Manga Collector

Programmentwurf

der Vorlesung „**A**dvanced **S**oftware **E**ngineering“

an der

Dualen Hochschule Baden-Württemberg Karlsruhe

von

Danny Kroll

Abgabedatum 15. Mai 2022

Matrikelnummer	7044515
Kurs	TINF19B4
Bearbeitungszeitraum	5. & 6. Semester
Gutachter der Studienakademie	Mirko Dostmann

Inhaltsverzeichnis

1. Domain Driven Design

Das Konzept der Ubiquitous Language ist einer der Grundpfeiler im Domain Driven Design: In der Kommunikation mit dem Kunden und im gesamten Quellcode des Projekts sollen hierbei einheitliche Begriffe verwendet werden.

1.1 Analyse der Ubiquitous Language

Ein Manga besitzt einen Namen und einen Publisher, wobei ein Publisher eine Vielzahl an Mangas herausbringen kann. Zudem besitzt ein Manga einen Author und ein Genre, wobei sowohl der Author als auch das Genre von mehreren Mangas verwendet werden kann. Ein Manga hat zudem eine Liste (episodeList), in welcher mehrere Episoden sein können. Diese Episoden besitzen außerdem eine ratingList, welche Bewertungen für eine spezifische Episode speichert.

1.2 Analyse und Begründung der verwendeten Muster

Im nächsten Schritt sollen die verwendeten Muster und deren Nutzen im Projekt erläutert werden.

1.2.1 Value Objects

Das Value Object ist ein in der Softwareentwicklung eingesetztes Entwurfsmuster. Wertobjekte sind unveränderbare Objekte, die einen speziellen Wert repräsentieren. Soll der Wert geändert werden, so muss ein neues Objekt generiert werden. Sei sind zudem nur über die Werte ihrer Attribute definiert und besitzen keine eigene Identität. Zwei Wertobjekte sind identisch, wenn alle Werte ihrer Attribute identisch sind.

1.2.2 Entities

Entities sind die Grundobjekte, welche durch ihre eigene Identität definiert werden und nicht durch ihre Attribute. In diesem Projekt sind dies die folgenden Domainobjekte:

- Country
- Manga
- Publisher

- Epiosde
- Author
- Rating

1.2.3 Aggregates

In diesem Objekt sind keine Aggregates vorhanden, da keine Entitätsgruppen gebildet werden müssen. Im Normalfall werden in Aggregates Entitäten in einer logischen Gruppe zusammengefasst.

1.2.4 Repositories

Die Repositories dienen als Schnittstelle zur Persistierungslogik. Hierbei können die Entitäten mit Hilfe dieser Logik persistiert oder aus einer persistierten Zustand zurück in die Anwendung geladen werden.

1.2.5 Application Services

Die Application Services werden verwendet, da die Use-Cases über den Bereich einer einzelnen Entität hinausgehen. Somit wurden diese Operationen in bestimmte Appliciation Services ausgelagert.

2. Clean Architecture

Die Clean-Architecture wurde für das Projekt gewählt, da hierbei die Modularität und die Kapselung der einzelnen Bestandteile der Anwendung im Mittelpunkt stehen. Die Clean Architecture wird mit Hilfe der Schichtenstruktur umgesetzt. Hierbei dürfen innere Schichten nie auf die äußeren Schichten zugreifen, sondern nur die äußeren auf die inneren.

2.1 Schichtenarchitektur

Die Schichtenarchitektur wurde mit Hilfe einzelner Maven Module umgesetzt und der Übersichtlichkeit halber getrennt. Außerdem können so die Abhängigkeiten pro Schicht festgelegt werden, sodass nicht alle Komponenten in jeder Schicht importiert werden müssen.

2.1.1 Domain

In der Domainschicht liegt das Modell der Anwendung aus den Entitäten. Diese Schicht bildet somit die organisationsweite Geschäftslogik ab. Außerdem werden hier die Abstraktionen für die Implementierung des Persistenzframeworks Hibernate gelegt.

2.1.2 Applikation

In der Applikation-Schicht der Anwendung liegen die Use-Cases, welche mit Hilfe von Services implementiert wurden. Diese bilden Sachverhalte und Routinen ab, die über den Zuständigkeitsbereich einer einzelnen Entität hinausgehen. Hierbei wird außerdem die Logik für z.B. die API-Aufrufe definiert.

2.1.3 Adapters

In dieser Schicht erfolgt die Umwandlung von serialisierten Objekten in die eigentlichen Domänenentities bzw. von Entität zu DTO. Diese Funktion wird mit Hilfe von sogenannten Mappern umgesetzt. Diese Schicht verarbeitet somit die entgegengenommenen Objekte und wandelt diese in ein Format, das für die Anwendung lesbar ist.

2.1.4 Plugins

In der letzten Schicht werden die Schnittstellen für die REST-Schnittstelle in Form von Controllern definiert. Außerdem können hier Frameworks importiert werden, die den Entwicklungs-

aufwand reduzieren oder bestimmte Funktionen implementieren. Im Projekt wurde hier die API-Test-Schnittstelle Swagger importiert, sowie Hibernate um mit der H2 Datenbank Entitäten auch über einen Neustart der Anwendung hinweg persistieren zu können.

3. Programming Principles

3.1 SOLID Prinzipien

Im folgenden Anbschnitt werden die SOLID-Prinzipien erläutert.

3.1.1 Single-Responsibility

Das Single-Responsibility-Prinzip besagt, dass eine Klasse nur eine Verantwortlichkeit haben soll. Dadurch wird der Code leichter verständlich und leichter wartbar, da er gezwungenermaßen modularer aufgebaut ist und dadurch leichter verständlich ist, was wo passiert. Dieses Prinzip wird in der gesamten Anwendung verwendet, zum Beispiel auch in den Services, die jeweils nur die Use-Cases für ein einzelnes Entity abbilden.

3.1.2 Open-Closed-Principle

Nach dem Open-Closed-Prinzip soll eine Klasse offen für Erweiterungen, aber geschlossen gegenüber Modifikationen sein. Das Verhalten einer Klasse darf erweitert, aber nicht verändert werden. Dieses Prinzip hilft, Fehler in schon fertigen Codeteilen zu vermeiden. Wenn eine Erweiterung nur durch Änderungen innerhalb einer Klasse erreicht werden kann, ist die Gefahr sehr groß, dass durch die Änderung schon fertig implementierte Funktionen neue Fehler bekommen. Dieses Prinzip ist zum Beispiel durch die Verwendung von Interfaces für die Repositories befolgt. Diese garantieren, dass neue konkrete Implementierungen hinzugefügt werden können, ohne den bestehenden Code anpassen zu müssen.

3.1.3 Liskov'sche Substitution

Das Liskovsche Substitutionsprinzip fordert, dass abgeleitete Klassen immer anstelle ihrer Basisklasse einsetzbar sein müssen. Subtypen müssen sich so verhalten wie ihr Basistyp. Auch dieses Prinzip ist durch die Verwendung der Repository-Interfaces gegeben. Die konkreten Implementierungen der Repositories lassen sich austauschen, ohne die Funktionsweise des restlichen Code zu beeinflussen.

3.1.4 Interface-Segregation

Das Interface-Segregation-Prinzip besagt, dass ein Client nicht von den Funktionen eines Servers abhängig sein darf, die er gar nicht benötigt. Ein Interface darf demnach nur die Funktionen

enthalten, die auch wirklich eng zusammengehören. Die Problematik ist, dass durch „fette“ Interfaces Kopplungen zwischen den ansonsten unabhängigen Clients entstehen. Das Prinzip wurde befolgt, indem die verschiedenen Repository-Interfaces nach Entity aufgeteilt sind und nur die Methoden vorgeben, die unbedingt benötigt und auch verwendet werden.

3.1.5 Dependency Inversion

Das Dependency-Inversion-Prinzip besagt, dass Klassen auf einem höheren Abstraktionslevel nicht von Klassen auf einem niedrigen Abstraktionslevel abhängig sein sollen. Dabei geht es nicht darum, die Abhängigkeiten einfach umzudrehen. Abhängigkeiten zwischen Klassen soll es nicht mehr geben; es sollen nur noch Abhängigkeiten zu Interfaces bestehen (beidseitig). Interfaces sollen nicht von Details abhängig sein, sondern Details von Interfaces. Auch dieses Prinzip wird durch die Repository-Interfaces befolgt.

3.2 GRASP

GRASP steht für *General Responsibility Assignment Software Patterns* und enthält neun Prinzipien, die im Folgenden näher beschrieben werden.

3.2.1 Information expert

Dieses Prinzip besagt, dass zusammengehörige Informationen auch am gleichen Ort liegen und neue Aufgaben von der Klasse übernommen werden, die schon am meisten Wissen über die Aufgabe besitzt. Dies ist zum Beispiel in den Entities mittels Getter und Setter gegeben. Diese prüfen dabei selbst die Integrität der eigenen Attribute, da sie selbst am meisten darüber wissen.

3.2.2 Creator

Das Creator-Prinzip besagt, dass neue Instanzen einer Klasse von derjenigen Klasse erzeugt werden sollten, die eng mit der Klasse zusammenarbeiten. Das Prinzip wird im Projekt nicht explizit befolgt, da nur an wenigen Stellen Objekte instanziiert werden, wodurch hier keine Unübersichtlichkeit besteht.

3.2.3 Controller

Controller sind in Form von REST-Controllern in den Plugins angesiedelt und werden dazu genutzt, um "Benutzereingaben" von der tatsächlichen REST-Schnittstelle zu abstrahieren.

3.2.4 Low coupling

Hier sollen Abhängigkeiten zwischen verschiedenen Klassen möglichst gering gehalten werden. Dadurch ist der Code leichter anpassbar, wartbar und auch wiederverwendbar. Dies wird zum Beispiel durch die Repository-Interfaces erreicht. Durch diese bestehen keine Abhängigkeiten von den konkreten Implementierungen, sondern nur vom Interface.

3.2.5 High cohesion

Eine Klasse soll nicht Verschiedene Aufgaben übernehmen, da dadurch die Komplexität steigen würde. Hat jede Klasse nur eine einzige fest definierte Aufgabe, wird der Code leichter lesbar und leichter wartbar. Zudem wird durch High Cohesion auch Low Coupling unterstützt. Das Prinzip der High Cohesion wird durch das ganze Projekt hindurch verwendet, indem Klassen möglichst kurz gehalten sind und dadurch nur genau eine Aufgabe erfüllen.

3.2.6 Polymorphism

Das Prinzip des Polymorphismus besagt, dass eine Instanz einer Klasse durch Instanzen von Child-Klassen oder Instanzen von Klassen, die das gleiche Interface implementieren, ersetzbar sein müssen. Durch das Ersetzen darf der restliche Code nicht fehlerhaft werden. Dies ist zum Beispiel bei den Repository-Interfaces der Fall. Die Implementierungen auf der Plugin-Schicht können dabei gewechselt werden, ohne die restliche Anwendung zu beeinflussen.

3.2.7 Pure fabrication

Pure Fabrication sind Klassen, die in der Problemdomäne nicht existieren. Dazu gehören zum Beispiel die DTOs in der Adapter-Schicht. Sie existieren nur in der Anwendung für geringere Kopplung und bessere Verwendbarkeit, nicht in der eigentlichen Domäne.

3.2.8 Indirection

Indirection beschreibt, dass zwei Klassen nicht direkt voneinander abhängen, sondern nur über einen Vermittler. Dies ist zum Beispiel durch die Application Services gegeben, da der Rest der Anwendung dadurch nicht direkt mit dem Repo arbeitet.

3.2.9 Protected variations

Durch Protected Variations wird bestehender Code vor Änderungen an anderem Code geschützt. Dies wird zum Beispiel durch Interfaces erreicht, da sie sicherstellen, dass sich die Schnittstelle (z.B. die Methodensignatur) nicht ändern dürfen.

3.3 DRY

Das "Don’t Repeat Yourself" Prinzip wird im ganzen Projekt befolgt, da Methoden kurz gehalten wurden. Zudem wurde wo immer möglich Teillogik in eigene Methoden extrahiert. Dadurch wird der Code wiederverwendbar und muss nicht kopiert werden. Dadurch müssen Änderungen nur an einer Stelle durchgeführt werden und der Code wird leichter lesbar und verständlich.

4. Refactoring

In diesem Kapitel sollen die Refactorings betrachtet werden.

4.1 Identifizieren von Codesmells

Hierbei wurde das Sonar Lint Plugin verwendet, um die bestehenden Code-Smells des Projektes anzuzeigen. Dabei wurden die gefundenen Fehler mit den RISC Nummern abgeglichen, um einen Lösungsansatz zu bekommen.

4.1.1 Codesmell 1

Abbildung 4.1: Codesmell 1 - Unnötige Imports löschen

Der erste Codesmell befasst sich mit den bei der Entwicklung importierten Bestandteilen. Bei der Entwicklung werden oft verschiedene Packages importiert, die wegen Codeänderungen am Ende gar nicht verwendet werden. Dieser Code-Smell mit der Kennung „java:S1700“ soll nun behoben werden.

Begründung

Diese Imports stellen ein Sicherheitsrisiko für die Anwendung dar und tragen außerdem nicht zur besseren Lesbarkeit des Codes bei, deshalb sollen sie entfernt werden.

Fix

Diese Imports müssen schlichtweg aus den implementierten Klassen entfernt werden. Dies kann manuell oder mit dem Refactoring Tool der IDE umgesetzt werden. Dieses Refactoring wurde im Commit „70c0170a518290b3449e383bd04e1c7e99e740fd“ umgesetzt.

4.1.2 Codesmell 2

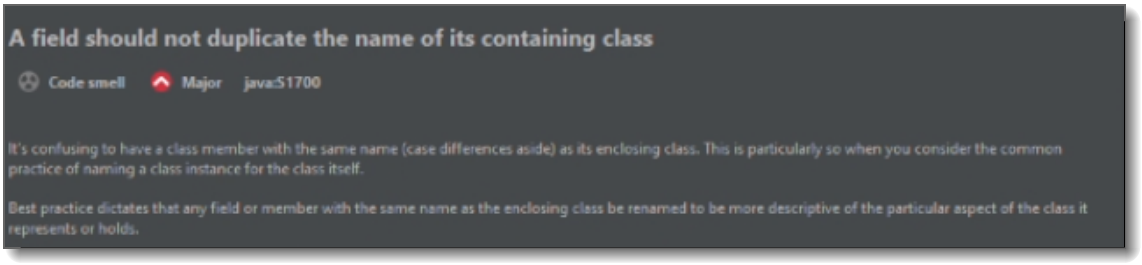

Abbildung 4.2: Codesmell 2 - Variablennamen gleich dem Klassennamen

Der zweite Codesmell bezieht sich auf die Namen von Variablen in Klassen. Hierbei hieß eine Variable wie die Klasse, in der sie verwendet wurde. Das ist nach java:S17000 ein schwerwiegender Codesmell

Begründung

Wenn das Attribut wie die Klasse heißt, verwirrt das den Entwicker und sollte neu benannt werden.

Fix

Umbenennung des Attributes wurde in Commit „d4bb6279cfc335f8967256a795db978ae6ad6930“ umgesetzt.

5. Entwurfsmuster

Im Projekt wurde das Bridge Pattern verwendet. Hierbei werden die Interfaces, welche in der Domain-Schicht entwickelt wurden mit Hilfe des Persistierungsformeworks Hibernate implementiert. Dabei wird die Implementierung von der Abstraktion entkoppelt. Dies hat zur Folge, dass die Persistierungslogik leicht ausgetauscht werden kann, wenn die neue Persistierungslogik die definierten Operationen besitzt.

5.1 UML Vorher

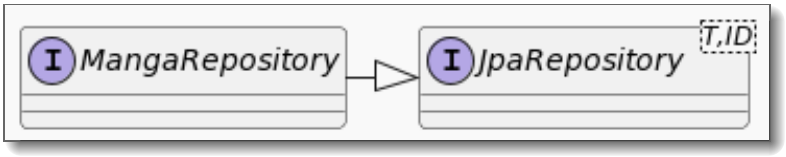

Abbildung 5.1: Vor der Benutzung des Bridge Patterns

5.2 UML Nachher

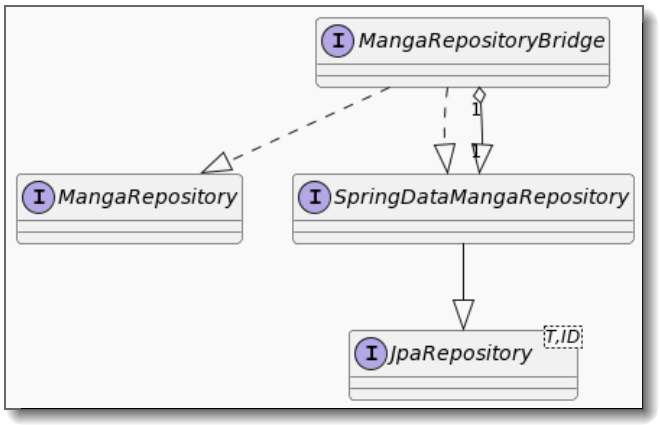

Abbildung 5.2: Nach der Benutzung des Bridge Patterns