

# 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

# Inhaltsverzeichnis

<b>1</b>	<b>Domain Driven Design</b>	<b>2</b>
1.1	Analyse der Ubiquitous Language . . . . .	2
1.2	Analyse und Begründung der verwendeten Muster . . . . .	2
1.2.1	Value Objects . . . . .	2
1.2.2	Entities . . . . .	3
1.2.3	Aggregates . . . . .	3
1.2.4	Repositories . . . . .	3
1.2.5	Domain Services . . . . .	3
<b>2</b>	<b>Clean Architecture</b>	<b>4</b>
2.1	Schichtarchitektur planen und begründen . . . . .	4
<b>3</b>	<b>Programming Principles</b>	<b>6</b>
3.1	Analyse und Begründung für SOLID . . . . .	6
3.1.1	Single Responsibility Principle . . . . .	6
3.1.2	Open Closed Principle . . . . .	6
3.1.3	Liskov Substitution Principle . . . . .	6
3.1.4	Interface Segregation Principle . . . . .	6
3.1.5	Dependency Inversion Principle . . . . .	7
3.2	Analyse und Begründung für GRASP . . . . .	7
3.3	Analyse und Begründung für DRY . . . . .	7
<b>4</b>	<b>Entwurfsmuster</b>	<b>8</b>
4.1	Dependency injection . . . . .	8
4.2	Singleton . . . . .	8
4.3	Composite pattern . . . . .	8
4.4	Einsatz begründen, UML vorher/nachher . . . . .	8
<b>5</b>	<b>Refactoring</b>	<b>9</b>
5.1	Code Smells identifizieren . . . . .	9
5.2	Refactorings begründen . . . . .	9
5.3	Dokumentation . . . . .	10
5.3.1	API . . . . .	10
5.3.2	Database . . . . .	10
5.3.3	Sonstiges . . . . .	10

# 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 besitzen 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 ihrer Lebenszeit. Aber die Entity sorgt auch dafür, dass keine ungültigen Werte gesetzt werden und sie nicht in einen ungültigen 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 verborgen 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 dass 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

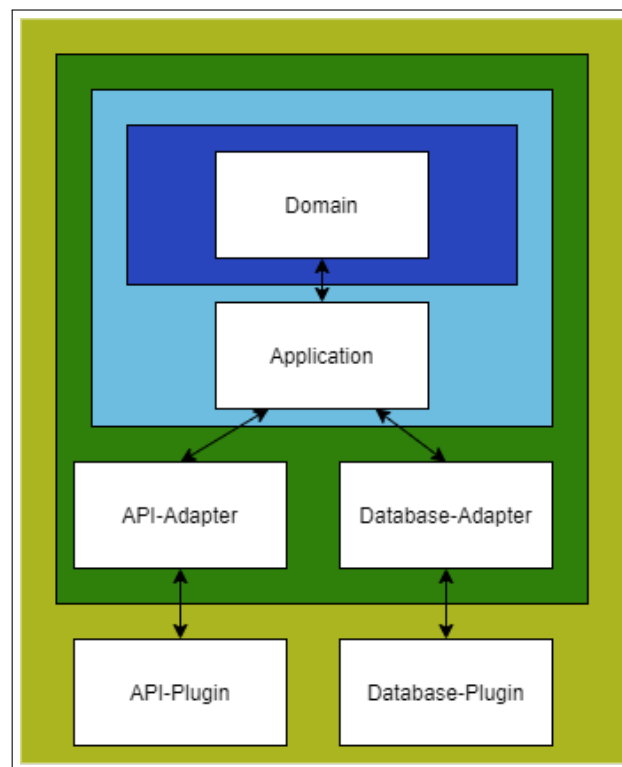


Abbildung 2.1: Schichtarchitektur

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 ist 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 abstrakte Abhängigkeiten genutzt. Ein großer Vorteil der dieses Prinzip ist die hohe Flexibilität der Software, denn Low-Level Module können somit einfach ausgetauscht werden ohne die High-Level Module zu beeinflussen.

## **3.2 Analyse und Begründung für GRASP**

## **3.3 Analyse und Begründung für DRY**



## Kapitel 4

# Entwurfsmuster

4.1 Dependency injection

4.2 Singleton

4.3 Composite pattern

4.4 Einsatz begründen, 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**