Entwicklungsportfolio

TEILKOMPONENTE CLIENT

Sven Ziörjen

MODUL 326 – OBJEKTORIENTIERT ENTWERFEN UND IMPLEMENTIEREN
INFW2017A
VERSION 2.0
03.12.2018

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Tagesjournals

1.1 26.22.2018

1.1.1 Ablauf

Tätigkeiten	Dauer (h)
Verbinden mit dem IET-Github und klonen des Projektes	0.5
Schnittstellen-Absprache mit den Server-Teams	0.5
Erstellen der Grundstruktur des Clients	1

1.1.2 Tätigkeiten

Zu Beginn haben ich und Florian, die beide in der Gruppe 2 für die Client-Anwendung eingetragen sind, uns mit dem Gitlab verbunden und das Projekt geklont. Der Test, ob wir pushen konnten, schlug fehl. Daraufhin haben wir das Projekt in ein privates Github Repository von uns selber eingefügt und haben von da an dort Programmiert.

Als nächster schritt stand das Absprechend der Schnittstellen mit den Server-Teams bevor. Nach bereits ziemlich kurzer Zeit waren wir uns einig und konnten uns dann an die Umsetzung der Projektstruktur setzen.

Bei der Umsetzung der Projektstruktur gingen wir nach dem MVC-Prinzip vor. Ich habe ein wenig die Führung übernommen und habe die ersten paar Klassen und Packages für mich und Florian erzeugt. Dann haben wir die Arbeit aufgeteilt, er hat sich um das GUI und ich habe mich um die Controller-Klassen gekümmert. Bis zum Ende der Schule hatten wir beide den Grossteil der Grobstruktur erstellt. Ausserdem wurde auch bereits ein statisches Fenster mit der Login-Sektion und der Konsolen-Sektion erstellt, wenn der Client gestartet wurde.

1.1.3 Reflexion

Bei dem Klonen des Projektes hatten wir keine Probleme, dies war schnell erledigt. Der merkwürdige Fehler mit dem Pushen war uns unerklärlich, doch um keine Zeit zu verlieren, haben wir einfach schnell ein eigenes privates erstellt. Später können wir unseren Code, wenn das Problem behoben ist, dann auf das GitLab Repository pushen.

Die Gespräche mit den Personen der Server-Komponenten wurden durch die bereits vorprogrammierten Schnittstellen (beispielsweise die Message-Klassen) stark vereinfacht. Da sogar kommentiert ist, welche Werte einzufüllen sind, mussten wir uns lediglich darauf einigen, strickt nach dem bereits angelegten Code zu arbeiten.

Die Umsetzung ging ziemlich schnell voran, da ich mir für das Klassendiagramm bereits im Voraus viele Gedanken dazu gemacht habe. Ich hatte somit bereits heute genügend Vorwissen, um wirklich produktiv arbeiten zu können. Es war praktisch, dass Florian mich bei Fragen konsultieren konnte.

Um mir das Vorwissen anzueignen haben mir die Videos von Scheidegger viel geholfen. Ich hatte zuvor keine Ahnung, was effektiv von meiner Gruppe verlangt wurde und wie es umgesetzt werden sollte, doch mit seiner Hilfe habe ich mir nun einen klaren Überblick über das Projekt verschaffen können.

Mit dem Projekt steht es momentan gut. Wenn wir die nächsten Male nicht auf grössere Probleme stossen, könnte die Clientapplikation rechtzeitig fertig werden.

1.2 03.12.2018

1.2.1 Ablauf

Tätigkeiten	Dauer (h)
Dem Florian den geschriebenen Code erklären.	0.2

1.2.2 Tätigkeiten

Heute hatten wir beinahe beide Lektionen lang Vorträge des letzten Moduls. Daher blieb kaum noch Zeit, wirklich etwas an dem Code zu schreiben. Ich habe Florian daher angeboten den Code, den ich über das Wochenende geschrieben hatte, zu erklären. Dies taten wir dann auch ungefähr 10 Minuten lang, bis wir die Stunde verliessen.

1.2.3 Reflexion

Zu der heute erledigten Arbeit gibt es nicht besonders viel zu sagen. Ich hatte am Wochenende bereits ziemlich viel gearbeitet, weswegen Florian den Code nun nicht mehr komplett verstand. Ich habe ihn über die erstellte Projektstruktur und die ersten Controller-Klassen unterrichtet, damit er zuhause auch weiterarbeiten kann.

Bei dem Arbeiten zuhause ist bei mir am Wochenende noch ein sehr mühsamer Fehler aufgetreten. Ich konnte Änderungen in der View nicht darstellen. Das Problem waren fehlende repaint() und revalidate() Aufrufe, welche mir etwas Zeit gekostet haben.

1.3 10.12.2018

1.3.1 Ablauf

Tätigkeiten	Dauer (h)
Alle Controller-Klassen erzeugen.	0.5
Ermöglichen ein Labyrinth zu empfangen und anzeigen.	1.5

1.3.2 Tätigkeiten

Heute Morgen haben ich und Florian zuerst alle Control-Klassen, die wir voraussichtlich brauchen werden, geschrieben. Parallel dazu haben wir auch den Dispatcher und die Control-Factory angepasst. Dafür haben wir ungefähr eine halbe Stunde aufgewendet. Nach dieser halben Stunde haben wir die Arbeit möglichst so aufgeteilt, dass es keine grossen Merge-Konflikte geben sollte. Ich habe mich um das Anzeigen eines Labyrinths gekümmert und Florian hat bei der Login-Funktion gearbeitet. Ich habe bis zum Ende der Stunde damit verbracht, er wurde aber früher fertig und hat noch versucht das Bewegen einer Figur zu implementieren.

1.3.3 Reflexion

Wir sind heute wirklich gut vorwärtsgekommen. Es sollten mehr oder weniger alle Klassen für unsere Komponente erstellt sein, somit lässt es sich nun einfacher arbeiten.

Bei dem Laden des Labyrinths gab es ziemlich viel zu schreiben, doch ich denke nach wenigen Änderungen, die ich noch in der Selbststudiums-Zeit vornehmen werde, sollte es schlussendlich funktionieren. Ich habe mich für ein GridLayout in einem JPanel entschieden, auch wenn ich mir immer noch nicht sicher bin, ob das wirklich der einfachste Weg ist, um dies umzusetzen. Eine Schwierigkeit war es, die Daten aus dem GridLayout abrufen zu können, also beispielsweise ob sich ein Spieler auf dieser Zelle befindet. Dies war nötig um Spieler aus einer Zelle löschen und in eine neue einfügen zu können. Schlussendlich habe ich es so gelöst, dass ich jedes JPanel im Grid auch noch in ein separates zweidimensionales Array gefügt habe, damit über welches ich diese auch im weiteren Spielverlauf bearbeiten konnte.

Alles in allem ist es ein guter Tag gewesen, doch ich werde bis nächste Woche noch zuhause arbeiten müssen, damit wir ungefähr im Zeitplan bleiben.

1.4 17.12.2018

1.4.1 Ablauf

Tätigkeiten	Dauer (h)
Legen einer Bombe implementieren.	0.5
Detonieren einer Bombe implementieren.	0.4
Aktualisieren des Labyrinths implementieren.	0.4
Error-Messages implementieren.	0.7

1.4.2 Tätigkeiten

Heute Morgen waren ich und Florian beide sehr produktiv. Wir haben beide Vollgas gegeben und schlussendlich sind wir tatsächlich beinahe fertig geworden. Zuhause haben wir in den vergangenen Tagen auch ziemlich viel gemacht, da wir nicht noch mehr über die Ferien zu erledigen haben wollten. Heute habe ich noch die Funktionen des Bombe-legens, des Detonierens einer Bombe, des Aktualisierens des Labyrinths und des Abbruchs bei Error-Messages implementiert.

1.4.3 Reflexion

Wir sind heute wirklich gut vorwärtsgekommen. Es ist sehr erleichternd gewesen, dass wir kaum Probleme gehabt haben und daher sehr viel in sehr kurzer Zeit erreicht haben. Zusammen mit der Arbeit, die wir zuhause erledigt haben, sind wir nun tatsächlich kurz vor der Vollendung unserer Komponente.

Ich hatte vor allem Probleme mit dem Aktualisieren der View nach dem Laden des Labyrinths bei einer Update-Message. Schlussendlich habe ich nach einer Suche im Internet gelernt, dass ich nicht die gesamte View neu erstellen muss, sondern besser die Objekte zwischenspeichere und bei der Update-Message dann anpasse.

Für die Error-Messages haben wir uns auch schon mit den Server-Leuten ausgetauscht, welche Error-Messages was genau bedeuten. Ich hoffe einfach, dass sie dies nicht vergessen und dass sie nicht nochmals im Nachhinein angepasst werden müssen.

Das nächste Mal müssen wir einfach den Code zusammen Mergen. Damit sollten wir fertig sein.

1.5 28.12.2018

1.5.1 Ablauf

Tätigkeiten	Dauer (h)
Merge-Konflikte beheben	1.5
Erstellen des Klassendiagramms	1.5
Erstellen des Sequenzdiagramms	1

1.5.2 Tätigkeiten

Heute haben ich und Florian geskyped, um das Programm komplett fertigzustellen. Wir haben direkt damit begonnen beide zu mergen und die Merge-Konflikte zusammen zu besprechen. Dabei hat jeweils nur er die Änderungen vorgenommen. Als wir dann nach ziemlich langer Zeit endlich fertig waren, hat er seinen Stand gepushed und ich habe ihn gepullt.

Anschliessend habe ich das Klassendiagramm erstellt. Dabei habe ich die Klassen aus dem Code heraus generieren lassen. Jedoch habe ich vergeblich versucht, auch vernünftige Assoziationen zu erzeugen, schlussendlich habe ich alle Verbindungen selber erstellt.

Zum Schluss erstellte ich auch noch das Sequenzdiagramm von der Aktion, wenn eine Bombe gelegt wird.

1.5.3 Reflexion

Es ist mühsamer gewesen, als wir gedacht hätten, die Mergekonflikte zu beheben und den Client zum Laufen zu bringen. Wir hätten besser öfter commited und kleine Konflikte behoben, anstatt zum Ende die riesige Menge an Merge-Konflikten klären zu müssen. Doch ich bin froh jetzt bereits fertig zu sein und mit einem guten Gefühl in den ersten Schultag von 2019 starten zu können.

Das Klassendiagramm ist auch ein wenig eine Herausforderung gewesen. Ich habe mehrmals wieder Optimierungen vornehmen müssen, weswegen es recht lange gedauert hat, bis ich mit dem Ergebnis endlich zufrieden war. Doch ich kann die Klassendiagramme nun deutlich schneller und besser zeichnen, dies habe ich heute gelernt.

Das Sequenzdiagramm ist keine grosse Herausforderung gewesen. Das, was nicht ganz klar gewesen ist, hatte bereits Florian abgeklärt.

Ich bin froh, fertig zu sein. In der Schule werden wir unseren Code lediglich noch auf das GitLab-Verzeichnis pushen müssen.

2 Entwurfsprinzipien / -muster

2.1 Entwurfsprinzip / -muster MVC

2.1.1 Problemstellung (zu allgemein, projektspezifischer)

Es kommt immer wieder vor, dass ein Entwickler in dem Code eines anderen Entwicklers verstehen muss. Meistens entweder um diesen zu erklären, um allfällige Fehler zu beheben oder um ihn zu erweitern. Dabei ist das Problem immer wieder, dass man sich nur sehr schwer in den Dateien zurechtfindet. Es wird viel Zeit benötigt, wenn sich ein Entwickler in die Applikation einarbeiten muss, um bestimmte Vorgänge oder Funktionen zu suchen. Aufgrund dessen empfiehlt es sich, ein bestimmtes Muster bei dem Aufbau der Applikation der Applikation umzusetzen, damit der Code für andere und auch für den Entwickler selbst übersichtlicher und somit einfacher zu verstehen wird.

Ein weiterer Vorteil eines Übersichtlichen Aufbaus ist die Sicherstellung der Wiederverwendbarkeit. Es wird einfacher Codestücke bei einem Ausbau wiederzuverwenden, vor allem, wenn nach demselben Muster vorgegangen wird. Grosse Verschachtelungen können verhindert werden, welche eine solche Wiederverwendbarkeit erschweren.

Bei der Entwicklung unserer Applikation haben wir uns gefragt, wo wir anfangen sollten. Unser Ziel war es, die Übersichtlichkeit zu optimieren, da wir beide zusammen daran arbeiten. Ein solches Entwurfsmuster sollte unsere Zusammenarbeit vereinfachen. Ausserdem könnten wir damit auch deutlich einfacher Hilfe anfordern, da sich andere Personen weniger lange in den Code einlesen müssten.

2.1.2 Auswahl des passenden Entwurfsprinzips / -musters

Wir haben uns schlussendlich für das Strukturierungsmuster Model View Controller (MVC) entschieden. In der Softwareentwicklung ist dieses am weitesten verbreitet. Es legt die Struktur fest, wie der Code angelegt wird. Die einzelnen Funktionen, Klassen und Methoden werden damit einfacher zu finden. Ausserdem wird es auch einfacher, gleichzeitig an unterschiedlichen Orten im Code zu arbeiten. Wenn nämlich mehrere Entwickler Änderungen an denselben Dateien vornehmen, kommt es zu Merge-Konflikten, welche unter Umständen sehr zeitaufwendig werden können.

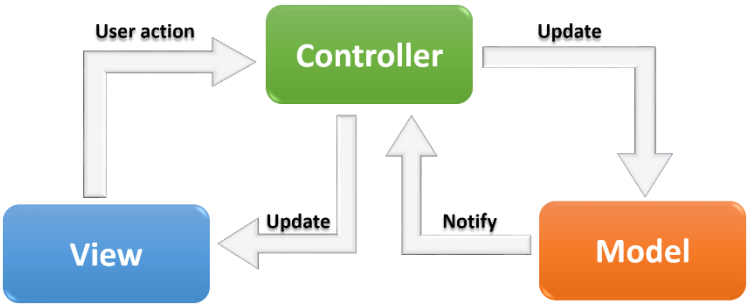

Abbildung 1: Übersicht der MVC-Struktur

Mit der MVC-Struktur wird der Code für die Logik, die Ansicht und die Datenerhaltung getrennt. Dadurch wird es einfacher die einzelnen Komponenten wiederzuverwenden oder auch den Code eines Teammitglieds zu verstehen. Auch Unabhängige können uns in diesem Projekt einfacher helfen, weil das Entwurfsmuster allgemein bekannt ist und es daher klar ersichtlich ist, wie die Komponenten miteinander agieren.

2.1.3 Anwendung auf das ursprüngliche Problem

Um das MVC umzusetzen haben wir drei verschiedene Packages erstellt, in welchen unsere Klassen verteilt wurden. Jeweils entstand ein Package für die drei Arten der Komponenten Model, View und Controller.

In dem View-Package befinden sind alle Klassen, die entweder ein Fenster oder GUI-Elemente in diesem Fenster erzeugen und verwalten. Um auf das GUI zugreifen zu können muss daher auf eine dieser Klassen zugegriffen werden.

Das Package Model beinhaltet die Klassen, welche die Daten in Form von Objekten zwischenspeichert und zur Verfügung stellt, während die Applikation läuft.

Die Klassen im Package Control steuern die gesamte Applikation. Sie sind dafür zuständig die Daten von der Server-Komponente zu erhalten, nach denen das Model zu aktualisieren und anschliessend die View nach dem neuen Spielzustand anzupassen. Auch nimmt er Benutzereingaben über die View-Klassen entgegen und sendet dementsprechend Nachrichten zu der Server-Komponente.

Mithilfe der MVC-Struktur behielten wir den Überblick über unsere Applikation. Ausserdem erhielten wir einen Startpunkt an welchem wir mit der Entwicklung beginnen konnten. Die Entwicklung dann konnte gut aufgeteilt werden, ohne dass wir uns gegenseitig in die Quere gekommen sind.

2.2 Entwurfsprinzip / -muster Singleton

2.2.1 Problemstellung

Die Client-Komponente muss Daten nur zur Laufzeit speichern. Dabei ist es jedoch wichtig, dass keine Inkonsistenzen entstehen. Die Daten werden in verschiedenen Objekten an unterschiedlichen Orten der Applikation verwendet. Daher ist es von Vorteil, wenn nur ein einziges Objekt zu haben, welches die Daten speichert und diese bei der Erzeugung neuer Objekte mitgibt. Dies versichert, dass alle Daten in der gesamten Applikation verfügbar sind und dass alle dieselbe Aktualität haben. Auch kann damit Code übersichtlicher gestaltet werden, da die Variablen, die wiederverwendet werden, nicht durch unzählige Funktionen hindurch mitgegeben werden oder global verfügbar sein müssen.

2.2.2 Auswahl des passenden Entwurfsprinzips / -musters

Eine Lösung dafür bietet das Singleton Design Pattern. Dieses Entwurfsmuster wird in der Softwareentwicklung verwendet. Damit wird exakt ein einziges Objekt einer Klasse erzeugt. Dabei ist der Zugriff auf dieses Objekt in dem gesamten Projekt sichergestellt. Um es zu erhalten muss über die Klasse darauf zugegriffen werden.

Der Aufbau einer solchen Klasse ist auf dem nebenstehenden Bild (Grundlegender Aufbau einer Singleton-Klasse) erkennbar. Dabei wird der Konstruktor privat gesetzt, damit er jeweils nur ein einziges Mal aufgerufen werden kann. Er speichert ein Objekt der Klasse in einem privaten static-Objekt der Klasse.

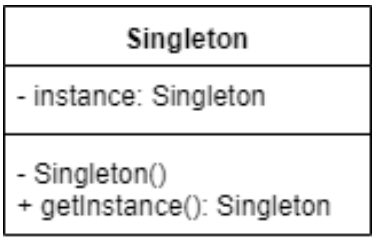

Abbildung 2: Grundlegender Aufbau einer Singleton-Klasse

Um ein Objekt dieser Klasse zu erhalten, kann die Methode «getInstance()» aufgerufen werden. Sie überprüft, ob bereits ein Objekt dieser Klasse existiert und erzeugt allenfalls ein neues, bevor sie es zurückgibt.

2.2.3 Anwendung auf das ursprüngliche Problem

In unserem Projekt ist das Singleton-Prinzip in der ControlFactory zum Einsatz gekommen. Es gibt viele verschiedene Control-Klassen, die alle dieselben Instanzvariablen benötigen. Diese können dem Singleton-Objekt mitgegeben werden, wenn es in der Main-Klasse erzeugt wird. Alle weiteren Control-Objekte, welche diese Instanzvariablen verwenden, werden nun über diese Factory erzeugt. Da sie in der gesamten Applikation verfügbar ist, müssen wir uns keine Gedanken zu der Erzeugung der anderen Control-Objekten mehr machen, dies übernimmt die Factory-Klasse für uns. So ist auch sichergestellt, dass überall dieselben Daten mit derselben Aktualität verwendet werden.

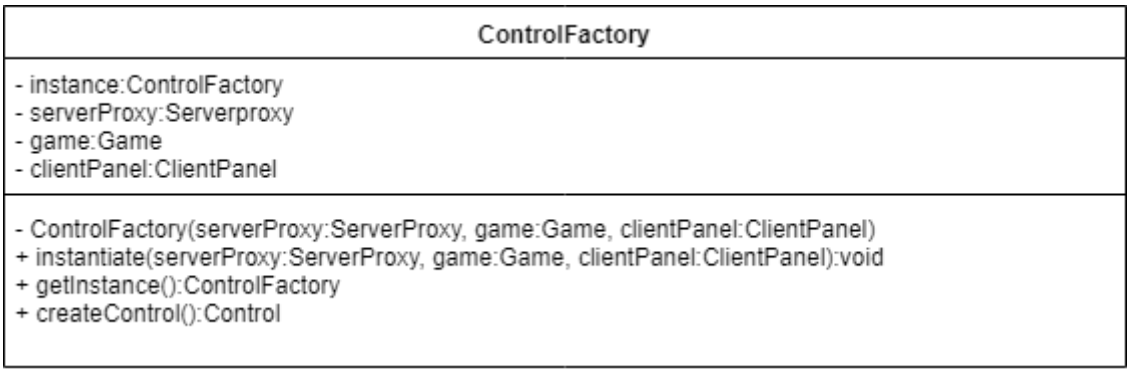

Abbildung 3: Struktur der ControlFactory

Auf der Abbildung 3 (Struktur der ControlFactory) ist unsere Implementation des Singleton-Prinzips ersichtlich. In unserem Fall ist die Methode «instantiate()» dafür zuständig, eine Objekt der Factory zu erzeugen, falls noch keines existiert. Diese sollte vor dem Aufruf von «instance()» aufgerufen werden. Der Grund, weshalb die zwei Funktionen «instantiate()» und «instance()» nicht beide zu einer einzigen

Funktion zusammengefasst sind, ist dass wir nach den Anleitungen in den Screencasts vorgegangen sind. Falls noch genügend Zeit bleibt, werden wir dies korrigieren. Es erscheint uns sauberer eine einzige Methode zu haben, welche prüft, ob ein Objekt bereits existiert und eines erzeugt, falls noch keines besteht.

Auf der Abbildung 3 ist auch die createControl()-Funktion abgebildet. Diese Control-Erzeugungsfunktionen, von denen für die Übersichtlichkeit nur eine Beispielsfunktion abgebildet ist, erzeugen ein Control-Objekt und geben dies zurück. Es ist erkennbar, dass man nun mit diesem Singleton nur ein einziges Mal die Parameter Game, ClientPanel und ServerProxy mitgeben muss, und zwar der ControlFactory. Von da an erledigt sie es, wenn sie die Control-Objekte, welche jeweils diese drei Parameter bei der Erzeugung benötigen, erzeugt.