DHBW
Duale Hochschule
Baden-Württemberg
Karlsruhe

Advanced Software Engineering

Programmentwurf

für die Prüfung zum

Bachelor of Science

des Studienganges Angewandte Informatik

an der

Dualen Hochschule Baden-Württemberg Karlsruhe

von

Dominik Klysch

Abgabedatum TODO

Matrikelnummer 1010579
Kurs TINF18B4

Inhaltsverzeichnis

Kapitel 1

Domain Driven Design

1.1 Analyse der Ubiquitous Language

Wort	Bedeutung
User	User spielen auf dem Game-Server
Register / Registrieren	Ein neuer User legt einen Account an
Login / Einloggen / Anmelden	Ein bestehender User meldet sich mit seinem Account an
Level	Gibt an wie hoch die Zugriffsrechte eines Users sind
Report	durch einen Report kann ein User einen anderen User zum Beispiel für einen Regelverstoß melden
Report Type	Gibt eine grobe Kategorie eines Reports an
Rank / Rang	Der Rang des Users, verschiedene Ränge haben verschieden hohe Level, möglich: User, Moderator, Admin
Moderator	Verwaltet User, User Reports und User Bans
Admin	Verwaltet die Ränge der User
Ban	Wenn ein User gegen Regeln verstößt kann er gebannt werden und ist dadurch eine Zeit lang gesperrt
todo	todo

1.2 Analyse und Begründung der verwendeten Muster

1.2.1 Value Objects

Value Objects wurden für die Ranks und ReportTypes verwendet, da diese zwar in der Datenbank eine ID besitzten aber im restlichen Code diese nicht benötigen. Dadurch profitieren die Klassen von der Unveränderlichkeit, dies bedeutet das ein Objekt welches gültig konstruiert wurde danach nicht mehr ungültig werden kann. Dadurch kann der Code weniger ungewollte Seiteneffekte erzeugen und ist leicht testbar.

1.2.2 Entities

Für die User, Ranks, Reports und Bans wurden Entities angelegt, da es für diese wichtig ist eine eindeutige ID zu besitzen und basierend auf dieser unterschieden zu werden. Im gegensatz zu den Value Objects haben die Entities einen Lebenszyklus und verändert ihre Werte während iherer Lebenszeit. Aber die Entity sorgt auch dafür, dass keine ungültigen Werte gesetzt werden und sie nicht in einen ungüligen Zustand versetzt werden kann. Zuletzt besitzen sie noch Methoden die verschiedenes Verhalten der Entities beschreiben.

1.2.3 Aggregates

Aggregate wurden keine verwendet, da die Komplexität der Entities und Value Objects nicht sehr hoch war.

1.2.4 Repositories

Repositories wurden für User, Ranks, Reports und Bans angelegt. Dadurch erhält der Domain Code Zugriff auf den persistenten Speicher, deren Implementierung wird der Domäne aber verborgern und ist somit flexibler. Sie bilden eine Art Anti-Corruption-Layer zur Persistenzschicht und wurden im Datenbank-Adapter Implementiert. Ein großer Vorteil ergibt sich dadurch, dass in Zukunft auch weitere Adapter für zum Beispiel andere Datenbanken hinzugefügt oder der alte ausgetauscht werden können, ohne das der Domain Code davon beeinflusst wird.

1.2.5 Domain Services

Domain Services wurden auch keine verwendet, da es kein Domänenweites komplexes Verhalten gab, welches nicht in Entities oder Value Objects abgebildet werden konnte.

Kapitel 2

Clean Architecture

2.1 Schichtarchitektur planen und begründen

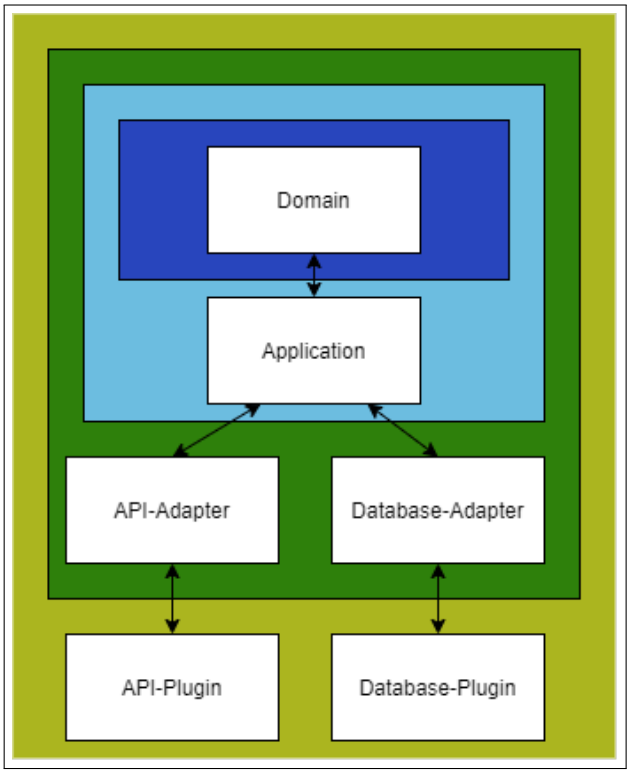

Abbildung 2.1: Schichtarchitektur

Die Schichtarchitektur ist in vier Schichten unterteilt: Plugin, Adapter, Application, Domain. Die Plugin-Schicht wurde nochmals in API und Datenbank aufgeteilt, dies erhöht die übersichtlichkeit da die beiden Schichten nicht direkt miteinander interagieren sollen. Sie interagieren nur mit ihrem eigenen Adapter. Dementsprechend wurde die Adapter-Schicht auch in API und Datenbank aufgeteilt, um die jeweiligen Objekt-Formate in die der inneren Schichten zu konvertieren. Die Adapter rufen dann Funktionen aus der Application Schicht auf. In der Application-Schicht liegt dann die eigentliche Logik der Anwendung und hier werden API und Datenbank zusammen

geführt. In der Domain Schicht befinden sich die Entities und Value Objects der Application Schicht, dies dient dem Ziel diese in Zukunft in weiteren Application-Schichten einheitlich verwenden zu können.

Kapitel 3

Programming Principles

3.1 Analyse und Begründung für SOLID

3.1.1 Single Responsibility Principle

Das Single Responsibility Principle sagt aus, dass eine Klasse nur eine Zuständigkeit haben sollte. Somit hat jede Klasse eine klar definierte Aufgabe und es entsteht eine niedrige Komplexität des Codes. Und um so niedriger die Komplexität des Codes desto besser lässt er sich warten und erweitern.

3.1.2 Open Closed Principle

Durch das Open Closed Principle wird die Software offen für Erweiterungen aber geschlossen für Änderungen. Das bedeutet der Code wird nur durch Vererbung bzw.Implementierung von Interfaces erweitert. Dies führt dazu das bestehender Code nicht geändert wird, wodurch die Anwendung für eine gute kompatibilität über die Versionen sorgt. Hierbei ist es wichtig viele Abstraktionen zu nutzen um die Erweiterbarkeit zu fördern.

3.1.3 Liskov Substitution Principle

Im Liskov Substitution Principle werden Ableitungsregeln stark eingeschränkt was zu einer Einhaltung von Invarianzen führt. Hierbei müssen abgeleitete Typen schwächere Vorbedingungen und stärkere Nachbedingungen haben. Durch dieses Prinzip ergeben sich somit im OOP "verhält sich wie"Beziehungen statt den üblichen ïst ein"Beziehungen. Somit kann man sich auf ein bestimmtes Verhalten der abgeleiteten Typen verlassen, wenn man das Verhalten des Basistyps kennt.

3.1.4 Interface Segregation Principle

Das Interface Segregation Principle sorgt dafür, dass Anwender nicht von Funktionen abhängig sind die sie gar nicht nutzen. Dies wird ermöglicht indem man statt einzelnen großen schweren Interfaces mit vielen Funktionen, viele kleine leichte Interfaces mit wenigen Funktionen implementiert. Somit kann man seine Klasse passend für den Anwendungszweck anpassen ohne unnötige extrafunktionen mit zu schleppen.

3.1.5 Dependency Inversion Principle

Im Dependency Inversion Principle wird die klassische Struktur in der High-Level Module von Low-Level Modulen abhängig sind umgekehrt. Denn Abstraktionen sollten nicht von Details abhängig sein, daher werden hierbei die Regeln durch High-Level Module vorgegeben und in Low-Level Modulen Implementiert. Dadurch bilden die High-Level Module ein Framework und es werden nur noch abtrakte Abhängigkeiten genutzt. Ein großer Vorteil der dieses Prinzip ist die hohe Fexibilität der Software, denn Low-Level Module können somit einfach ausgetauscht werden ohne die High-Level Module zu beinflussen.

3.2 Analyse und Begründung für GRASP

GRASP steht für General Responsibility Assignment Software Patterns, dies sind Basis Prinzipien auf denen Entwurfsmuster aufbauen. Das Ziel hierbei ist es, die Low Representational Gap (LRG) möglichst klein zu halten. Also die Lücke zwischen gedachten Domänenmodell und Softwareimplementierung (Designmodell).

3.2.1 Low Coupling

Die Kopplung ist ein Maß für die Abhängigkeit zwischen Objekten, wobei versucht wird eine möglichst geringe Kopplung zu erreichen. Dadurch hat man eine geringere Abhängigkeit zu Änderungen in anderen Teilen des Codes und die Software wird einfacher testbar und wiederverwendbar. Zudem wird die Software verständlicher, da weniger Kontext benötigt wird um einzelne Ausschnitte des Codes zu verstehen. Kopplung entsteht in Java zum Beispiel bei der Implementierung von Interfaces, durch das halten eines Attributs vom Typ einer anderen Klasse oder durch das Besitzen einer Methode mit Referenz zu einer anderen Klasse. Durch eine lose Kopplung werden Komponenten austauschbar. Man kann an konkrete oder abstrakte Datentypen gekoppelt sein, aber auch durch verschiedene Threads mit gemeinsamen Sperren oder durch Resourcen wie gemeinsame Dateien. Dabei ist eine Kopplung zu stabilen Komponenten weniger problematisch.

3.2.2 High Cohesion

Kohäsion ist ein Maß für den Zusammenhalt einer Klasse, es beschreibt die semantische Nähe der Elemente einer Klasse. Hohe Kohäsion und Lose Kopplung sind das Fundament für idealen Code, da sie zu einem einfacheren und verständlicheren Design führen. Die Semantische Nähe der Attribute und Methoden ist nur schwer automatisiert testbar und es ist Menschliche Einschätzung notwendig. Zu den Automatisch bestimmbaren technischen Metriken gehören Anzahl Attribute und Methoden einer Klasse und Häufigkeit der Verwendung der Attribute in allen Methoden aber diese sind nicht immer ideal.

3.2.3 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 enstehen. Aber es kann zu Problemen mit anderen Prinzipien führen.

3.2.4 Creator

Im Creator-Prinzip wird festgelegt wer für die Erzeugung von Objekten zuständig ist. Allgemein gesehen 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, was zu den bereits genannten Vorteilen führt.

3.2.5 Indirection

Durch Indirection werden Systeme oder Teile von Systemen voneinander entkoppelt. Es bietet mehr Freiheitsgrade als Vererbung oder Polymorphismus, aber benötigt auch mehr Aufwand. Dadurch ensteht das gewollte Low Coupling wodurch die Software viel flexibler wird.

3.2.6 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. Zudem werden Polymorphe Methodenaufrufe erst zur Laufzeit gebunden. Mithilfe von Polymorphismus wird die Software einfacher erweiterbar und bestehende die Implementierung muss nicht verändert werden.

3.2.7 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. Zu den verschiedene Arten von Controllern gehören der System Controller, wobei nur ein Controller für alle Aktionen genutzt wird, der nur für kleine Anwendungen praktikabel ist und der Use Case Controller, welcher einen Controller pro Use Case nutzt.

3.2.8 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. Damit werden diese Softwareteile auch außerhalb der Domäne einfach wiederverwendbar und durch die Kapselung spezieller Funktionalität wird High Cohesion begünstigt.

3.2.9 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 Schutzöglichkeiten gibt es durch Stylesheets im Webumfeld, Spezifikation von Schnittstellen oder Betriebssystemen und Virtuellen Maschinen.

3.3 Analyse und Begründung für DRY

Die Abkürzung DRY steht für Don't Repeat Yourself und kann auf alles mögliche wie zum Beispeil Datenbankschemata, Testpläne, Buildsysteme oder Dokumentation angewendet werden. Es darf dann nur noch eine Quelle der Wahrheit geben und alle anderen Quellen werden davon abgeleitet. Das ist Vergleichbar zu den Normalformen bei RDBMS. Die Auswirkungen der Modifikation eines Teils haben eine definierte Reichweite und betreffen keine unbeteiligten Teile, wobei sich alle relevanten Teile automatisch ändern. Für die Entstehung von dupliziertem Code gibt es drei Hauptgründe:

3.3.1 Imposed Duplication

Hierbei handelte es sich um eine auferlegte Duplikation und der Entwickler glaubt die Duplikation ist unumgänglich.

3.3.2 Inadvertent Duplication

Dies ist eine versehentliche Duplikation da der Entwickler die Duplikation nicht bemerkt, dies kann aber teilweise durch diverse Tools vermieden werden indem sie erkannte Duplikate markieren.

3.3.3 Impatient Duplication

Diese ungeduldige Duplikation ensteht durch Entwickler die zu faul sind die Duplikation zu beseitigen.

Kapitel 4

Entwurfsmuster

4.1 Singleton

Das Singleton wurde für die beiden Klassen MariaDBConnector und JWTProvider eingesetzt.

4.1.1 Einsatz begründen

Das Singleton kam zum Einsatz da es gerantiert, dass systemweit nur eine Instanz vorhanden ist und es nur eine einheitliche aktive Datenbank Verbindung geben soll. Zudem benötigt das System auch nur eine globale Instanz des JWTProvider. Weiterhin ermöglicht es einen einfachen Zugriff auf diese Instanz, ist einfach zu verstehen und anzuwenden. Durch das Singleton gibt es zwar keine Möglichkeiten für polymorpe Aufrufe mehr aber die wird in diesem Fall auch nicht benötigt.

4.1.2 UML vorher/nachher

TODO

4.2 Dependency injection

Unsicher ob das hier zählt?

4.2.1 Einsatz begründen

4.2.2 UML vorher/nachher

Kapitel 5

Refactoring

5.1 Code Smells identifizieren

5.2 Refactorings begründen

5.3 Dokumentation

5.3.1 API

5.3.2 Database

5.3.3 Sonstiges