0.1 Entwurfsmuster

Quelle: lehrer-online

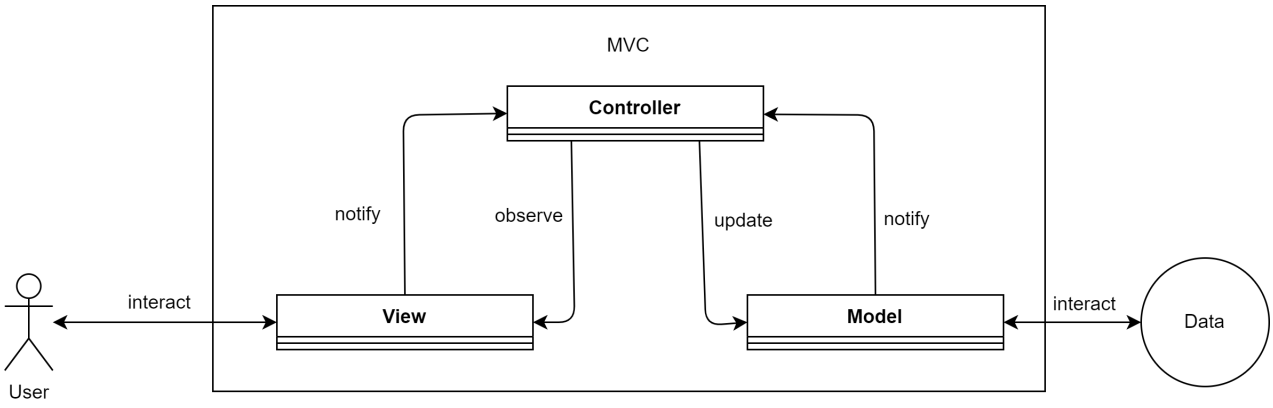
Wäre dies ein Skript zu einem anderen Fach, so wäre die erste Aufgabe die Analyse des obigen Comics. Glücklicherweise ist das in Informatik nicht notwendig - die Kernaussage ist klar. Zwar ist es durchaus nützlich für das Verständnis, Dinge selbst zu programmieren, aber Zeit ist eine begrenzte Ressource. Deswegen ist es unerlässlich, dass auf die Arbeit von von anderen Personen zurückgegriffen wird. Möchte man fremden Code nicht direkt kopieren - oder kann das nicht, weil er doch nicht zu 100 Prozent auf den eigenen Fall zutrifft - so kann man sich zumindest an der Art und Weise der Problemlösung orientieren.
Da es Problemklassen gibt, die häufig ein ähnliches Vorgehen erfordern, wurden bereits Lösungsschablonen entwickelt, die im Allgemeinen **Entwurfsmuster** heißen. Das bedeutet, dass ein grobes Klassendiagramm der Struktur bereits vorhanden ist, eine konkrete Ausgestaltung ist aber dem Programmiery überlassen.
Wir haben bereits mit dem **Kompositum** ein Entwurfsmuster kennengelernt, auch dort haben wir nicht direkt „**Das Kompositum**“ implementiert, sondern die grundlegende Idee, die dahinter steckt, verwendet.
Es gibt noch unzählige weitere Entwurfsmuster, hier eine kleine Auswahl, jeweils mit kurzer Erläuterung:

- **Singleton**: stellt sicher, dass von der betrachteten Klasse nur ein einziges Objekt erzeugt werden kann und bietet eine globale Zugriffsmöglichkeit auf dieses Objekt. Z.B. braucht ein Login-System nur eine Instanz (dieses soll alle Logins verwalten!).
- **Factory**: Abstrahiert die Erstellung von Objekten, indem sie eine gemeinsame Schnittstelle für die Erstellung von Objekten definiert und die spezifische Omplementierung der Objekterstellung in weitere Unterklassen auslagert. Dieses Muster kann z.B. bei der Erstellung von Spielfiguren o.Ä. nützlich sein.
- **Observer**: stellt eine Möglichkeit bereit, um auf Änderungen eines Objekts zu reagieren, indem andere Objekte (die Observer) benachrichtigt werden. Ein klassischer Anwendungsfall wäre ein Benachrichtigungsticker (siehe auch Benachrichtigungen am Handy!).
- **Decorator**: ermöglicht die dynamische Erweiterung von Objekten durch Hinzufügen von zusätzlichen Funktionen oder Eigenschaften (die Dekorationen). Ein Bild kann z.B. mit Filtern „dekoriert“ werden.
- **Adapter**: ermöglicht die Integration von inkompatiblen Klassen durch die Entwicklung einer gemeinsamen Schnittstelle (sehr allgemeines Konzept!).
- **Iterator**: es wird eine Möglichkeit bereitgestellt, sequentiell auf Elemente in einer Zusammenstellung zuzugreifen, z.B. in einer Liste.

Das Muster, das für unser Projekt zentral sein soll, ist noch etwas allgemeiner als die Meisten der oben bereits erwähnten - **Model View Controller**. Es kann sich nicht nur auf einzelne Klassen beziehen (auch wenn das durchaus möglich ist), sondern beschreibt die grundlegende Implementierungsidee einer ganzen **Softwarearchitektur**.
Bevor wir uns Beispielen zuwenden ein kurzer Überblick über die drei Komponenten:

1. **Modell (Model)**: Das Modell beschreibt die konkreten Daten, die mit dieser Anwendung verknüpft sind. Die Daten können dabei in vielerlei Form vorliegen, von definierten Attributen in einfachen Klassen über Datenformate wie **JSON** bis hin zu ganzen Datenbanken. Hier kann gegebenenfalls auch noch Logik implementiert sein, wenn die Daten, z.B. bei Speicherung, noch verarbeitet werden müssen.
2. **Ansicht (View)**: Die Ansicht ist dazu da, die Daten des Modells in einer Form darzustellen, sodass der Anwender mit ihnen interagieren kann. Letztendlich läuft es im modernen Kontext in der Regel auf ein **Graphical User Interface (GUI)** hinaus. GUIs verwenden häufig das Entwurfsmuster Kompositum, da sie aus Komponenten mit ähnlichen Grundfunktionen (z.B. Menüeintrag, klickbares Objekt, Datendarstellung etc.) zusammengesetzt werden.
3. **Steuerung (Controller)**: Die Steuerung verwaltet die anderen beiden Komponenten. In der Regel wird das Entwurfsmuster „Observer“ verwendet, um auf Interaktion des Anwenders mit der Ansicht zu „**lauschen**“. Anschließend veranlasst die Steuerung alle notwendigen Änderungen, so werden beispielsweise neue Menüeinträge angezeigt, oder neue Daten werden von einem Eingabefeld in der

In einem sehr vereinfachten Schema mit jeweils einer Klasse dargestellt:



Ein mögliches Grundgerüst der Implementierung dieses Diagramms sieht so aus:

```
//Unser Model entspricht hier einfach nur einer Datenklasse, die Zugriff und
//Änderungen regelt.
public class Model {
    private String data;

    public String getData() {
        return data;
    }

    public void setData(String data) {
        this.data = data;
    }
}

public class Controller {
    //Der Controller hat Referenzen auf ein Model und einen View
    private Model model;
    private View view;

    //Der Controller könnte Model und View im Konstruktur auch direkt
    //erzeugen - man könnte hier auch das Singleton-Pattern verwenden,
    //wenn man sichergehen möchte, dass in einer Instanz des Programms
    //nur ein Controller existieren kann!
    public Controller(Model model, View view) {
        this.model = model;
```

```
        this.view = view;
    }

    //Um Daten zu modifizieren (z.B. durch eine Eingabe im View getriggert)
    //mus der Controller auch auf die Methoden des Models zugreifen:
    public void setData(String data) {
        model.setData(data);
    }

    public String getData() {
        return model.getData();
    }

    //Der View wird mit dieser Methode aktualisiert, d.h. hier:
    //Die aktuellen Daten werden auf der Konsole ausgegeben.
    public void updateView() {
        view.printData(model.getData());
    }
}

//Der View ist hier am einfachsten:
//Er gibt einfach nur Daten auf der Konsole aus.
public class View {
    public void printData(String data) {
        System.out.println("Data: " + data);
    }
}
```

Diese Implementierung ist noch recht abstrakt, geben wir den Klassen ein klein wenig mehr Leben:

```
//Das Model hat sich nicht wesentlich verändert, es soll jetzt aber
//einen Spielstand modellieren statt einfach nur eines Textes
public class Model {
    private int score;

    public int getScore() {
        return score;
    }

    public void increaseScore() {
        score += 10;
    }
}

import java.util.Random;

public class Controller {
    private Model model;
    private View view;
    //Um Zufallszahlen zu erzeugen
    private Random random;

    public Controller(Model model, View view) {
        this.model = model;
        this.view = view;
        random = new Random();
```

```
    }

    public void performAction() {
        //In diesem kleinen Beispiel wird eine Zufallszahl erzeugt, ist
        //diese gerade, so haben wir "gewonnen" und der Score wird erzeugt
        int randomValue = random.nextInt(10) + 1;
        if (randomValue % 2 == 0) {
            model.increaseScore();
            //außerdem wird angezeigt, dass wir Erfolg hatten!
            view.printSuccessMessage();
        } else {
            //andernfalls wird die Versagens-Meldung angezeigt - Schade :)
            view.printFailureMessage();
        }
        //In jedem Fall wird der derzeitige Spielstand ausgegeben.
        view.printScore(model.getScore());
    }
}

//Der View interagiert weiterhin nur mit der Konsole:
public class View {
    public void printSuccessMessage() {
        System.out.println("Great job! You earned 10 points.");
    }

    public void printFailureMessage() {
        System.out.println("Sorry, try again.");
    }

    public void printScore(int score) {
        System.out.println("Your current score is: " + score);
    }
}
```

Ausgehend von diesen sehr grundlegenden Beispielen können jetzt die einzelnen Klassen sukzessive ausgebaut werden - es muss auch nicht zwingend bei diesen drei Klassen bleiben! Z.B. könnten die Daten in mehreren Klassen organisiert sein. Wollen wir beispielsweise verschiedene Karten modellieren, könnte eine Oberklasse Karte verwendet werden und für die einzelnen speziellen Kartenarten jeweils eine Unterkarte, also z.B. wie bei Uno:

1. Zieh Vier Farbwahl
2. Aussetzen
3. Richtungswechsel
4. Zieh 2
5. Zahlenkarte

Alle erben von Karte, da sie z.B. alle ausgepielt werden können, die Eigenschaften unterscheiden sich aber und sie können spezielle Effekte haben.
Anmerkung: Weitere Hinweise zur weiteren Ausdifferenzierung bzw. Ausgestaltung der einzelnen Komponenten finden sich im nächsten Kapitel.