Inhalt

13MVC-Architektur

13.1 Das MVC-Konzept

13.1.1 Entwurfsmuster

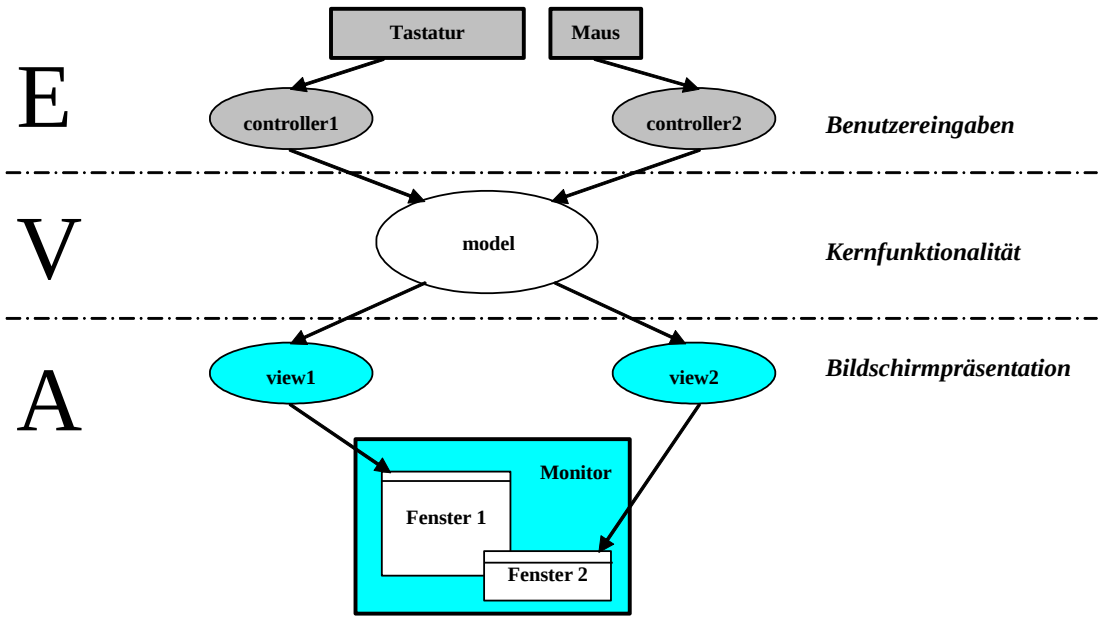
Ein **Entwurfsmuster** (*design pattern*) beschreibt eine bewährte Schablone für ein Entwurfsproblem. Es stellt damit eine wieder verwendbare Vorlage zur Problemlösung dar. Der Begriff ist aus der Architektur für die Softwareentwicklung übernommen wurden.

Ein gutes Muster sollte

- ein oder mehrere Probleme lösen,
- ein erprobtes Konzept bieten,
- über das rein Offensichtliche hinausgehen,
- den Benutzer in den Entwurfsprozess einbinden,
- Beziehungen aufzeigen, die tiefer gehende Strukturen und Mechanismen eines Systems umfassen.

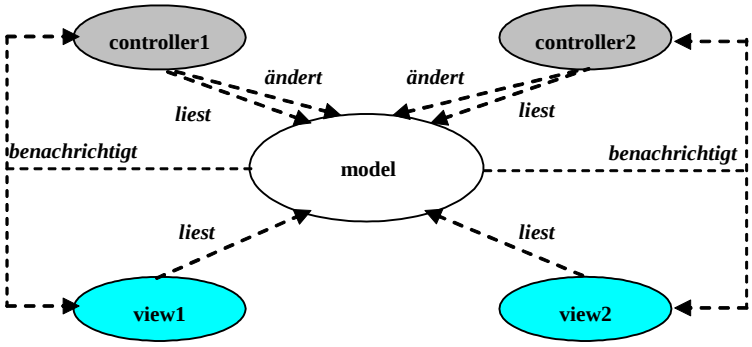
Es gibt eine Reihe vorgefertigter **Entwurfsmuster**, die eine Kombination mehrerer Objekte verschiedener Klassen darstellen und ihre Beziehungen untereinander festlegen.

Ein Beispiel für eine häufig vorkommende Kombination von Objekten ist die bereits mit Smalltalk eingeführte **MVC-Architektur** (*M .. Model*, *V .. View*, *C .. Controller*) zur Konstruktion von Benutzeroberflächen. Ein Objekt **model** realisiert die *Kernfunktionalität* und ist das *Anwendungsobjekt*, ein Objekt **view** ist die Schnittstelle für die *Bildschirmrepräsentation* und ein Objekt **controller** für die *Benutzereingaben*. Durch diese Architektur wird die Verarbeitung eines Problems von dessen Präsentation und Manipulation strikt getrennt. Damit ist es möglich, zu einem *Model mehrere Views* und *mehrere Controller* zur Verfügung zu stellen.



13.1.2 Mechanismus der MVC-Architektur

Durch die MVC-Architektur wird eine Anwendung von seiner Programmsteuerung entkoppelt. Bei der Programmsteuerung wiederum werden Eingaben und Ausgaben getrennt behandelt. Die drei Klassen (bzw. Klassenhierarchien) sorgen für eine strenge Abgrenzung zwischen diesen Aufgaben. Die miteinander kommunizierenden Objekte dieser Klassen müssen einen reibungslosen Ablauf garantieren.



Ein *Model* wird zuerst implementiert. Es enthält Daten und Kernfunktionalität der Anwendung, ist unabhängig vom *View* und *Controller* und hat insbesondere noch folgende zusätzliche Aufgaben: *Ein Model kennt seine Views und Controller, die zum Einsatz kommen, und informiert diese über Änderung in den Anwenderdaten.*
Ein *View* stellt Daten des Models am Bildschirm dar. *Er ermöglicht also die Sicht auf das Model und reagiert auf Änderungen im Model.*
Ein *Controller* ist die Eingabeschnittstelle zwischen Benutzer und Model. *Er interpretiert die empfangenen Eingabedaten und übergibt sie dem Model.* Ein *Controller* ist meist auf einen speziellen *View* zugeschnitten. Ist sein Verhalten vom Model abhängig, so muss er auch auf Änderungen im Model reagieren können.

Offen sind noch die Fragen:

- Woher weiß ein *View*, was er darstellen soll?
- Wie erfährt ein *View* bzw. *Controller*, dass sich *Model*-Daten geändert haben?
- Wie merkt ein *Model*, dass ein *Controller* Eingabedaten empfangen hat?

13.1.3 Klassen `java.util.Observable` und `java.util.Observer`

Das Verhalten eines *View* und unter Umständen auch eines *Controller* ist von Änderungen im *Model* abhängig. Das bedeutet, sie müssen das *Model* überwachen und auf Änderungen im *Model* reagieren.
Java stellt zwei Klassen zur Verfügung, welche den Aufbau eines solchen *Überwachungsmechanismus* ermöglichen: *Zu überwachende* Klassen werden von der Klasse **`Observable`** abgeleitet und *überwachende* Klassen implementieren das Interface **`Observer`**.
Die Klasse **`Observable`** ermöglicht die Verwaltung beliebig vieler Überwacher (Objekte der Klasse **`Observer`**):

- Sie registriert Observer in einer Observerliste,
- vermerkt Änderungen in den Anwenderdaten,
- teilt allen registrierten Observern diese mit und
- streicht Observer aus der Observerliste.

Methoden (Auswahl) der Klasse `java.util.Observable`

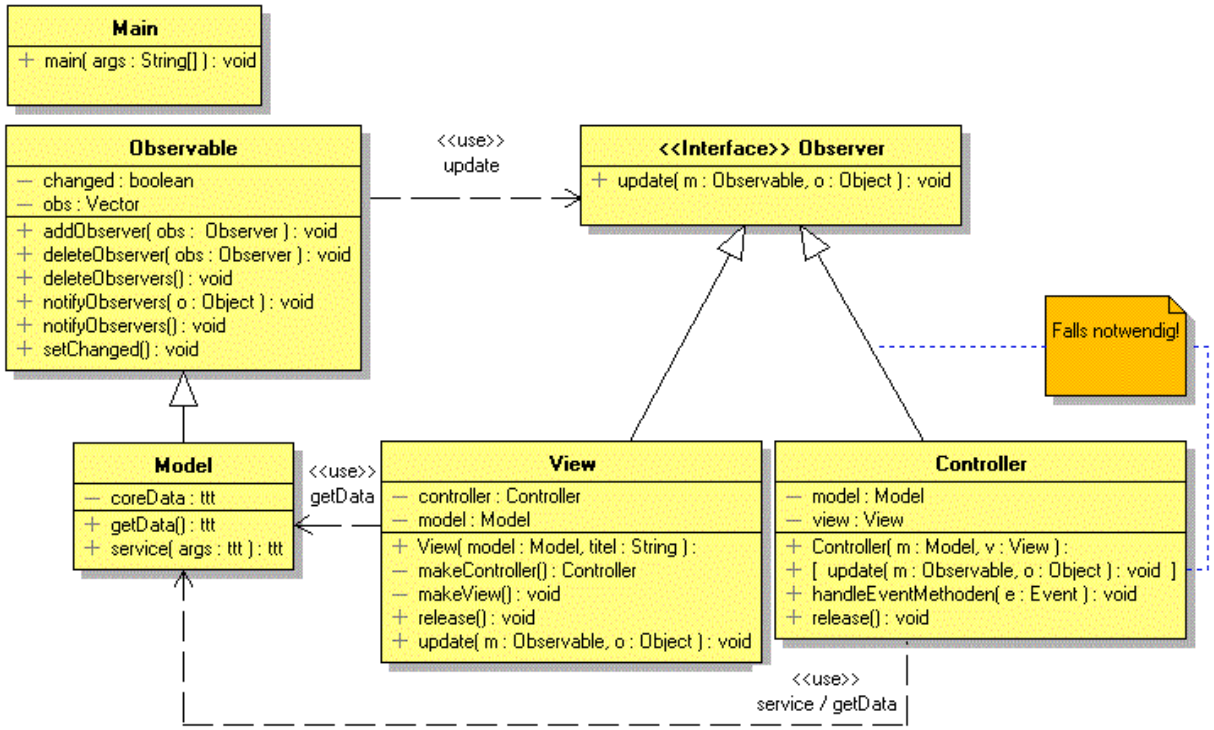
Name	Parameter-anzahl	Parameter-typ	Ergebnis-typ	Beschreibung
`addObserver`	1	`Observer`		nimmt Observer in Liste auf
`deleteObserver`	1	`Observer`		löscht Observer aus Liste
`deleteObservers`	0			löscht alle Observer aus Liste
`notifyObservers`	0			teilt den registrierten Observern Änderungen mit
`notifyObservers`	1	`Object`		teilt den Observern geändertes Objekt mit
`setChanged`	0			setzt ein Änderungsflag

Klassen, welche andere Klassen überwachen wollen, müssen das Interface **`Observer`** implementieren. Das Interface besitzt nur eine Methode, die Methode **`update`**. In ihr wird die Reaktion auf Änderungen der zu überwachenden Klasse festgelegt.

Methode der Klasse `java.util.Observer`

Name	Parameter-anzahl	Parameter-typ	Ergebnis-typ	Beschreibung
`update`	2	`Observable, Object`		wird aufgerufen, wenn das Objekt der überwachten Klasse geändert wurde

Verallgemeinerter Überwachungsmechanismus



Ein *Model* wird als Ableitung der Klasse **`Observable`** implementiert. Es sind Zugriffe zum *Lesen* (**`getData`**) darzustellender Daten (**`coreData`**) und zum *Ändern* (**`service`**) dieser vorzusehen. Die Abfragemethoden werden auch als **Getter** und die Änderungsmethoden als

Setter bezeichnet. Die *Benachrichtigung* über Änderungen erfolgt in den Methoden durch die **`Observable`**-Methoden **`setChanged`** und **`notifyObservers`**.

Model.java

```
// Model                                                    MM 2014
// MVC-Model-Grundstruktur, einfache Variante

import java.util.*;                                  // Observable

/**
 * Model-Grundstruktur
 * (ttt steht fuer einen beliebigen Datentyp).
 */
public class Model
  extends Observable
{
/**
 * coreData.
 */
  private ttt coreData = value;

/* ------------------------------------------------------ */
/**
 * getData-Methoden, Lesen von Daten.
 * @return coreData
 */
  public ttt getData()
  {
    return coreData;
  }

/* ------------------------------------------------------ */
/**
 * service-Methoden, Aendern von Daten.
 */
  public void service( ttt args)
  {
    // ...                                 // coreData aendern

    setChanged();                        // Aenderung anzeigen
    notifyObservers( coreData);
  }
}
```

Eine Klasse **`View`** implementiert das Interface **`Observer`**. Ihr Konstruktor und eine Methode **`makeController`** stellt die Verbindung zum *Model* und zum *Controller* her und richtet einen Überwachungsmechanismus ein, d.h. der *View* meldet sich beim *Model* als Überwacher an (**`addObserver`**). Die Darstellung am Bildschirm organisiert eine Methode **`makeView`**. Dazu greift diese unter Verwendung von **`getData`**-Methoden auf *Model*-Daten (**`coreData`**) zu und baut einen Bildschirm auf.

Eine **update**-Methode greift zur Bildschirmaktualisierung mittels der **getDate**-Methoden auf *Model*-Daten (**coreData**) zu und aktualisiert die Bildschirmdarstellung entsprechend.
Die Auflösung des Überwachungsmechanismus wird ebenfalls vom *View* gesteuert. Eine Methode **release** meldet den *View* beim *Model* als Überwacher ab (**deleteObserver**), trennt die Verbindungen zum *Model* und gibt nicht mehr benötigte Ressourcen frei.

View.java

```
// View                                                    MM 2014
// MVC-View-Grundstruktur, einfache Variante

import java.util.*;                                  // Observer

/**
 * View-Grundstruktur,
 * stellt Model dar, installiert Controller.
 */
public class View
  implements Observer
{
/* ------------------------------------------------------ */
                                                       // MVC
/**
 * Mathematisches Modell,
 * enthaelt Funktionalitaet des Problems.
 */
  private Model model;

/**
 * Zum View gehoeriger Controller.
 */
  private Controller controller;

/* ------------------------------------------------------ */
                                            // MVC-Installation
/**
 * Konstruktor, setzt Ueberschrift und installiert MVC.
 * @param model Model, welches dargestellt werden soll
 * @param titel Ueberschrift
 */
  public View( Model model, String titel)
  {
// Titelzeile
    // ...
// Model
    this.model = model;
    this.model.addObserver( this);           // Ueberwachung
// Controller
    controller = makeController();
// View
    makeView();
  }
```

```
/**
 * Erzeugt Controller, Empfaenger fuer Ereignisse.
 * @return Controller fuer View
 */
  private Controller makeController()
  {
    Controller controller
    = new Controller( model, this);
    // ...
    return controller;
  }

/**
 * Erzeugt View, baut die Oberflaeche auf.
 */
  private void makeView()
  {
    // ... model.getData() - Methoden
  }

/* ------------------------------------------------------ */
                                        // MVC-Deinstallation
/**
 * Deinstalliert MVC,
 * setzt Model und Controller zurück.
 */
  public void release()
  {
// Controller
    controller.release();
    controller = null;
// Model
    model.deleteObserver( this);
    model = null;
  }

/* ------------------------------------------------------*/
                                          // Observer-Methode
/**
 * Ueberschreibt Interfacemethode,
 * legt Reaktion auf Aenderungen fest.
 * @param m Model, welches Aenderungen meldet
 * @param o geaenderte Objekte
 */
  public void update( Observable m, Object o)
  {
    if( model != m) return;
    if( o == null)
    {
      // Model wird neu dargestellt
      // ... model.getData() - Methoden
    }
```

```
      else
      {
        // Objekt wird neu dargestellt
        // ... model.getData() - Methoden
      }
    }
  }
```

Die Klasse **`Controller`** implementiert *nur dann* das Interface **`Observer`**, wenn die Nutzereingaben von *Model*-Daten (**`coreData`**) abhängen. Hier wird dieser Fall *nicht* betrachtet.
Ein Konstruktor einer Klasse **`Controller`** stellt eine Verbindung zum *Model* und zum *View* her. Ein *Controller* reagiert auf Benutzereingaben durch spezielle Methoden (**`handleEvent`**). Mittels der **`service`**-Methoden des *Models* übergibt er diesem die Eingaben.
Eine Freigabe von Ressourcen des *Controllers* selbst erfolgt durch eine **`release`** Methode. Sie trennt die Verbindungen zum *Model* und zum *View*.

Controller.java

```
// Controller.java                                        MM 2014
// MVC-Controller-Grundstruktur, einfache Variante

/**
 * Controller-Grundstruktur,
 * installiert und deinstalliert Controller,
 * als Ereignisverarbeiter.
 */
public class Controller
{
/* ------------------------------------------------------ */
                                                      // MVC
/**
 * Mathematisches Modell,
 * enthaelt Funktionalitaet des Problems.
 */
  private Model model;

/**
 * Zum Controller gehoeriger View.
 */
  private View view;

/* ------------------------------------------------------ */
                                            // MVC-Installation
/**
 * Konstruktor, initialisiert Model und View.
 * @param m Mathematisches Modell
 * @param v zum Controller gehoeriger View
 */
  public Controller( Model m, View v)
```

```
    {
      model = m;
      view = v;
    }

  /* ------------------------------------------------------ */
                                          // MVC-Deinstallation
  /**
   * Deinstalliert MVC,
   * setzt Model und View zurück.
   */
    public void release()
    {
      model = null;
      view = null;
    }

  /* ------------------------------------------------------ */
                                        // handleEvent-Methode
  /**
   * Verarbeiten von Eingaben.
   */
    public void handleEventMethoden()
    {
      // ...
    }
  }
```

Wir starten das Programm, indem eine Benutzeroberfläche mit dem darzustellenden *Model* aufgerufen wird. Die Steuerung des weiteren Ablaufs übernimmt der *View*.

Main.java

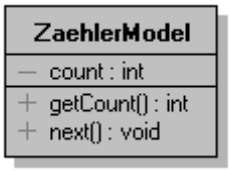
```
  // Main.java                                          MM 2014
  // MVC-Main-Grundstruktur, einfache Variante

  /**
   * Initialisieren eines Model mit einem View.
   */
  public class Main
  {
  /**
   * Starten eines Programms mit MVC-Architektur,
   * Initialisiere ein Model mit einem View.
   */
    public static void main( String args[])
    {
      View view = new View( new Model(), "Titel");
    }
  }
```

13.2 Beispiel „Zähler modulo 10“

Als erstes, sehr einfaches Beispiel soll ein Zähler modulo 10 mit einer einfachen MVC-Architektur ohne Benutzeroberfläche implementiert werden. Ein- und Ausgaben erfolgen über Tastatur bzw. Konsole. Das Zählen erfolgt schrittweise nach vorheriger Abfrage. Ein Abbruchmechanismus sorgt für einen kontrollierten Abbruch des Programms.
Der Überwachungsmechanismus wird fast vollständig übernommen, nur die Methoden **`makeView`** und **`update`** müssen im *View* spezifiziert und entsprechende **`handleEvent`**-Methoden im *Controller* hinzugeführt werden.

13.2.1 Design und Implementierung des Models

Ein *Model* wird durch die Klasse **`ZaehlerModel`** als Ableitung der Klasse **`Observable`** definiert. In einem Attribut **`count`** (≙ **`coreData`**) wird der Zählerstand festgehalten. Ein Zugriff auf diesen erfolgt durch eine Methode **`getCount`** (≙ **`getData`**) und das Zählen durch eine Methode **`next`** (≙ **`service`**). In ihr wird der geänderte Zählerstand mittels der **`Observable`**-Methoden **`setChanged`** und **`notifyObservers`** angezeigt.



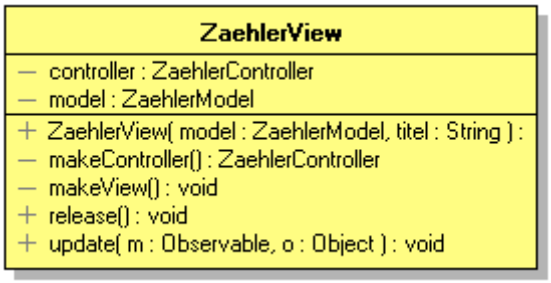
ZaehlerModel.java

```
// ZaehlerModel                                          MM 2014
// MVC-Model

import java.util.*;                                  // Observable

/**
 * Zaehler modulo 10,
 * Version Konsole/Tastatur.
 */
public class ZaehlerModel
  extends Observable
{
                                                       // coreData
/**
 * Zaehler.
 */
  private int count = 0;

/* ------------------------------------------------------ */
                                                   // getData-Methode
/**
 * Lesen des Zaehlers.
 * @return Zaehlerstand
```

```
   */
    public int getCount()
    {
      return count;
    }

  /* ------------------------------------------------------ */
                                                  // service-Methode
  /**
   * Zaehlen.
   */
    public void next()
    {
      count = (count + 1) % 10;       // Wert wurde veraendert

      setChanged();                        // Aenderung anzeigen
      notifyObservers( new Integer( count));
    }
  }
```

13.2.2 Design und Implementierung des Views

Die zwei Methoden **`makeView`** und **`update`** sind für einen spezifischen *View* dem Problem anzupassen. Alle anderen Bestandteile eines *Views* werden im Wesentlichen unverändert übernommen.



ZaehlerView.java

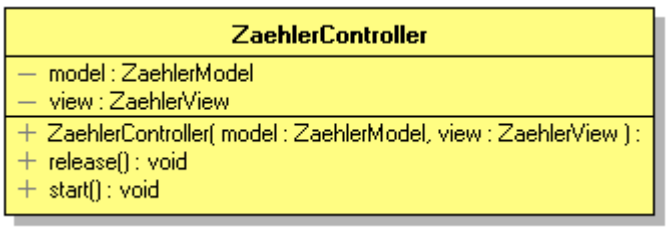
```
  // ZaehlerView                                          MM 2014
  // MVC-View

  import java.util.*;                                  // Observer

  /**
   * View zum Zaehler modulo 10,
   * Version Konsole/Tastatur.
   */
  public class ZaehlerView
    implements Observer
  {
```

```
/* ------------------------------------------------------ */
                                                    // MVC
/**
 * Mathematisches Modell,
 * enthaelt Funktionalitaet des Zeahlers.
 */
  private ZaehlerModel model;

/**
 * Zum View gehoeriger Controller.
 */
  private ZaehlerController controller;

/* ------------------------------------------------------ */
                                          // MVC-Installation
/**
 * Konstruktor, setzt Ueberschrift und installiert MVC.
 * @param model Model, welches dargestellt werden soll
 * @param titel Ueberschrift
 */
  public ZaehlerView( ZaehlerModel model, String titel)
  {
// Titelzeile
    System.out.println( titel);
// Model
    this.model = model;
    this.model.addObserver( this);          // Ueberwachung
// Controller
    controller = makeController();
// View
    makeView();
  }

/**
 * Erzeugt Controller.
 * @return Controller fuer View
 */
  private ZaehlerController makeController()
  {
    return new ZaehlerController( model, this);
  }

/**
 * Erzeugt View, startet Controller.
 */
  private void makeView()
  {
    System.out.println( "" + model.getCount());
    System.out.print( "Weiter (j/n): ");
    controller.start();
  }
```

```
/* ------------------------------------------------------ */
                                          // MVC-Deinstallation
/**
 * Deinstalliert MVC,
 * setzt Model und Controller zurück.
 */
  public void release()
  {
// Controller
    controller.release();
    controller = null;
// Model
    model.deleteObserver( this);
    model = null;
  }

/* ------------------------------------------------------ */
                                             // Observer-Methode
/**
 * Ueberschreibt Interfacemethode,
 * legt Reaktion auf Aenderungen fest.
 * @param m Model, welches Aenderungen meldet
 * @param o geaenderte Objekte
 */
  public void update( Observable m, Object o)
  {
    if( model == m)
    {
      System.out.println( "" + model.getCount());
      System.out.print( "Weiter (j/n): ");
    }
  }
}
```

13.2.3 Design und Implementierung des Controllers

Eine **handleEvent-Methode**, hier die Methode **start**, beschreibt die Ereignisverarbeitung und ist deshalb dem Problem anzupassen. Alle anderen Bestandteile eines *Controllers* werden im Wesentlichen unverändert übernommen.



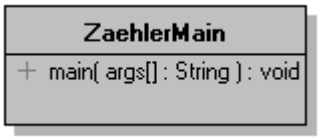
ZaehlerController.java

```
// ZaehlerController.java                                MM 2014
// MVC-Controller

import Tools.IO.*;                          // Tastatureingaben

/**
 * Controller zum ZaehlerView.
 */
public class ZaehlerController
{
/* ------------------------------------------------------ */
                                                      // MVC
/**
 * Mathematisches Modell,
 * enthaelt Funktionalitaet des Zaehler.
 */
  private ZaehlerModel model;

/**
 * Zum Controller gehoeriger View.
 */
  private ZaehlerView view;

/* ------------------------------------------------------ */
                                            // MVC-Installation
/**
 * Konstruktor, initialisiert Model und View.
 * @param model Mathematisches Modell
 * @param view zum Controller gehoeriger View
 */
  public ZaehlerController
     ( ZaehlerModel model, ZaehlerView view)
  {
    this.model = model;
    this.view = view;
  }

/* ------------------------------------------------------ */
                                          // MVC-Deinstallation
/**
 * Freigabe des Controllers,
 * setzt Model und View zurück.
 */
  public void release()
  {
    model = null;
    view = null;
  }

/* ------------------------------------------------------ */
                                        // handleEvent-Methode
```

```
/**
 * Verarbeiten der Tastatureingaben.
 */
  public void start()
  {
    String s = "";
    do
    {
      s = IOTools.readLine();
      if( s.equals( "j")) model.next();
      if( s.equals( "n")) break;
    }
    while( true);
    view.release();
  }
}
```

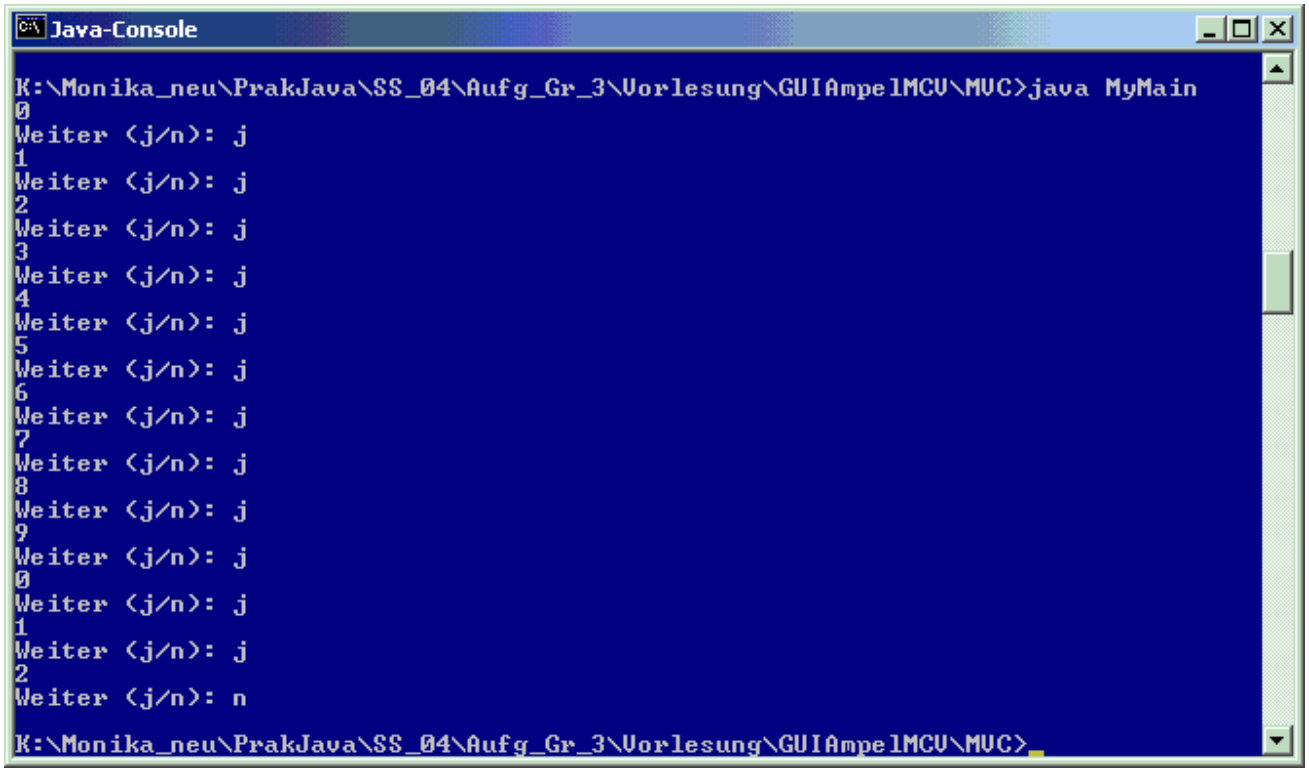
13.2.4 Implementierung des Programmaufbaus

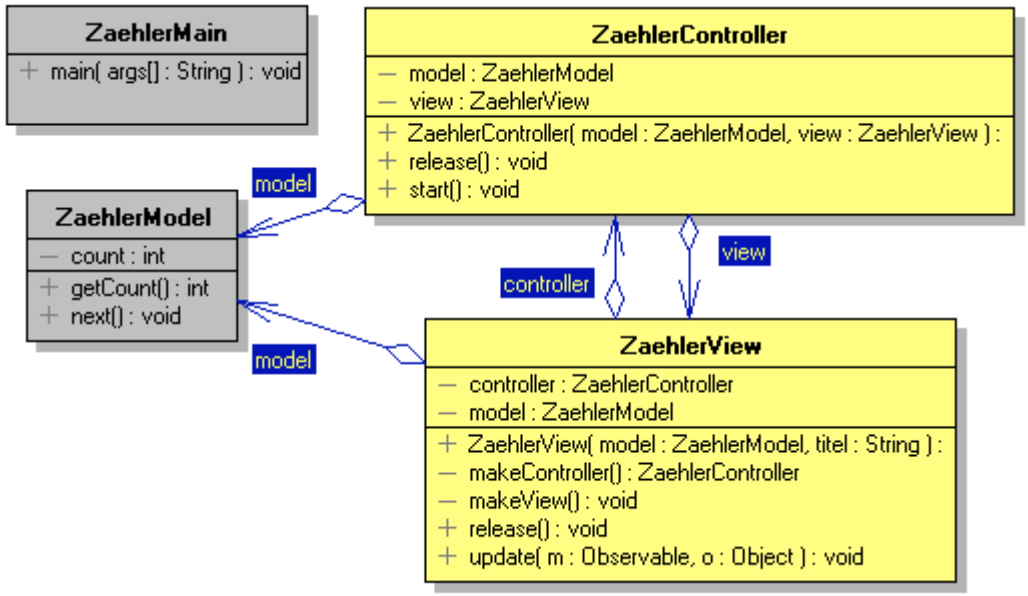
Der Programmaufruf *Main* wird im Wesentlichen unverändert übernommen.



ZaehlerMain.java

```
// ZaehlerMain.java                                    MM 2014
// MVC-Main

/**
 * Simulation eines Zaehlers modulo 10,
 * Version Konsole/Tastatur.
 */
public class ZaehlerMain
{
/**
 * Starten des Zaehlerprogramms mit MVC-Architektur,
 * Initialisiere ein Model mit einem View.
 */
  public static void main( String args[])
  {
    ZaehlerView view = new ZaehlerView
            ( new ZaehlerModel(), "Zaehler modulo 10");
  }
}
```





Modellierung und Programmierung

Klassendiagramm, **Dokumentation**, ***ZaehlerMVC.zip***

13.2.5 Beispiel „Zähler modulo 10“ mit grafischer Oberfläche

Model, Main
Durch die MVC-Architektur bleiben die Klassen **`ZaehlerModel`**, **`ZaehlerMain`** unabhängig von der Benutzerschnittstelle erhalten.

View
Notwendige Attribute und zwei Methoden **`makeView`** und **`update`** sind für einen spezifischen *View* dem Problem anzupassen. Alle anderen Bestandteile eines *Views* werden im Wesentlichen unverändert übernommen.

ZaehlerView.java

```
// ZaehlerView.java                                        MM 2014
// MVC-View

import java.util.*;                                  // Observer
import javax.swing.*;     //JFrame, JPanel, JLabel, JButton

/**
 * View zum Zaehler modulo 10,
 * Version mit Benutzeroberflaeche.
 */
public class ZaehlerView
  extends JFrame
  implements Observer
{
/* ------------------------------------------------------- */
                                                       // GUI
/**
 * Konstante, Zaehlen.
 */
  public static final String ACTION_NEXT = "Next";

/**
 * Konstante, Beenden des Programms.
 */
  public static final String ACTION_QUIT = "Quit";

/**
 * Label, fuer Zahl.
 */
  private JLabel lbZahl;

/* ------------------------------------------------------- */
```

```
                                                        // MVC
  /**
   * Mathematisches Modell,
   * enthaelt Funktionalitaet des Zaehlers.
   */
    private ZaehlerModel model;

  /**
   * Zum View gehoeriger Controller.
   */
    private ZaehlerController controller;

  /* ------------------------------------------------------ */
                                          // MVC-Installation
  /**
   * Konstruktor, setzt Ueberschrift und installiert MVC.
   * @param model Model, welches dargestellt werden soll
   * @param titel Ueberschrift
   */
    public ZaehlerView( ZaehlerModel model, String titel)
    {
  // Titelzeile
      super( titel);
  // Model
      this.model = model;
      this.model.addObserver( this);
  // Controller
      controller = makeController();
  // View
      makeView();
    }

  /**
   * Erzeugt Controller,
   * Empfaenger fuer Ereignisse.
   * @return Controller fuer View
   */
    private ZaehlerController makeController()
    {
      return new ZaehlerController( model, this);
    }

  /**
   * Erzeugt View,
   * baut die grafische Oberflaeche auf,
   * Controller verarbeitet Fensterereignisse.
   */
    private void makeView()
    {
  // ContentPane
      setContentPane( createContentPane());
```

```
// Fenster mit Controller als Listener
    addWindowListener( controller);
    pack();
    setVisible( true);
  }

/**
 * Erzeugt Darstellungsbereich mit Steuerbuttons
 * und Label fuer Zaehler,
 * Controller verarbeitet Buttonereignisse.
 */
  private JPanel createContentPane()
  {
    JPanel contentPane = new JPanel();

            // Button Next mit Controller als Listener
    JButton btNext = new JButton( ACTION_NEXT);
    contentPane.add( btNext);
    btNext.addActionListener( controller);

    lbZahl = new JLabel( "0");       // Label Augenzahl
    contentPane.add( lbZahl);

            // Button Quit mit Controller als Listener
    JButton btQuit = new JButton( ACTION_QUIT);
    contentPane.add( btQuit);
    btQuit.addActionListener( controller);

    return contentPane;
  }

/* ------------------------------------------------- */
                                      // MVC-Deinstallation
/**
 * Entfernt Fenster, deinstalliert MVC.
 */
  public void release()
  {
// View (Fenster)
    dispose();
// Controller
    controller.release();
    controller = null;
// Model
    model.deleteObserver( this);
    model = null;
  }

/* -------------------------------------------------*/
                                        // Observer-Methode
/**
 * Ueberschreibt Interfacemethode,
```

```
   * legt Reaktion auf Aenderungen fest.
   * @param m Model, welches Aenderungen meldet
   * @param o geaenderte Objekte
   */
    public void update( Observable m, Object o)
    {
      if( model == m)
        lbZahl.setText( "" + model.getCount());
    }
}
```

Zwei Ergänzungen sind für grafische Oberflächen noch wichtig:

- Im Konstruktor wird der Titel mit **`super( titel)`** in die Menüleiste eingetragen, einem Konstruktor der übergeordneten Klasse **`JFrame`**.
- Ein Fenster wird von der Oberfläche durch die Methode **`dispose`** der Klasse **`JFrame`** entfernt. Dieser Aufruf muss in der Methode **`release`** erfolgen.

Controller

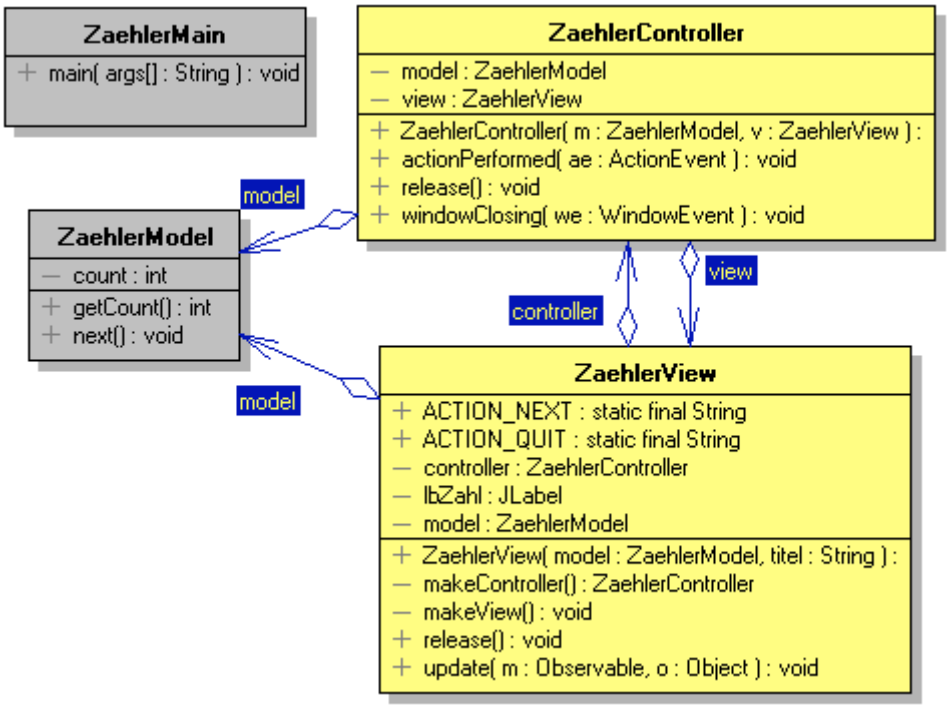
Als **handleEvent-Methoden** sind hier zwei Methoden zu implementieren:
`actionPerformed` für das Betätigen der Button und **`windowClosing`** zum Fensterschließen. Alle anderen Bestandteile eines *Controller* werden im Wesentlichen unverändert übernommen.

ZaehlerController.java

```
// ZaehlerController.java                                MM 2014
// MVC-Controller

import java.awt.event.*;                    // Listener, Event

/**
 * Controller zum ZaehlerView.
 */
public class ZaehlerController
  extends WindowAdapter
  implements ActionListener
{
/* ------------------------------------------------------ */
                                                        // MVC
/**
 * Mathematisches Modell,
 * enthaelt Funktionalitaet des Zaehlers.
 */
  private ZaehlerModel model;

/**
 * Zum Controller gehoeriger View.
 */
  private ZaehlerView view;
```

```
/* ------------------------------------------------------ */
                                          // MVC-Installation
/**
 * Konstruktor, initialisiert Model und View.
 * @param model Mathematisches Modell
 * @param view zum Controller gehoeriger View
 */
  public ZaehlerController
     ( ZaehlerModel model, ZaehlerView view)
  {
    this.model = model;
    this.view = view;
  }

/* ------------------------------------------------------ */
                                       // MVC-Deinstallation
/**
 * Freigabe des Controllers,
 * setzt Model und View zurück.
 */
  public void release()
  {
    model = null;
    view = null;
  }

/* ------------------------------------------------------*/
                                     // handleEvent-Methode
/**
 * ActionListener,
 * Ereignisverarbeitung Betaetigen eines Button.
 */
  public void actionPerformed( ActionEvent ae)
  {
    String command = ae.getActionCommand();

// Zaehlen durch Button 'Next'
    if( command.equals( ZaehlerView.ACTION_NEXT))
      model.next();                          // Weiterzaehlen

// Programmabbruch durch Button 'Quit'
    if( command.equals( ZaehlerView.ACTION_QUIT))
      view.release();                  // Programm  beenden
  }

/**
 * WindowAdapter,
 * Ereignisverarbeitung Schliessen des Fensters,
 * Programm wird beendet.
 */
  public void windowClosing( WindowEvent we)
```

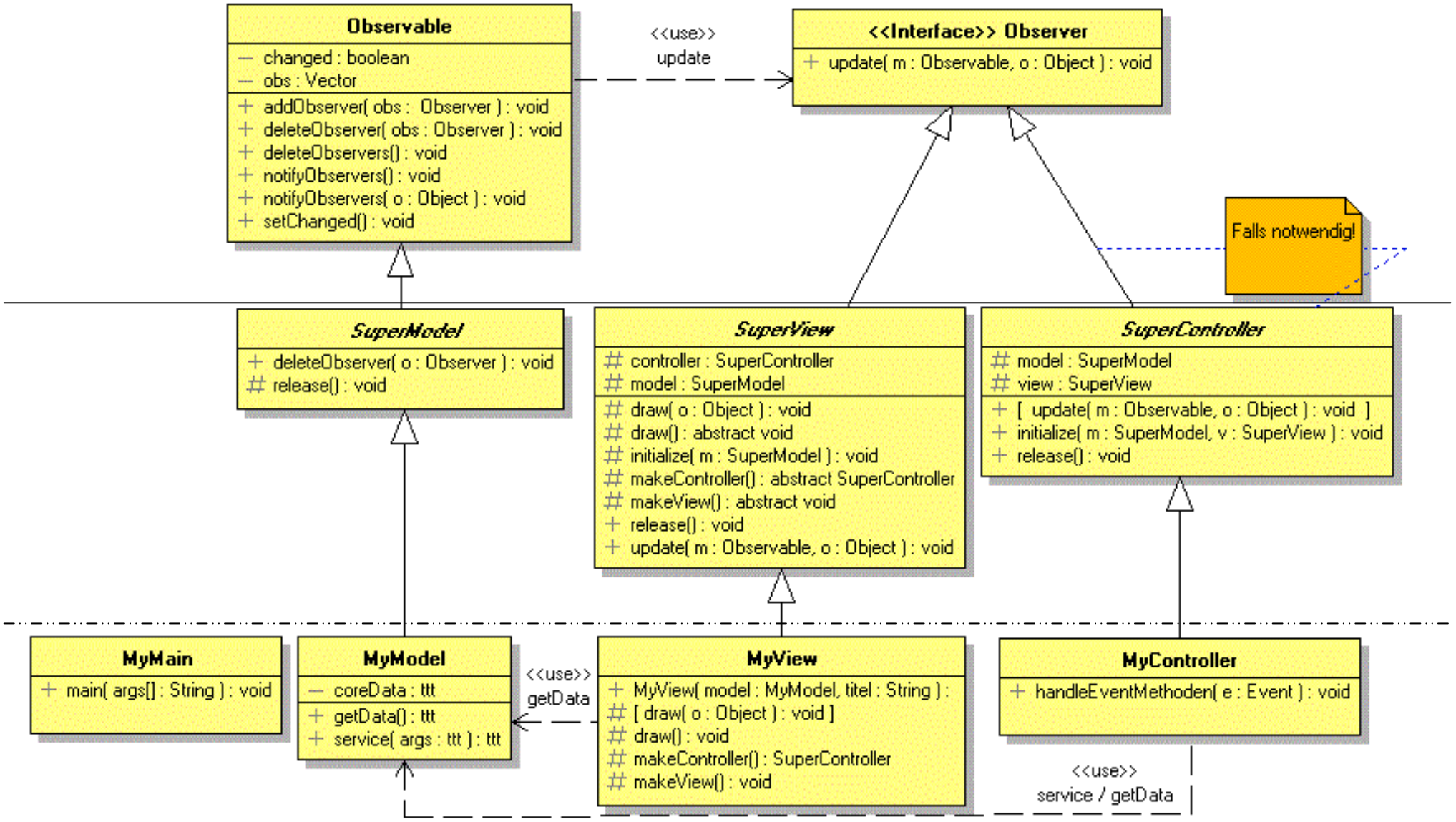
```
    {
      view.release();
    }
}
```



Modellierung und Programmierung

Klassendiagramm, **Dokumentation**, ***ZaehlerMVC_GUI.zip***

13.3 Steuerung mehrerer View und Controller zu einem Model



In den Klassen **ZaehlerView** und **ZaehlerControlle**r des letzten Beispiels findet man Methoden bzw. Teile von Methoden, welche spezifisch für das hier dargestellte Problem notwendig sind (**makeController**, **makeView**, **update**, handleEvent-Methoden). Andere sind unabhängig von der Problemstellung, d.h. sie können unverändert bei anderen Problemen so übernommen werden (Konstruktoren, **model**, **view** und **controller** als Attribute, **release**). Deshalb bietet es sich im Sinne der Wiederverwendung an, diese als Generalisierung bereitzustellen.

Die allgemeingültigen Eigenschaften und Methoden der MVC-Architektur wurden als abstrakte Superklassen im Paket **Tools.MVC** zusammengefasst.

Wir betrachten hier den Fall, dass mehrere unterschiedliche *View/Controller* parallel eingerichtet werden sollen. Dabei wird das Programm selbst erst dann beendet, wenn sich alle *View/Controller* verabschiedet haben.

13.3.1 Abstrakte Klassen

SuperModel.java

```
// SuperModel.java                                        MM 2014
// MVC-Model-Superklasse
package Tools.MVC;

import java.util.*;                                  // Observable

/**
 * Model-Superklasse, abstract,
 * Steuerung fuer ein oder mehrere Views.
 */
public abstract class SuperModel
  extends Observable
{
/**
 * Entferne Observer.
 */
  public void deleteObserver( Observer o)
  {
    super.deleteObserver( o);
    if( countObservers() == 0) release();
  }

/**
 * Beende Programm.
 */
  protected void release()
  {
    System.exit( 0);
  }
}
```

SuperView.java

```
// SuperView.java                                        MM 2014
// MVC-View-Superklasse
package Tools.MVC;

import java.util.*;                                  // Observer

/**
 * View-Superklasse, abstract,
 * stellt Model dar, installiert Controller.
 */
public abstract class SuperView
  implements Observer
{
/**
 * Mathematisches Modell,
 * enthaelt Funktionalitaet des Problems.
 */
  protected SuperModel model;

/**
 * Zum View gehoeriger Controller.
 */
  protected SuperController controller;

/* ------------------------------------------------------ */
                                          // MVC-Installation
/**
 * MVC-Installation,
 * meldet den View als Observer des Models an,
 * erzeugt Controller fuer Model und View,
 * erzeugt View und uebergibt ihm die Aktivitaeten.
 * @param m Mathematisches Modell
 */
  protected void initialize( SuperModel m)
  {
// Model
    model = m;
    model.addObserver( this);                 // Ueberwachung
// Controller
    controller = makeController();
    controller.initialize( model, this);
// View
    makeView();
  }

/**
 * Erzeugt Controller, abtract.
 * @return Controller fuer View
 */
  protected abstract SuperController makeController();
```

```
/**
 * Erzeugt und startet View, abtract.
 */
  protected abstract void makeView();

/* ------------------------------------------------------ */
                                        // MVC-Deinstallation
/**
 * Deinstalliert MVC,
 * setzt Model und Controller zurück.
 */
  public void release()
  {
// Controller
    controller.release();
    controller = null;
// Model
    model.deleteObserver( this);
    model = null;
  }

/* ------------------------------------------------------ */
                                              // draw-Methoden
/**
 * Darstellung des Model, abstract.
 */
  protected abstract void draw();

/**
 * Darstellung eines Objektes,
 * wird die Methode nicht ueberschrieben,
 * so wird das ganze Model neu dargestellt.
 * @param o Object geaendertes Objekt
 */
  protected void draw( Object o)
  {
    draw();
  }

/* ------------------------------------------------------ */
                                          // Observer-Methode
/**
 * Ueberschreibt Interfacemethode,
 * legt Reaktion auf Aenderungen fest.
 * @param m Model, welches Aenderungen meldet
 * @param o geaenderte Objekte
 */
  public void update( Observable m, Object o)
  {
    if( model != m) return;
    if( o == null)
```

```
        draw();                  // Model wird neu dargestellt
      else
        draw( o);               // Objekt wird neu dargestellt
    }
  }
```

SuperController.java

```
  // SuperController.java                                MM 2014
  // MVC-Controller-Superklasse
  package Tools.MVC;

  /**
   * Controller-Superklasse zur View-Superklasse,
   * installiert und deinstalliert Controller
   * als Ereignisverarbeiter.
   */
  public class SuperController
  {
  /* ------------------------------------------------------ */
                                                      // MVC
  /**
   * Mathematisches Model,
   * enthaelt Funktionalitaet des Problems.
   */
    protected SuperModel model;

  /**
   * Zum Controller gehoeriger View.
   */
    protected SuperView view;

  /* ------------------------------------------------------ */
                                           // MVC-Installation
  /**
   * Initialisiert Model und View.
   * @param m Mathematisches Modell
   * @param v zum Controller gehoeriger View
   */
    public void initialize( SuperModel m, SuperView v)
    {
      model = m;
      view = v;
    }

  /* ------------------------------------------------------ */
                                         // MVC-Deinstallation
  /**
   * Deinstalliert MVC,
   * setzt Model und View zurück.
   */
    public void release()
```

```
    {
      model = null;
      view = null;
    }
  }
```

13.3.2 Abgeleitete Klassen

Konkrete Anwendungen, hier **`MyModel`**, **`MyView`** und **`MyController`**, werden von diesen abstrakten Klassen abgeleitet. Dazu muss das Paket **`Tools.MVC`** importiert werden.

MyModel.java

```
// MyModel.java                                          MM 2014
// MVC-Model-Grundstruktur

import Tools.MVC.*;                                 // SuperModel

/**
 * Model-Grundstruktur
 * (ttt steht fuer einen beliebigen Datentyp).
 */
public class MyModel
  extends SuperModel
{
/**
 * coreData.
 */
  private ttt coreData = value;

/* ------------------------------------------------------ */
/**
 * getData-Methoden, Lesen von Daten.
 * @return coreData
 */
  public ttt getData()
  {
    return coreData;
  }

/* ------------------------------------------------------ */
/**
 * service-Methoden, Aendern von Daten.
 */
  public void service( ttt args)
  {
    // ...                                 // coreData aendern
    setChanged();                       // Aenderung anzeigen
    notifyObservers( coreData);
  }
}
```

MyView.java

```
// MyView.java                                          MM 2014
// MVC-View-Grundstruktur

import Tools.MVC.*;                                  // SuperView

/**
 * View-Grundstruktur,
 * View mit Controller wird installiert.
 */
public class MyView
  extends SuperView
{
/* ------------------------------------------------- */
                                          // MVC-Installation
/**
 * Konstruktor, setzt Ueberschrift und installiert MVC.
 * @param model Model, welches dargestellt werden soll.
 * @param titel Ueberschrift
 */
  public MyView( MyModel model, String titel)
  {
// Titelzeile anzeigen
    // ...
// MVC-Installation
    initialize( model);
  }

/**
 * Erzeugt Controller,
 * Empfaenger fuer Ereignisse,
 * ueberschreibt abstrakte Methode.
 * @return Controller fuer View
 */
  protected SuperController makeController()
  {
    MyController controller = new MyController();
    // ...
    return controller;
  }

/**
 * Erzeugt,
 * ueberschreibt abstrakte Methode,
 * baut die Oberflaeche auf.
 */
  protected void makeView()
  {
// Praesentation aufbauen
    // ...
  }
```

```
  /* ------------------------------------------------------ */
                                                 // draw-Methoden
  /**
   * Darstellung des Model,
   * ueberschreibt abstrakte Methode.
   */
    protected void draw()
    {
      // ...
    }

  /**
   * Darstellung eines Objekts, optional.
   */
    protected void draw( Object o)
    {
      // ...
    }
  }
```

MyController.java

```
  // MyController.java                                  MM 2014
  // MVC-Controller-Grundstruktur

  import Tools.MVC.*;                        // SuperController

  /**
   * Controller-Grundstruktur,
   * verwaltet Eingaben.
   */
  public class MyController
    extends SuperController
  {
  /* ------------------------------------------------------ */
                                          // handleEvent-Methode
  /**
   * Verarbeiten von Eingaben.
   */
    public void handleEventMethoden()
    {
      // ...
    }
  }
```

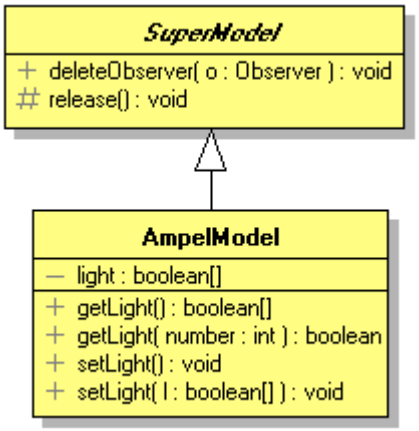
Schließlich ist noch die Startklasse zu vereinbaren, welche das konkrete *Model* mit ein oder mehreren *View* bereitstellt:

MyMain.java

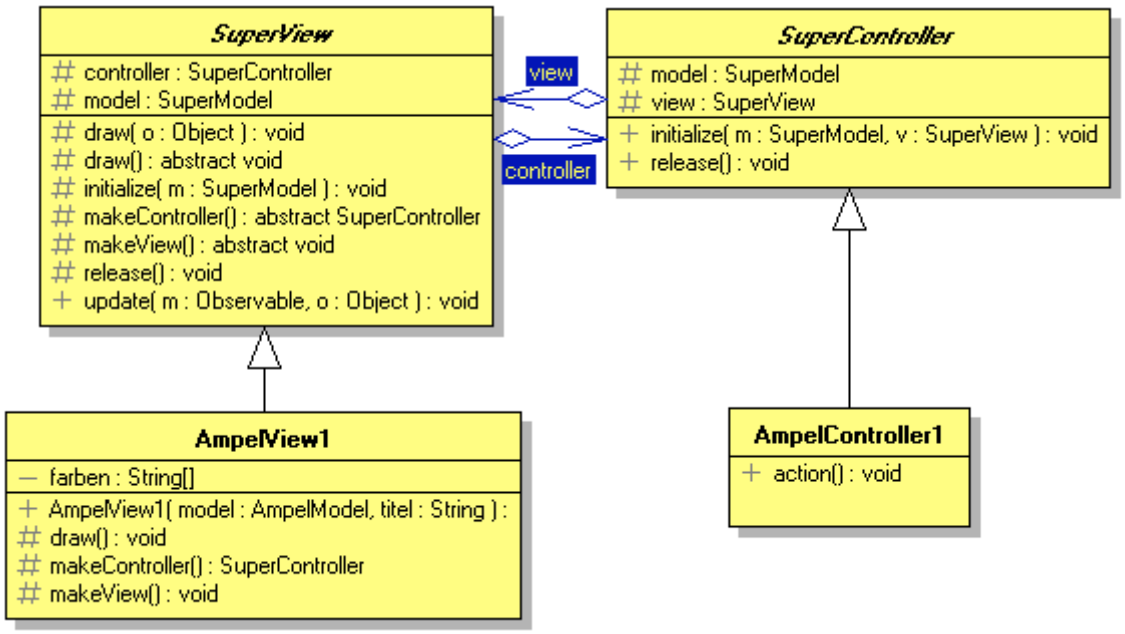
```
// MyMain.java                                          MMM 2014
// MVC-Main-Grundstruktur

/**
 * Initialisieren eines Model
 * mit einem oder mehreren View.
 */
public class MyMain
{
/**
 * Starten eines Programms mit MVC-Architektur,
 * Initialisiere ein Model mit mehreren View.
 */
  public static void main( String args[])
  {
// Model
    MyModel model = new MyModel();

// View 1, Uebergabe des Model
    MyView view1 = new MyView( model, "Titel 1");

// View 2, Uebergabe des Model
    MyView view2 = new MyView( model, "Titel 2");

// ...
  }
}
```

13.3.3 Beispiel „Ampelsteuerung“

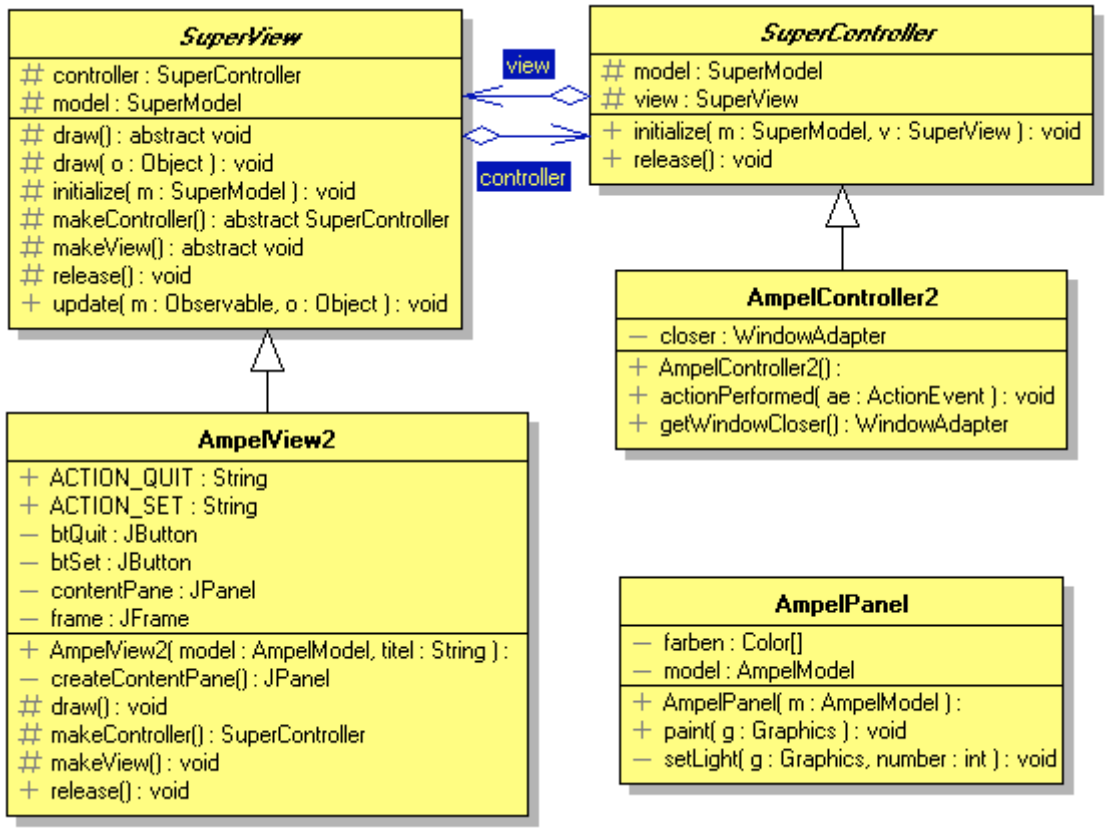
In diesem Beispiel werden die abstrakten Klassen **`SuperModel`**, **`SuperView`** und **`SuperController`** in einer einfachen Ampelsteuerung zur Anwendung kommen. Die Ein- und Ausgaben erfolgen sowohl über Tastatur und Konsole als auch durch eine grafische Oberfläche. Beide Möglichkeiten können parallel genutzt werden.



In der ersten Version sollen Ein- und Ausgaben über Konsole und Tastatur erfolgen:



In der zweiten Version wird eine Benutzeroberfläche aufgebaut. Ein **`AmpelPanel`** ist für das Zeichnen der Ampel zuständig.

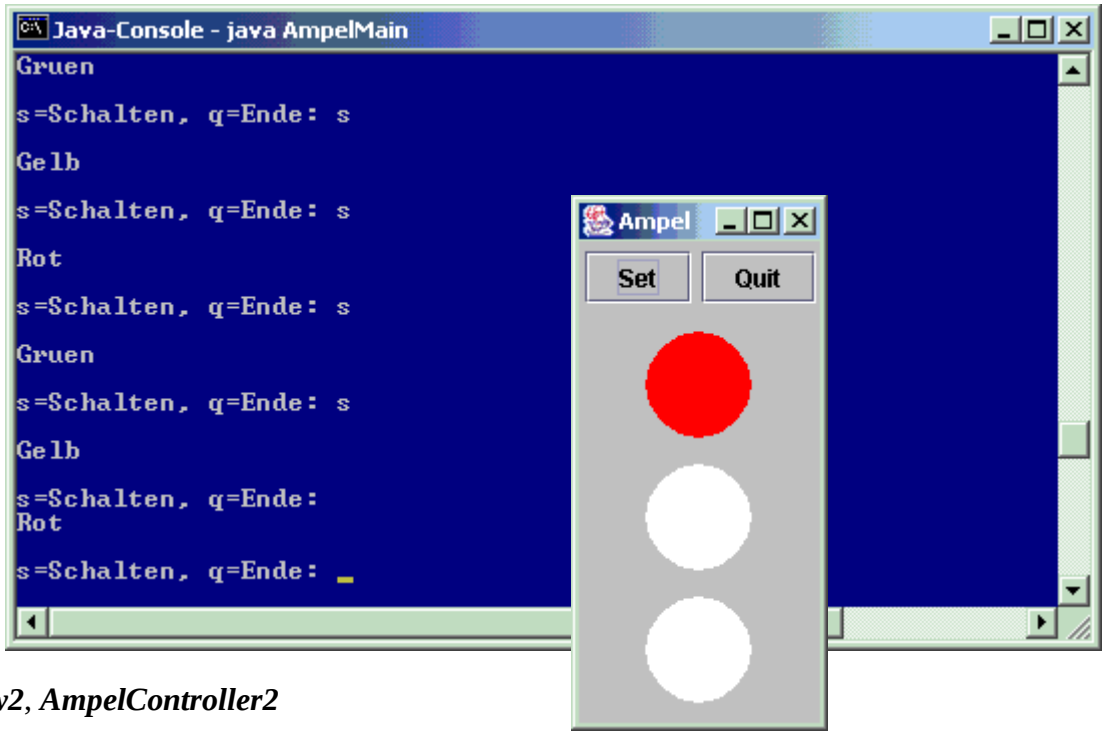


Im Hauptprogramm **`AmpelMain`** werden beide **`AmpelView/AmpelController`**-Versionen für das **`AmpelModel`** installiert.

AmpelMain.java

```
// AmpelMain.java                                    MM 2014
// MVC-Main

/**
 * Simulation einer Ampelschaltung,
 * Version mit zwei Views und zwei Controller.
 */
public class AmpelMain
{
/**
 * Starten eines Ampelprogramms mit MVC-Architektur,
 * Initialisiere ein Model mit zwei View.
 */
  public static void main( String args[])
  {
// Model
    AmpelModel trafficLight = new AmpelModel();

// Views
    AmpelView2 viewTrafficLight2 =
      new AmpelView2( trafficLight, "Ampel");

    AmpelView1 viewTrafficLight1 =
      new AmpelView1( trafficLight, "Ampel");
  }
}
```

AmpelView1, AmpelController1



AmpelView2, AmpelController2

Dokumentation, ***AmpelMVCTools.zip***

13.4 Zusammenfassung

Zusammenfassend kann man den Aufbau eines Programms durch die folgende Struktur beschreiben: