Projektdokumentation Seriephant

Programmentwurf

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

an der

Dualen Hochschule Baden-Württemberg Karlsruhe

von

Kristin Agne

Abgabedatum 16. Mai 2022

Matrikelnummer	TBD
Kurs	TINF19B4
Bearbeitungszeitrum	5. & 6. Semester
Gutachter der Studienakademie	Mirko Dostmann

Inhaltsverzeichnis

Inhaltsverzeichnis **ii**

Abbildungsverzeichnis **iii**

Codeverzeichnis **iv**

Abkürzungsverzeichnis **v**

---

1 Domain Driven Design **1**
1.1 Ubiquitous Language 1
1.2 Analyse und Begründung der verwendeten Muster 2

2 Clean Architecture **5**
2.1 Schichtenarchitektur 5
2.2 Frontend 7

3 Programming Principles **8**
3.1 SOLID 8
3.2 GRASP 9
3.3 DRY 12

4 Refactoring **13**
4.1 Identifizieren von Codesmells 13

5 Entwurfsmuster **16**
5.1 Begründung des Einsatzes 16
5.2 Unified Modeling Language (UML) Vorher 16
5.3 UML Nachher 17

A Anhang **I**
A.1 Ubiquitous Language I

Abbildungsverzeichnis

4.1 Code Smell "Long method"before fix 13
4.2 Code Smell "Long method"before fix 14
4.3 Code Smell "Long method"before fix 14
4.4 Code Smell "Long method"before fix 15

5.1 UML Diagram ohne Benutzung des Bridge Pattern 17
5.2 UML Diagram mit Benutzung des Bridge Pattern 18

A.1 Gemeinsame Projektsprache von Domänenexperten und Entwickler I

Liste der Algorithmen

Abkürzungsverzeichnis

UML	Unified Modeling Language	ii
DDD	Domain Driven Design	1
UL	Ubiquitous Language	1

1. Domain Driven Design

Unter Domain Driven Design (DDD) ist eine Philosophie für die Modellierung komplexer Software. Der Fokus liegt hierbei auf dem präzisen und tiefgreifenden Verständnis der Problemdömane. Desweiteren bietet es Muster, um ein konsistentes, erweiterbares, funktionierendes Modell der Problemdomäne zu entwickeln.

1.1 Ubiquitous Language

Die Ubiquitous Language (UL) ist das wichtigste Konzept des DDD. Es bezeichnet die von Dömanenexperten und Entwicklern gemeinsam im Projekt verwendete Sprache. Dadurch sollen gleiche Begriffe in der Domäne sowie in dem Sourcecode verwendet werden. Durch die UL sollen die Verständnisschwierigkeiten zwischen Domänenexperten und Entwicklern minimieren. Durch UL ist der Code näher an der Sprache der Domäne. Als UL bezichnet man den Teil des Projektes, der sich von den Entwicklern und den Domänenexperten überschneiden (siehe Anhang A.1).

1.1.1 Analyse

Die Fachdomäne ist in diesem Projekt sehr überschaubar. Damit in dem Projekt keine Sprachprobleme aufkommen, wurde als Sprache Englisch gewählt, da diese international verständlich ist. Die Fachbegriffe der einzelnen Objekte und deren Bedeutung sowie Attribute wurde im Voraus wie folgt festgelegt:

Genre

Unter einem Genre wird eine Kategorie verstanden, die einer Serie zugeordnet werden kann. Diese sind global definiert.

Actor

Unter einem Actor versteht man einen Schauspieler. Dies ist eine Person, die in einzelnen Episoden mitspielen kann.

Serie

Eine Serie ist eine Fernsehshow, bei der regelmäßig Folgen von Sendungen ausgestrahlt werden.

Season

Eine Season stellt eine Staffel von einer Serie dar. Hierbei werden Episoden thematisch gegliedert in Handlungsabschnitte, und alle Folgen werden in zeitlicher Nähe ausgestrahlt.

Episode

Eine Episode stellt eine einzelne Folge da, die einer Staffel und somit auch einer Serie zugeordnet ist. Eine Episode kann Schauspieler, die in ihr mitspielen, speichern sowie Nutzer, die die Episode angeschaut haben, und deren Ratings.

User

Ein User stellt Personen da, die Episoden anschauen und bewerten können. Außerdem können Nutzer auch neue Serien, Seasons und Episodes erstellen.

Rating

Ratings sind Bewertungen von Nutzern zu einzelnen Episoden.

1.2 Analyse und Begründung der verwendeten Muster

1.2.1 Value Objects

Value Objects sind sogenannte Wertobjekte, die unveränderbar sind, da sie einen bestimmten Wert bzw. eine bestimmte Kombination aus Werten repräsentieren. Um Werte eines Value Objects zu ändern, muss ein neues erstellt werden.

Analyse

In diesem Projekt werden Value Objects dafür benutzt, Daten vom Frontend zum Backend und wieder zurück zu schicken. Bei dem Aufruf eines Create oder Update Endpunkts wird vom Frontend ein value object mitgeschickt. Auch vom Backend bekommt man ein Value Object zurück. In diesem Projekt werden Value Objects von den „DTO“-Klassen repräsentiert und sind im Modul „1 - Adapters“ zu finden. Ein weiteres Beispiel für ein Value Object in dem Projekt ist der Key des Ratings, der als Embeddable umgesetzt wurde. Dieser Key setzt sich aus der Id des Users und aus der Id der Episode zusammen und ist der Primary Key für das Rating. Dieser Key kann nicht verändert werden.

Begründung

Der Grund für die Benutzung von Value Objects ist, dass so viele Daten vom Frontend zum Backend geschickt werden können, ohne dass sie verändert werden können. Die Serialisierbarkeit wird durch das Verwenden von Value Objects sichergestellt. Da es bei der Create oder Update Methode eine Vielzahl von Parametern gibt, ist eine Kapselung in einem Value Object übersichtlicher. Der Key ist ein Value Object, das nicht verändert werden darf, weil die Entität Rating an den Primary Key gekoppelt ist. Könnte man den Key ändern, würde das Rating Objekt nicht mehr richtig sein.

1.2.2 Entities

Entitäten stellen die Basis eines Modells dar. Sie sind Objekte, die durch ihre Identität definiert werden und Attribute besitzen. Verschiedene Instanzen einer Entität können die gleichen Eigenschaften haben. Attribute könne geupdatet werden, hierbei verändert dich die Instanz nicht.

Analyse

In diesem Projekt gibt es sieben Entitäten (Attribute):

- Genre (Title, Description)
- Actor (FirstName, LastName)
- Serie (Title, Description, Genre, List<Season>)
- Season (SeasonNumber, Serie, List<Episode>)
- Episode (Title, EpisodeNumber, Season, List<Actor>, List<User>, List<Rating>)
- User (FirstName, LastName, List<Episode>, List<Rating>)
- Rating (User, Episode, Rating)

Jede Entität wird mit einer Id identifiziert. Die Entitäten haben Eigenschaften, die sich ändern können. In diesem Projekt sind die Entitäten im Modul „3 - Domain“ zu finden.

Begründung

Während der Laufzeit ändern sich die Attribute dieser Objekte regelmäßig. Daher ist es sinnvoll, diese Objekte als Entitäten umzusetzen, zu persistieren, bei Bedarf Attribute zu updaten und anschließend erneut zu persistieren.

1.2.3 Aggregates

Aggregates umfassen mindestens eine Entität, im sogenannten Aggregatstamm. An diesem können weitere (Unter-)Entitäten oder Objekte angehängt werden, um sogenannte Aggregates zu bilden.

Analyse

Aggregates finden in diesem Projekt Anwendung bei der Klasse RatingAverage. Hierbei werden Ratings und Episoden zusammengefasst um einen Durchschnitt oder die Anzahl der Bewertungen anzeigen zu lassen.

Begründung

Grund für die Benutzung des Aggregate ist, dass die verschiedenen Entitäten zusammengefasst werden können. So kann mit den Daten besser interagiert werden, der Durchschnitt und die Anzahl an Bewertungen kann im Aggregate Objekt gespeichert werden und ans Frontend weitergegeben werden.

1.2.4 Repositories

Repositories sind Klassen, die eine Schnittstelle zwischen Domäne und der Datenzuordnungsebene bilden. Diese dienen meist dazu die Basisoperationen CRUD zu implementieren. Hier lassen sich Entitäten erstellen, lesen, updaten oder löschen.

Analyse

Repositorys sind im Backend, im Modul „0 - Plugins“ zu finden. Diese werden mit Hilfe der JPA-Repository Schnittstelle implementiert, um auf die von Hibernate ORM geschaffenen Strukturn zuzugreifen. Hier wird das Objekt sowie die Id festegelegt. Mit Hilfe der Repository-Interfaces in der Domänenschicht und Implementierung in der Pluginsschicht kann zwischen den Schichten kommuniziert werden.

Begründung

Hierbei wird das ORM-Framework Hibernate verwendet, um die definierten Operationen von Repositories zu implementieren. Diese Implementierung implementiert automatisch die grundlegenden Funktionen und kann um weitere, eigene, Funktionen erweitert werden. Dadurch können Fehler vermieden werden, und man hat eine klare Trennung der Logik zu den Operationen mit den Daten.

1.2.5 Domain Services

Domain Services bilden die Prozesse ab, welche über den Verantwortungsbereich einer Domäne hinaus gehen. Diese werden als eigenständige Services ausgelagert.

Analyse

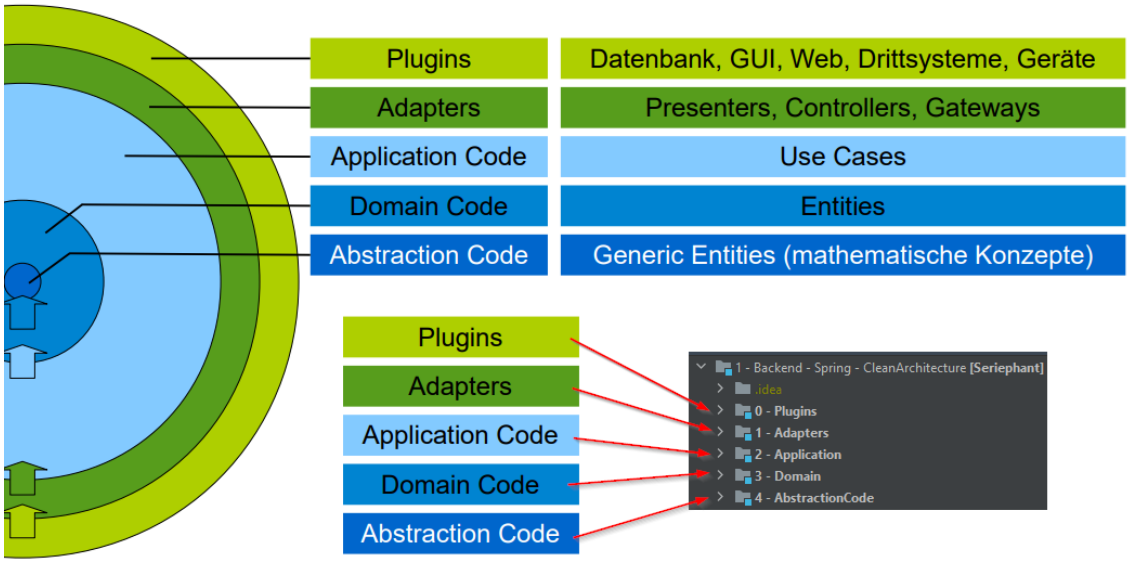
Die Services in diesem Projekt sind im Backend zu finden. Hierbei ist für jede Entität ein entsprechender Service vorhanden, der die Operationen implementiert. Konkret liegen diese Services im Modul „2 - Application“ im Package »de.dhbw.ase.seriephant« - Package

Begründung

Da es viele Operationen gibt, die über den Zuständigkeitsbereich einer Entität oder eines Value Objects hinausgehen, müssen diese Prozesse in einer eigenen Klasse implementiert werden. Diese steht hierbei zwischen dem eigentlichen Modell, der Domäne, und den Controllern in der Plugins-Schicht.

2. Clean Architecture

Die Clean Architecture beschreibt eine bestimmte Methode, wie Software aufgebaut werden kann. Hierbei werden im weiteren 5 Schichten analysiert, welche im Projekt umgesetzt wurden.



2.1 Schichtenarchitektur

Das Projekt soll sich an der Schichtenarchitektur orientieren. Hierbei sollen die einzelnen Module die einzelnen Schichten darstellen. Der Grund für die Benutzung einzelner Schichten ist die fachliche Unabhängigkeit der Anwendung von der sonstigen Infrastruktur. Dadurch können die einzelnen Teile leichter wiederverwendet, getestet und weiterentwickelt werden. Es ist auch leichter, einzelne Komponenten der Infrastruktur auszutauschen.

2.1.1 Abstraction - Schicht

Diese Schicht stellt den Kern der Applikation dar. Hier werden Fundamente, oft durch die verwendete Programmiersprache geschaffen. Ein Beispiel hierfür ist die Implementierung des „Strings“ in Java, welche in der Spring Boot Anwendung verwendet wird.

Implementierung im Projekt

Diese Schicht wird durch Spring Boot,welches auf Java aufbaut, implementiert.

2.1.2 Domain - Schicht

In der Domain Schicht liegen die Business Objekte der Anwendung. Hier werden unternehmensweite Geschäftslogiken implementiert.

Implementierung im Projekt

In dieser Schicht liegen die Domänenobjekte, die Entities (Abschnitt 1.2.2).

2.1.3 Application - Schicht

In der Applikationsschicht liegen die Services, welche die Use-Cases implementieren.

Implementierung im Projekt

Die Use-Cases wurden in dieser Schicht implementiert, da sie über die Verantwortlichkeit einer einzelnen Entität hinausgehen. Diese Logiken und Prozesse wurden in sogenannte Services ausgelagert.

2.1.4 Adapters - Schicht

In der Adapterschicht erfolgt die Umwandlung von Business-Objekten (interne Objekte) in Transferobjekten (externe Objekte).

Implementierung im Projekt

Hierzu werden Mapper verwendet, welche eine einfache Konvertierung von DTOs zu Entitäten und umgekehrt zulassen.

2.1.5 Plugins - Schicht

In der Plugin-Schicht werden Frameworks verwendet. Diese unterstützen bestimmte Programmfunktionen, ohne diese selbst zu entwickeln und somit den selbt-geschriebenen Code zu minimieren.

Implementierung im Projekt

In diese Projekt sind mehrere Frameworks im Einsatz.

- Persistierungsframework –> Hibernate
- Debug-SQL-Console –> H2 Console
- API-Dokumentation & Tests –> Swagger

2.2 Frontend

Im Frontend mussten keine Prinzipien der Clean Architecture verwendet werden. Hier kommt React zum Einsatz, um die visuelle Darstellung der Use-Cases zu implementieren.

3. Programming Principles

3.1 SOLID

SOLID steht für weiter fünf Prinzipien, die im folgenden betrachtet werden.

3.1.1 Single Responsibility

Analyse

Das Prinzip wird bei den Repository Klassen eingesetzt. Hierbei hat jedes Repository nur eine Funktion. Diese Funktion ist die Schittstelle zur Datenbank.

Begründung

Der Grund dafür ist, dass eine Klasse nur eine Zuständigkeit haben soll. Es kapselt so die Zuständigkeit, diese sind klar definiert, da jede Klasse nur einen Nutzen hat. Das führt dazu, dass Änderungen der Software nur zu Änderungen in einem eingegrenzten Bereich führen.

3.1.2 Open Closed Principle

Analyse

Hierbei geht es darum, dass Objekte einfach erweiterbar sind, ohne sie grundsätzlich zu ändern. In dem Projekt kommt dieses Prinzip zum Beispiel in den Services oder den Controllern zum Einsatz. Hierbei kann der Controller um einen Endpunkt erweitert werden, ohne dass die Klasse grundlegend modifiziert werden muss. Auch bei den Services trifft dies zu.

Begründung

Der Grund für die Benutzung dieses Prinzips ist, dass eine Erweiterung an Funktionen oft vorkommt. Wenn man jedes Mal die Klasse von Grund auf ändern müsste, wäre das sehr aufwendig. Durch das Open Closed Principle kann man neue Features ohne großen Aufwand hinzufügen.

3.1.3 Liskov Substitution Principle

Unter dem Liskov Substitution Principle wird zusammengefasst, dass eine Instanz einer Klasse durch eine Instanz einer anderen Klasse, die entweder von dieser Klasse erbt oder das gleiche

Interface erweitert, ersetzt werden kann. Hierbei darf es nicht zu einem logischen Bruch innerhalb der Anwendung kommen. In diesem Projekt gibt es zwar keine Vererbung, aber es wurden Interfaces benutzt. Dieses Prinzip kommt also bei den Repository Interfaces zum Einsatz. Die Bridge Klassen (z.B. GenreBridge) implementiert hierbei das Repository Interface und jede Instanz der Bridge Klasse kann durch eine Instanz ersetzt werden, die das gleiche Interface erweitert. Dies wird mit dem Liskov Substitution Principle umgesetzt, um die implementierte Logik in der Bridge Klasse einfach austauschen zu können, ohne alle Klassen anpassen zu müssen, die von den Repository Interfaces abhängig sind (wie die Application Services).

3.1.4 Interface Segregation Principle

Analyse

Das Interface Segregation Principle wurde in den Repository Interfaces verwendet. Hierbei wurden keine unnötigen Schnittstellenteile implementiert.

Begründung

Der Grund hierfür ist, dass Schnittstellen so kleiner sind und nur verwendete Methoden implementieren. Dadurch wird die Wartbarkeit erheblich verbessert und der Code wird kompakter und besser wiederverwendbar. Die implementierten Klassen gewinnen so an Übersichtlichkeit, fachlich zusammengehöriges Verhalten wird so gruppiert und die Modularität wird dadurch verbessert.

3.1.5 Dependency Inversion Principle

Analyse

Das Dependency Inversion Principle kommt vorallem bei objektorientierten Entwürfen zum Einsatz und strukturiert das Projekt in Module. Die einzelnen Module stellen hierbei verschiedenen Ebenen dar. Die Ebenen haben verschiedene Ordnungen, und sind abhängig voneinander. In diesem Projekt ist die niedrigste Ebene die Plugin-Schicht (siehe Kapitel 2). Module niedriger Ebenen sind immer von Modulen höherer Ebenen abhängig, da Module niedrigerer Ebenen die Vorgänge der höheren Ebene spezifizieren.

Begründung

Der Grund, warum man die verschiedenen Ebenen in Module aufteilt und somit eine Abhängigkeit von niedrigen zu höheren Ebenen realisiert, ist, dass man so dem Prinzip der Hierarchie nicht widersprechen kann. Des Weiteren wird so die Vorstellung der Clean Architecture zwangsweise umgesetzt und Veränderungen in der Architektur oder des Design sind leicht umsetzbar. So wird auch Komplexität verringert.

3.2 GRASP

GRASP steht für General Responsibility Assignemnt Software Patterns. Darunter werden viele Muster vertsanden, wovon im folgenden zwei näher betrachtet werden, die voneinander abhängen.

3.2.1 Creator

Unter dem Erzeuger-Prinzip versteht man das Prinzip, dass eine Instanz einer Klasse nur von einem Experten A erzeugt wird. In diesem Projekt findet das Creator-Pattern keinen Einsatz.

3.2.2 Controller

Das Controller-Pattern findet in dem Projekt in den Controllern in der Plugins-Schicht Anwendung. Sie sind dafür zuständig, um die Benutzereingaben zu empfangen und zu verarbeiten. Ein Request wird mit Parametern von einem Controller entgegengenommen und von diesem wird die entsprechende Methode aufgerufen.

3.2.3 Indirection

Das Indirection Pattern wird verwendet, um eine low coupling umzusetzen. Es wird ein sogenannter Vermittler zwischen Client und Server eingesetzt. In diesem Projekt findet dieses Pattern keinen Einsatz, da die Leistungsfähigkeit so vermindert wird.

3.2.4 Information expert

Assign a responsibility to the information expert– the class that has the information necessary to fulfill the responsibility. Bei dem Information Expert Pattern geht es darum, dass eine Klasse nur genau die Informationen kennt, die sie auch benötigt, um ihrer Verantwortung gerecht zu werden. Das heißt, eine Aufgabe wird immer von der Klasse übernommen, die das meiste benötigte Wissen besitzt. Dieses Pattern fördert high cohesion. Umgesetzt wird dieses Pattern in den Entitäten und Repositories. Die Entitäten kapseln das Wissen, sodass nur mit Gettern und Settern darauf zugegriffen werden kann. Bei den Repositories übernimmt immer genau das Repository die Aufgabe, welches das meiste Wissen hat.

3.2.5 Polymorphism

Das Polymorphism-Pattern wird in diesem Projekt nicht umgesetzt, da es keine Vererbung in diesem Projekt gibt.

3.2.6 Protected variations

Das Protected variations pattern wird in diesem Projekt durch die Repository Interfaces in der Domänenschicht, die in der Plugins Schicht implementiert werden, umgesetzt. Dadurch wird die eigentliche Implementierung in der Plugins Schicht über die Interfaces verschleiert. So hat eine Veränderung an der Implementierung keine Auswirkung auf das restliche System.

3.2.7 Pure Fabrication

Das Pure Fabrication Pattern stellt Klassen da, die in der Dömane nicht existieren. In diesen Klassen werden Methoden ausgelagert und die Klassen werden als Erfindung bezeichnet, da die nichts reales aus der Domänenschicht repräsentieren. Dieses Pattern wird zur Unterstützung von high cohesion und low coupling eingesetzt. In diesem Projekt wird Pure Fabrication durch die Repositories umgesetzt.

3.2.8 Low coupling and high cohesion (Service und Controller)

Im folgenden wird die Betrachtung von Low Coupling und High Cohesion in den Services und Controllern betrachtet.

Low Coupling

Analyse
Bei dem Prinzip Low Coupling geht es um die Minimierung der Abhängigkeiten einer Klasse von der Umgebung. Hierbei sollten die einzelnen Klassen und Objekte möglichst wenig untereinander vernetzt sein, sodass die Abhängigkeiten so gering wie möglich gehalten sind. Dieses Prinzip wurde in einigen ApplicationServices umgesetzt, aber nicht in allen. Der ActorApplicationService oder der GenreApplicationService sind jeweils nur von einem Repository, von dem ApplicationRepository bzw. dem GenreRepository abhängig. Hier ist das Prinzip des Low Coupling umgestezt. Im Gegensatz dazu wurde das Prinzip im EpisodeApplicationService nicht umgesetzt, da hier eine Abhängigkeit zu vier Repositories besteht. Hier spricht man von High Coupling.

Begründung
Der Grund für das Einsetzen von Low Coupling ist die leichte Anpassbarkeit. Durch die geringen Abhängikeiten können ActorApplicationService und GenreApllication im Gegensatz zum EpisodeApllicationService leicht angepasst werden. Auch die Verständlichkeit der beiden Klassen mit Low Coupling ist einfacher und man kann die beiden Klassen besser wieder verwenden. Zusätzlich ist das Testen von Klassen mit Low Coupling einfacher, da man nur auf wenige Abhängigkeiten achten muss.

High Cohesion

Analyse
Bei dem Prinzip High Cohesion geht es um die Vermeidung von verschiedenen Verantwortlichkeiten bzw. Aufgaben innerhalb einer Klasse. Hierbei wird betrachtet, inwieweit die Objekte und Attribute einer Klasse zusammenarbeiten und wie viel sie über andere Objekte wissen müssen. Im Projekt wurde dieses Prinzip zum Beispiel in den Controllern umgesetzt. Hierbei bekommen die Controller eine Anzahl von Parametern und eine Methode, und sie verarbeiten diese Daten mit Hilfe des Services. Dabei haben die Controller nur eine Art von Verantwortlichkeit und somit ist das Prinzip der hohen Kohäsion hier umgesetzt.

Begründung
Der Grund für die Umsetzung des Prinzips ist, dass man so die Übersichtlichkeit der Klasse deutlich verbessert und die Komplexität minimiert.

Zusammenhang Low Coupling und High Cohesion

Diese zwei Prinzipien stehen hier in Korrelation zueinander. Desto höher die Cohesion wird und desto mehr man Verantwortlichkeiten in andere Klassen auslagert, desto höher wird auch die Abhängigkeit. High Cohesion geht also mit High Coupling einher, beides voll umzusetzen ist schwierig, man sollte ein Mittel finden, bei dem das Coupling möglichst low und die Cohasion möglichst high ist.

3.2.9 Loose Coupling und high cohesion (Domain)

Im Bereich der Domäne verhält sich die Verbindung zwischen Low Coupling und High Cohesion synchron.

Loose Coupling

Analyse
Bei dem Prinzip Loose Coupling geht es um die Minimierung der Abhängigkeiten einer Klasse von der Umgebung. Betrachtet man die Klassen innerhalb der Domäne, sieht man, dass hier eine Tight Coupling ist. Da die beispielsweise die Klasse Genre eine List<Serie> und jede Serie ein Genre Objekt hat, sind die beiden Klassen gekoppelt.

Begründung
Der Grund für diese Umsetzung und die hohe Kopplung ist, wie in Spring die Entities angelegt wurden und wie mit deren Beziehungen umgegangen wird. Hierbei werden komplette Objekte anstelle von ihren Id's referenziert.

High Cohesion

Analyse
Bei dem Prinzip High Cohesion geht es um die Vermeidung von verschiedenen Verantwortlichkeiten bzw. Aufgaben innerhalb einer Klasse. Innerhalb der Domäne hätte man aufgrund er Tight Coupling eigentlich eine low cohesion, da durch die hohe Abhängigkeit von der Genre und Serie Klasse Funktionen gegenseitig übernommen werden müssten. Jedoch ist dies durch Spring nicht der Fall, weil das Genre theoretisch nur in der Serie gespeichert ist und beim Persistieren die List<Serie> in der Genre Klasse von Spring automatisch geupdatet wird.

Begründung
Auch hierfür ist der Grund Verbesserung der Übersichtlichkeit der Klasse sowie die Minimierung der Komplexität.

Zusammenhang Loose Coupling und High Cohesion

Normalerweise stehen Loose Coupling und High Cohesion hier synchron in Beziehung, da bei einer geringen Anzahl von Abhängikeit auch eine hohe Kohäsion umgesetzt wird. Allerdings ist durch Spring und die Verwendung von Hibernate hier Tight Coupling umgesetzt, und trotzdem ist noch eine hohe Kohäsion vorhanden.

3.3 DRY

Das Prinzip DRY bedeutet „Don't Repeat yourself". Hierbei handelt es sich um ein Prinzip des Clean Code, da Codeabschnitte nicht wiederholt, sondern ausgelagert und wiederverwendet werden sollen.

3.3.1 Analyse

Diese Prinzip ist ein grundlegegndes Prinzip und wurd prinzipiell in dem gesamten Projekt umgesetzt. Im Kapitel 4 sieht man die Extraktion von Teilen einer Methode in einzelne neue Methoden. Diese neuen Methoden werden anschließend auch von anderen Methoden benutzt, wodurch eine Wiederholung vermieden wird.

3.3.2 Begründung

Diese Prinzip hat mehrere Vorteile, da man so weniger Code schreiben muss und sich somit Zeit spart. Außerdem nimmt die Übersichtlichkeit enorm zu, wenn man gleiche Teile in Methoden auslagert und mehrfach verwendet. Nicht nur die Übersichtlichkeit, sondern auch die Verständlichkeit und Fehleranfälligkeit des Codes wird so verbessert. Desweiteren wird die Wartbarkeit vereinfacht, weil man so bei Logikfehlern oder Veränderungen nur eine Methode und nicht alle Codestellen anpassen muss.

4. Refactoring

4.1 Identifizieren von Codesmells

4.1.1 Code Smell 1

Hierbei handelt es sich um den Code Smell "Long method". Methoden sollen kurz und übersichtlich sein, die folgende Methode ist aber lang und verschachtelt.

```
public User updateUser(User user) throws ValidationException {
    if (user != null
            && user.getId() != null
            && this.episodeRepository.existsById(user.getId())) {
        User foundUser = this.userRepository.getById(user.getId());
        foundUser.setFirstName(user.getFirstName());
        foundUser.setLastName(user.getLastName());
        if (user.getWatchedEpisodes() != null) {
            for (Episode episode : foundUser.getWatchedEpisodes()) {
                this.removeSeenEpisodeOfUser(user.getId(), episode.getId());
            }
            for (Episode episode : user.getWatchedEpisodes()) {
                this.updateSeenEpisodesOfUser(user.getId(), episode.getId());
            }
        }
        return this.userRepository.save(foundUser);
    }
    throw new ValidationException("Id of User is not known.");
}
```

Abbildung 4.1: Code Smell "Long method"before fix

Begründung

Die Methode ist 20 Zeilen lang und sehr verschachtelt. Dadurch ist es schwer, auf den ersten Blick zu erkennen, was die Methode macht. Auch ist die Methode nicht übersichtlich.

Fix

Durch das Auslagern von einzelnen Teilen der Methode wird klarer, was wo passiert und warum. In meinem Projekt gab es einige weitere lange Methoden, die sehr unübersichtlich waren. Durch die Umstrukturierung von einer langen Methode in mehrere kurze werden die einzelnen Teile der Methode übersichtlicher und sind leichter verständlich.

```
public User updateUser(User user) throws ValidationException {
    if (user != null
            && user.getId() != null
            && this.userRepository.existsById(user.getId())) {
        User foundUser = this.userRepository.getById(user.getId());
        foundUser.setFirstName(user.getFirstName());
        foundUser.setLastName(user.getLastName());
        this.updateWatchedEpisodes(user, foundUser);
        return this.userRepository.save(foundUser);
    }
    throw new ValidationException("Id of User is not known.");
}
```

Abbildung 4.2: Code Smell "Long method"before fix

4.1.2 Code Smell 2

Bei diesem Code Smell geht es darum, dass ungenutze private Variablen entfernt werden.

```
@Service
public class ActorApplicationService {
    private final ActorRepository actorRepository;
    private final EpisodeRepository episodeRepository;


    @Autowired
    private ActorApplicationService(ActorRepository actorRepository, EpisodeRepository episodeRepository) {
        this.actorRepository = actorRepository;
        this.episodeRepository = episodeRepository;
    }
```

Abbildung 4.3: Code Smell "Long method"before fix

Begründung

Wenn eine private Variable oder ein Import in der Klasse nicht benutzt werden, sind diese unnötig und verschlechtern nur die Übersichtlichkeit und Fehleranfälligkeit des Codes.

Fix

Für den Fix wurden hierbei einfach die unbenutze private Variablendeklaration entfernt und auch die Zuweisung aus dem Konstruktor wurde entfernt.

Abbildung 4.4: Code Smell "Long method"before fix

5. Entwurfsmuster

Das Objektmuster „Bridge Pattern“ kam in diesem Projekt zum Einsatz.

5.1 Begründung des Einsatzes

Dieses Entwurfsmuster wurde eingesetzt, da so eine Entkopplung von Domänenmodell und Persistierungslogik gelingt. Die Trennung von Domänenmodell und Persistierungslogik ist Teil von Clean Architecture, da so die Persistierung leicht ausgetauscht werden kann. Hierbei erfolgt die Implementierung in Form eines Interfaces im Domänenmodell, welches von einer Bridge mit Hilfe eines ORM-Plugins in der Plugins-Schicht umgesetzt wird.

5.2 UML Vorher

Im folgenden sieht man das UML Diagramm vor der Verwendung des Bridge Patterns. Hierbei hat das Repository als Child von einem JPA Repository direkt auf die Datenbank zugegriffen. Die Persistierungslogik war so nicht getrennt und konnte nur mit einigen Anpassungen ausgetauscht werden.

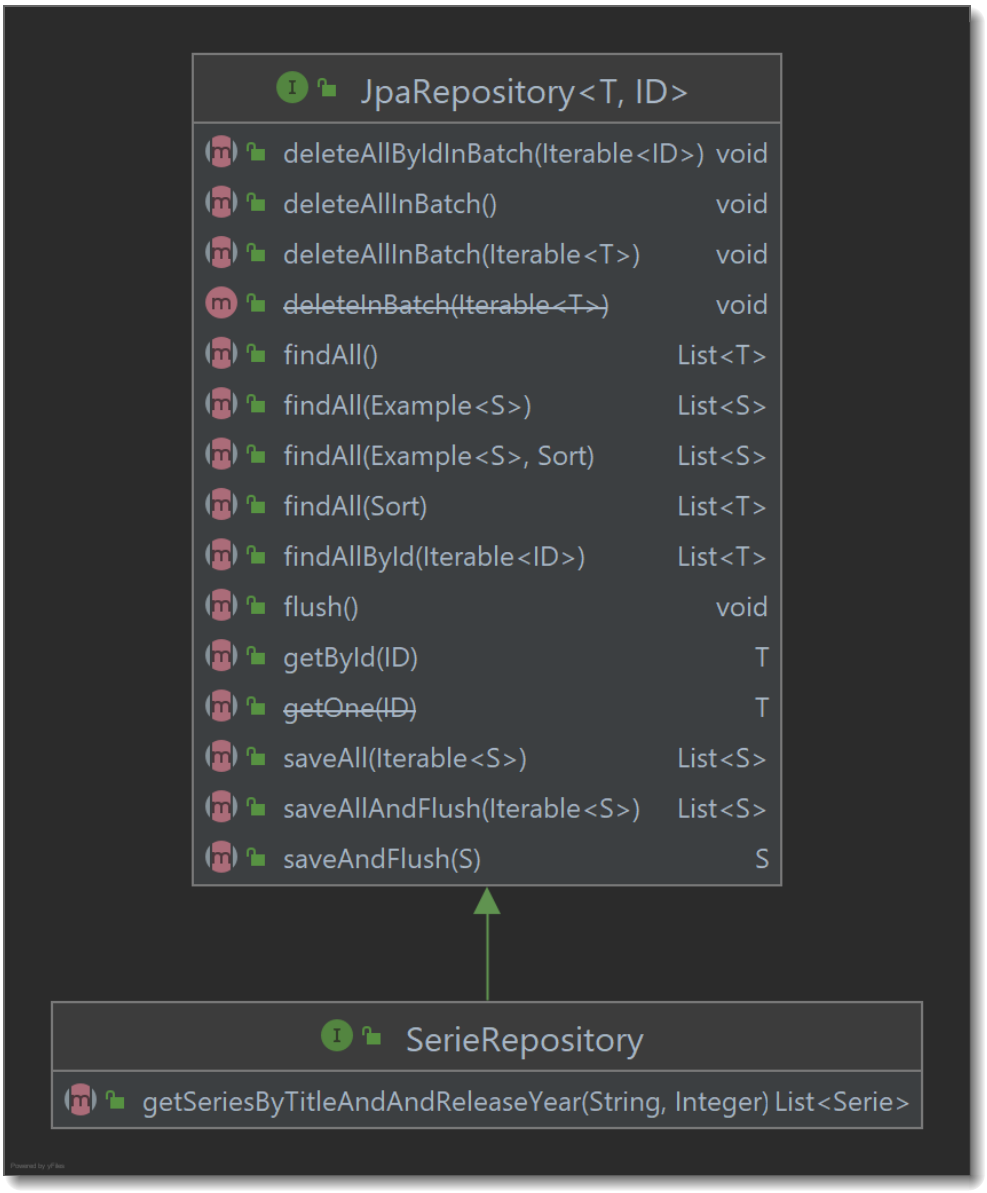


Abbildung 5.1: UML Diagram ohne Benutzung des Bridge Pattern

5.3 UML Nachher

Beim UML Diagramm nach der Verwendung des Bridge Patterns sieht man, das mehr Klassen beteiligt sind. Zwischen Repository und JPARepository sind zum einen die Bridge sowie das Spring DataRepository dazwischen. Dadurch wird die Persistierung nicht mehr direkt, sondern über die Bridge durchgeführt.

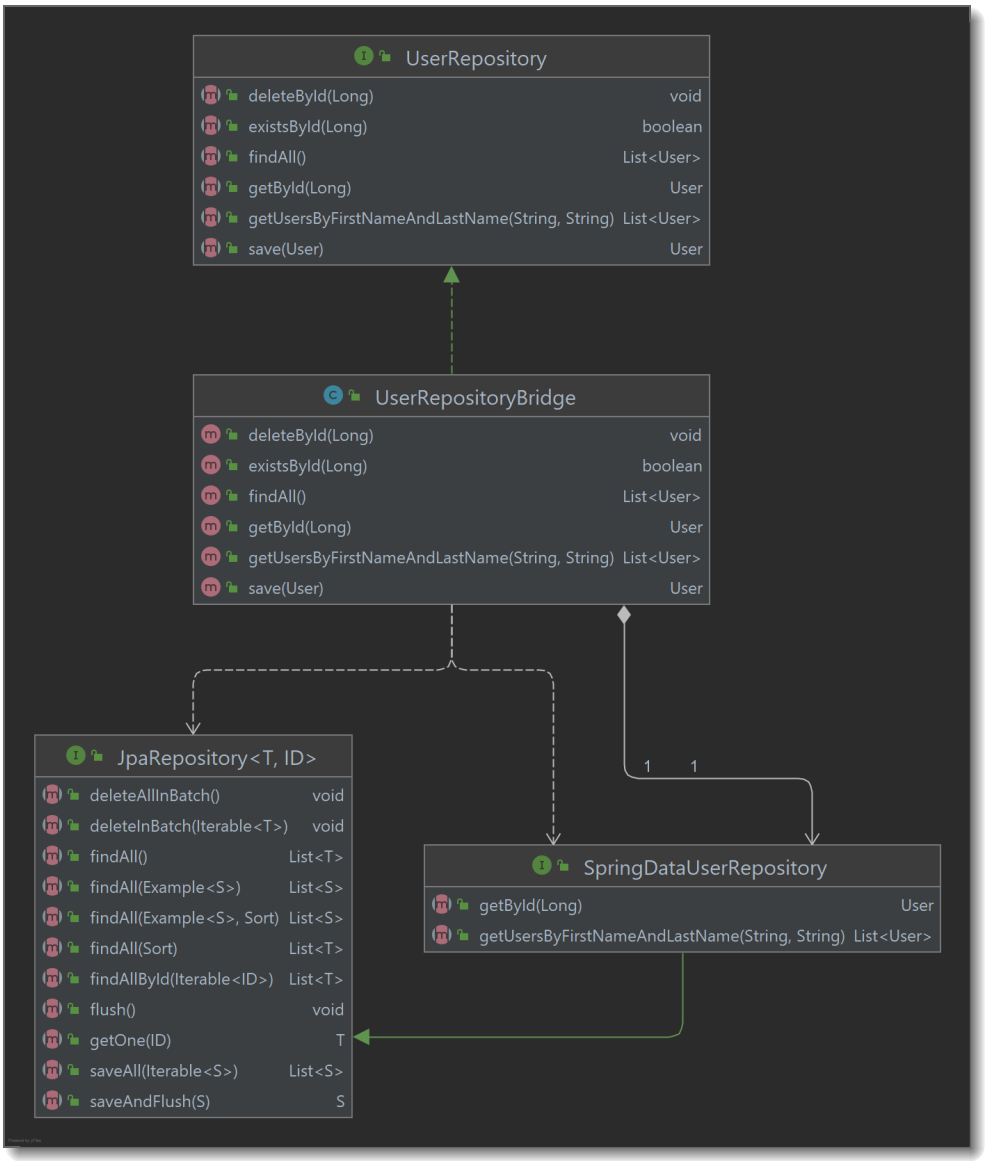


Abbildung 5.2: UML Diagram mit Benutzung des Bridge Pattern

A. Anhang

A.1 Ubiquitous Language

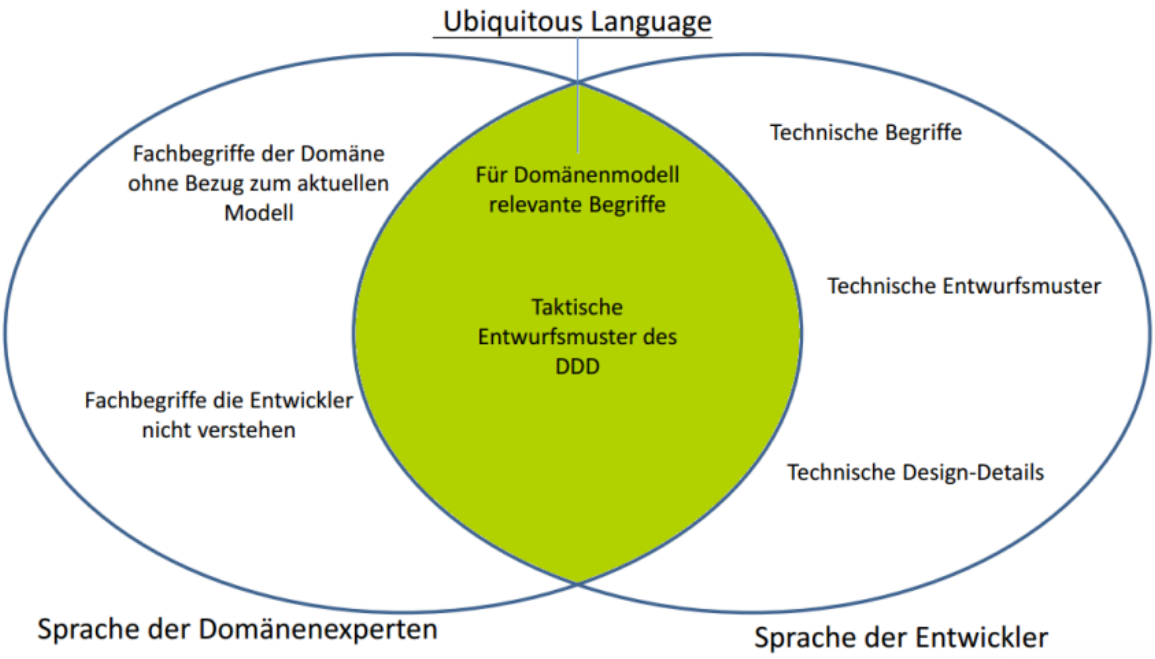


Abbildung A.1: Gemeinsame Projektsprache von Domänenexperten und Entwickler