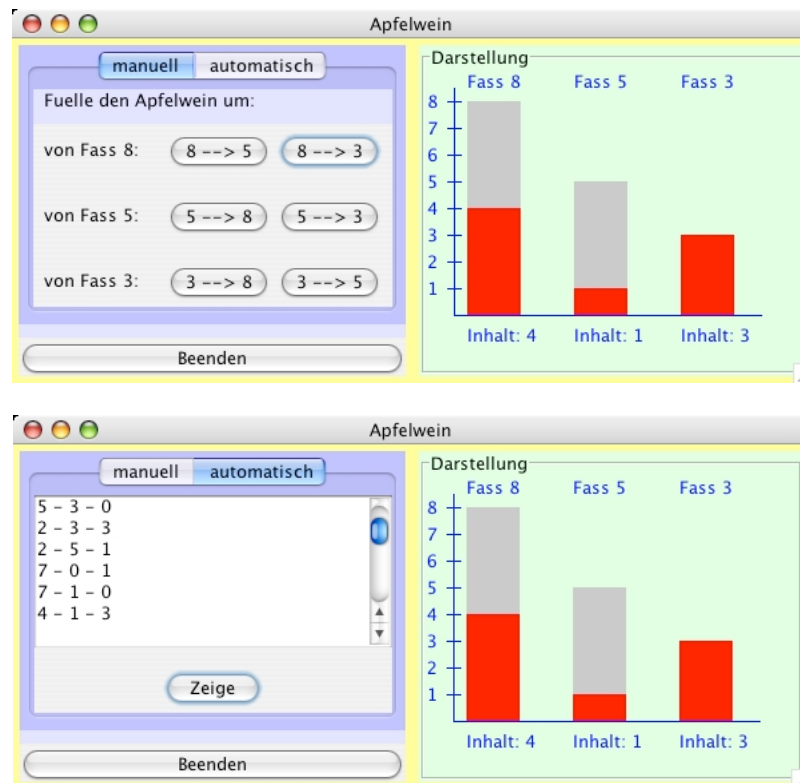


Abbildung 6.32: Projekt *Apfelwein03* mit Registerkarten *automatisch* und *manuell*

Zustands- und Zustandsübergangstabelle:

Zustand	F8	F5	F3	Folgezustand bei					
				8/5	8/3	5/8	5/3	3/8	3/5
0	8	0	0	1	2	-	-	-	-
1	3	5	0	-	3	0	4	-	-
2	5	0	3	5	-	-	-	0	6
3	0	5	3	-	-	2	-	1	-
4	3	2	3	3	-	2	-	7	1
5	0	5	3	-	-	2	-	1	-
6	5	3	0	1	8	0	2	-	-
7	6	2	0	1	4	0	9	-	-
8	2	3	3	3	-	2	-	6	10
9	6	0	2	11	2	-	-	0	7
10	2	5	1	-	3	12	8	1	-
11	1	5	2	-	3	9	13	1	-
12	7	0	1	10	2	-	-	0	14
13	1	4	3	3	-	2	-	15	11
14	7	1	0	1	16	0	12	-	-
15	4	4	0	1	13	0	16	-	-
16	4	1	3	3	-	2	-	14	15

Bemerkung:

Das Zeichen „-“ bedeutet, dass hier kein Zustandsübergang existiert ist und somit auch kein Folgezustand existiert. Bei der Implementierung muss hier die Nummer des jeweiligen Zustands eingetragen werden.

Lösungswege:

a)	0	2	6	8	10	12	14	16	15		
b)	0	1	4	7	9	11	13	15			
c)	0	1	4	2	6	8	10	12	14	16	15
d)	0	1	3	2	6	8	10	12	14	16	15