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 31.05.2021

Matrikelnummer	1010579
Kurs	TINF18B4

Inhaltsverzeichnis

Kapitel 1

Einleitung

In diesem Programmentwurf wurde ein User Verwaltungs System Backend für einen beliebigen Game Server entworfen. Das bedeutet dieses System soll auf die Datenbank eines Game Servers mit Usern zugreifen und diese über eine API Schnittstelle verwalten. Es wurde auch ein simples UI entworfen um die Funktionalität der API Schnittstelle einfacher überprüfen zu können. Dieses UI gehört aber nicht zum eigentlichen Programmentwurf und sollte daher als von der Bewertung ausgeschlossen betrachtet werden.

1.1 Installation

1.1.1 Requirements

Benötigte Software zum ausführen des Programmentwurfs:

- Java 11
- Docker Desktop
- docker-compose

1.1.2 Starten der Anwendung

Befehle im root Order der Repo ausführen. Das erste bauen könnte etwas länger dauern.
Bauen und ausführen: docker-compose up –build
Starten ohne neu bauen: docker-compose start
Stoppen: docker-compose stop
Stoppen + alles löschen: docker-compose down –volumes

Nach dem starten sollte gewartet werden bis in der Console der Datenbank steht: mysqld: ready for connections.

Default Admin User: email: root@example.com pw: example

Frontent: http://localhost:3000/ Wenn noch keine Reports / Bans in der DB sind zeigt das UI dauerhaft einen Ladebalken auf diesen Seiten an. Wenn ein User einen neuen Rang bekommt, aber noch im UI angemeldet ist: Einmal neu im UI Anmelden damit eine neue Session mit dem aktuellen Rang ausgestellt wird.

API-Server: http://localhost:7001/
Datenbank: localhost:3306

1.1.3 Ausführen der Tests

Befehl im root Order des Backends ausführen: ./gameserver-backend
Wichtig die Mockito version benötigt Java 11 !
Befehl zum ausführen der Unit tests: mvn test

Kapitel 2

Domain Driven Design

2.1 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
Grant / Revoke Rank	Einem User einen Rang geben bzw. seinen Rang wegnehmen
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

2.2 Analyse und Begründung der verwendeten Muster

2.2.1 Value Objects

Value Objects wurden für die Ranks und ReportTypes verwendet, da diese rein auf ihre Werte reduziert werden können und über keine zusätzlichen Funktionen verfügen. Dadurch profitieren die Klassen von der Unveränderlichkeit, dies bedeutet das ein Objekt welches gültig konstruiert wurde danach nicht mehr ungültig werden kann. Zum Beispiel wird verhindert, dass das Rank Value Object ein Level kleiner als 0 oder einen Leeren Text als Namen gesetzt bekommt. Dadurch kann der Code weniger ungewollte Seiteneffekte erzeugen und ist leicht testbar.

2.2.2 Entities

Für die User, 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.

2.2.3 Aggregates

Es wurden verschiedene Aggregate verwendet um die Abhängigkeiten zwischen verschiedenen Entities und Value Objects zu modellieren. Hierfür gruppieren die Aggregate die Entities und Value Objects zu gemeinsam verwalteten Einheiten. Dadurch reduzieren Aggregate die Komplexität der Beziehungen zwischen den Objekten. In jedem Aggregat übernimmt eine Entity die Rolle der Aggregate Root Entity beziehungsweise des Aggregat Root. Für das Aggregat bestehend aus UserEntity und Rank Value Object ist das Beispielsweise das UserRankAggregate. Zudem gibt es noch das Aggregat BanAggregate welches die Beziehung zwischen der BanEntity, der gebannten UserEntity, und der UserEntity die den Ban durchgeführt hat. Und zuletzt noch das ReportAggregate welches die Beziehung zwischen der ReportEntity, der reporteten UserEntity, und der UserEntity die den Report abgesendet hat. Die Aussenwelt muss eine Methode auf dem Aggregat Root aufrufen, wenn der innere Zustand verändert werden soll um dafür zu sogen, dass sein Zustand immer den Domänenregeln entspricht.

2.2.4 Repositories

Repositories wurden für User, Ranks, Reports und Bans angelegt. Sie stellen der Domäne Methoden bereit, um Aggregates aus dem Persistenzspeicher zu lesen, zu speichern und zu löschen. Die Repositories liefern dabei immer die Aggregate Roots und damit den Zugriff auf den Rest zurück. 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. Die Repositories könnnen auch für Unit Tests leicht gemockt werden wie in den implementierten Tests zu sehen ist. Die Implementierung erfolgte in der technischen Schicht Plugin-Database.

2.2.5 Domain Services

Es wurden auch diverse Domain Services implementiert wobei auch die beiden Haupteinsatzzwecke abgebildet wurden. Für das Abbildung von komplexem Verhalten, wie zum Beispiel Regeln der Problemdomäne, welche nicht eindeutig einer bestimmten Entity oder einem bestimmten Value Object zugeordnet werden können wurde zum Beispiel ein CredentialService erstellt welcher für das Einhalten der Namens-, Email- und Passwortkonventionen in der Domäne sorgt. Der andere Einsatzzweck welcher die Definition eines Erfüllungs-Vertrages für externe Dienste beschreibt wurde zum Beispiel im AuthService abgebildet. Dieser bietet in einem Interface die Funktion an, das Passwort eines Users zu verifizieren, dabei ist die Domäne aber nicht von der technischen Implementierung abhängig, wodurch dieses Verhalten in Zukunft eine andere Implementierung erhalten kann ohne die Domäne zu beeinflussen. Wichtig zu beachten ist noch das diese Services selbst zustandslos sind.

Kapitel 3

Clean Architecture

3.1 Schichtarchitektur planen und begründen

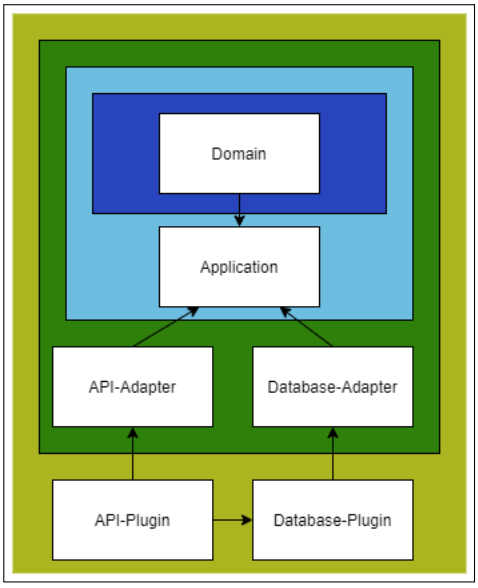

Abbildung 3.1: Schichtarchitektur

Die Schichtarchitektur wurde nach den Regeln der Clean Architecture in vier Schichten unterteilt:

- Plugin
- Adapter
- Application
- Domain

Zudem wurden in diesen Schichten 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 dieser Regeln ist dabei die Richtung von außen nach innen, wodurch die äußeren Schichten immer nur von inneren Schichten abhängig sind. Somit können äußere Schichten jederzeit ausgetauscht oder angepasst werden ohne dabei innere Schichten zu beeinflussen.

Die Plugin-Schicht wurde nochmals in API und Datenbank aufgeteilt um die übersichtlichkeit des Codes etwas zu erhöhen. Zudem wird dadurch ermöglicht nicht nur verschiedene Datenstrukturen zwischen Plugin- und Domaincode sondern auch zwischen API und Datenbank zu haben. Ein wichtiges Beispiel hierfür ist der User, welcher in der Datenbank ein Passwort besitzt aber in der API ohne dieses ausgegeben werden soll. Die Plugin Schicht ist die äußerste Schicht und hat somit als einzige externe Abhängigkeiten, neben den Test-Dependencies welche von allen benötigt werden. Was ja auch eine der fundamentalen Regeln der Clean Architecture ist. Sie enthält allgemein also Frameworks, Datentransportmittel und andere Werkzeuge. In diesem Fall befinden sich hier die Datenbank Implementierung und das API Framework. Die Schicht ist aber vom Funktionsumfang nur für die Delegation der externen Anfragen an die Adapter zuständig.

Die Adapter-Schicht wurde auch in API und Datenbank aufgeteilt und konvertiert die jeweiligen Objekt Datenstrukturen aus API und Datenbank in die Strukturen der inneren Schichten. Die Adapter rufen dann Funktionen aus der Application-Schicht mit den konvertierten Daten auf und dient als ein Anti-Corruption Layer zwischen den Technischen Schichten und der Geschäftslogik.

In der Application-Schicht liegt dann die eigentliche anwendungsspezifische Geschäftslogik und hier werden API und Datenbank zusammen geführt. Hier werden unter anderem Domain Service Interfaces Implementiert und die Funktionen der Repositories und Aggregates aufgerufen. Die steuert also sozusagen den Fluss der Daten und Aktionen von und zu den Aggregaten und somit auch den darin enthaltenen Entities.

In der Domain Schicht befinden sich die Entities, Value Objects, Aggregate, Repositories und Domain Services auf welche die Application Schicht zugreift. Hier liegt also die organisationsweit gültige und somit anwendungsunabhänige Geschäftslogik, dies dient dem Ziel diese in Zukunft zum Beispiel in anderen Anwendungen verwenden zu können. Die Repository-Interfaces welche von darunter liegenden Schichten implementiert werden erlauben der Application und Domain Schicht auf die Persistierung zuzugreifen ohne dabei von deren Implementierung abhängig zu sein indem eine Inversion of Control erzeugen wird.

Die Abstraction Schicht, welche die innerste Schicht ist, wurde nicht Implementiert, da es sich hierbei um eine Schicht handelt welche zum Beispiel allgemeine Mathematische Konzepte, Algorithmen oder Abstrahierte Muster beeinhaltet. Diese kommen in dieser Domäne aber nicht vor wodurch diese Schicht in diesem Fall keinen Sinn hätte.

Kapitel 4

Programming Principles

4.1 Analyse und Begründung für SOLID

4.1.1 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 und Kopplung des Codes. Und um so niedriger die Komplexität des Codes desto besser lässt er sich warten und erweitern. Angewendet wurde das vor allem in den Klassen der Application-Schicht, denn hier wurden für die Anfragen verschiedene Klassen erstellt. Die Klasse LoginUser bzw. RegisterUser kümmern sich zum Beispiel nur um den Login bzw. die Registrierung der User. Ein weiteres Beispiel wäre die Klasse ReportUser, da diese nur für die Report Anfrage zuständig ist.

4.1.2 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. Dieses Prinzip wurde hauptsächllich für die Interfaces im Domaincode angewendet, zum einen mit den Repository Interfaces aber auch durch die Domain Service Interfaces welche auf verschiedene Arten implementiert werden können. Ein Beispiel hierfür wäre es eine neue Datenbank hinzuzufügen welche dann den Code durch das Implementieren der Repository Interfaces erweitert. Ein weiterer Bereich ist die Database-Adapter-Schicht in der die Controller der einzelnen Tabellen von einem Allgemeinen Database Controller erben. Um den Code mit einer neuen Datenbanktabelle zu erweitern wird hier eine neue Klasse erstellt welche Vererbung einsetzt um die Funktionalität zu erweitern. Zudem werden diese dann von der Datenbank Implementiert und könnte in Zukunft durch mehrere Datenbanken implementiert werden.

4.1.3 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 „ist ein“ Beziehungen. Somit kann man sich auf ein bestimmtes Verhalten der abgeleiteten Typen verlassen, wenn man das Verhalten des Basistyps kennt. Dieses Prinzip wurde zum Beispiel für die Domain Service Interfaces angewendet welche ja darauf ausgelegt sind dem Domaincode eine „verhält sich wie“ Beziehung darzubieten.

4.1.4 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. Da der Code keine Interface Implementierungen beeinhaltet in denen Funktionen des Interfaces nicht genutzt würden sind alle Eigenschaften des Prinzips erfüllt. Das bedeutet jedes Interface welches im Programmentwurf enthalten ist dieses Prinzip erfüllt.

4.1.5 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. Dieses Prinzip wurde durch die Schichtenarchitektur der Clean Architecture verwirklicht, in welcher die Repositories und Domain Services des High-Level Moduls Domain in den darunterliegenden Implementiert und aufgerufen werden. Beim initialen Aufbau der Anwendung werden dann die Instanzen der Repositories und Domain Services erzeugt.

4.2 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).

4.2.1 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. Durch die Inversion of Control sind die meisten Klassen von Klassen innerer Schichten abhängig, somit also von stabileren Komponenten. Denn je tiefer eine Klasse liegt desto selterner wird diese Klasse geändert wodurch die Abhängigkeiten relativ stabil sind. Beispielsweise sind die Mapper in der Database-Adapter Schicht nur von den Objeten welche sie umwandeln anhängig wodurch sie sehr einfach zu testen sind.

4.2.2 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. Der Zusammenhalt der Klassen in diesem Projekt ist ziemlich gut da die Attribute, wie zum Beispiel in den Entities oder Value Objects der Domain-Schicht zu sehen ist, semantisch gut zu den Klassen passen. Das Rank Value Object enthält zum Beispiel nur Attribute welche semantisch zu einem Rang des Users gehören und verwendet jedes Attribut auch in mindestens einer Methode.

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 enstehen. 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 UserRankAggregate übernimmt zum Beispiel die Zuständigkeit über die Informationen der UserEntity und des Rank Value Object.

4.2.4 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. Die Creator der Objekte sind hier zum Beispiel die Repositories, da in deren Implementierung die Instanzen der Aggregate und Entities 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

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 die gewollte High Cohesion und die Software wird viel flexibler. Die Table Wrapper sorgen hierbei beispielsweise dafür den übergeordneten Code von den SQL Strings zu entkoppeln.

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. Zudem werden Polymorphe Methodenaufrufe erst zur Laufzeit gebunden. Mithilfe von Polymorphismus wird die Software einfacher erweiterbar und bestehende die Implementierung muss nicht verändert werden. Angewendet wurde das zum Beispiel in den Database Controllern in der Database-Adapter-Schicht aber auch für die Table Wrapper in der Database-Plugin-Schicht. Hier werden die Funktionen die Allgemein für alle Tabellen benötigt werden durch die Tabellen spezifischen Funktionen erweitert.

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. 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. Implementiert wurden die Use Case Controller in der API-Plugin-Schicht. Hierbei wurde je ein Controller für die folgenden Usecases definiert:

- Authentifizierung (Register / Login)
- User
- Rank
- Report
- Ban

Diese Controller leiten die Anfragen dann an die API-Adapter-Schicht weiter in denen dann die Daten für die inneren Schichten umgemappt 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. Damit werden diese Softwareteile auch außerhalb der Domäne einfach wiederverwendbar und durch die Kapselung spezieller Funktionalität wird High Cohesion begünstigt. Dies wurde nicht angewandt da alle Klassen einen Bezug zur Domäne hatten und das Projekt zum Beispiel keine allgemein Verwendbaren Algorithmen oder ähnliches enthällt.

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 Schutzöglichkeiten gibt es durch Stylesheets im Webumfeld, Spezifikation von Schnittstellen oder Betriebssystemen und Virtuellen Maschinen. Um diese einheitlichen Schnittstellen zu ermöglichen beinhaltet zum Beispiel die Domain-Schicht die Repository Interfaces und Domain Services, welche dann für verschiedene Datenbanken abgekapselt implementiert werden können.

4.3 Analyse und Begründung für DRY

Die Abkürzung DRY steht für Don't Repeat Yourself und kann auf alles mögliche wie zum Beispiel 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:

4.3.1 Imposed Duplication

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

4.3.2 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.

4.3.3 Impatient Duplication

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

4.3.4 Anwendung

In der Erstellung dieses Projektes wurde darauf geachtet keinen duplizierten Code zu erzeugen und stattdessen bestehenden Code wiederzuverwenden. Denn durch duplizierten Code kann es zu einigen Problemen kommen:

- Es soll eine Regelung / Einstellung geändert werden und es werden manche Stellen vergessen => Inkonsistenz, Sicherheits-Risiko
- Großer Aufwand wenn sich z.B eine Datenstruktur ändert und dass an vielen Stellen angepasst werden muss => Großer Aufwand
- Duplizierter Code macht den Programmcode länger und unübersichtlicher
- Duplizierter Code führt zu Duplizierten Bugs => Shotgun Surgery

Folgende Funktionen werden zum Beispiel an mehreren Stellen aufgerufen statt den Code zu duplizieren:

Im CredentialService wurde ein Check für Name, Email und Passwort einmalig Implementiert und von vielen Stelle naufgerufen. Dadurch wird sichergestellt, dass überall einheitliche Richtlinien gelten und diese auch bei einer Änderung überall automatisch korrekt sind.

Die Mapper wie der UserMapper sorgen zum Beispiel dafür dass überall eine korrekte umwandlung der Datenstrukturen stattfindet. Dadurch muss bei der Änderung einer Datenstruktur nur der Mapper angepasst werden und alle umwandlungen bleiben funktionell.

Kapitel 5

Entwurfsmuster

5.1 Singleton

Das Singleton wurde für die beiden Klassen MariaDBConnector und JWTProvider eingesetzt. Hierbei wurde eine Lazy Instantiation eingesetzt da dadurch die Systemstartzeit nicht verlängert wird. Es wurde keine Synchronized Lazy Instantiation eingesetzt, da sowieso alles auf dem selben Thread läuft.

5.1.1 Einsatz begründen

Das Singleton kam zum Einsatz da es gerantiert, dass Systemweit nur eine Instanz vorhanden ist und es zum nur eine einheitliche aktive Datenbank Verbindung und eine JWTProvider Instanz 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. Und eine Instanz wird auch nur angelegt wenn diese auch benötigt wird. Durch das Singleton gibt es zwar keine Möglichkeiten für polymorpe Aufrufe mehr aber diese werden in diesem Fall auch nicht benötigt.

5.1.2 UML

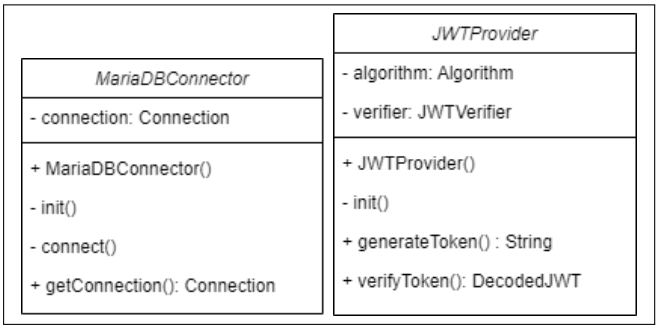

Abbildung 5.1: UML vor Singleton

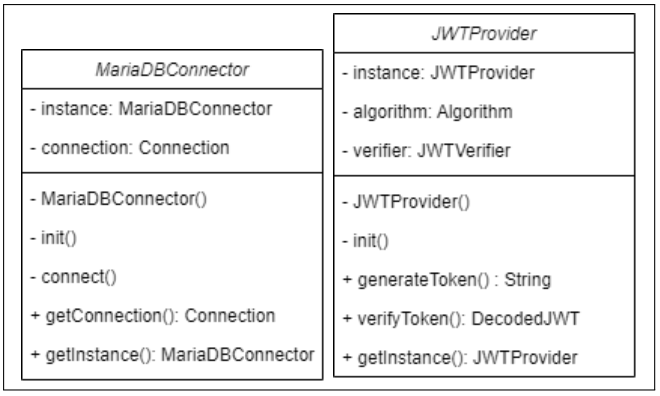

Abbildung 5.2: UML nach Singleton

Kapitel 6

Refactoring

6.1 Long Method in APIServer

6.1.1 Identifizierung

Der erste Code Smell den ich im Refactoring fand war der Lange Konstruktor der APIServer Klasse. Da sollten vor dem Starten des Servers verschiedene Dinge initialisiert und konfiguriert werden. Da häufte sich aber über die Zeit immer mehr an, was zu diesem Code Smell führte.

6.1.2 Begründung des Refactorings

In diesem Refactoring wurde zuerst das Konfigurieren der Kontroller in eine extra Klasse mit dem Namen APIServerConfig ausgelagert. Diese Klasse wird dann als Parameter des Konstruktors übergeben was es in Zukunft ermöglicht den Server einfacher mit verschiedenen Konfigurationen zu starten. Danach wurde der AccessManager in die Klasse APIServerAccessManager und die Konfiguration der Endpunkte in die Klasse APIServerEndpointGroup ausgelagert, wodurch der Code viel verständlicher und überschaubarer wird. Zuletzt wurde das erstellen der Serverinstanz, für eine bessere Übersichtlichkeit, noch in eine extra Funktion verschoben. Dadurch wurde der Code insgesammt auf viele kleinere und logische Bausteine verteilt wodurch er einfacher zu verstehen ist. Zudem konnten durch das aufteilen einfacher sinnvolle Namen für die Methoden der einzelnen Schritte gefunden werden.

Commit zum Refactoring: 04e489aab4cc16ecb07dde6ed08f51ef80c86231

6.2 Duplicated Code

6.2.1 Identifizierung

Ein weiterer Code Smell ist Duplicated Code, dieser wurde in der update Methode der diversen TableWraper Klassen in der Plugin-Database-Schicht gefunden. Allgemein ist es der wichtigste Code Smell da er am häufigsten vorkommt und meistens durch Unwissenheit entsteht. Dabei ist die gleiche Code-Struktur an mehr als einerStelle im Code vorhanden. Das bedeutet die gleiche Anweisung kommt mehrfach in einer Methode oder Klasse vor oder der ähnliche Code verteilt sich auf mehreren Methoden oder Klassen. In meinem Fall wurde in fast allen Klassen auf die selbe Art ein SQL String gebaut, dies könnte aber auch an einer anderen Stelle einmalig für alle implementiert werden.

6.2.2 Begründung des Refactorings

Dieser Code könnte in Zukunft Auseinanderdriften wodurch der Wartungsaufwand unnötigerweise erhöht wird. Durch das Auslagern des gemeinsamen Codes können neue Strukturen geschaffen und die Wiederverwendbarkeit des Codes stark erhöht werden. Die Funktion wurde deshalb verallgemeinert und in die abstrakte Superklasse TableWraper ausgelagert.

Commit zum Refactoring: 07c1d0806ac9774b6f8914010928ec39a0de33f1