ENTWICKLUNGSPORTFOLIO TEILKOMPONENT CLIENT

MODUL 326 OBJEKTORIENTIERT ENTWERFEN UND IMPLEMENTIEREN

LUKAS NYDEGGER INF13F
VERSION 1.0
16.11.2016

Entwurfprinzip 1: MVC

Problemstellung

Wir mussten uns für ein Architektur Muster unserer JavaFX Applikation entscheiden.

Auswahl des passenden Entwurfsprinzips / -musters.

Da wir den Client entwickeln, haben wir uns für die JavaFX Variante entschieden. Für beide ist es das erste Mal, dass wir mit JavaFX arbeiten. Deshalb haben wir uns eingelesen um einen möglichst guten Aufbau unserer Applikation sicher zu stellen.

Als Muster haben wir uns aus folgenden Gründen für die MVC Architektur entschieden:

- JavaFX sieht vor, dass es in verschiedene Views unterteilt wird, zwischen denen man wechseln kann.
- Für jede View kann auf einen Controller verwiesen werden um Aktionlistener zu implementieren und Benutzereingaben zu verarbeiten.

Diese wichtigen Eigenschaften von JavaFX lassen sich wunderbar mit dem MVC Muster umsetzen.

Anwendung auf das ursprüngliche Problem

Durch die angewandte MVC Architektur lässt sich unsere Applikation in verschiedene Views mit entsprechenden Controllern umsetzten. Das vereinfacht die Umsetzung des ganzen Clients erheblich, denn alle Teile lassen sich unabhängig voneinander entwickeln. Jede Klasse hat somit seine eigene Aufgabe.

Lernprozess

Das Wählen einer geeigneten Architektur seiner Applikation sollte immer als erstes geschehen. Die Umsetzung wird erheblich einfacher. Anpassungen wie auch Wartungen solcher MVC Architekturen sind einfacher, da sich die Klassen unabhängig voneinander austauschen lassen.

Für uns als Team ist es von Nutzen wenn alle nach demselben Muster programmieren. Der Austausch und die Lesbarkeit des Codes ist einfacher.

Entwurfprinzip 2: Single Responsibility

Problemstellung

Bei der nicht Anwendung des Single Responsibility Prinzips kann es sein, dass in unserer Applikation eine Redundanz der Funktionalität zweier Klassen entstehen kann. Dadurch wird die Lesbarkeit nicht gewährleistet.

Auswahl des passenden Entwurfsprinzips / -musters.

Das passende Entwurfsprinzip ist das Single Responsibility Prinzip, es sagt vor, dass eine Klasse nur einen Zweck hat. Die Funktionen in der Klasse sollten auch nur diesem Zweck dienen. So kann keine Funktionalität einer Klasse doppelt oder auf mehrere verteilt werden.

Anwendung auf das ursprüngliche Problem

Bei unserer Applikation haben wir das Single Responsibility Prinzip angewendet und beim Erstellen einer Klasse genau überlegt, welchem Zweck es dienen soll und was für Funktionen sie haben soll.
Falls der Zweck schon zu einer bestehenden Klasse gehört hat, haben wir die Funktionen in diese implementiert oder die Klasse gegebenenfalls unterteilt.

Lernprozess

Das Angewendete Entwurfsprinzip finde ich persönlich sehr sinnvoll.
Es lässt sich leicht umsetzten und dadurch ist die Motivation, seine Klassen dementsprechend zu gestalten und allenfalls die Architektur nochmal zu überdenken, gross.

Was mir am meisten aufgefallen ist, ist dass die Lesbarkeit des Codes meines Partners viel einfacher wurde. Man weiss, was die Klasse macht, meist schon an dem Namen und deshalb weiss man auch wo man ansetzten muss.

Bei dem Entwurfsprinzip finde ich es wichtig, dass man seine Klassen demnach passend benennt. Dabei ist es auch wichtig die gleichen Begriffe zu benutzen und sich nicht abzuwechseln.

Entwurfprinzip 3: Singleton

Problemstellung

Unser Problem besteht darin, dass wir aus verschiedenen Teilen unserer Applikation auf das Spielfeld und deren Komponente zugreifen müssen. In dem Spielfeld befinden sich alle möglichen Daten zu dem aktuellen Spielstand, diese müssen wir weitergeben können und dabei sicher gehen, dass sie immer auf dem neusten Stand sind.

Auswahl des passenden Entwurfsprinzips / -musters.

Bei dieser Art von Problem bietet sich das Singleton Entwurfs Muster an, welches wir im Unterricht behandelt haben. Ein Singleton stellt sicher, dass nur eine Instanz von einer Klasse existiert und diese meistens Global verfügbar ist.

Diese Eigenschaften des Entwurfsmusters bieten sich bei unserem Problem als gute Lösung an.

Anwendung auf das ursprüngliche Problem

Der Singleton war aus meiner Sicht sehr einfach anzuwenden. Wir haben eine Klasse erstellt - „FieldService“ – welche alle Konfigurationen beinhaltet: Die Spieler, Die Steuerung, Das Spielfeld und deren Komponenten. Wenn nun eine andere Klasse auf die Spielkonfigurationen oder das Spielfeld zugreifen will, ist nur ein Aufruf der getInstance Methode der Klasse notwendig. Die Instanz wird schon am Anfang des Programmes erstellt und es wird immer die gleiche Instanz aufgerufen.

```
public class FieldService {

    private static  FieldService fieldService = new FieldService();

    private Maze maze;
    private Gear gear;
    private GameMode gameMode;
    private List<Player> players = new ArrayList<>();
    private List<Bomb> bombs = new ArrayList<>();
    private String playerName;

    private FieldService() {
        ArrayList<Key> keys = new ArrayList<>();
        keys.add( new Key(KeyCode.W, PlayerFunctions.UP));
        keys.add( new Key(KeyCode.S, PlayerFunctions.DOWN));
        keys.add( new Key(KeyCode.D, PlayerFunctions.RIGHT));
        keys.add( new Key(KeyCode.A, PlayerFunctions.LEFT));
        keys.add( new Key(KeyCode.SHIFT, PlayerFunctions.DROPBOMB));


        this.gear = new Gear(true, keys);

        this.gameMode = new GameMode(60000, 3);
    }
//Getter und Setter
…
    public static FieldService getInstance(){
        return fieldService;
    }

}
```

LERNPROZESS

Das Muster hat uns sehr viel Arbeit erspart. Dass wir nun jederzeit im Programmablauf auf die Spielressourcen zugreifen können und dabei sicher sein können die aktuellsten Werte zu haben, erleichtert uns die Architektur des Programmes sehr.
Das Singleton Muster war notwendig, da das Spielfeld und die Spieler dauernd abgefragt und weiterverarbeitet werden müssen.

Entwurfprinzip 4: Open Closed

Problemstellung

Ein Spiel sollte immer erweiterbar sein. Ein zweites Problem ist auch, dass wir nur einen Teil Komponenten realisieren und die Erweiterbarkeit möglichst keine Änderungen der Hauptteile unserer Applikation beinhalten sollte.

Auswahl des passenden Entwurfsprinzips / -musters.

Für dieses Problem gibt es kein Entwurfsmuster. Die Lösung unseres Problems ist ein Entwurfsprinzip namens Open Closed. Das Open Closed Prinzip sagt folgendes aus:
„Module sollten sowohl offen (für Erweiterungen) als auch verschlossen (für Modifikationen) sein.“

Konkret in unsere Umsetzung integriert, sollte dieses Prinzip dafür sorgen, dass wir möglichst offen für neue Spielideen oder Änderungen der anderen Teil-Komponenten sein sollten, ohne die bestehenden Codes zu ändern oder gar zu löschen.

Anwendung auf das ursprüngliche Problem

Die Umsetzung auf dieses Problem war sehr aufwendig und beinhaltete viele andere Entwurfs Muster und Entwurfsprinzipien.

Das oben erwähnte Prinzip MVC ist eines davon. Andere Lösungen die wir implementiert haben ist der Singleton und das Factory Pattern.
Dieses Pattern haben wir zum Beispiel bei der Steuerung umgesetzt. Dadurch sind in diesem Bereich unseres Programmes nur minimale Änderungen notwendig.

```
public enum PlayerFunctions implements PlayerFunctionsImp {
       UP {
           @Override
           public void action() {
                  //Spieler hoch bewegen
               }
           }
       }
       ...
}

public interface PlayerFunctionsImp{
        public void action();
}
```

Diese einzelnen Anwendungen der Entwurfsprinzipien oder Muster sollten sicherstellen, dass das Open Closed Prinzip erfüllt wird. Hinzu kommt noch, dass wir für zukünftige Implementationen mit diesem Hintergrundgedanken arbeiten und für Probleme ein passendes Entwurfsprinzip oder Muster finden.

LERNPROZESS

Für mich ist dieses Entwurfsprinzip eine Überordnung aller Entwurfsprinzipien- oder Muster, denn diese sind für die Umsetzung des Open Closed unerlässlich. Sie sind die Lösung der Probleme die notwendig sind, damit der Code veränderbar und geschlossen bleibt.