

the One

PROGRAMMENTWURF

der Vorlesung „Advanced Software Engineering“

an der

Dualen Hochschule Baden-Württemberg Karlsruhe

von

Georg Reichert

Abgabedatum 5. Mai 2022

Kurs
Bearbeitungszeitrum
Gutachter der Studienakademie

TINF19B4
5. & 6. Semester
Mirko Dostmann

Inhaltsverzeichnis

Inhaltsverzeichnis	ii
1 Domain Driven Design	1
1.1 Analyse der Ubiquitous Language	1
1.2 Analyse und Begründung der verwendeten Muster	1
1.2.1 Value Objects	1
1.2.2 Entities	1
1.2.3 Aggregates	2
1.2.4 Repositories	2
1.2.5 Domain Services	2
2 Clean Architecture	3
2.1 Schichtenarchitektur	3
2.1.1 Planung	3
2.1.2 Umsetzung	3
3 Programming Principles	4
3.1 SOLID	4
3.2 GRASP (insb. Kopplung/Kohäsion)	5
3.3 DRY	5
4 Refactoring	6
4.1 Identifizieren von Codesmells	6
4.1.1 Code Smell 1	6
4.1.2 Code Smell 2	7
5 Entwurfsmuster	9
5.1 Begründung des Einsatzes	9

1. Domain Driven Design

1.1 Analyse der Ubiquitous Language

In diesem Projekt soll die Ubiquitous Language möglichst einfach gehalten werden. Hierzu werden die gängigen Geschäftsobjekte definiert und deren Verbindung untereinander klar dargestellt.

Ein Hersteller kommt immer aus genau einem **Land**, jedoch können mehrere Hersteller einem Land zugeordnet werden. Ein Land ist mit dem alltäglichen Land gleichzusetzen und besitzt einen **Namen** sowie eine **Abkürzung**. Ein **Hersteller** kann **endlich viele Parfüms** herstellen und besitzt einen **Namen**, ein **Land**. Ein **Parfüm** hat immer zwingend einen **Hersteller**, ein **Herstellungsjahr**, eine **Basisnote**, eine **Herznote**, eine **Kopfnote**, einen **Namen**, eine **Größe**, einen **Preis** und ist *optional Teil einer Kollektion*. Jede **Note** besteht aus einem **Namen**, sowie einem **Geruch**. Die **Kollektionen** haben ebenfalls einen **Namen** und beinhalten **endlich viele Parfüms**. Ebenfalls können **endliche viele Parfüms** auf eine **Wunschliste** hinzugefügt werden, die einen **Namen** und die **erwähnten Parfüms** hat. Zuletzt kann ein **Rating** erstellt werden, welches das zu bewertende **Parfüm**, die **Rating-Attribute (Duft, Haltbarkeit, Sillage, Flakon & Preis/Leistung)**, sowie einen **Benutzer** beinhalten. Jedes **Parfüm** kann **endlich viele Ratings** besitzen und auf **endlich vielen Wunschlisten** vorhanden sein. Jedoch kann jede Wunschliste und jede Kollektion ein Parfüm nur **einmal** enthalten.

1.2 Analyse und Begründung der verwendeten Muster

1.2.1 Value Objects

„Value Objects“ sind unveränderbare Objektinstanzen. Diese sind im Projekt in Form der *Ressourcen* Klassen implementiert. Diese Ressourcen sind serialisierbare Kopien der Business Objekte / Entities, um mit dem Frontend kommunizieren zu können, welche durch die Mapper erschaffen werden.

1.2.2 Entities

Entities sind die Basisobjekte, die durch ihre Identität definiert werden. Im Projekt sind dies Konkret die Domainobjekte

- Country
- Manufacturer

- Perfume
- Rating
- Collection
- Wishlist
- Note (konkret hierbei Base-,Heart & Headnote)

1.2.3 Aggregates

Ein Aggregat umfasst mindestens eine Entität, die zu einer Gruppe zusammengefasst werden. Das prominenteste Beispiel im Projekt ist das Aggregate *Note*, welches aus den drei verschiedenen Noteklassen besteht.

1.2.4 Repositories

Die Repositories sind in diesem Projekt sowohl in der Domänenschicht (formale Definition) als auch in der Plugins Schicht (konkrete Implementierung) zu finden. Sie umfassen die JPA Repositories, welche durch Hibernate eine Implementierung schaffen um auf die Persistierungsebene zuzugreifen.

1.2.5 Domain Services

Domainservices sind Operationen, die über die Verantwortung einer einzelnen Entität hinausgehen. Diese Services implementieren somit die vorher definierten Use-Cases. Konkret sind diese DomainServices in der Application-Schicht zu finden. Hierbei sind die UC konkret in einem separaten Service von den eigentlichen CRUD Operationen der Entitäten abgetrennt.

2. Clean Architecture

2.1 Schichtenarchitektur

Die Schichtenarchitektur besteht aus fünf Schichten.

2.1.1 Planung

Diese fünf Schichten wurden im Projekt in Form von Modulen umgesetzt. Hierbei wurde strikt darauf geachtet, dass keine innere Schicht auf eine äußere Schicht zugreift.

2.1.2 Umsetzung

Im Projekt wurde jede Schicht umgesetzt.

Abstraction Schicht Diese Schicht ist der Kern der Architektur. Hier wurden die nötigen Packages importiert, die im kompletten Projekt (alle darüberliegenden Schichten) verwendet werden.

Domain Schicht Die Domain-Schicht beinhaltet die Business Objekte und implementiert die organisationsweiten Geschäftslogiken. In dieser Schicht sind somit die Entitäten zu finden.

Applikation Schicht In der Applikationsschicht werden die Use-Cases implementiert. Dabei wurde darauf geachtet, dass die UC von den eigentlichen CRUD Operationen getrennt werden, um das Projekt übersichtlicher zu gestalten.

Adapter Schicht In dieser Schicht findet das Mapping von den eigentlichen Businessobjekten zu den serialisierbaren Ressourcen statt. Hierbei wurde sowohl ein Mapper für den Weg von den Ressourcen zu Businessobjekten als auch vice versa für jede Entität implementiert.

Plugin Schicht In dieser Schicht werden die geschaffenen Schnittstellen aus der Domainschicht konkret mit Hilfe des Frameworks Hibernate implementiert. Ebenfalls wurde in dieser Schicht die Controllerlogik für die API geschaffen und diverse Plugins für die Unterstützung des Entwicklungsprozesses integriert. Konkret wurde die **H2Console** für eine Ansicht der Datenbank, sowie **Swagger** für die Anzeige der APIs verwendet.

3. Programming Principles

3.1 SOLID

Das Programmierprinzip **SOLID** umfasst fünf Programmierprinzipien.

Single - Responsibility - Prinzip Das Single-Responsibility Prinzip besagt, dass eine Klasse nur immer genau **eine** bestimmte Zuständigkeit hat. Das beste Beispiel für den Einsatz dieses Prinzips sind die Repository Klassen. Diese Hibernate Klassen haben nur einen bestimmten Einsatz, nämlich die Persistierung und den Abruf von Entitäten.

Open-Closed-Prinzip Das „Open-Closed-Prinzip“ besagt, dass Klassen einfach erweiterbar sein sollen, ohne dass die ganze Klassenlogik abgeändert werden muss. Ein gutes Beispiel hierfür sind die Controller. Hier können einfach neue API-Calls eingefügt werden, ohne andere Methoden abändern zu müssen.

Liskov'sche Substitutions-Prinzip Das „Liskov'sche Substitutions-Prinzip“ besagt, dass ein Programm, welches ein Objekt einer Basisklasse T verwendet auch mit den davon abgeleiteten Klassen S korrekt funktionieren muss. Konkret wird das im „NoteToNoteResourceManager“ umgesetzt. Hierbei funktioniert die Methode *map* mit der Hauptklasse Note, sowie allen abgeleiteten Klassen.

Interface-Segregation-Prinzip Das „Interface-Segregation-Prinzip“ besagt, dass große Schnittstellen in kleinere (Teil-)Schnittstellen geteilt werden. Konkret wurden im Projekt die Schnittstellen direkt als kleinere Bestandteile, für jede einzelne Entität entwickelt. Im Gegensatz hierzu könnten alle API-Schnittstellen in eine einzelne Klasse gepackt werden. Dabei würde jedoch das „Interface-Segregation-Prinzip“ verletzt.

Dependency-Inversion-Prinzip Das „Dependency-Inversion-Prinzip“ besagt, dass Module in höheren Ebenen nicht von Modulen niedrigerer Ebenen abhängen dürfen. Hierbei sollten durch Interfaces Möglichkeiten geschaffen werden, um die Module abzukoppeln. Im Projekt wird es konkret beim Bridge-Entwurfsmuster in der Pluginschicht verwendet. Hierbei werden die internen Repository-Interfaces in der Pluginschicht implementiert.

3.2 GRASP (insb. Kopplung/Kohäsion)

Unter dem Prinzip von GRASP versteht man neun Programmiermuster. Hierbei soll insbesondere auf die Kopplung und die Kohäsion eingegangen werden. Jedoch werden noch weitere dieser neun Prinzipien im Projekt verwendet.

Informationsexperte Beim Prinzip des „Informationsexperten“ geht es darum, dass geklärt werden muss an welcher Stelle oder in welche Entität bestimmte Funktionalitäten implementiert werden. Dieses Prinzip ist im Projekt in der Domainschicht zu finden, bei dem bestimmte Entitäten bestimmte Funktionen haben. So ist zum Beispiel das Land ein Informationsexperte der z. B. überprüft ob eine neue Abkürzung den Anforderungen entspricht.

Polymorphie Die „Polymorphie“ ist ein Prinzip in der Programmierung. Diese findet im Projekt am Beispiel der Duftnoten ihren Einsatz. So erben alle Duftnoten, egal ob Kopf-, Herz- oder Basisnote von der abstrakten Implementierung „Note“. Im weiteren Verlauf können alle Noten in der API als ein Array des Typen „NoteRessource“ zurückgegeben werden. Diese Polymorphie vereinfacht die Entwicklung dieses Teilaspektes des Projektes wesentlich.

Kopplung Unter Kopplung versteht man die Abhängigkeiten einer Klasse. So gibt es *low* und *high coupling*. Ein einfaches Beispiel ist hier an der Entität des Landes festzumachen. Diese besitzt keine Abhängigkeiten zu anderen Entitäten und ist daher eine Klasse mit einem *low coupling*. Im Gegensatz hierzu steht die Entität/Klasse Parfüm. Diese benötigt viele Klassen, um zu funktionieren. Hier spricht man von *high coupling*.

Kohäsion Die Kohäsion einer Klasse befasst sich mit deren Spezialisierung bzw. den logischen Zusammenhang in einer einzelnen Klasse. So sollte in einer Klasse immer auf sich selbst fokussiert, überschaubar und verständlich sein. Außerdem sollten keine Methoden vorhanden sein, die in keinem engen Verhältnis zur Klasse selbst stehen. Ein Beispiel für eine hohe Kohäsion wären die implementierten JPA-Repositories von Hibernate. Diese verfügen nur über Funktionen die sie selbst, also die persistierte Datenbasis, betreffen.

3.3 DRY

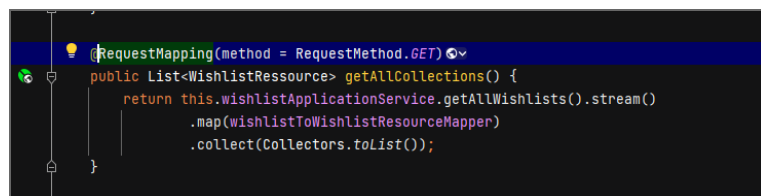
„DRY“ oder **Don't repeat yourself** steht für ein Programmierprinzip, bei dem Codeduplikation verhindert werden soll. Grundlegend soll hier der Code, welcher wiederverwendet wird in eigene Methoden ausgelagert werden, damit er an allen benötigten Codestellen einfach verwendet werden kann.

Bekannte Beispiel in diesem Projekt sind zum Beispiel die Zuweisungen in den unterschiedlichen Konstruktoren. Hier wird in einem „Treppenkonzept“ vorgegangen und immer der nächst kleinere Konstruktor mithilfe des Befehls „this(...)“ aufgerufen. Danach erfolgt die zusätzliche Zuweisung der weiteren Variablen. Ein Beispiel hierfür ist die Klasse „Manufacturer“.

4. Refactoring

4.1 Identifizieren von Codesmells


4.1.1 Code Smell 1



```
@RequestMapping(method = RequestMethod.GET)
public List<WishlistResource> getAllCollections() {
    return this.wishlistApplicationService.getAllWishlists().stream()
        .map(wishlistToWishlistResourceMapper)
        .collect(Collectors.toList());
}
```

Abbildung 4.1: Codesmell 1 - Vor der Beseitigung

Begründung



Composed "@RequestMapping" variants should be preferred

Code smell Minor java:S4488

Spring framework 4.3 introduced variants of the @RequestMapping annotation to better represent the semantics of the annotated methods. The use of @GetMapping, @PostMapping, @PutMapping, @PatchMapping and @DeleteMapping should be preferred to the use of the raw @RequestMapping(method = RequestMethod.XYZ).

Abbildung 4.2: Codesmell 1 - Erklärung SonarLint

Bei diesem Codesmell geht es darum, dass die Annotation

1 `@RequestMapping(method = RequestMethod.GET)`

eine veraltete Schreibweise darstellt. Besser geeignet ist die seit Spring Version 4.3 enthaltene Schreibweise

1 `GetMapping(path = "")`

Diese neue Schreibweise vereinfacht das Finden von entsprechenden Schnittstellen in einem Controller sehr, da nur der Anfang der Annotation gelesen werden muss, und nicht ein Konfigurationsparameter. Somit trägt die Behebung dieses Codesmells zur **besseren Lesbarkeit** des Codes bei.

Fix

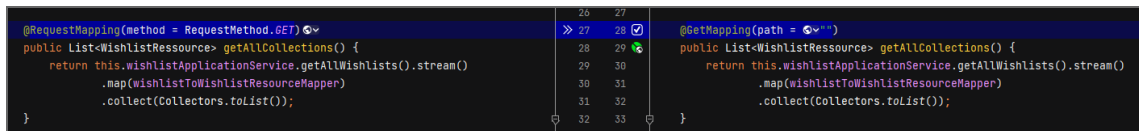


Abbildung 4.3: Codesmell 1 - Fix

4.1.2 Code Smell 2

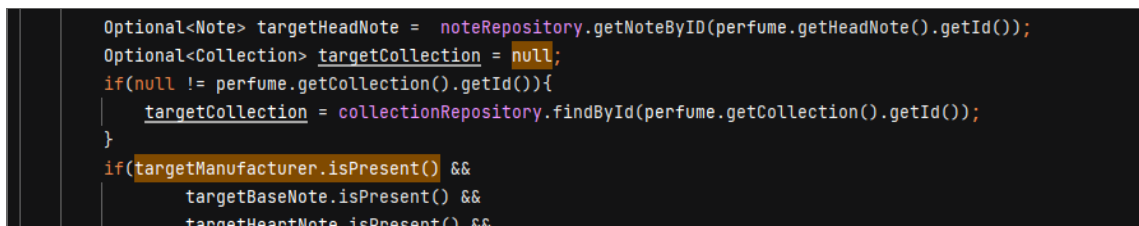


Abbildung 4.4: Codesmell 2 - Vor der Beseitigung

Begründung

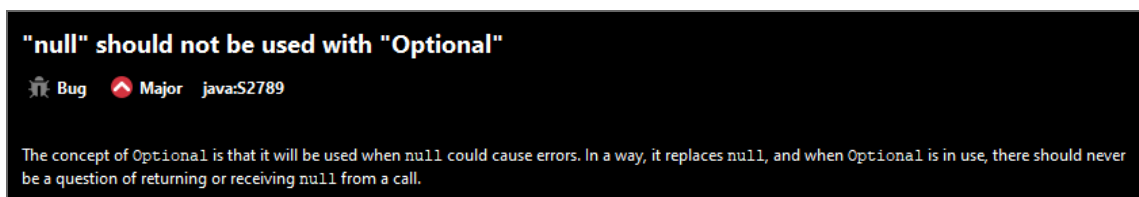


Abbildung 4.5: Codesmell 1 - Erklärung SonarLint

Bei diesem Codesmell liegt ein kritischer Bug vor, der ein grundlegendes Konzept des Typen „Optional“ bricht. Hierbei entsteht zur Laufzeit **kein** Fehler, jedoch ist dies nicht die Art wie ein Optional genutzt werden sollte. Um diese Zuweisung zu entfernen wird einfach in der Abfrage eine leere Instanz des Optional einer Kollektion erschaffen und zugewiesen. Mithilfe dieser kleinen Änderung kann im weiteren Bauvorgang des Parfüms eine einfache Abfrage gemacht werden, ob die Optional-Instanz einer Kollektion eine Kollektion enthält oder das Parfüm in keiner Kollektion eingepflegt werden soll.

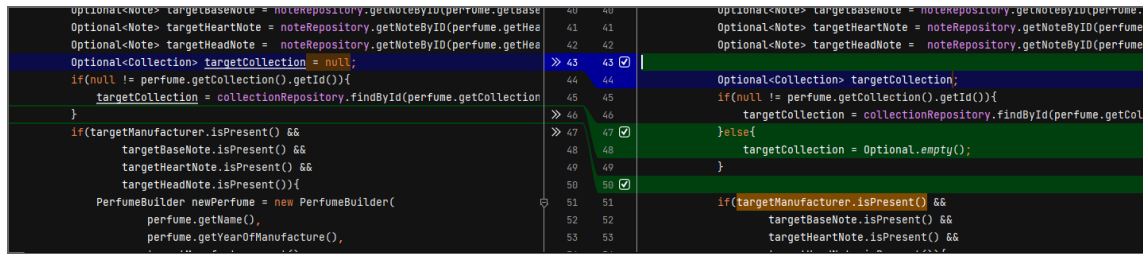
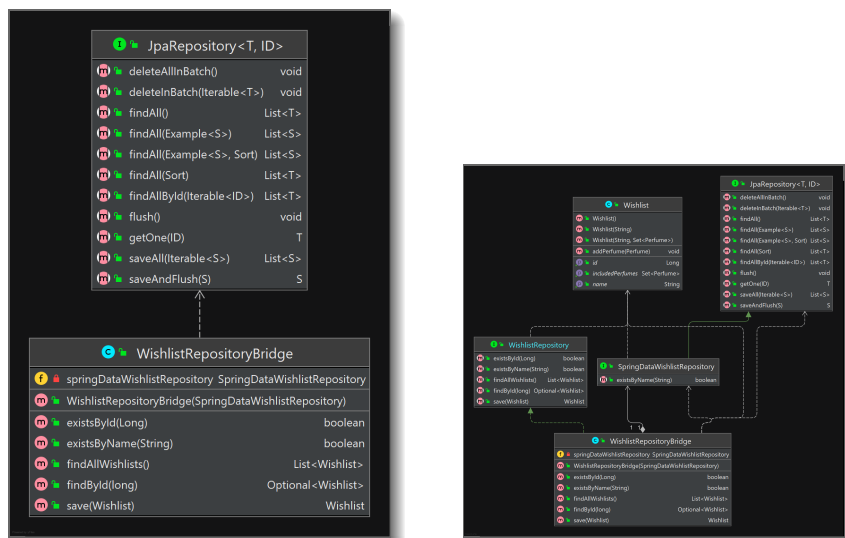
Fix

Abbildung 4.6: Codesmell 2 - Fix

5. Entwurfsmuster



(a) UML vor dem Refactoring

(b) UML nach dem Refactoring

Abbildung 5.1: UML im Vergleich

5.1 Begründung des Einsatzes

Mithilfe des Bridge Musters kann die Implementierung von der Abstraktion entkoppelt werden.

Konkret bedeutet das, dass in der Domainschicht ein Interface geschaffen wird, welches durch eine entsprechende Persistenzlogik in der Pluginschicht implementiert werden muss. Hierbei ergeben sich einige Vorteile durch diese vollzogene Entkopplung beider Komponenten.

Durch diese Aufteilung wird zuerst die Clean Architecture umgesetzt. Des Weiteren ist es durch dieses Entwurfsmuster einfach, die Persistenzlogik in der Pluginschicht auszutauschen, oder simultan mehrere Lösungen zu betreiben.