Entwicklungsportfolio

TEILKOMPONENTE CLIENT

Florian Moser

MODUL 326 – OBJEKTORIENTIERT ENTWERFEN UND IMPLEMENTIEREN
INFW2017A
VERSION 2.2
04.01.2019

Inhalt

Abbildungsverzeichnis

1 Tagesjournals

1.1 Journal vom 26.11.2018

Ablauf

Tätigkeiten	Dauer (h)
IET-GitLab: Klonen des Projektes	0.5
Schnittstellen-Absprache mit den Server-Teams	0.5
Erstellen des minimalen GUI's.	1

Tätigkeiten

Als erstes haben wir uns das Projekt vom GitLab geholt und wir uns zuerst die verschiedenen Schnittstellen angeschaut, die uns zur Verfügung gestellt werden. Wir versuchten etwas zu pushen. Dies ist aber bei beiden fehlgeschlagen. Aus diesem Grund haben wir unser Projekt aufs GitHub von Sven geladen. Danach haben uns danach haben mit den Servergruppen darüber unterhalten.

Als die Fragen geklärt waren haben wir begonnen, die Projektstruktur aufzusetzen. Ich habe ein minimales GUI erstellt. Meine java-swing Kenntnisse waren nicht mehr so frisch, darum habe ich mir die Screencasts zur Hilfe genommen.

Reflexion

Ich fand die Gespräche mit den Server-Teams nicht sehr aufschlussreich. Einige waren noch etwas ahnungslos, weil sie wohl den bestehenden Code noch nicht so genau angeschaut hatten. Wir einigten uns darauf, dass wir uns an den vorgegebenen Schnittstellen organisieren und dass wir nachfragen, wenn Fragen auftauchen.

Die Screencasts zum Minimal-GUI haben mir viel gebracht: Ich kam dadurch sehr gut voran und bin froh, dass nun meine Java-Swing Kenntnisse wieder etwas aktueller sind und ich neue Sachen betreffend Java-Swing gelernt habe.

1.2 Journal vom 27.12 bis 03.12.2018

Ablauf

Tätigkeiten	Dauer (h)
Sven erklärte mir den Code, den er geschrieben hat	0.2

Tätigkeiten

Wir hatten heute nicht gross Zeit, um am Projekt weiterarbeiten zu können, denn wir hatten die Präsentationen des Fotomodules. In den letzten Minuten vor der Pause hat mir Sven gezeigt, an was er unter der Woche gearbeitet hat.

Reflexion

Ich fand es eine gute Idee, dass mir Sven in der verbleibenden Zeit den Code gezeigt hat. So war ich wieder auf dem neusten Stand. Ich habe noch nicht alles zu 100% verstanden, darum will ich mir bis nächsten Montag die Screencasts anschauen und ich will auch noch einmal einen Blick in den bereits existierenden Code werfen.

1.3 Journal vom 04.12 bis 10.12.2018

Ablauf

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

Tätigkeiten

Unter der Woche habe den Code von Sven angeschaut und einige Kommentare hinzugefügt.

Zusammen mit Sven habe ich heute Morgen zuerst die restlichen Control-Klassen geschrieben. Hierfür mussten wir auch Änderungen an der Control-Factory und dem Dispatcher vornehmen. Danach habe ich mich um das Login gekümmert. Am Ende befasste ich mich mit der Bewegung der Spielfiguren. (Action-Listener usw.)

Reflexion

Wir konnten zu zweit effizient arbeiten und konnten die restlichen Controller-Klassen ohne Probleme in kurzer Zeit programmieren: Es war eine gute Idee, dass wir zusammengearbeitet hatten.
Auch beim Login hatte ich keine Mühe, es war aber sicherlich auch nicht der schwerste Auftrag. Dafür hatte ich beim Bewegen der Spieler einige Probleme mit den Action-Listener. Ich konnte bis zum Ende der Lektion die Key-Events nicht abfangen. Ich habe mit Sven abgemacht, dass wir dem Problem in den nächsten Tagen noch nachgehen.

1.4 Journal vom 11.12 bis 17.12.2018

Ablauf

Tätigkeiten	Dauer (h)
Bewegen der Gegner	1
Error-Messages	1
Entwurfsprinzipien / -muster schreiben	1

Tätigkeiten

Zu Hause habe ich an den Entwurfsmuster gearbeitet. In der Schule habe ich mich wie immer zuerst mit Sven abgesprochen. Danach haben wir uns an die Arbeit gemacht. Ich habe das Bewegen der Gegner implementiert und Error-Messages hinzugefügt, wenn das Spiel abgebrochen wird.

Reflexion

Ich bin froh, dass wir heute sehr effizient arbeiten konnten und dass wir aus diesem Grund quasi fertiggeworden sind. Ich hatte heute Probleme mit dem Aktualisieren des Spielfeldes. Aus irgendeinem Grund haben sich die Gegner nicht bewegt, weil das Spielfeld nicht richtig aktualisiert worden war. Nachdem ich verschiedene Varianten ausprobiert hatte, habe ich Hilfe bei Sven geholt. Zusammen mit Sven konnte ich das Problem lösen.

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.

Jetzt müssen wir in den Ferien nur noch die beiden Branches zusammenmergen.

1.5 Journal vom 18.12 bis 05.01.2019

Ablauf

Tätigkeiten	**Dauer (h)**
Mergen	0.5
Packagediagramm erstellen	1
Sequenzdiagramm erstellen	1
Entwurfsprinzipien / -muster schreiben	1.5

Tätigkeiten

Sven und ich haben heute zusammen geskypet, damit wir mit dem Programm fertig werden. Gemergt habe nur ich. Ich habe jeden Konflikt mit Sven jeden Konflikt besprochen. Als wir fertig waren, habe ich das Packagediagramm und ein Sequenzdiagramm erstellt.

Reflexion

Die Merge-Konflikte haben uns noch etwas Zeit gekostet. Es war vielleicht nicht die beste Idee, dass wir nicht so oft commited und aus diesem Grund unsere Branches sehr unterschiedliche Stände hatten.

Beim Sequenzdiagramm hatte ich keine grossen Probleme. Ich habe dies schon öfters gemacht. Das einzig mühsame war das Arbeiten mit Modelio. Mehr Probleme hatte ich beim Packagediagramm: Dies habe ich noch nie gemacht. Zuerst musste ich nachlesen, wie man ein Paketdiagramm zeichnet.

2 Entwurfsprinzipien / -muster

2.1 MVC

Problemstellung

In unserer Teilkomponente haben müssen wir mit dem User interagieren. Für dies verwenden wir Java-Swing. Wenn man ohne Nachzudenken einfach beginnt daran zu arbeiten, dann wird das Projekt ziemlich schnell unübersichtlich. Mit jedem neuen File wird man sich weniger gut zurechtfinden können. Dies Kostet wertvolle Zeit, die ein Entwickler meist nicht hat. Für den Entwickler wird es schwieriger einen Fehler zu finden. Weiter ist es auch schwieriger, dass Projekt sinnvoll zu erweitern, ohne dass dabei Code doppelt vorkommt.
Oft wird der Code eines Projektes auch wieder für ein neues Projekt verwendet. Für einen neuen Entwickler wird es aber beinahe unmöglich in kurzer Zeit den bestehenden Code zu übernehmen, wenn dieser nicht generell strukturiert ist, sondern nur passend aufs Projekt angelegt wurde.

Die Probleme bei einem strukturlosen Vorgehen sind also offensichtlich: Bei der Entwicklung kann wertvolle Zeit durch z.B. Suchen verloren gehen und später wird es schwieriger, den Code wiederverwenden zu können.

Auswahl des passenden Entwurfsprinzips / -musters

Es gibt eine einfache Lösung zu den oben erläuterten Problemen: MVC (Model-View-Control). MVC ist ein sehr weit verbreitetes Model um eine Software in einzelne Komponente zu unterteilen, dabei werden die verschiedenen Klassen entweder dem Model, dem View oder dem Control-Komponenten hinzugefügt. MVC legt also die Codestruktur fest. Durch die MVC-Struktur wird der Code sofort übersichtlicher: Wenn man auf der Suche nach einem GUI-Element ist, weiss man sofort, dass dies unter View zu finden ist. Dasselbe gilt für die Logik unter Control und für Daten unter Model.

MVC löst aber nicht nur das Problem von unübersichtlichem Code: Wir benutzten MVC auch, um die Arbeit am Projekt etwas aufteilen zu können. Ich habe zum Beispiel zuerst an View-Komponenten gearbeitet, während Sven in der Control-Komponente sein Code schreib. So konnten wir Mergekonflikte verhindern.

Für MVC spricht auch, dass es nicht aufwendig ist, wenn man die einzelnen Komponenten wiederverwenden will. (z.B., wenn man dieselbe Applikation auch für ein anderes Betriebssystem schreiben will, kann man z.B. die Model-Komponente übernehmen) Dies hatte aber keinen Einfluss auf unseren Entscheidungsprozess, da eine Wiederverwendung unserer Applikation ausgeschlossen werden kann.

Anwendung auf das ursprüngliche Problem.

Wie man in unserem Paketdiagramm (Abbildung 1: Paketdiagramm des) erkennen kann, haben wir für jede MCV-Komponente ein eigenes Paket erstellt. Nun haben wir jede Klasse, die wir erstellt haben, einer diesen drei Packages zugeteilt:

Alle Klassen, die Logik enthalten und aus diesem Grund unsere Applikation steuern haben wir dem Control-Package zugeordnet. Die Klassen in diesem Package regeln die Kommunikation zwischen dem Server, von dem jegliche Daten kommen, und der View-Komponente, wo dann alle Daten angezeigt werden.

Klassen die GUI-Elemente enthalten, fanden unter dem View-Package ihren Platz.

Für die Spielobjekte haben wir Model-Klassen erstellt. Wenn Daten zum Spiel vom Server kommen werden sie in einer dieser Klassen als Objekt zwischengespeichert. Diese Klassen haben wir logischerweise im Model-Package angesiedelt.

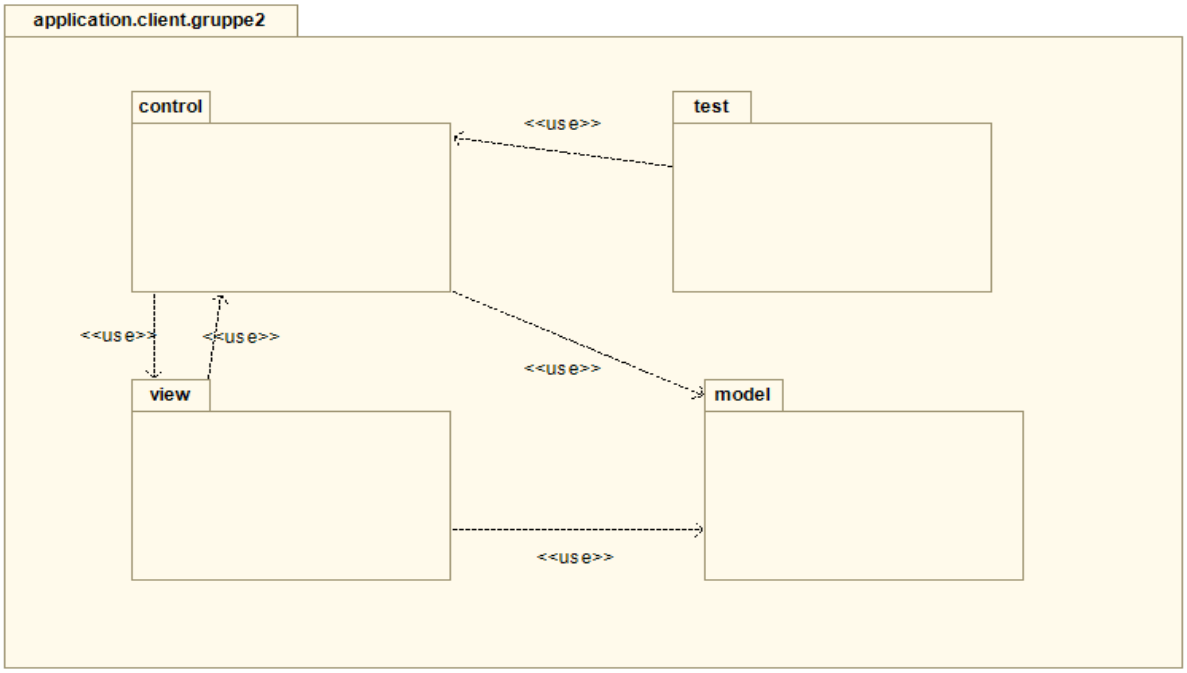

Abbildung 1: Paketdiagramm des Clients

2.2 Factory-Design-Pattern

Problemstellung

In unserem Projekt haben wir für jede Aktion eine eigene Controller-Klasse, die von einer allgemeinen Control-Klasse erbt. Wenn wir nun eine Meldung vom Server bekommen, sollte diese Aktion ausgeführt werden. Nun sollte man auf die Methoden der verschiedenen Control-Klassen zugreifen können, damit die Aktion ausgeführt werden kann. Ohne Design-Pattern wird es an dieser Stelle sehr aufwendig, für jede eigene Control-Klasse ein neues Objekt zu erstellen, nur damit die Methode dieser Klasse ausgeführt werden kann.
Auch brauchen alle Control-Klassen dieselben Parameter: Ohne Design-Pattern entsteht also auch duplizierter Code.

Auswahl des passenden Entwurfsprinzips / -musters

Die oben beschriebenen Probleme gilt es natürlich zu verhindern. Darum haben wir uns für das Factory-Pattern entschieden. Auch dieses Pattern wird heutzutage bei vielen Softwartes verwendet.

Bei diesem Pattern erstellt man eine Factory, die sich um das Erstellen der Objekte kümmert. Dies bedeutet, dass so die oben erwähnten Probleme umgehen werden können. Daher ist es naheliegend, dass wir uns für dieses Patter entschieden haben.
Wenn man dieses Pattern verwendet, ist die einfache Erweiterbarkeit der Applikation auch gewährleistet.

Anwendung auf das ursprüngliche Problem.

In unserem Projekt haben wir das Factory-Design-Pattern bei den Controller-Klassen verwendet. Wir haben eine ControlFactory erstellt. In der ControlFactory-Klasse gibt es für jede Control-Klasse eine eigene Methode: Diese erstellt mit den nötigen Parametern das gewünschte Objekt und gibt dieses zurück. Im Code kann man danach die Control-Klasse verwenden, ohne dass man mit «new» ein neues Objekt erstellt hat und ohne, dass man die Instanzvariablen – die immer dieselben sind – der Klasse mitgegeben hat.

Das Gesamte haben wir mit einem Singleton kombiniert: Die ControlFactory wird nur beim Starten des Spieles über eine Methode instanziiert. Dies stellt sicher, dass es nur eine einzige ControlFactory existiert und diese immer dieselben Daten verwendet. Auf die ControlFactory kann man von überall im Projekt zugreifen.

In Abbildung 2: Ausschnitt aus Sequenzdiagram mit ControlFactory sieht man den Mechanismus mit der ControlFactory: Anstatt im BombermanPanel ein Objekt zu erstellen, wird das Objekt über die Factory geholt.

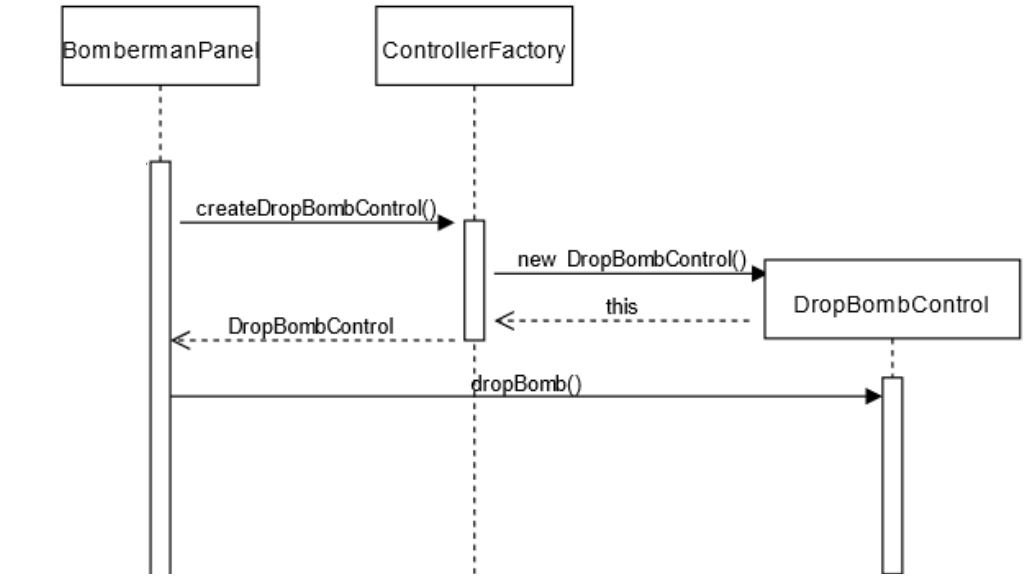

Abbildung 2: Ausschnitt aus Sequenzdiagram mit ControlFactory