ENTWICKLUNGSPORTFOLIO TEILKOMPONENT CLIENT

MODUL 326 OBJEKTORIENTIERT ENTWERFEN UND IMPLEMENTIEREN

TOBIAS LÜSCHERINF7F
VERSION 1.0
16.11.2016

ENTWURFPRINZIP 1 / SINGLE RESPONSIBILITY

PROBLEMSTELLUNG

Das es Klassen gibt welche mehr als nur eine Aufgabe haben und diese auch ausführen. somit hat man das Problem, dass es oft mehrere Gründe gibt eine Klasse anzupassen.

AUSWAHL DER PASSENDEN ENTWURFSPRINZIPS / -MUSTERS.

Wir suchten nach einer Lösung die Klassen zu kapseln aber trotzdem noch OO zu programmieren. Wir suchten in der Liste der Entwurfsprinzipe nach einem ähnlichen fall und fanden die Single Responsibility. Dieses Muster gibt vor das jede Klasse nur eine Aufgabe hat und keine Klasse doppelt vorkommt.

ANWENDUNG AUF DAS URSPRÜNGLICHE PROBLEM

Wir haben die Klassennamen geändert und deren Inhalte gekürzt so dass sie nur noch dies tun was ihnen als Klassenname vorgibt, z.B. haben wir den GameManager welcher nur die verschiedenen GameElemente managet. Dieser führt z.B. den GameCreater aus welcher das Spielfeld und die Listener dafür erstellt. So hat jede Klasse nur noch eine Aufgabe welche aber perfekt ausgeführt wird. Zudem haben wir geschaut das bei neu erstellten Klassen Single Responsibility ebenfalls eingehalten wird.

LERNPROZESS

Wir gingen wie schon gesagt sehr Systematisch vor. Wir schauten uns jede Klasse an und überdachten deren Aufgabe. So erstellten wir neue Klassen oder strukturierten sie um. Dabei gab es Klassen welche genau eine Aufgabe hatten, diese aber geordnet ausführten. Wir teilten bei diesem Prozess die Arbeit auf. Der Weg war sehr einfach und verständlich, da jeder bereits für einen Teil zuständig war, überarbeitete er auch seinen Teil. Verständlicher Weise gab es bei diesem Factoring neue Überschneidungen welche wir zusammen angeschaut und umgesetzt haben.

Ich sehe den Sinn hinter Single Responsibility sehr gut und ich denke wir werden es auch weiterhin anwenden. Ich finde es eine sehr gute Lösung da man ab sofort schon ohne dass man den ganzen Code liest die Aufgabe sieht und diese dann in einem neuen Code teil gleich benutzen kann. Ebenfalls sind lange Dokumentation so nicht mehr wirklich nötig.

ENTWURFPRINZIP 2/ MVC

PROBLEMSTELLUNG

Wir hatten alle Models Views und Controller wild vermischt und mussten somit immer viel an verschiedene Klassen anpassen.

AUSWAHL DER PASSENDEN ENTWURFSPRINZIPS / -MUSTERS.

Wir kannten MVC schon aus dem Betrieb und für uns war ohne lang zu suchen gleich klar, dass wir dieses Entwurfsprinzip benutzen werden.

Zudem passte es zu dem benutzten Fronten JavaFX welches selber schon auf MVC setzt.

ANWENDUNG AUF DAS URSPRÜNGLICHE PROBLEM

Wir haben 3 Ordner erstellt welche Model, View und Controller heissen. Wir haben danach die Klassen überarbeitet und in eines dieser 3 Typen eingefügt. So gab es aus vielen Klassen plötzlich 3 oder 4 und nicht mehr nur eine.

LERNPROZESS

Auch hier haben wir die Arbeit logischer weise aufgeteilt. Jeder hat seinen Teil im vorherein schon umgesetzt also ist er ebenfalls für das Factoring zuständig. So habe ich meine Teile überarbeitet und Lukas seine. Dabei haben wir immer möglichst schnell kommuniziert was der andere erstellt da wir ja z.B. Models mehrfach in verschiedenen Klassen hatten und wir die Arbeit nicht doppelt umsetzen wollten. Das Praktische dabei war das wir in der Schule immer nahe bei einander arbeiteten und so bei Fragen direkt fragen konnten. Ich denke das Prinzip ist sehr gut und wir werden es weiterhin nutzen.

Der Mehrwert für das Team ist extrem gross.

SINGELTON PATTERN

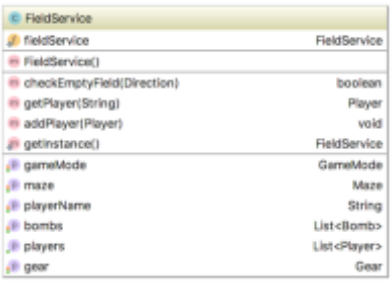
PROBLEMSTELLUNG

Wir hatten das Problem das wir das Labyrinth sowie auch die Spieler und Objekte irgendwie zwischenspeichern mussten. Deshalb haben wir eine Klasse erstellt welche all dies beinhaltet. Nun kam aber das 2te Problem, wie stellen wir sicher das die Daten auch wirklich gespeichert werden. Sobald man eine neue Instanz der Klasse erstellt hat diese keine Daten mehr oder nicht die wir wollen.

AUSWAHL DER PASSENDEN ENTWURFSPRINZIPS / -MUSTERS.

Wir überlegten uns wie das ganze ablaufen wird. Wie Java das Erstellen der Klassen handelt und wann er welche Instanz erstellt oder zerstört. Wir wussten das die Daten einer Klasse nur in einer Instanz enthalten sind und nicht in einer 2ten. Deshalb dachten wir zuerst an daran eine Instanz bei der Initialisierung zu erstellen und immer weiter zu geben. Dies wäre aber in gewisser Weise nicht sehr schön gelöst und könnte sicher anders gelöst werden. Deshalb haben wir das Singelton Pattern gewählt, da diese immer nur eine Instanz einer Klasse erlaubt und wir so immer die Instanz der Klasse haben welche alle Daten beinhaltet.

ANWENDUNG AUF DAS URSPRÜNGLICHE PROBLEM



Wir haben die Klasse FieldService erstellt welche alle Attribute hat welche wir während dem Spiel gespeichert haben müssen.

Sie hat statische Variablen und ein paar Methoden für das Auslesen von Player, hinzufügen von Player, Überprüfung der Felder und das bekommen der Instanz. Nun wird dieser Singelton immer wieder abgerufen z.B. sobald ein neuer Spieler Joined, wird dieser dem Singelton hinzugefügt.

Somit haben wir nun eine Instanz welche wir immer wieder abrufen können und müssen nicht immer die gleiche Instanz als Parameter weitergeben.

LERNPROZESS

Wir haben zuerst analysiert welche Daten wir zu jederzeit an mehreren Orten brauchen. Bei uns wären dies des Players, die Bomben, der Name unseres Spielers und unser Maze. Diese haben wir dann als Attribute in der Klasse Fieldservice definiert. Die Klasse Fieldservice haben wir dann als Singelton definiert, sie wird das aller erste Mal beim starten des Spiels aufgerufen. Dabei wird auch gleich das Maze und der eigene Spieler hinzugefügt. Bei empfangen Veränderung wie z.B. playerMoved, wird der Player ausgelesen und angepasst.

Was haben ich gelernt:
Ich habe sehr viel über das Verarbeiten von Klassen und den Garbitch Collecter gelernt. Ich weiss nun auch wie ich und wann ich Singelton brauchen sollte und wann nicht. Ich kenne nun auch die Vor- und Nachteile des Singelton.

Woran merkten wir das wir auf dem richtigen Weg sind:
Wir merkten das wir auf dem richtigen Weg sind als wir sahen das wir nun egal wo wir uns befinden genau das abrufen können was wir brauchen und somit auch immer gleich alles bearbeiten können. In der Zukunft würde ich den Singelton immer wieder für Speicherung von zwischenständen während der Laufzeit brauchen.

ENTWURFPRINZIP 3/ OPEN-CLOSED PRINZIP

PROBLEMSTELLUNG

Wir hatten das Problem das wir für jede Message welche der Server oder das Protokoll hinzugefügt hat einen sehr grossen Mehraufwand hatten. Wir mussten jede Message abfangen und dann anhand des Inhaltes erkennen um was für eine Klasse es sich handelt. Dies war sehr aufwändig da wir nicht immer wussten welche Klasse was beinhaltet.

AUSWAHL DER PASSENDEN ENTWURFSPRINZIPS / -MUSTERS.

Wir suchten nach einer Möglichkeit möglichst ohne grossen Aufwand die Klassen zu erkennen und somit offen für Erweiterungen zu sein. Dabei stossen wir auf das Open-Closed Prinzip. Dies gibt vor das man möglichst wenig ändern muss um eine Erweiterung hinzuzufügen. Aber man sollte nichts Modifizieren müssen. Als Idee wird bei diesem Prinzip oft die Vererbung erwähnt.

ANWENDUNG AUF DAS URSPRÜNGLICHE PROBLEM

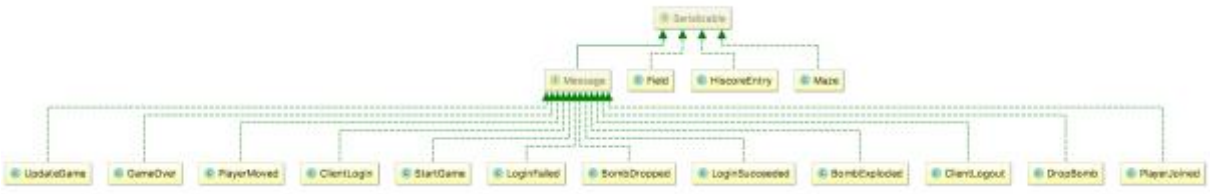
Die Klassen welche wir als Message bekommen erben alle von der Oberklasse Message. Dies spielt uns natürlich in die Hand. Wir haben deshalb einen Message Handler erstellt welcher die Message in eine Art Factory sendet, wo die Klasse der Nachricht ermittelt wird und je nach Art der Klasse wird eine andere Funktion aufgerufen. Hier ist ein Teil der Factory dargestellt.

```
public void executeMessageMethod(Message message){
    if(message instanceof UpdateGame){
        GameCreater creater = new GameCreater();
        creater.createMaze((UpdateGame) message);

    } else if(message instanceof StartGame){
        GameCreater creater = new GameCreater();
        creater.createMaze((StartGame) message);

    } else if(message instanceof PlayerJoined){
        PlayerJoined playerJoined = (PlayerJoined)message;
        if(FieldService.getInstance().getPlayers().isEmpty()){
            FieldService.getInstance().setPlayerName(playerJoined.getPlayerName());
        }
        Player player = new Player(playerJoined.getPlayerName(), playerJoined.getPositionX(), playerJoined.getPositionY());
        FieldService.getInstance().addPlayer(player);
        displayAllElementsService.displayAll();
```

Das vereinfachte Klassendiagramm der Messages sieht man hier.



LERNPROZESS

Wir haben zuerst die Klassen der Messages analysiert und danach einen Message Handler erstellt welcher die Messages zu einer MessageFactory weiterleitet. Diese führt dann anhand der Instance der Klasse eine andere Funktion aus. Dabei haben wir in meinem Sinn das Open-Closed Prinzip sehr gut umgesetzt. Falls nun eine weitere Message hinzugefügt wird, kann man ganz einfach nur in der Factory eine weitere Überprüfung hinzufügen und schon ist alles erledigt.

Was habe ich gelernt:
Ich weiss nun wann und wie ich eine Factory einsetzten kann. Ich kenne nun das Open-Closed Prinzip und werde, wenn möglich alle meine Applikationen danach gestalten.

Woran merkten wir das wir auf dem richtigen Weg sind:
Als wir immer weniger Aufwand für das Abfangen und bearbeiten der Messages hatten.