MVC – Model-View-Controller

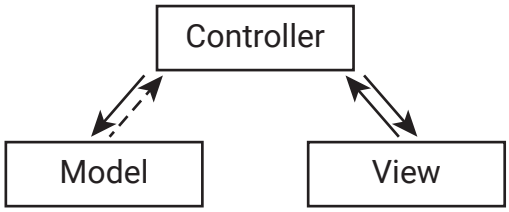
MVC ist eine Struktur genauer gesagt ein Entwurfsmuster, um Programme mit einer grafischen Benutzeroberfläche in möglichst getrennte Funktionsblöcke aufzuteilen. Das geschieht aus folgenden Gründen:

- Klassen können so besser wieder verwendet werden
- Man kann Klassen leichter gegen andere Klassen austauschen, um z.B. eine bessere GUI-Oberfläche für eine bestehende Funktion entwerfen zu können
- Die einzelnen Funktionsblöcke insbesondere das Model können unabhängig von einander getestet werden.
- Programme können leichter im Team entwickelt werden, da die einzelnen Funktionsblöcke klar getrennt sind

Um das zu erreichen, teilt man eine Aufgabe mit GUI in die folgenden Teilbereiche:

- Model: repräsentiert die Funktionalität, die Logik, den Zustand des Programms (d.h. die für die Funktionalität gespeicherten Werte)
- View: Stellt die im Model gespeicherten Werte dar, entspricht daher der Layout-Klasse bzw. der Frame-Klasse
- Controller: steuert den Ablauf des Programms, reagiert auf Benutzerereignisse, verbindet das Model mit der View.

Das Ziel von MVC ist es die einzelnen Bereiche möglichst unabhängig von einander zu gestalten. Es soll zum Beispiel möglich sein, statt einem Button einen Radiobutton zu verwenden ohne den Controller oder das Model ändern zu müssen. Oder eine andere Punkteberechnung zu verwenden, ohne Controller und View ändern zu müssen. Die Struktur sollte sich im Idealfall folgendermaßen darstellen:



Das bedeutet, dass keine Abhängigkeit zwischen Model und View herrscht. Der Controller (oder die Controller-Klassen) haben dabei eine zentrale Rolle. Sie rufen abwechselnd die Methoden aus den Model-Klassen und den View-Klassen auf und bekommen nur über Rückgabewerte die Ergebnisse gemeldet. Die View und der Controller hängen noch etwas stärker zusammen, da die Komponenten der View über die Listener ebenfalls die Methoden aus dem Controller aufrufen. Deshalb ist in der obigen Grafik die Beziehung von Model zum Controller schwächer (strichliert) dargestellt (es wird hier nur über Rückgabewerte gearbeitet) als zwischen View und Controller. Die stärkere Entkopplung von View und Controller erfordert leider Techniken, die im zweiten Jahrgang noch nicht Thema sind, deshalb ist hier auch nur eine rudimentäre Version von MVC vorgestellt.

Wie erreiche ich diese Struktur:

Model:

Model-Klassen enthalten alle Daten, die für den Ablauf wichtig sind und die Berechnungen dazu. Viele Klassen, die bisher geschrieben wurden sind typische Model Klassen (Musikalbum, Produkt,

Magisches Quadrat). Neben den Daten enthalten sie auch noch relevante Berechnungen und Abläufe (Suche, Gesamtlängenberechnung, Überprüfungen, ...) als Methoden. Diese Methoden arbeiten mit den Daten (Attributen).

View:

Die View-Klassen enthalten alle GUI-Komponenten, Grafiken und das Layout. Zusätzlich stellt die View-Klasse auch eigene Methoden zur Verfügung, damit der Controller Werte setzen kann (z.B. `setzeStatusMeldung(String meldung)`). Wie diese Meldung angezeigt wird (in einem Label, mit einer JOptionPane-Message, mit einer blinkenden Animation, ...) bleibt komplett der View überlassen. Daher dürfen diese Methoden **keine GUI-Komponente als Parameter und auch nicht als Rückgabetyp** haben. Für den Informationsaustausch zwischen Controller und View sind im Allgemeinen die folgenden Datentypen geeignet:

- Primitive Datentypen
- String
- Eigene Datenklassen
- Enums und Arrays auf Basis der obigen

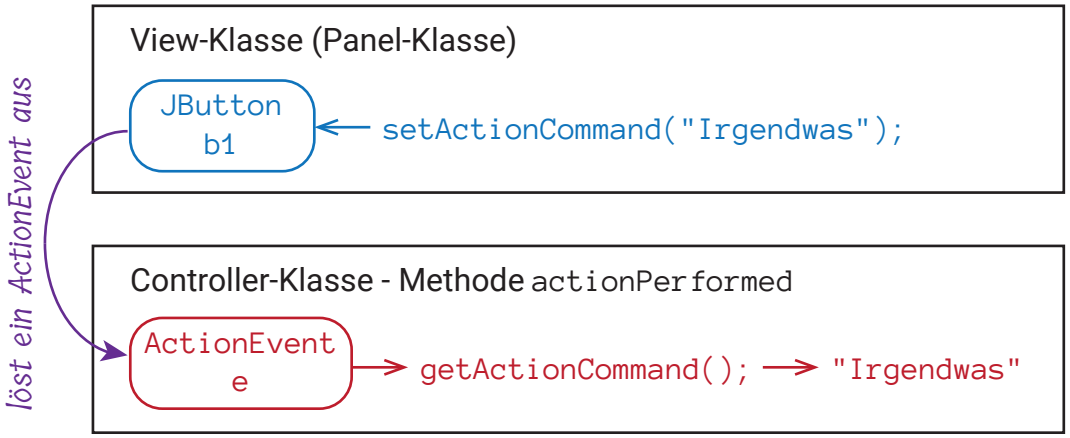
Controller

Der oder die Controller-Klassen implementieren die notwendigen Listener-Interfaces und enthalten die main-Methode. Nachdem der Controller sowohl Methoden der Model-Klassen als auch Methoden der View-Klassen verwendet muss er die Objekte dafür erzeugen und deren Referenzen speichern (als Attribtue), sonst kann er die Referenzen nicht für den Methodenaufruf verwenden.

Zusätzlich muss die Haupt-Controller-Klasse beim Erzeugen des Layout-Panel-Objekts diesem Panel auch noch mitteilen, dass er selbst für die Ereignisverarbeitung zuständig ist und sich selbst als Parameter "mitgeben" (`... = new LayoutPanel(this);`).

Um die View möglichst vom Controller zu entkoppeln ist es auch nicht sinnvoll, mit `getSource()` zu überprüfen, von welcher GUI-Komponente die Aktion ausgelöst wurde (sonst bräuchte man erst wieder die Referenzen auf die GUI-Komponenten im Controller). Bei Ereignissen, die vom `ActionListener`-Interface abgefangen werden, kann man dafür das Attribut `actionCommand` verwenden. Das `actionCommand` ist bei **`JButton`-Objekten standardmäßig die Beschriftung**, man kann es aber auch mit `setActionCommand(String command)` beim `JButton`-Objekt auf einen beliebigen Text setzen.

Um herauszufinden, welche GUI-Komponente die Aktion ausgelöst hat, kann man in der Controller-Klasse das `ActionEvent`-Objekt fragen, welches `actionCommand` mitgesendet wurde. Damit muss man dann nur einen String vergleichen und nicht eine GUI-Komponente mit Referenzvergleich.



Nun muss der Controller nur noch entscheiden, welche Methoden er im Model aufrufen muss, um auf das registrierte `ActionCommand` zu reagieren und welche Methoden dann als Reaktion auf das Ergebnis im Model in der View aufgerufen werden müssen, um das Ergebnis darzustellen.

Ein Beispiel mit Model-View-Control entwickeln

Die einzelnen Aspekte von Model-View-Control sollen noch einmal anhand eines Beispiels verdeutlicht werden.

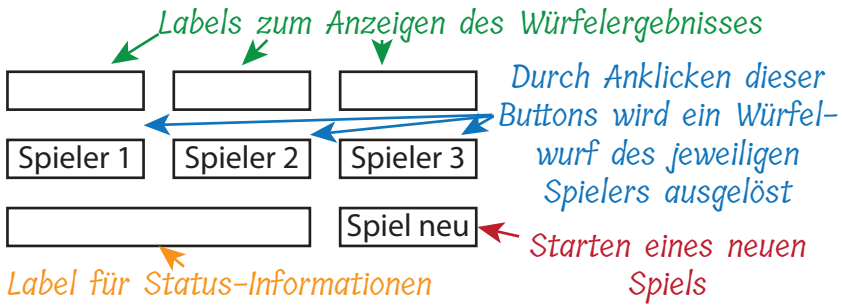
Beispiel:

Angabe: *Schreibe ein GUI-Programm, welches es ermöglicht, dass mehrere Spieler gegeneinander in einem einfachen Würfelspiel antreten. Der Spieler mit dem höchsten Würfelwurf gewinnt.*

Zunächst müssen die drei Bereiche identifiziert werden

View:

Für viele ist es am einfachsten, sich zunächst die Möglichkeiten der View vorzustellen. Eine Skizze kann dabei zusätzlich helfen. Das folgende Beispiel wurde für drei Spieler entworfen:



Neben der Frame-Klasse muss die Oberfläche also jedem Spieler die Möglichkeit bieten, einen Würfel-Wurf auszulösen kann. Das Ergebnis eines jeden Würfelwurfes muss irgendwo angezeigt werden und es muss ein Anzeigefeld für den Gewinner geben. Weiters muss es eine Möglichkeit geben ein neues Spiel zu initiieren und die Oberfläche zurück zu setzen. Nachdem die Steuerung dieser Funktionen aber nun von außen erfolgt, muss auch die View Methoden anbieten, mit denen diese Funktionen gesteuert werden können.

Im Detail könnte die View folgende Methoden benötigen:

- Eine Methode zum Setzen des Würfelergebnisses (kommt vom Controller, der diese Informationen aus dem Model erhält)
- Eine Methode zum Setzen der Status-Informationen (kommt vom Controller, der diese Informationen aus dem Model erhält oder auch selbst liefert)
- Eine Methode zum Anzeigen/Markieren des Spielers, der gewonnen hat.
- Eine Methode zum Zurücksetzen der Oberfläche für einen Neustart
- Ev. Eine Methode um einen Button zu disablen, damit ein Spieler in einer Runde nicht öfters würfeln kann.
- Ev. Eine Methode zum Abfragen, des Würfelwurfs eines bestimmten Spielers

Welche Methoden die View genau benötigt ergibt sich aus dem Zusammenspiel zwischen Model, Controller und View. Aber unabhängig von der genauen Ausprägung der Methoden dürfen diese Methoden **weder GUI-Komponenten als Parameter übernehmen noch GUI-Komponenten zurückgeben**.

Model

Ein Model darf nicht im Hinblick auf eine bestimmte GUI entworfen werden. Es dürfen daher auch hier keine GUI-Komponenten verwendet werden. Genauso wenig dürfen Ein- und Ausgabeanweisungen enthalten sein. Man kann sich vor Augen führen, dass ein Model genauso gut mit einer einfachen main-Methode und der Konsole als Ein- und Ausgabemedium funktionieren muss, wie im Zusammenspiel mit einer komplexen GUI. Ein Model kann und soll auch mit einer eigenen automatisierten Testklasse (main-Methode ohne Eingaben oder JUnit-Tests – siehe spätere Module) getestet werden. Für das obige Beispiel reicht eine recht einfache Klasse, die aber folgende Grundfunktionen (= Methoden) bieten muss:

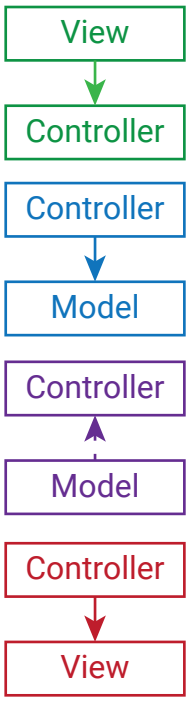
- Speichern und abfragen der Würfelergebnisse eines jeden Spielers
- Würfel-Funktion für jeden Spieler
- Ermitteln wer gewonnen hat
- Zurücksetzen des Spiels

Zusätzlich könnten noch Methoden zur Überprüfung, ob schon jeder Spieler an der Reihe war, bzw. zum Zusammenfassen des Ergebnisses als Text zur Verfügung gestellt werden.

Controller:

Der Controller muss die View mit dem Model verbinden. Darum erzeugt und speichert der Controller die View- und die Model-Objekte und startet das Programm. Außerdem implementiert er die Ereignissteuerung und damit die Listener-Interfaces, damit die Kontrolle bei jeder Benutzerakton auch wieder an ihn zurückfällt. In diesem Fall genügt das ActionListener-Interface.

Beim Ablauf im Controller werden Informationen zwischen View und Model ausgetauscht. Dabei ist der Controller derjenige, der diesen Austausch lenkt. Im folgenden ist ein Ablauf aus dem Würfelspielbeispiel bei Klicken eines "Spieler ..."-Buttons dargestellt (am besten mit dem entsprechenden Code im Vergleich gemeinsam anschauen):



Eine Komponente der View löst ein ActionEvent aus, dessen ActionCommand mit `Spieler` anfängt. Der Controller erkennt dies und holt sich aus diesem ActionCommand die Spielernummer heraus

Der Controller „beauftragt" das Model einen Würfelwurf für den Spieler mit der entsprechenden Spielernummer durchzuführen

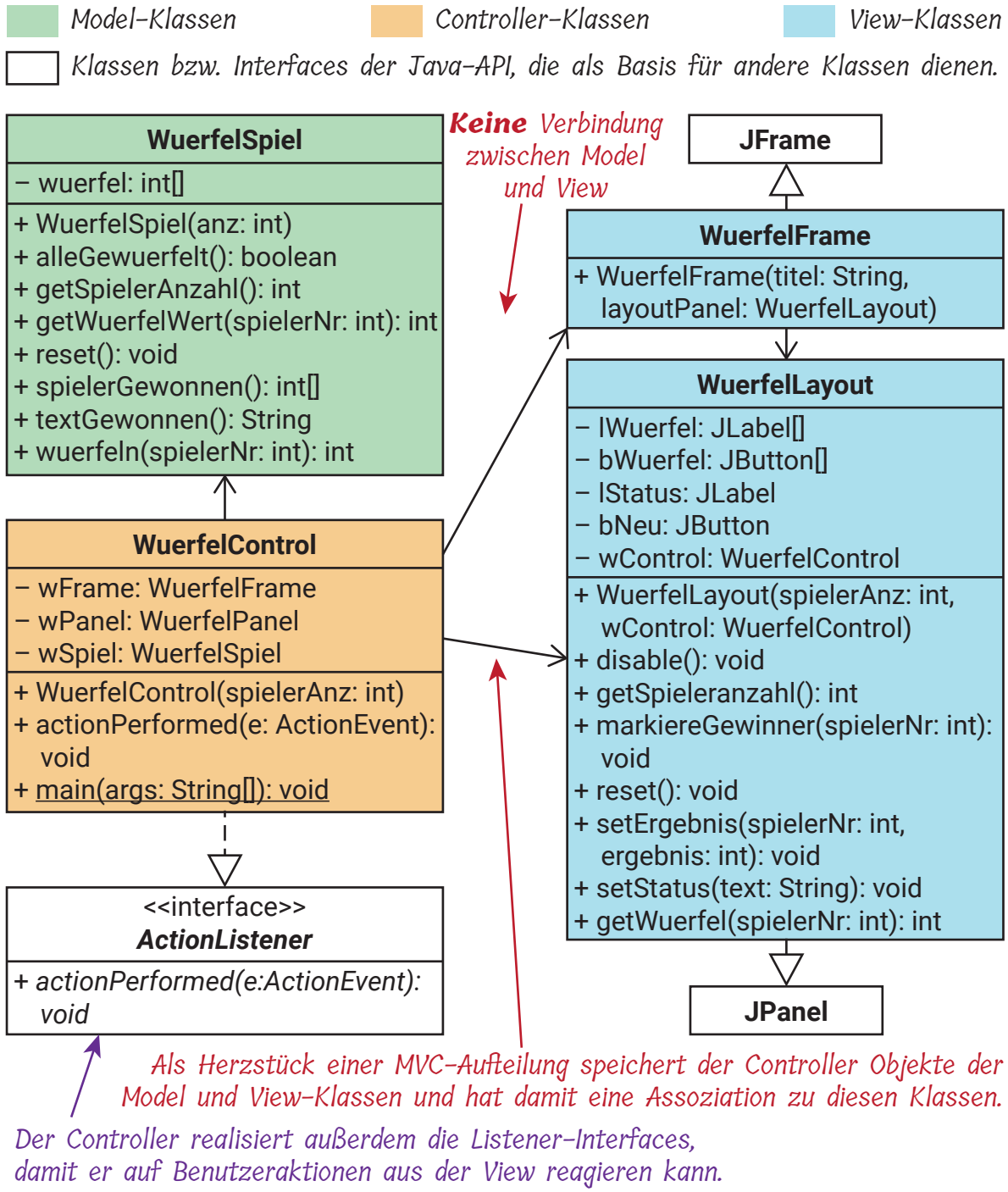
Das Model speichert den Würfelwurf in seinen Attributen und „meldet" das Ergebnis über einen Rückgabewert and den Controller

Wenn ein gültiges Ergebnis geliefert wurde, „beauftragt" der Controller die Layout-Klasse (View) das Ergebnis in irgendeiner Form anzuzeigen

Welche Methoden Model und View zur Erfüllung der Aufgaben zur Verfügung stellen muss, wird daher durch das bestimmt, was der Controller benötigt. Er ist das "Herzstück".

Implementierung des Beispiels:

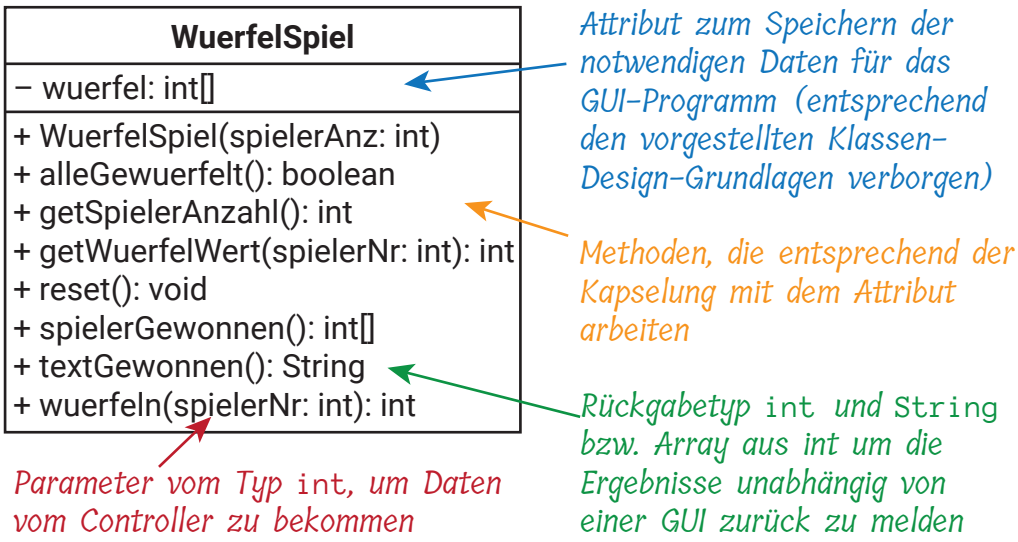
Entsprechend der vorangegangenen Überlegungen kann nun ein UML-Klassendiagramm der Aufteilung in die einzelnen Komponenten angegeben werden:



Nachdem in der Angabe keine genaue Spieleranzahl gegeben war, wurde das Spiel so gestaltet, dass die Anzahl der Mitspieler in gewissem Rahmen (2 – 9) über Konstruktorparameter konfigurierbar ist. Die tatsächliche Spieleranzahl wird beim Start des Spiels in der main-Methode abgefragt. Die Model- und View-Klassen bieten auch Methoden, die vom Controller in dieser Version nicht verwendet werden, aber vielleicht für zukünftige Updates sinnvoll sind.

Model:

Als ganz „normale“ Java-Klasse bietet das Model ein Array-Attribut, in dem für jeden Spieler ein Würfelergebnis gespeichert werden kann und Methoden, die mit diesen Werten arbeiten.



Das gesamte Beispiel wurde in einem Package namens `bsp.mvc` gespeichert. Um die Bereiche noch sichtbarer zu trennen und die Funktionsbereiche auch im Code sichtbar zu machen, wurde die Model-Klasse in einem eigenen Sub-Package (`bsp.mvc.`**`model`**) gespeichert.

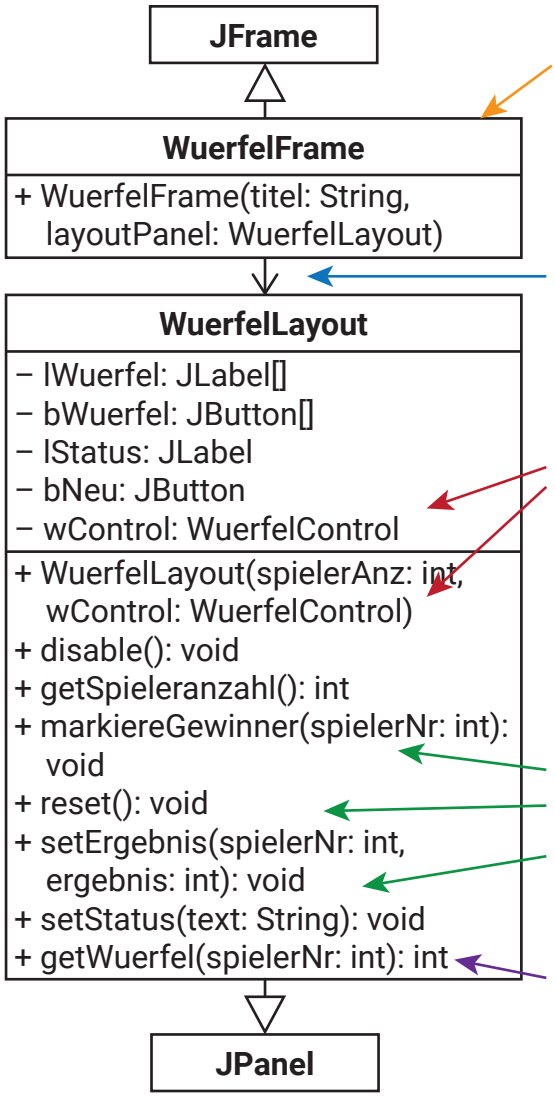
```
package bsp.mvc.model;
/**
 * Enthält die relevante Logik für ein einfaches Würfelspiel, bei dem beliebig
 * viele Spieler würfeln können. Derjenige bzw. diejenigen mit dem höchsten
 * Würfelwurf gewinnen, sofern schon ein Spieler gewürfelt hat.
 * @author Lisa Vittori
 * @version 2019-04-25
 */
public class WuerfelSpiel {
  private int[] wuerfel;

  /**
   * Erstellt ein Wuerfel-Spiel für eine beliebige Anzahl an Spielern
   * @param spielerAnz die Anzahl an Spielern, wenn eine Anzahl weniger als 2
   *                   übergeben wird, dann wird das Spiel für 2 Spieler erzeugt.
   */
  public WuerfelSpiel(int spielerAnz) {
    if (spielerAnz < 2) {
      spielerAnz = 2;
    }
    wuerfel = new int[spielerAnz];
    for (int i = 0; i < wuerfel.length; i++) {
      wuerfel[i] = -1;
    }
  }

  /**
   * Gibt zurück, ob schon alle Spieler gewürfelt haben.
   * @return true, wenn alle Spieler bereits einen Würfelwurf gemacht haben, im
   *         anderen Fall false.
   */
  public boolean alleGewuerfelt() {
    boolean alleGewuerfelt = true;
    for (int i = 0; i < wuerfel.length; i++) {
      if (wuerfel[i] < 0) {
        alleGewuerfelt = false;
      }
    }
    return alleGewuerfelt;
  }

  /**
   * Gibt die Anzahl der Spieler im Würfelspiel zurück
   * @return die Anzahl der Spieler
   */
  public int getSpielerAnzahl() {
    return wuerfel.length;
```

```
  }

  /**
   * Gibt den gewürfelten Wert des Spielers mit dem Index spielerNr zurück
   * @param spielerNr die Nummer des Spielers als Index (von 0 beginnend)
   * @return den gewürfelten Wert des Spieler mit dem Index spielerNr oder -1
   *         falls der Spieler noch nicht gewürfelt hat oder der Spieler nicht
   *         existiert
   */
  public int getWuerfelWert(int spielerNr) {
    if (spielerNr >= 0 && spielerNr < wuerfel.length) {
      return wuerfel[spielerNr];
    } else {
      return -1;
    }
  }

  /**
   * Setzt alle Spieler auf den Status "Noch nicht gewürfelt" zurück
   */
  public void reset() {
    for (int i = 0; i < wuerfel.length; i++) {
      wuerfel[i] = -1;
    }
  }

  /**
   * Gibt die Indizes jener Spieler zurück, die gewonnen haben.
   * @return die Indizes der Spieler mit dem höchsten Würfelergebnis oder null,
   *         wenn noch kein Spieler gewürfelt hat.
   */
  public int[] spielerGewonnen() {
    int max = -1;
    int anzahl = 0;
    int[] indizesGewonnen = null;
    for (int i = 0; i < wuerfel.length; i++) {
      if (wuerfel[i] > max) {
        max = wuerfel[i];
      }
    }
    if (max > -1) {
      // Zählen wieviele Maximal-Werte es gibt
      for (int i = 0; i < wuerfel.length; i++) {
        if (wuerfel[i] == max) {
          anzahl++;
        }
      }
      // Speichern der Spieler-Indizes
      indizesGewonnen = new int[anzahl];
      int indexErgebnis = 0;
      for (int i = 0; i < wuerfel.length; i++) {
        if (wuerfel[i] == max) {
          indizesGewonnen[indexErgebnis++] = i;
        }
      }
    }
    return indizesGewonnen;
  }

  /**
   * Gibt die Nummern der Spieler, die gewonnen haben, in natürlicher Nummerierung
   * (bei 1 beginnend) als Text mit und als Bindewort zurück
   * @return den Text mit den Nummern all jener Spieler, die gewonnen haben mit
   *         und als Bindewort.
   */
  public String textGewonnen() {
    String gewText = "";
    int[] gew = this.spielerGewonnen();
    if (gew != null) {
      gewText = "Spieler ";
      if (gew.length == 1) {
        gewText += (gew[0] + 1) + " gewinnt";
```

```
      } else {
        gewText += (gew[0] + 1);
        for (int i = 1; i < gew.length; i++) {
          gewText += " und " + (gew[i] + 1);
        }
        gewText += " gewinnen";
      }
    }
    return gewText;
  }

  /**
   * Erstellt einen Würfelwurf für den Spieler mit der angegebenen Spieler-Nummer
   * @param spielerNr die Nummer des Spielers als Index (von 0 beginnend)
   * @return -1, wenn der Spieler schon an der Reihe war und nicht würfeln darf
   *         bzw. wenn die SpielerNr nicht existiert, sonst den gewürfelten Wert
   */
  public int wuerfeln(int spielerNr) {
    int ergebnis = -1;
    if (spielerNr >= 0 && spielerNr < wuerfel.length && wuerfel[spielerNr] < 0) {
      ergebnis = (int) (Math.random() * 6 + 1);
      wuerfel[spielerNr] = ergebnis;
    }
    return ergebnis;
  }
}
```

View

Die View besteht in diesem Fall aus 2 Klassen: der Frame-Klasse und des Layout-Panels.



Eine einfache Frame-Klasse, die das Layout-Panel als Parameter übernimmt und daher ohne weitere Änderungen für ähnliche Projekte wieder verwendet werden kann.

Das Frame übernimmt und speichert intern eine Referenz auf ein Objekt der Layout-Klasse, deshalb besteht zwischen diesen beiden Klassen eine Assoziation.

Zusätzlich zu den GUI-Komponenten verwaltet das Layout auch eine ***Referenz auf das Objekt der Controller-Klasse****. Dies ist notwendig, damit der Controller für die Ereignissteuerung zu den GUI-Komponenten hinzugefügt werden kann. (Deshalb besteht auch eine Assoziation vom Layout zum Controller)*

Die View bietet für jede notwendige Änderung der grafischen Oberfläche eine eigene Methode, die über `int`*- oder* `String`*-Parameter die notwendigen Informationen bekommt.*

Um dem Controller Abfragen über Werte in der GUI zu ermöglichen, gibt es weitere Methoden, die diese Werte bereits in int umgewandelt an den Controller zurück geben.

Die Frame-Klasse ist Teil des Subpackages bsp.mvc.**view** und übernimmt das Layout-Panel und den Titel als Parameter und könnte daher ohne Änderung für weitere Projekte wieder verwendet werden.

```
package bsp.mvc.view;

import javax.swing.*;

/**
 * Stellt einen Rahmen für die Würfelspiel-GUI zur Verfügung
 * @author Lisa Vittori
 * @version 2019-04-25
 */
public class WuerfelFrame extends JFrame {
  /**
   * Initialisiert den Rahmen für das Würfel-Spiel im Zentrum des Bildschirms in
   * minimaler Größe
   * @param layoutPanel den Inhalt für das Fenster
   */
  public WuerfelFrame(String titel, JPanel layoutPanel) {
    super(titel);
    this.add(layoutPanel);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.pack();
    this.setLocationRelativeTo(null);
    this.setVisible(true);
  }
}
```

Das Layout-Panel (ebenfalls im Subpackage bsp.mvc.**view**) übernimmt ein Objekt der Controller-Klasse als Parameter im Konstruktor. Da sich die Klasse WuerfelControl in einem anderen Package befindet, muss diese Klasse Dieses Objekt wird dann für die Ereignissteuerung verwendet. Außerdem muss die Layout-Klasse Methoden zur Verfügung stellen, mit denen die Anzeige geändert werden kann, damit der Controller diese Arbeit erledigen kann.

Beim Design der Methoden für die View-Klasse ist es wichtig die Grenzen zwischen den Verantwortlichkeiten des Controllers und der View zu beachten. Das bedeutet, dass der Controller mit Hilfe der Methoden eine Änderung der View anstoßen kann, aber nicht das genaue Verhalten der View bestimmt. Die Methode `markiereGewinner` verdeutlicht dieses Prinzip:

- Der **Controller** teilt der View über die Methode mit, **was für eine Änderung** erforderlich ist (in diesem Fall welcher Spieler als Gewinner markiert werden soll).
- **Wie** diese Markierung aussieht (rote Farbe oder Animation oder eigenes `JOptionPane`-Fenster oder …) **bestimmt die View** in der Methode selbst.

Dadurch bleibt die Verantwortlichkeit der Darstellung in der GUI-Klasse (wo sie auch hingehört).

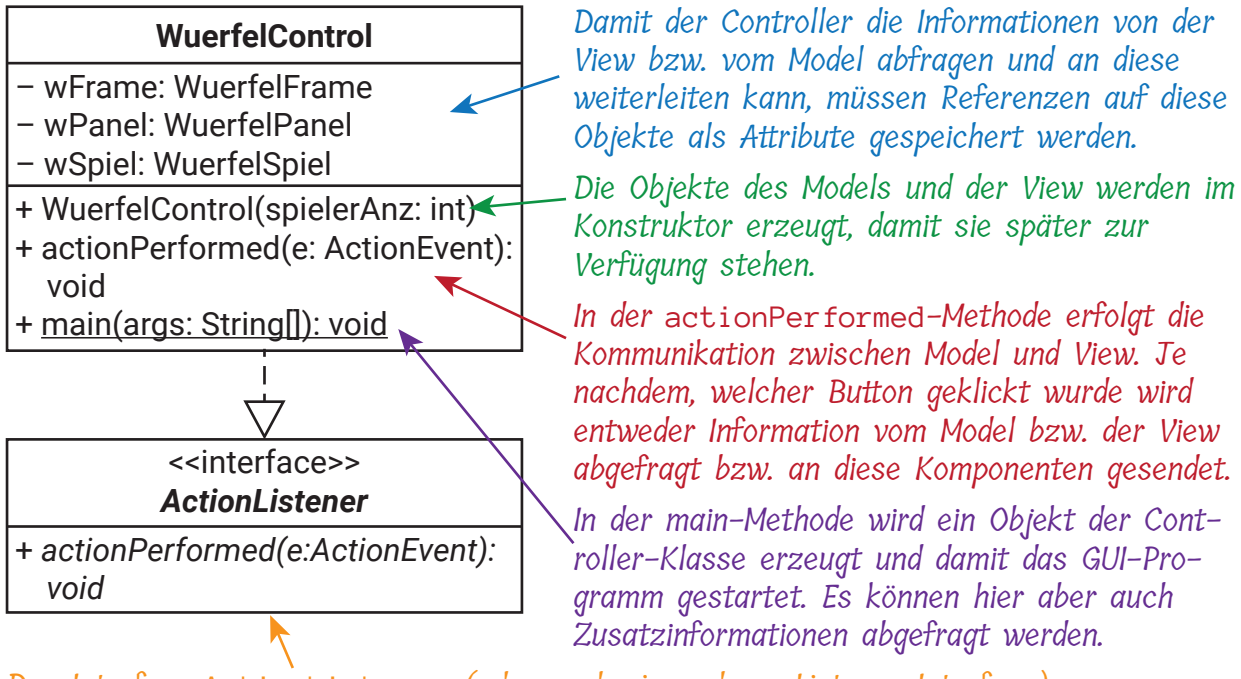
```
package bsp.mvc.view;

import javax.swing.*;
import java.awt.*;
import bsp.mvc.control.WuerfelControl;

/**
 * Erstellt ein Layout für das Würfelspiel
 * @author Lisa Vittori
 * @version 2019-04-25
 */
public class WuerfelPanel extends JPanel {
  private JLabel[] lWuerfel;
  private JButton[] bWuerfel;
  private JLabel lStatus;
  private JButton bNeu;
  private WuerfelControl wControl;

  /**
   * Übernimmt den Controller und initialisiert das Layout
```

```
   * @param spielerAnz die Anzahl der Spieler
   * @param wControl   der Controller für diese GUI
   */
  public WuerfelPanel(int spielerAnz, WuerfelControl wControl) {
    this.wControl = wControl;
    this.setLayout(new BorderLayout());

    JPanel main = new JPanel(new GridLayout(2, spielerAnz));
    lWuerfel = new JLabel[spielerAnz];
    for (int i = 0; i < lWuerfel.length; i++) {
      lWuerfel[i] = new JLabel("", SwingConstants.CENTER);
      lWuerfel[i].setOpaque(true);
      lWuerfel[i].setBackground(Color.WHITE);
      main.add(lWuerfel[i]);
    }

    bWuerfel = new JButton[spielerAnz];
    for (int i = 0; i < bWuerfel.length; i++) {
      bWuerfel[i] = new JButton("Spieler " + (i + 1));
      main.add(bWuerfel[i]);
      bWuerfel[i].addActionListener(this.wControl);
    }
    this.add(main);

    JPanel unten = new JPanel(new FlowLayout());
    lStatus = new JLabel("Zum Würfeln entspr. Button klicken!");
    unten.add(lStatus);

    bNeu = new JButton("Neues Spiel");
    bNeu.setActionCommand("neu");
    bNeu.addActionListener(this.wControl);
    unten.add(bNeu);

    this.add(unten, BorderLayout.PAGE_END);
  }

  /**
   * Deaktiviert den Button für den Spieler mit dem entsprechenden Index
   * @param spielerNr der Index des Spieler, dessen Button deaktiviert werden muss
   */
  public void disable(int spielerNr) {
    if (spielerNr >= 0 && spielerNr < lWuerfel.length) {
      bWuerfel[spielerNr].setEnabled(false);
    }
  }

  /**
   * Markiert den betreffenden Spieler als Gewinner
   * @param spielerNr der Index des Spieler, der als Gewinner markiert wird
   */
  public void markiereGewinner(int spielerNr) {
    if (spielerNr >= 0 && spielerNr < lWuerfel.length) {
      lWuerfel[spielerNr].setBackground(Color.RED);
    }
  }

  /**
   * Aktiviert alle Buttons und löscht die Labels für das WürfelErgebnis
   */
  public void reset() {
    for (int i = 0; i < lWuerfel.length; i++) {
      lWuerfel[i].setText("");
      lWuerfel[i].setBackground(Color.WHITE);
      bWuerfel[i].setEnabled(true);
    }
    lStatus.setText("Zum Würfeln entspr. Button klicken!");
  }

  /**
   * Setzt das Ergebnis für einen bestimmten Spieler im entsprechenden Label
   * @param spielerNr der Index des betreffenden Spielers
   * @param ergebnis  das Würfelergebnis, das im Label angezeigt werden soll
```

```
   */
  public void setErgebnis(int spielerNr, int ergebnis) {
    if (spielerNr >= 0 && spielerNr < lWuerfel.length) {
      lWuerfel[spielerNr].setText(String.valueOf(ergebnis));
    }
    this.disable(spielerNr);
  }

  /**
   * Setzt den Text für das Status-Label
   * @param text der neue Text für das Satus-Label
   */
  public void setStatus(String text) {
    lStatus.setText(text);
  }

  /**
   * Eine Methode zum Abfragen des Würfelergebnisses eines Spielers
   * @param spielerNr der Index des Spielers, dessen Ergebnis abgefragt wird
   * @return der im Label angezeigte Würfelwurf des Spielers
   */
  public int getWuerfelwurf(int spielerNr) {
    int wuerfel = -1;
    if (spielerNr >= 0 && spielerNr < lWuerfel.length) {
      wuerfel = Integer.parseInt(lWuerfel[spielerNr].getText());
    }
    return wuerfel;
  }
}
```

Controller

Der Controller verbindet nun das Model mit der View und steuert den Ablauf. Daher muss er Referenzen auf diese Klassen speichern, damit er Zugriff auf sie hat. In unserem Fall werden die Objekte auch gleich im Controller erzeugt. Der Ablauf des Programms wird Großteils über die Ereignissteuerung abgewickelt und damit in diesem Fall in der `actionPerformed`-Methode der Ereignissteuerung.



Wenn die Klassen der View und des Models in anderen Packages gespeichert sind, müssen diese Klassen in einem ersten Schritt importiert werden:

```
import java.awt.event.*;
import javax.swing.JOptionPane;

import bsp.mvc.model.WuerfelSpiel;
import bsp.mvc.view.WuerfelFrame;
import bsp.mvc.view.WuerfelPanel;

/**
 * Übernimmt die Steuerung der Applikation und stellt die Verbindung zwischen
 * View und Model her.
 *
 * @author Lisa Vittori
 * @version 06.05.2014
 */
public class WuerfelControl implements ActionListener {
  private WuerfelFrame wFrame;
  private WuerfelPanel wPanel;
  private WuerfelSpiel wSpiel;

  /**
   * Initialisiert die View-Elemente und das Model
   * @param spielerAnz die Anzahl der Spieler, für die das Spiel
   * erzeugt wird.
   */
  public WuerfelControl(int spielerAnz) {
    wPanel = new WuerfelPanel(spielerAnz, this);
    wFrame = new WuerfelFrame("Würfelspiel", wPanel);
    wSpiel = new WuerfelSpiel(spielerAnz);
  }

  /**
   * Die Steuerungsmethode, die beim Button-Klick ausgelöst wird.
   * @param e das Ereignis, das den Aufruf der Methode ausgelöst hat
   */
  @Override
  public void actionPerformed(ActionEvent e) {
    String ac = e.getActionCommand();
    if (ac.equals("neu")) {
      wPanel.reset();
      wSpiel.reset();
    } else if (ac.startsWith("Spieler")) {
      String nummer = ac.substring(ac.length() - 1);
      int spielerNummer = Integer.parseInt(nummer) - 1;
      int erg = wSpiel.wuerfeln(spielerNummer);
      if (erg > -1) {
        wPanel.setErgebnis(spielerNummer, erg);
        if (wSpiel.alleGewuerfelt()) {
          wPanel.setStatus(wSpiel.textGewonnen());
          int[] ergInd = wSpiel.spielerGewonnen();
          for (int i = 0; ergInd != null && i < ergInd.length; i++) {
            wPanel.markiereGewinner(ergInd[i]);
          }
        }
      }
    }
  }

  /**
   * Diese Methode bildet den Start-Punkt der Applikation und erfragt
   * zunächst die Anzahl der Mitspieler. Auf Basis dieser Information
   * wird die GUI gebildet.
   * @param args nicht verwendet
   */
  public static void main(String[] args) {
    try {
      int anzahl = Integer.parseInt(JOptionPane.showInputDialog(
          "Wieviele Spieler sollen mitspielen? (2-9)"));
      if (anzahl >= 2 && anzahl <= 9) {
        new WuerfelControl(anzahl); // Starten der GUI mit entspr. Anzahl
```

```
      } else {
        JOptionPane.showMessageDialog(null, "Für die Eingabe sind nur " +
            "ganze Zahlen von 2 - 9 erlaubt!", "Fehler",
            JOptionPane.WARNING_MESSAGE);
      }
    } catch (NumberFormatException ex) {
      JOptionPane.showMessageDialog(null, "Für die Eingabe sind nur " +
          "ganze Zahlen von 2 - 9 erlaubt!\n" + ex.getMessage(), "Fehler",
          JOptionPane.WARNING_MESSAGE);
    }
  }
}
```