Was versteht man unter *Model View Controller* (MVC)?

Das Model-View-Controller Entwurfsmuster (MVC Design Pattern) ist eines der gebräuchlichsten Muster zur Strukturierung von Software. Es ermöglicht eine weitgehende Trennung von Daten-Modell und dessen graphischer Repräsentation.

Eine klare, übersichtliche Strukturierung, die damit verbundene Erleichterung von Wartungsarbeiten und die Wiederverwendbarkeit von Programmteilen gehören mit zu den obersten Zielen eines gelungenen Programmentwurfes. Das MVC-Muster ermöglicht dies dadurch, dass die architektonischen Hauptteile einer Software voneinander weitgehend separat gehalten und somit voneinander unabhängig bearbeitet und gewartet werden können. Drei Teile sind zu unterscheiden:

Das Modell (engl. model)
: Es enthält die Arbeitsdaten eines Programms, z.B. Nutzereingaben, aus Datenbanken gelesene Daten, etc. Es enthält niemals Referenzen auf die beiden anderen Teile.

Die Präsentation (engl. view)
: Die (bei einem Desktop-Programm z.B. graphische) Darstellung von Daten, die Programmoberfläche (GUI), etc. Sie enthält im Quelltext weder Daten, noch Teile der Geschäfts- oder Programmlogik vor, sondern ist lediglich für deren Darstellung verantwortlich. Hierzu definiert sie entsprechende Schnittstellen.

Die Steuerung (engl. controller)
: Die vermittelnde Schicht, die für die Interaktion zwischen Präsentationsschicht und Daten zuständig ist. Sie wird oft mittels eines Observer-Patterns realisiert, das die Daten des Modells beobachtet und diese ggf. mit der Präsentationsschicht wechselseitig aktualisiert. Die Steuerung muss somit sowohl zu den Daten, als auch zu Teilen der Darstellungsschicht Zugang haben.

Das Beispiel demonstriert den grundsätzlichen Aufbau eines MVC-Musters, bleibt dabei aber der Übersichtlichkeit halber sehr einfach (z.B. ohne Observer-Pattern). Innerhalb der einzelnen Quelltextabschnitte wurde aus dem gleichen Grund zudem auf import-Anweisungen verzichtet. Der Quelltext kann in seiner Gesamtheit weiter unten bezogen werden.

Das Modell

Das Modell wurde extremst einfach gehalten. Die Klasse definiert lediglich eine `String`-Instanzvariable, deren Wert durch eine Getter-Methode zu beziehen ist.

```
class MVCModel {
    private String text = "Hallo Welt!";
    public String getText() {
        return text;
    }
}
```

Die Präsentation

Sie stellt im gezeigten Beispiel eine Klasse, die von `JFrame` abgeleitet wurde mit einem `JLabel` und einem `JButton` dar. Die Initialisierung der gesamten graphischen Oberfläche findet in der Methode `init()` statt, die im Konstruktor aufgerufen wird. Nach Klick auf den Button wird der Inhalt der `String`-Variablen aus dem Modell auf dem Label dargestellt. Hierzu wird der Button beim *Controller* angemeldet. Er implementiert in diesem Fall `ActionListener` und bekommt die aktuelle Instanz des View übergeben.

```
class MVCView extends JFrame {

    JLabel label;

    public MVCView() {
        init();
    }

    private void init() {
        label = new JLabel(" ");
        label.setHorizontalAlignment(JLabel.CENTER);
        JButton button = new JButton("klick");
        button.addActionListener(new MVCController(this));
        this.add(label, BorderLayout.NORTH);
        this.add(button, BorderLayout.CENTER);
        this.pack();
        this.setTitle("MVC");
        this.setLocationRelativeTo(null);
        this.setDefaultCloseOperation(EXIT_ON_CLOSE);
        this.setVisible(true);
    }

    public void setText(String s) {
        label.setText(s);
    }
}
```

Die Steuerung

Die Klasse implementiert `ActionListener` und übernimmt die Ereignis-Behandlung des Buttons. Der Konstruktor bekommt beim Aufruf das `MVCView`-Objekt der Präsentationsschicht übergeben und erzeugt eine neue Instanz des Modells.
In der die Aktion realisierenden Methode `actionPerformed()` wird dann der Text des Modells über den Getter des Modells bezogen und durch eine Accessor-Methode der View auf das Label gesetzt.

```
class MVCController implements ActionListener {

    private MVCView view;
```

```
    private MVCModel model;

    public MVCController(MVCView view) {
        this.view = view;
        this.model = new MVCModel();
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        view.setText(model.getText());
    }
}
```

Wollte man zukünftig das Programm durch weitere Darstellungselemente erweitern, so müssen lediglich die Präsentations- und die Steuerungsklasse angepasst werden. Das Modell kann unberücksichtigt bleiben. Umgekehrt ist bei einer Änderung der Daten kein Eingriff in Steuerung und Darstellung notwendig.
Selbstverständlich muss hierbei berücksichtigt werden, dass in konkreten Realisierungen zusätzlicher Aufwand bei der Implementierung notwendig wird, etwa zur Gewährleistung der Typsicherheit des Modells, etc.

```
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class MVC {

    public static void main(String[] args) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new MVCView();
            }
        });
    }
}

class MVCView extends JFrame {

    JLabel label;

    public MVCView() {
        init();
    }

    private void init() {
```

```
        label = new JLabel(" ");
        label.setHorizontalAlignment(JLabel.CENTER);
        JButton button = new JButton("klick");
        button.addActionListener(new MVCController(this));
        this.add(label, BorderLayout.NORTH);
        this.add(button, BorderLayout.CENTER);
        this.pack();
        this.setTitle("MVC");
        this.setLocationRelativeTo(null);
        this.setDefaultCloseOperation(EXIT_ON_CLOSE);
        this.setVisible(true);
    }

    public void setText(String s) {
        label.setText(s);
    }
}

class MVCController implements ActionListener {

    private MVCView view;
    private MVCModel model;

    public MVCController(MVCView view) {
        this.view = view;
        this.model = new MVCModel();
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        view.setText(model.getText());
    }
}

class MVCModel {
    private String text = "Hallo Welt!";

    public String getText() {
        return text;
    }
}
```