Ein in C-Sharp entwickeltes Backend für ein Party Game

ASE - Programmentwurf

im Rahmen des Studiums zum

Bachelor of Science

des Studienganges Angewandte Informatik

an der Dualen Hochschule Baden-Württemberg Karlsruhe

von

Kevin Bobbe

Mai 2021

Matrikelnummer: 6887164

Kurs: TINF18B4

Inhaltsverzeichnis

1 Einleitung

1.1 Projektidee

Idee des Projektes ist die Implementierung eines Partyspiels. Das Spielprinzip hierbei ist, dass in einer Gruppe von Mitspielern immer einer an der Reihe ist. Der Spieler, welcher an der Reihe ist zieht eine karte. Auf dieser Karte steht eine Aufgabe, welche der Spieler in einer gewissen Zeit absolvieren muss. Nachdem die Zeit abgelaufen ist können die anderen Mitspieler bewerten, wie erfolgreich die Aufgabe absolviert wurde. Je nach dem wie gut oder schlecht die Aufgabe umgesetzt wurde wird der Spieler bestraft oder auch nicht. Die Härte der Strafe hängt von den Bewertungen der anderen Spieler ab. Nach der Bestrafung ist der nächste Spieler an der Reihe.
Zur Umsetzung dieser Idee wird eine Datenbank für die Verwaltung der Spiel und Spielerdaten benötigt. Ebenfalls wird ein Backend und ein Frontend benötigt. Aus Zeitgründen wird das Frontend in dieser Dokumentation, sowie dem gesamten Programmentwurf nicht betrachtet. Da es aber später implementiert werden soll, wird eine API bereitgestellt, mit der das Frontend süäter mit dem Backend kommunizieren kann.

1.2 Use Cases

Die Use Cases die bei der Projektidee anfallen sind ein Login, sowie eine Registrierung um sich als Spieler anzumelden. Ein Freundesystem um zusammen mit Freunden zu spielen, da darauf die gesamte Idee basiert. Ein Use Case für das eigentliche spielen des Spiels. Einer Historie um Vergleiche zu ermöglichen. Sowie ein Profil, welches alle wichtigen Informationen des eigenen Accounts beinhalten.

1.2.1 Login / Registration

Beim Login soll eine Abfrage an die Datenbank geschickt werden ob die eingegeben Userdaten, d.h. Username und Passwort, korrekt sind. Bei einer korrekten Eingabe wird der Login akzeptiert. Bei einer ungültigen Eingabe gibt es einen Fehler, welcher das Problem beschreibt, z.B. "Passwort ist falsch.". Falls der User nicht vorhanden sein sollte, dann soll der Fehler auf die Registrierung verweisen. Bei der Registrierung wird bei korrekter Eingabe der Userdaten, d.h. Username mit der Einschränkung der gewählte Username ist noch nicht

vorhanden und Passwort, ein neuer User in der Datenbank angelegt und der User kann sich jetzt einloggen.

1.2.2 Freunde

Ein User soll die Möglichkeit haben Freundschaftsanfragen an andere User zu schicken. Werden dieser angenommen, dann sind die User befreundet und können sich gegenseitig zu Spielen einladen und ihre History anschauen. Ebenfalls sehen sie den aktuellen Status des anderen, d.h. ob dieser z.B. Online oder Offline ist. Eine Freundschaft kann auch beendet werden.

1.2.3 Spiel

Das Spiel ist der Hauptbestandteil des Projektes. Es besteht aus Lobbyphase und Spielphase. In der Lobbyphase kann das Spiel eingestellt werden, d.h. der Spielmodus kann festgelegt werden, Mitspieler bestimmt, Decks ausgewählt, etc. In der Spielphase wird das Spiel durchgeführt, d.h, ein Spieler zieht eine Karte und bekommt eine Aufgabe. Alle anderen Spieler sehen die Aufgabe ebenfalls und welcher Spieler an der Reihe ist. Wenn der Spieler an der Reihe seine Aufgabe erfüllt hat, bewerten seine Mitspieler die Umsetzung. Bei zu schlechter Wertung wird bestraft. Nach der Bestrafung ist der nächste Spieler an der Reihe.

1.2.4 Historie

Jeder User besitzt eine eigene Historie. In dieser Historie ist vermerkt, wie viele Spiele der User gemacht hat und wie oft er bestraft wurde. Ein Nutzer kann sich seine Historie jederzeit anschauen. Ebenfalls können seine Freunde seine Historie einsehen, sowie er ihre.

1.2.5 Profil

Im Profil eines Users stehen alle seine Accountdaten, d.h. sein Username, seine Freunde und seine Historie. Im Profil kann der User sein Passwort ändern.

2 Domain Driven Design

2.1 Ubiquitous Language

Die Nutzung einer einheitlichen Sprache und einheitlicher Begriffe in einem Projekt und dessen Domäne sind essentiell, damit es nicht zu Missverständnissen kommt. In diesem Projekt soll die Domäne ein Spiel mit Spielclient sein. Dadurch ergeben sich zwei Kontexte, der Kontext Spiel und der Kontext Client. Dadurch gibt es zwei getrennte UL.

Wort	Bedeutung
User	User enthalten die Benutzerinformationen des späteren Spielers
Register	Ein neuer User legt einen Account an
Login	Ein bestehender User meldet sich mit seinem Account an
Friend	Eine Verknüpfung von zwei Usern
History	Auflistung von Benutzerinformationen über bisherige Spiele

Tabelle 1: Client Kontext

Wort	Bedeutung
Game	Spielinformationen über ein bestimmtes Spiel und dessen Zustand
Player	Verknüpfung User und Game
Carddeck	Beinhaltet die Spielkarten (Taskcards)
Gamemode	Bestimmt wie ein Spiel abläuft und was für Decks im Spiel genutzt werden
Taskcard	Beinhaltet die Aufgaben und Bestrafungen

Tabelle 2: Spiel Kontext

2.2 Verwendeten Muster

2.2.1 Value Objects

Value Objects, sind Objekte, die keine eigene Identität besitzen. Das heißt ein Value Object wird nur durch seine Werte beschrieben. Es wurden mehrere Value Objects implementiert. Dazu gehören das Careddeckgenre, der Gamemode, das RecommendedAge, der Status und die Taskcard. Jedes dieser Objekte ist unveränderlich, das bedeutet wenn es einmal gültig

konstruiert wurde, dann ist es immer gültig. Dadurch können bei der Erstellung dieser Objekte die gewünschten Eigenschaften geprüft werden, z.B. das ein RecommendedAge immer größer oder gleich 0 sein muss. Durch diese Unveränderbarkeit und die initialen Prüfungen, ist der Code leichter testbar, da weniger ungewollte Seiteneffekte entstehen.

2.2.2 Entities

Im Gegensatz zu den Value Objects verändern sich Entitäten während ihres Lebenszyklus. Ebenfalls unterscheidet sich zwei Entitäten einzig in ihrer Id, welche immer eindeutig sein muss. Dadurch sind ihre Eigenschaften irrelevant beim Vergleich, wodurch theoretisch zwei Entitäten mit exakt den selben Eigenschaften existieren könnten. Entitäten repräsentieren Dinge der Domäne, weshalb das Carddeck, das Game, der User und die History als Entitäten angelegt sind.

2.2.3 Aggregates

Beim maßstabsgetreuen modellieren einer Domäne werden viele Entities und VO, die große Graphen mit oft Bidirektionale Abhängigkeiten bilden, gefunden. Da dies oft und sehr schnell sehr unübersichtlich wird, werden die Entitäten und Value Objects zu gemeinsam verwalteten Einheiten, den Aggregaten zusammengefasst. Friend und Player sind so ein Aggregat, denn sie verbinden zwei Userentitäten miteinander, um diese Verbindungen einfacher verwalten zu können und die Viele zu Viele Beziehung einfach abzubilden. Eine Verknüpfung aus Enität und Value Object ist Beispielsweise das GamemodeIncludesDeck Aggregat. Hierbei werden Ein Deck und ein Gamemode miteinander verknüpft.

2.2.4 Repositories

Repositories vermitteln zwischen Domäne und Datenmodell. Sie stellen der Domäne Methoden bereit, um Aggregates aus dem Persistenzspeicher zu lesen, zu speichern und zu löschen. Dadurch wird der konkrete technische Zugriff auf den Speicher vor der Domäne verborgen. Sie bilden eine Art Anti-Corruption-Layer zur Peristenzschich und wurden im API-Adapter implementiert. Die Repositories ermöglichen das einfache austauschen von Komponenten aus der Adapter Schicht, z.b. der Datenbank, da bei einem Tausch nur die Implementierung des Repositories geändert werden muss und nichts an der Domäne. Ebenfalls lassen sich Repositories gut mocken, wodurch sie das Testen vereinfachen. Es wurde

ein Repsoitory für das Carddeck, den User und das Game angelegt.

2.2.5 Domain Services

Domain Services bilden komplexe Verhalten oder Prozesse ab, die keiner Entität eindeutig zugeordnet werden können. Zudem definieren sie eine Erfüllungsvertrag für externen Dienste. Solch ein Erfüllungsvertrag ist der AuthService, dieser bietet in einem Interface die Funktion zur Passwortverifizierung des Users, wobei die Domäne aber nicht abhängig von der Implementierung ist. Ein komplexer Prozess, der keiner Entität zugeordnet werden kann ist der CredentialService. Dieser ist für die Einhaltung der Namens-, Email-, und Passwortkonventionen zuständig.

3 Clean Architecture

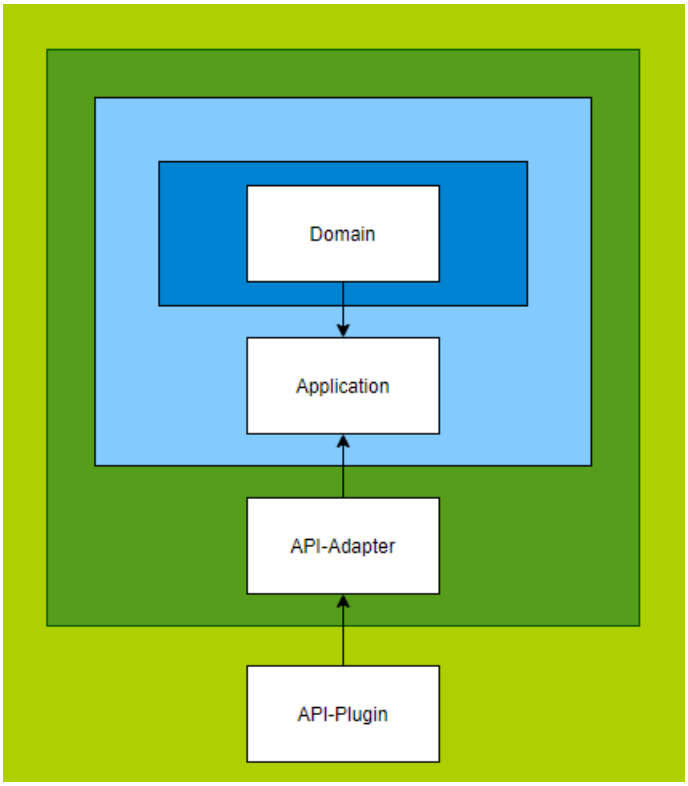

Abbildung 1: Schichtarchitektur

Die Architektur des Projektes wurde nach der Clean Architecture in vier Schichten unterteilt. Diese Schichten sind in 1 zu erkennen. Sie heißen Plugin, Adapter, Application und Domain. Bei der Implementierung dieser Schichten wurden die fundamentalen Grundregeln beachtet:

- Der Anwendungs- und Domaincode ist frei von Abhängigkeiten
- Sämtlicher Code kann eigenständig verändert werden
- Sämtlicher Code kann unabhängig von Infrastruktur kompiliert und ausgeführt werden
- Innere Schichten definieren Interfaces, äußere Schichten implementieren diese
- Die äußeren Schichten koppeln sich an die inneren Schichten in Richtung Zentrum

Die wichtigste Regel ist dabei die Richtung von außen nach innen, wodurch die äußeren Schichten immer nur von den inneren Schichten abhängig sind. Dadurch bleibt eine innere Schicht immer von einer äußeren Schicht unbeeinflusst, wodurch äußere Schichten jederzeit

angepasst oder ausgetauscht werden können, ohne dass innere Schichten verändert werden müssen.

Die Plugin-Schicht beinhaltet den API-Controller. In diesem wird die API mit ihren Pfaden definiert. In der Plugin Schicht werden einzig und allein die Funktionen der im Adapter implementierten Repositories genutzt. Dadurch ist es in der Plugin Schicht irrelevant, wie das anzeigen der Daten implementiert ist und was mit den daten passiert, den es werden nur noch die fertig gemappten Daten aus der Adapter Schicht abgefragt.

In der Adapter-Schicht sind zwei Dinge implementiert. Einerseits die Repositories, um die Funktionen der Domäne abzubilden und Nutzen zu können und andererseits das Mapping. Das bedeutet in der Adapter Schicht, werden die Daten von außerhalb so verarbeitet, dass sie in den inneren Schichten genutzt werden können.

Die Application Schicht beinhaltet die eigentliche anwendungsspezifische Geschäftslogik. Hier werden die Funktionen der Repositories und Aggregates aufgerufen. Damit steuert diese Schicht den Fluss der Daten und Aktivitäten auf die Aggregate und den zugehörigen Entitäten und Value Objects.

Die Domain Schicht enthält die Entitäten, Value Objects, Aggregate, Repositories und Domain Services auf welche die Application Schicht zugreift. In dieser Schicht liegt somit die organisationsweite und anwendungsunabhängige Geschäftslogik. Diese hat das Ziel in zukünftigen Anwendungen verwendet werden zu können, ohne dass sie erneut implementiert werden muss. Die Repositoryinterfaces erlauben anderen Schichten auf die Persistierung zuzugreifen, ohne dabei von deren Implementierung abhängig zu sein.

Die Abstraction Schicht wurde nicht implemntiert, da dies allgemeine mathematische Konzepte, Algorithmen oder ähnliches beinhaltet. Da diese in der Domäne nicht vorkommen, wäre eine Implementierung somit sinnfrei.

4 Programming Principles

4.1 SOLID

4.1.1 Single Responsibility Principle

Das Single Responsibility Principle sagt aus, dass jede Klasse nur eine Zuständigkeit haben sollte. Dadurch hat jede Klasse eine klare Aufgabe und klare Gründe warum sie sich wie verhält. Dadurch sinkt die Komplexität und Kopplung des Codes, wodurch die Wartbarkeit enorm ansteigt. Angewendet wurde diese Prinzip vor allem in den Klassen der Application Schicht, so haben beispielsweise die LoginUser und AddNewcard Klasse nur einen Nutzen. Der LoginUser ist dazu da um den User einzuloggen und die AddNewCard Klasse erstellt eine neue Karte und fügt sie einem Deck hinzu.

4.1.2 Open Closed Principle

Das Open Closed Principle besagt, dass Klassen, Module und Funktionen offen für Erweiterungen und geschlossen für Änderungen sein sollen. Erweiterungen sollen nur durch Vererbung oder die Implementierung von Interfaces durchgeführt werden, dadurch wird gewährleistet, dass bestehender Code nicht verändert wird bzw. verändert werden muss. Dieses Prinzip wird hauptsächlich für die Interfaces im Domaincode angewendet, d.h. für die Repositories.

4.1.3 Liskov Substitution Principle

Beim Liskov Substitution Principle werden Ableitungsregeln stark eingeschränkt, sodass Invarianzen eingehalten werden. Dadurch entstehen bei Ableitungen in OOP aus ïst ein" Beziehungen, "verhält sich wie" Beziehungen. Abgeleitete Typen haben schwächere Vor-, aber stärkere Nachbedingungen. Dadurch kann sich auf ein bestimmtes Verhalten der abgeleiteten Typen verlassen werden, wenn das Verhalten des basistyps bekannt ist. Angewendet ist dieses Prinzip zum Beispiel bei den Domain Service Interfaces, da diese darauf abzielen dem Domaincode eine "verhält sich wie" Beziehung anzubieten.

4.1.4 Interface Segregation Principle

Das Interface Segregation Principle besagt, dass Anwender nicht von Funktionen abhängig sein sollten, welche sie gar nicht nutzen. Deshalb sollten große schwere Interfaces und Klassen, welche viele Funktionalitäten bündeln, vermieden werden, denn ein Anwender einer Interface Methode ist automatisch abhängig von Änderungen an anderen Methoden des Interfaces. Der Code beinhaltet keine Interface Implementierung, bei denen nicht alle Funktionen genutzt werden und somit sind alle Eigenschaften des Prinzips erfüllt.

4.1.5 Dependency Inversion Principle

Das Dependency Inversion Principle invertiert die Abhängigkeit von Low.Level- und High-Level-Modulen. Denn beide Module sollten von Abstraktionen abhängen und Abstraktionen sollten nicht von Details abhängig sein, sondern umgekehrt. Bei diesem Prinzip, werden die Regeln durch die HLM vorgegeben und von den LLM implementiert. Damit bilden die HLM ein wiederverwendbares Framework. Umgesetzt ist diese Prinzip durch die Schichtarchitektur. In dieser werden die Repositories des HighLevel Moduls Domain in den darunterliegenden Schichten Implementiert und aufgerufen.

4.2 GRASP

Das Akronym GRASP sthet für General Responsibility Assignment Software Patterns, welches Basisprinzipien für Entwurfsmuster sind. Ziel heirbei ist es die Lücke zwischen gedachtem Domänemodel und Softwareimplementierung möglichst klein zu halten.

4.2.1 Low Coupling

Die Kopplung ist ein Maß für die Abhängigkeit zwischen Objekten, wobei wie der Name des Prinzips schon sagt, versucht wird eine möglichst geringe Kopplung zu erzielen. Positive Effekte einer geringen Kopplung sind die geringeren Abhängigkeiten zu Änderungen in anderen Teilen, die einfachere Testbarkeit, eine einfachere Wiederverwend- und Verständlichkeit. Formen der Kopplung beispielsweise in Java sind Interfaces, abgeleitete Klassen, Attribute. Durch eine lose Kopplung werden Komponenten austauschbar. Durch die Inversion of Control sind die meisten Klassen von Klassen innerer Schichten abhängig, welche zumeist stabiler sind und daher unproblematisch. Beispielsweise sind die Mapper der Adapter Schicht

nur von den Objekten, welche sie umwandeln abhängig, wodurch sie einfach zu testen sind.

4.2.2 High Cohesion

Die Kohäsion ist ein Maß für den ZUsammenhalt einer Klasse und beschreibt semantische Nähe. Hohe Kohäsion und Lose Kopplung dienen als Fundament für idealen Code. Durch eine hohe Kohäsion wird das Design einfacherer und verständlicher, sowie die Komponenten wiederverwendbar. Da semantische Nähe schwierig automatisch einschätzbar ist, ist menschliche Einschätzung notwendig, was zu Fehlern führt. Durch die automatisch bestimmten Metriken, wie z.B. Anzahl Attribute und Methoden einer Klasse oder Häufigkeit der Verwendung der Attribute, welche nicht immer ideal sind, wird diese Einschätzung jedoch vereinfacht. Der Zusammenhalt der Klassen in diesem Projekt ist recht gut, da die Attribute der Entitäten und Value Objects semantisch gut zu den Klassen passen. Das Recommended Age Value Object beinhaltet zum Beispiel nur Attribute, welche semantisch gut zu einem Alter passen und diese werden auch alle in mindestens einer Methode verwendet.

4.2.3 Information Expert

Hierbei geht es um eine allgemeine Zuweisung einer Zuständigkeit zu einem Objekt. Die einfachste Möglichkeit ist es dem Objekt, das die Informationen besitzt, die Verantwortung dafür zu überreichen. Wenn im Designmodell eine passende Klasse existiert wird diese verwendet, ansonsten wird im Domänenmodell eine passende Repräsentation gesucht und dafür eine Klasse im Designmodell erstellt. Objekte sind dann zuständig für Aufgaben über die sie Informationen besitzen, wodurch eine Kapselung von Informationen und leichtere Klassen entstehen. Aber es kann zu Problemen mit anderen Prinzipien führen. Auch hier passen die Aggregate Root Klassen gut rein, da diese die Verantwortung zu ihren Informationen und denen der enthaltenen Entities beziehungsweise Value Objects übernehmen. Das GamemodeIncludesDeck Aggregat übernimmt beispielsweise die Zuständigkeit für die zugehörige CardDeck Entität und das zugehörige Gamemode Value Object.

4.2.4 Creator

Dieses Prinzip legt fest, wer für die Erzeugung von Objekten zuständig ist. Im allgemeinen kommt ein Objekt als Creator eines anderen in Frage, wenn es zu jedem erstellten Objekt eine Beziehung hat. Ein geeigneter Creator verringert die Kopplung von Komponenten. Die

Creator der Objekte sind hier zum Beispiel die Repositories, da in deren Implementierung die Instanzen der Aggregate und Entitäten erzeugt werden. Ein weiteres Beispiel wären auch die Mapper welche die Objekt-Formate umwandeln und dabei neue Objekte erzeugen. Dabei wird die Kopplung zwischen den inneren und den äußeren Schichten verringert.

4.2.5 Indirection

Indirektion bzw. Delegation kann Systeme oder Teile von Systemen voneinander entkoppeln. Dabei bietet Indirektion mehr Freiheitsgrade als Vererbung bzw. Polymorphismus, aber unter höherem Aufwand.

4.2.6 Polymorphism

Polymorphismus ist ein grundlegendes OO Prinzip zum Umgang mit Variation. Dabei erhalten Methoden je nach Typ eine andere Implementierung, wodurch auf Fallunterscheidungen verzichtet werden kann. Die Konditionalstruktur wird sozusagen im Typsystem codiert und es werden Abstrakte Klassen oder Interfaces als Basistyp genutzt. Mithilfe von Polymorphismus wird die Software einfacher erweiterbar und bestehende die Implementierung muss nicht verändert werden.

4.2.7 Controller

Der Controller verarbeitet einkommende Benutzereingaben und koordiniert zwischen Benutzeroberfläche und Businesslogik. Seine Hauptaufgabe ist die Delegation zu anderen Objekten und er enthält keine Businesslogik. Für die verschieden API Services wurden verschieden Controller implementiert. Diese Controller leiten die Anfragen an die API Adapter Schicht weiter, in denen dann die Daten für die inneren Schichten umgemapped und weitergereicht werden.

4.2.8 Pure Fabrication

Bei der Pure Fabrication besitzt eine Klasse keinen Bezug zur Problemdomäne, wodurch eine Trennung zwischen Technologie und Problemdomäne und eine Kapselung von Algorithmen entsteht. Da in diesem Projekt alle Klassen einen Bezug zu Domäne haben und keine speziellen und wiedervendbaren Algorithmen entwickelt wurden, wurde dieses Prinzip nicht verwendet.

4.2.9 Protected Variations

Mit der Kapselung verschiedener Implementierungen hinter einer einheitlichen Schnittstelle wird die Software vor Variation gesichert. Der Einfluss von Variabilität einzelner Komponenten soll dadurch nicht das Gesamtsystem betreffen. Mit Polymorphie und Delegation lässt sich ein System gut schützen. Weitere Schutzmöglichkeiten gibt es durch Stylesheets im Webumfeld, Spezifikation von Schnittstellen oder Betriebssystemen und Virtuellen Maschinen. Diese einheitlichen Schnittstellen wurden einerseits durch die Repsoitory Interfaces der Domain Schicht, welche dann für verschieden Systeme unterschiedlich implementiert werden können, und andererseits durch die einheitliche API nach außen implementiert.

4.3 DRY

Das Akronym DRY steht für Don´t repeat yourself und ist auf alles mögliche, wie z.B. Datenbankschemata, Testpläne oder Dokumentationen anwendbar. Dabei darf es nur eine Quelle der Wahrheit geben und alle anderen Quellen werden davon abgeleitet. Dadurch haben die Auswirkung der Modifikationen eines Teils eine definierte Reichweite und alle relevanten Teile ändern sich automatisch. Es gibt drei verschiedene Arten der Duplikation.

4.3.1 Imposed Duplication

Hierbei glaubt der Entwickler die Duplikation sei unumgänglich und erlegt sie dem Code somit auf.

4.3.2 Inadvertent Duplication

Hierbei übersieht der Entwickler eine Codeduplikation und dupliziert somit Code versehentlich. Diesem Problem kann mithilfe von Duplikatserkennungstools, aber teilweise vorgebeugt werden.

4.3.3 Impatient Duplication

Hierbei ist der Entwickler schlichtweg zu faul eine erkannte Duplikation zu entfernen.

4.3.4 Anwendung

In der Erstellung des Projektes wurde darauf geachtet keinen duplizierten Code zu erzeugen und stattdessen bestehenden Code wiederzuverwenden. Dies wurde deshalb gemacht, da duplizierter Code zu einigen Problemen führen kann. Durch duplizierten Code wird der Programmcode länger und damit unübersichtlicher. Zudem kommt es dadurch auch zu duplizierten Bugs. Ein weiteres Problem ist der steigende Aufwand bei der Änderung von Datenstrukturen, da dies an vielen Stellen geändert werden müssen und bei vergessen kann es zu Problemen kommen. Umgesetzt wurde DRY bei den verschieden Bridges in der Adapter-Schicht, welche die Daten von extern zu intern mappen. Diese Stellen damit sicher, dass überall die richtigen Daten genutzt werden und bei Änderung einer datenstruktur muss lediglich die Bridge angepasst werden.

5 Entwurfsmuster

Für die Klasse DatabaseContext wurde das Singleton Muster angewendet. Dabei wurde eine Lazy Instantiation eingesetzt, da diese den Systemstart nicht verlängert. Die Threadunsicherheit ist in sofern nicht relevant, da alles auf einem Thread läuft. Singleton kam zum Einsatz, da es bei korrekter Implementierung garantiert, dass es Systemweit nur eine einheitliche Instanz des DatabaseContext gibt. Zudem benötigt das System auch nur einen DatabaseContext, weshalb es sinnvoll ist die Erstellung mehrerer Instanzen einzuschränken. Außerdem erleichtert das Singleton das verstehen und verwenden der Instanz.

Abbildung 2: UML vor Singleton

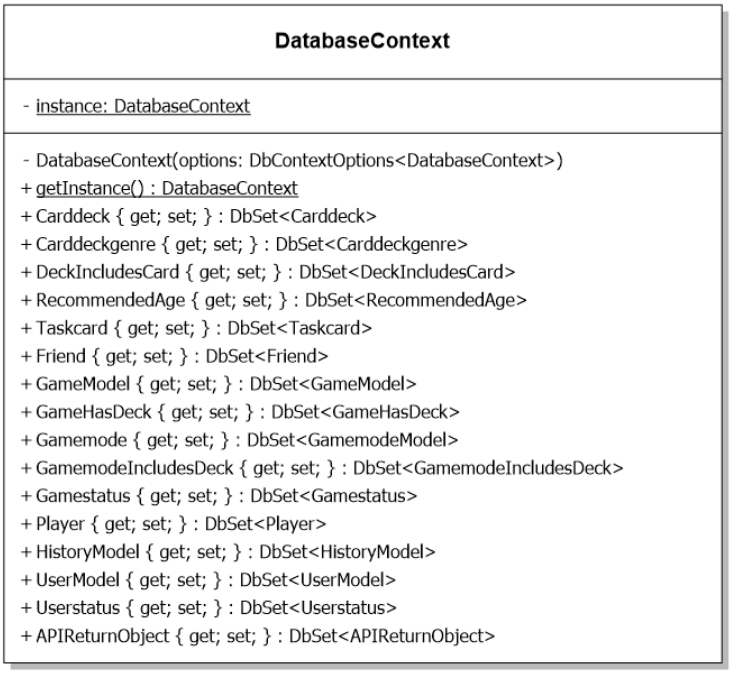

Abbildung 3: UML nach Singleton

6 Refactoring

6.1 Large Class

Die APIController Klassen für sowohl den User-, als auch den Gameservice haben zu viel Verantwortung. Das ist erkennbar an den unglaublich vielen Methoden der Controller. Entstanden ist dieser Code Smell durch eine unzureichende Aufteilung der Methoden, d.h. ein Controller für alle API Methoden des jeweiligen Services. Um diesen Code Smell aufzulösen und die Klassen kleiner und übersichtlicher zu gestalten wurden die APIControllerKlassen in kleiner Klassen mit weniger Methoden aufgeteilt. Dadurch wird der Code wesentlich überschaubarer und es ist viel einfacherer die verschiedenen API-Routen zu erkennen und ihre Funktionen zu prüfen.
Zu finden ist diese Änderung auf dem Refactoring/LargeClass - Branch.
Commit: cd38e88ca26e98bd1a15d79282239eb37d601df5

6.2 Duplicated Code

Duplicated Code ist ein Code Smell der besagt, dass die gleiche Code-Struktur an mehreren Stellen vorhanden ist. Das heißt die gleiche Anweisung ist mehrfach in einer Methode oder

Klasse oder es gibt ähnlichen Code in mehreren Methoden. Zumeist wird Duplicated Code durch Unwissenheit verursacht und ist dadurch auch gleichzeitig jener, der am häufigsten vor kommt. So auch in diesem Projekt. In den verschiedenen Mappern, den BridgeKlassen, kommen einige Mappingmethoden mehrfach vor. Außerdem werden in den ServiceKlassen und den APIControllern alle Collections gleich gemapped. In Zukunft könnte dadurch der Code auseinander driften, wodurch der Code unnötigerweise schwieriger zu warten wird. Beim Refactoring wurden einerseits die doppelt vorkommenden Mappingmethoden identifiziert und so umgesetzt, dass sie nur noch einmal vorkommen. Andererseits wurden einheitliche Methoden zum Mapping von Collections eingebaut.
Zu finden ist diese Änderung auf dem Refactoring/DuplicatedCode - Branch.
Commit: 345a03915c0d7522160fa389fab307932da5a206

7 Unit Tests

Mithilfe von Unit Tests wird die anwendungsspezifische Logik innerhalb der Application Schicht getestet. Dazu wurden die Repositories in den Tests manuelle gemocked. Mithilfe der Tests soll festgestellt werden, ob die implementierte Logik so funktioniert, wie es geplant wurde und ob die Daten korrekt verarbeitet und weitergegeben werden.