



# Whiskey 'o clock

## PROGRAMMENTWURF

der Vorlesung „Advanced Software Engineering“

an der

Dualen Hochschule Baden-Württemberg Karlsruhe

von

**Nico Holzhäuser**

Abgabedatum 5. Mai 2022

Matrikelnummer

Kurs

Bearbeitungszeitrum

Gutachter der Studienakademie

...TBD...

TINF19B4

5. & 6. Semester

Mirko Dostmann

# Inhaltsverzeichnis

Inhaltsverzeichnis	ii
Abbildungsverzeichnis	iv
Codeverzeichnis	v
Abkürzungsverzeichnis	vi
<hr/>	
<b>1 Domain Driven Design</b>	<b>1</b>
1.1 Ubiquitous Language . . . . .	1
1.2 Analyse und Begründung der verwendeten Muster . . . . .	2
<b>2 Clean Architecture</b>	<b>5</b>
2.1 Schichtenarchitektur . . . . .	5
<b>3 Programming Principles</b>	<b>6</b>
3.1 SOLID [SEVINC 2022] . . . . .	6
3.2 GRASP (insb. Kopplung/Kohäsion) . . . . .	7
3.3 DRY . . . . .	9
<b>4 Refactoring</b>	<b>10</b>
4.1 Codesmell 1 - RSPEC-1128 - Unnecessary imports should be removed [SONAR 2022c] . . . . .	10
4.2 Codesmell 2 - RSPEC-5411 - Boxed „Boolean“ should be avoided in boolean expressions [SONAR 2022a] . . . . .	10
4.3 Code Smell 3 - RSPEC-1186 - Functions should not be empty [SONAR 2022b] . . . . .	11
<b>5 Entwurfsmuster</b>	<b>13</b>
5.1 Erzeugungsmuster . . . . .	13
5.2 Analyse der verwendeten Muster . . . . .	15

---

<b>A</b>	<b>Anhang</b>	<b>I</b>
A.1	Aggregate Visualisierung . . . . .	I
A.2	Domain Driven Design Visualisierung . . . . .	II
A.3	Clean Architecuture in Spring Boot . . . . .	III
A.4	Code Smell - Backend - 1 - Before . . . . .	IV
A.5	Code Smell - Backend - 1 - Fix . . . . .	IV
A.6	Code Smell - Backend - 1 - After . . . . .	V
A.7	Code Smell - Backend - 2 - Before . . . . .	VI
A.8	Code Smell - Backend - 2 - After . . . . .	VI
A.9	Code Smell - Frontend - 2 - Before . . . . .	VII
A.10	Code Smell - Frontend - 2 - After . . . . .	VII
A.11	Entwurfsmuster - Bridge - Before . . . . .	VIII
A.12	Entwurfsmuster - Bridge - After . . . . .	IX
	<b>Literaturverzeichnis</b>	<b>X</b>

# Abbildungsverzeichnis

A.1	Aggregate Visualisierung [ANIL 2021] . . . . .	I
A.2	Domain Driven Design Visualisierung [STEMMLER 2019] . . . . .	II
A.3	Clean Architecuture in Spring Boot [HEWAGAMAGE 2020] . . . . .	III
A.4	Code Smell - Backend - 1 - Before . . . . .	IV
A.5	Code Smell - Backend - 1 - Fix . . . . .	IV
A.6	Code Smell - Backend - 1 - After . . . . .	V
A.7	Code Smell - Backend - 2 - Before . . . . .	VI
A.8	Code Smell - Backend - 2 - After . . . . .	VI
A.9	Code Smell - Backend - 2 - Before . . . . .	VII
A.10	Code Smell - Backend - 2 - Fix . . . . .	VII
A.11	Entwurfsmuster - Bridge - Before . . . . .	VIII
A.12	Entwurfsmuster - Bridge - After . . . . .	IX

# Liste der Algorithmen

4.1	Non Compliant Version - Codesmell 1	11
4.2	Compliant Version - Codesmell 1	11
4.3	Non Compliant Version - Codesmell 3	11
4.4	Compliant Version - Codesmell 3	12
5.1	Beispiel eines Constructors des BottleBuildes	15
5.2	Beispiel einer Builder Methode	15
5.3	Beispiel einer Builder Methode	16
5.4	Beispiel einer build() Methode eines Builders	16
5.5	Beispiel einer build() Methode eines Builders	16

# Abkürzungsverzeichnis

<b>API</b>	Application Programming Interface . . . . .	4
<b>DDD</b>	Domain Driven Design	
<b>IDE</b>	Integrated Development Environment . . . . .	10
<b>UML</b>	Unified Modeling Language . . . . .	15
<b>UC</b>	Use Case . . . . .	4

# 1. Domain Driven Design

## 1.1 Ubiquitous Language

Die „ubiquitous language“ ist ein Begriff, den EVANS in seinem Buch *Domain-Driven Design: Tackling Complexity in the Heart of Software* eingeführt hat. Dieser Begriffs beschreibt die allgegenwärtige Sprache, die von Softwareentwickler\*innen und Fachexpert\*innen gemeinsam gesprochen wird [PLÖD 2022]. Sie soll der Basis für die Entwicklung des Softwaremodells sein.

### 1.1.1 Analyse der Ubiquitous Language

In diesem Projekt ist die „ubiquitous language“ relativ einfach gehalten, da die Fachdomäne überschaubar ist. Aufgrund dieser beschränkten Problemdomäne sollten keine schwerwiegenden Kommunikationsprobleme in der gemeinsamen Sprache auftreten. Jedoch ist es auch hier wichtig, bestimmte Thematiken und Objekttypen exakt zu spezifizieren um Problemen direkt vor deren Entstehung entgegenzuwirken.

Ubiquitous Language	
Ubiquitous Language	„normale“ Definition
Country	Entität für das Herkunftsland einer Flasche
Manufacturer	Hersteller / Abfüller der Flasche
Whiskeyflasche	Kleinste Entität des Systems. Einfach nur eine schöne, meist teure, Flasche Whiskey
Serie	0 .. n Whiskeyflaschen bilden zusammen eine Serie. Diese haben meist einen gemeinsamen Faktor, wie z.B. die Herkunft

### Verbindungen zwischen den Objekten [Holzhäuser 2021]

Die Grundlage sollen einzelne Whiskeyflaschen-Objekte bilden. Diese werden durch ihre **Herkunft, das Alter, den Kaufpreis, die Sorte und den aktuellen Status** definiert. Hierbei müssen bei einer Objektanlage zwingend ein **label** und eine **Manufacture** definiert werden. Diese Objekte können als **einzelne Flasche** oder als **Teil einer Serie** angelegt werden. Hierbei soll ein Objekt nur Teil **einer oder keiner** Serie sein können. Außerdem sollen Objekte als 'Favoriten', 'Unverkäufliche' bzw. als im Angebot zum Verkauf gekennzeichnet werden können. Hierbei können **favorisierte** sowie **unverkäufliche** Flaschen nicht in den Status 'zum Verkauf' wechseln ohne vorher die entsprechenden Kennzeichnungen zu entfernen.

Eine Serie besteht aus einer Anzahl an Objekten  $> 1$ . Hierbei können verschiedene Flaschen zu einer Serie zusammengefasst werden. Eine Serie benötigt immer zwingend eine Bezeichnung und eine Menge an zugehörigen Objekten. Sind alle Objekte in einer Serie mit dem Status 'zum Verkauf' gekennzeichnet kann die ganze Serie '\*\*zum Verkauf\*\*' angeboten werden. Werden

Objekte gelöscht so soll die Serie automatisch angepasst, und bei einem Inhalt von weniger als zwei Flaschen gelöscht werden.

Objekte sowie können des weiteren gelöscht, sowie verändert werden.

### 1.1.2 Probleme bei der Ubiquitous Language & deren Lösung [Batista 2019]

- Übersetzen von komplexen Sachinhalten in einfache Sprache
  - Durch Wortdefinitionen eliminiert
- Unterschiedliche Bezeichnungen für ein und das Selbe
  - Begriffe aus dem Definitionspool verwenden
- Abstraktion der technischen Sachverhalte für die Domain Experten
  - Klar und deutliche Definitionen im Team und ständige Weiterentwicklung der Ubiquitous Language
- Keine Rücksichtnahme der Domain Experten bei der Entwicklung des technischen Modells
  - Mit einbeziehen aller Parteien für das bestmögliche Ergebnis

## 1.2 Analyse und Begründung der verwendeten Muster

### 1.2.1 Analyse

Eine komplette Übersicht des Domain Driven Design ist in Anhang in Abb. A.2 beigelegt.

#### Value Objects [Milian 2019]

Das Value Object ist ein in der Softwareentwicklung häufig eingesetztes Entwurfsmuster. Wertobjekte (engl. „Value Objects“) sind unveränderbare Objekte, die einen speziellen Wert repräsentieren. Soll der Wert geändert werden, so muss ein neues Wertobjekt erzeugt werden.

#### Entities [Anil 2021]

Entitäten stellen im Domänenmodell Objekte da, welche nicht nur durch die darin enthaltenen Attribute definiert, sondern primär durch ihre Identität, Kontinuität und Persistenz im Laufe der Zeit definiert werden. Entitäten sind im Domänenmodell sehr wichtig, da sie die Basis eines Modells darstellen.

#### Aggregates [Anil 2021]

Ein Aggregat umfasst mindestens eine Entität: den sogenannten Aggregatstamm, der auch als Stammentität oder primäre Entität bezeichnet wird. Darüber hinaus kann es über mehrere untergeordnete Entitäten und Wertobjekte verfügen, wobei alle Entitäten und Objekte zusammenarbeiten, um erforderliche Verhaltensweisen und Transaktionen zu implementieren. Abb. A.12



### Repositories [Šimara 2020]

Ein Repository vermittelt zwischen der Domänen- und der Datenzuordnungsebene mithilfe einer sammlungsähnlichen Schnittstelle für den Zugriff auf Domänenobjekte. Hierbei kann die Anwendung einfach vorhandene Entitäten aus der Persistenzebene laden, speichern, updaten oder löschen.

### Domain Service [Gorodinski 2012]

Eric Evans beschreibt in seinem Buch *Domain-Driven Design: Tackling Complexity in the Heart of Software* einen Domänen Service wie folgt :

When a significant process or transformation in the domain is not a natural responsibility of an ENTITY or VALUE OBJECT, add an operation to the model as standalone interface declared as a SERVICE. Define the interface in terms of the language of the model and make sure the operation name is part of the UBIQUITOUS LANGUAGE. Make the SERVICE stateless.

Hierbei ist grob gesagt, dass ein signifikanter Prozess oder eine Transformation, die über die Verantwortung einer Entität hinaus geht, als eine eigenständige Operation des Modells als Interface mit dem Namen „Service“ hinzugefügt werden muss. Somit beschreibt ein Service einen Prozess, der über die Verantwortung einer Entität oder eines Wertobjektes hinaus geht.

## 1.2.2 Begründung

### Value Objects

In diesem Projekt sollen Wertobjekte dazu dienen, die Übertragung zwischen Backend und Frontend zu gewährleisten. Meist werden Sie für Read-, Update- oder Modifizierungsaufrufe verwendet. Hierbei sind diese „DTO“ Objekte unveränderlich und haben meist die Inhalte einer Entität. So können Entitäten in einer Serialisierten Form ausgetauscht werden.

### Entities

In diesem Projekt sind vier Entitäten vorhanden, die die Basis des Domänenmodells bilden vorhanden :

- »Country« - Land
- »Manufacturer« - Hersteller/Abfüller
- »Bottle« - Flasche
- »Serie« - Flaschenserie

Diese werden durch ihre Identität selbst beschrieben, werden persistiert und haben veränderliche Attribute.

### **Aggregates**

Aggregate sind in diesem Projekt im eigentlichen Sinne nicht vorhanden. Jedoch sind nach der Definition in Abschnitt 1.2.1 auch einzelne Entitäten ein Aggregatstamm und somit ein Aggregate im jeweiligen Bereich. Folgen wir diesem Prinzip gibt es in diesem Projekt **vier** Aggregate.

### **Repositories**

In diesem Projekt sind Repositories im Backend zu finden. Hierbei werden sie durch Implementierungen des Interfaces »JpaRepository<Object,PrimaryKey>« umgesetzt. Mit der Hilfe von Hibernate wird so die Persistierungsschicht mit der Applikationsschicht verbunden und Methoden geschaffen, um Objekte aus der Datenbasis zu filtern.

### **Domain Service**

Service's finden in diesem Projekt ebenfalls im Backend ihre Anwendung. Hierbei wird die Logik in sogenannte Services ausgelagert, die bestimmte Entitäten oder Beziehungen anhand eines Use Case (UC) bearbeiten. Hierbei sind in Spring Boot oft die Application Programming Interface (API) Aufrufe einzelne gekapselte Logiken, welche durch einen entsprechenden Serviceaufruf umgesetzt werden.

Desweiteren sind Security oder Transaktionale Komponenten denkbare Services für eine Spring Anwendung.

## **2. Clean Architecture**

### **2.1 Schichtenarchitektur**

In Abb. A.3

#### **2.1.1 Planung**

#### **2.1.2 Entscheidung anhand von Kriterien**

## 3. Programming Principles

### 3.1 SOLID [Sevinc 2022]

Hinter der Bezeichnung „SOLID“ stehen fünf Prinzipien

- **SRP** Single-Responsibility-Prinzip
- **OCP** Open-Closed-Prinzip
- **LSP** Liskov'sche Substitutions-Prinzip
- **ISP** Interface-Segregation-Prinzip
- **DIP** Dependency-Inversion-Prinzip

#### 3.1.1 Single-Responsibility-Prinzip

Dieses Prinzip findet zum Beispiel in den Repositor's seinen Nutzen. Dieses haben nur eine Funktion, die Schnittstelle zur Datenbank darzustellen. Dies ist ihr einziger definierter Aufgabenbereich.

#### 3.1.2 Open-Closed-Prinzip

Das Open-Closed-Prinzip besagt, dass Objekte einfach erweiterbar sind und die bestehende Logik nicht in ihren fundamentalen Bestandteilen modifiziert werden muss.

Ein Beispiel ist hier der „BottleApplicationService“, der mit Hilfe der Klasse „BottleBooleanType“ eine Unterscheidung der „checkbaren“ Boolean Werte ermöglicht. Kommt hier ein neuer hinzu muss nur das Enum erweitert werden und im Switch-Case ein neuer Fall implementiert werden.

#### 3.1.3 Liskov'sche Substitutions-Prinzip

Das Liskov'sche Substitutions Prinzip findet keine Anwendung, da keine Vererbung implementiert wurde.

### 3.1.4 Interface-Segregation-Prinzip

Das Interface-Segregation-Prinzip besagt, dass Entwickler nicht dazu gezwungen werden sollen Teile von Schnittstellen zu implementieren, die später nicht verwendet werden. Prinzipiell wurde dieses Prinzip, durch den Aufbau der Interfaces in verschiedene Zuständigkeitsbereiche, direkt umgesetzt und die Interfaces müssen nicht weiter aufgeteilt werden.

### 3.1.5 Dependency-Inversion-Prinzip

Das Dependency-Inversion-Prinzip besagt, dass Systeme am flexibelsten sind, wenn Codeabhängigkeiten ausschließlich auf Abstraktion beziehen, statt auf Konkretionen. In Java soll sich hier bei der Nutzung von import nur auf abstrakte Quellmodule bezogen werden, wie z.B Schnittstellen, abstrakte Klassen oder Module, die jede andere Form der Abstraktion gewährleisten.

In der Praxis ist diese Regel aber kaum umsetzbar, da Softwaresysteme auch von konkreten Entitäten abhängig sind. Im groben folgen die Module der Kapitel 2 diesem Prinzip, da nur Module von höheren Schichten die Interfaces aus den niedrigeren Schichten implementieren. Somit liegen die Regeln in der inneren Schicht, wohingegen die Implementierung in den äußeren Schichten vorgenommen werden.

## 3.2 GRASP (insb. Kopplung/Kohäsion)

Unter der Bezeichnung „GRASP“ (**G**eneral **R**esponsibility **A**ssignment **S**oftware **P**inciples) versteht man neun grundlegende Muster

1. Controller(engl. Controller)
2. Ersteller (engl. Creator)
3. Indirektion (engl. Indirection)
4. Informationsexperte (engl. Information expert)
5. Hohe Kohäsion (engl. High cohesion)
6. Wenig Koppelnd (engl. Low coupling)
7. Polymorphie (engl. Polymorphism)
8. (engl. Protected variations)
9. (engl. Pure fabrication)

Im weiteren werden die **Prinzipien 5,6** genauer betrachtet.

### 3.2.1 Analyse

1. Bei Spring werden die Events (in unserem Fall API Calls) durch sogenannte Controller behandelt. Diese liegen im Modul »0 - Plugins« im »de.dhbw.ase.plugin.rest« - Package
2. -
3. -
4. -
5. Näher in Abschnitt 3.2.2 behandelt
6. Näher in Abschnitt 3.2.3 behandelt
7. -
8. -
9. -

### 3.2.2 High cohesion [Sites 2022]

Das Prinzip „Hohe Kohäsion“ (engl. high cohesion) besagt, dass darauf geachtet werden soll, dass diese Klasse nicht mehrere Verantwortlichkeiten in sich trägt. Die Kohäsion ist ein Maß für den logischen Zusammenhang der Daten und Methoden der Klasse. So sollten keine Verantwortlichkeiten und Aufgaben **in** einer Klasse vermischt werden.

Dieses Prinzip wurde im ganzen Projekt umgesetzt. Einige Beispiele hierfür sind die JPA-Repository's, die nur Kenntnisse von ihrer Anbindung an die Datenbasis haben und keine weiteren Verbindungen zu anderen Objekten benötigen, oder die Controller, die nur Events entgegennehmen und sie entsprechend mit Hilfe der Services verarbeiten. Hier ist jeweils eine hohe Kohäsion gegeben.

### 3.2.3 Low coupling [Sites 2022]

Das Prinzip „wenig Kopplung“ (engl. low coupling) besagt, dass möglichst wenig Abhängigkeiten zwischen den Komponenten bestehen sollte. Die einzelnen Klassen/Objekte/Module sollten möglichst wenig voneinander wissen. Das Ziel ist die Abhängigkeiten zwischen einzelnen Klassen/Objekte/Module so gering wie möglich zu halten.

Dieses Prinzip wurde im ganzen Projekt umgesetzt. Ein Beispiel ist, dass z.B. der CountryApplicationService kein Wissen über andere Repository's als das „CountryRepository“ benötigt. Hier herrscht also „low coupling“. Im Gegenzug benötigt der „BottleApplicationService“ drei verschiedene Repository's, was eine hohe Kopplung mit sich bringt.

### 3.2.4 Schlussfolgerung

Prinzipiell geht die Kohäsion mit der Kopplung einher. Umso mehr Verantwortlichkeiten und Teilaufgaben in andere Klassen ausgelagert werden, umso höher wird die Kohäsion der Ursprungsklasse, da hier eine Spezialisierung stattfindet. Im gleichen Schritt nimmt ebenfalls die Kopplung zu, da die Ursprungsklasse von mehreren Unterklassen abhängig ist. Hierbei gilt es ein gesundes Mittelmaß zu finden und eine überlegte Trennung durchzuführen. Generell gilt jedoch die Kopplung unter Klassen gering und damit die Kohäsion der Klassen hoch zu halten.

### 3.3 DRY

Das Programmierprinzip „DRY“, dt. für „wiederhole dich nicht“ (engl. **D**on't **R**epeat **Y**ourself), steht für ein Prinzip in der Programmierung. Hierbei sollen Redundanzen vermieden bzw. beseitigt werden. Es ist ein fundamentales Prinzip von Clean Code, dass von Programmieren wie folgt definiert wird :

Every piece of knowledge must have a single, unambiguous, authoritative representation within a system. [CNEUDE 2022]

#### 3.3.1 Analyse

DRY ist ein grundlegendes Prinzip in der Programmierung und prinzipiell überall im Projekt zu finden. So wurde beispielsweise die Konstruktoren der »Bottle« ineinander verzahnt, um die Zuweisung lediglich einmal entwickeln zu müssen. Ebenso wurden Mapper geschaffen, um die Umwandlungslogik zentral an einem Ort abzulegen und nicht an jeder Stelle, an der diese Logik benötigt wird eine eigene Implementierung zu benötigen.

#### 3.3.2 Begründung

Die Anwendung der „DRY“ Prinzipien bringt einige Vorteile mit sich. So ist weniger Code generell besser da es Zeit und Aufwand mit sich bringt diesen Code zu warten. Außerdem werden Fehler durch weniger Code vermieden, da schlichtweg weniger Platz vorhanden ist um Fehler zu machen. Des weiteren hilft es dabei, wenn bestimmte Logiken ausgelagert werden, wie z.B die Mapper Logiken da diese an vielen Stellen in der Anwendung verwendet und dadurch wiederverwendet werden können. Zuletzt muss, wenn ein Logikfehler auftritt nicht jede Codestelle einzeln angepasst werden, wenn die „DRY“-Logiken umgesetzt wurden.

## 4. Refactoring

### 4.1 Codesmell 1 - RSPEC-1128 - Unnecessary imports should be removed [Sonar 2022c]

Dieser Codesmell wird in der Regel „RSPEC - 1128“ behandelt. Er befasst sich damit, dass ungenutzte Imports (dt. Einbettungen) aus den entsprechenden Klassen entfernt werden sollen.

#### Begründung

Dieser Fehler soll verhindern, dass der Entwickler durch zu viele Imports in einer Klasse verwirrt wird.

#### 4.1.1 Fix

Entfernen der ungenutzten Imports. Dies kann oft durch die Option „Code Refactoring“ der IDE automatisch übernommen werden.

#### 4.1.2 Projektbezug

Hierbei wurde der Fehler erkannt (Anhang A.4) und im gleichen durch die Refactoring Funktion (Anhang A.5) der Integrated Development Environment (IDE) beseitigt (Anhang A.6).

### 4.2 Codesmell 2 - RSPEC-5411 - Boxed „Boolean“ should be avoided in boolean expressions [Sonar 2022a]

Dieser Codesmell wird in der Regel „RSPEC - 5411“ behandelt. Er befasst sich damit, dass Boolean Variablen nicht direkt als Entscheidungskriterium einer Verzweigung verwendet werden sollen.

#### 4.2.1 Begründung

Dieser Fehler verhindert, dass bei einem Nullwert der Boolean Variable eine *NullPointerException* geworfen wird. Hierbei wird die Robustheit der Anwendung gesteigert.



### 4.3. CODE SMELL 3 - RSPEC-1186 - FUNCTIONS SHOULD NOT BE EMPTY [CODESMELL3.SONAR]

#### 4.2.2 Fix

Um diesen Fehler zu vermeiden sollten der Boolean Ausdruck in einen Boolean Wrapper geschrieben werden. Hierbei muss die nicht gültige Version

```
1 Boolean b = getBoolean();
2 if (b) { // Noncompliant, it will throw NPE when b == null
3     foo();
4 } else {
5     bar();
6 }
```

Algorithmus 4.1: Non Compliant Version - Codesmell 1

wie folgt abgewandelt werden.

```
1 Boolean b = getBoolean();
2 if (Boolean.TRUE.equals(b)) {
3     foo();
4 } else {
5     bar(); // will be invoked for both b == false and b == null
6 }
```

Algorithmus 4.2: Compliant Version - Codesmell 1

#### 4.2.3 Projektbezug

Hierbei wurde der Fehler erkannt (Anhang A.7) und im gleichen Zug eine Lösung für den Code Smell implementiert (Anhang A.8).

### 4.3 Code Smell 3 - RSPEC-1186 - Functions should not be empty [Sonar 2022b]

Dieser Codesmell wird in der Regel „RSPEC - 1186“ behandelt. Er befasst sich damit, dass es keine leeren Funktionen in Typescript geben sollte.

#### 4.3.1 Begründung

Leere Funktionen sind allgemein zu entfernen, um ungewollte Verhaltensmuster in der Produktivumgebung zu verhindern. Des weiteren könnte, wenn diese Methode nicht unterstützt oder später entwickelt wird, eine Fehlermeldung geworfen werden.

#### 4.3.2 Fix

Hierbei sollten leere Funktionen entweder entfernt oder,

```
1 function foo() {
2 }
3
4 var foo = () => {};
```

Algorithmus 4.3: Non Compliant Version - Codesmell 3

wie folgt mit einem Kommentar abgewandelt werden, warum diese Methode einen leeren Rumpf aufweist.

```
1  function foo() {  
2      // This is intentional  
3  }  
4  
5  var foo = () => {  
6      do_something();  
7  };
```

Algorithmus 4.4: Compliant Version - Codesmell 3

### 4.3.3 Projektbezug

Hierbei wurde der Fehler erkannt (Anhang A.9) und im gleichen Zug eine Lösung für den Code Smell implementiert (Anhang A.10).

# 5. Entwurfsmuster

Im Buch [GAMMA u. a. 1994] werden Muster anhand zweier Kriterien klassifiziert, der Zweck (engl. purpose) und der Bereich (engl. scope). Anhand dieser beiden Merkmale können die Muster in verschiedene Kategorien eingeordnet werden. Es gibt nach [GAMMA u. a. 1994] drei Kategorien, in welche Entwurfsmuster eingeteilt werden. Außerdem erfolgt in jeder Kategorie eine Einteilung in Klassen- und Objektmuster

## 5.1 Erzeugungsmuster

Erzeugungsmuster abstrahieren Objekterzeugungsprozesse. Hier erfolgt eine weitere Einteilung in Klassen- und Objektmuster

### 5.1.1 Klassenmuster

Klassenmuster nutzen Vererbung um die Klasse des zu erzeugenden Objektes zu variieren. Bekannte Vertreter sind hierbei

- Fabrikmethode (factory Method)

### 5.1.2 Objektmuster

Objektmuster delegieren die Objekterzeugung an andere Objekte. Bekannte Vertreter sind hierbei

- Abstrakte Fabrik (engl. abstract factory, kit)
- Einzelstück (engl. singleton)
- Erbauer (engl. builder)
- Prototyp (engl. prototyp)

### 5.1.3 Strukturmuster

Strukturmuster fassen Klassen und Objekte zu größeren Strukturen zusammen.

#### 5.1.4 Klassenmuster

Klassenmuster fassen dabei Schnittstellen und Implementierungen zusammen. Hierbei werden die Strukturen zur Übersetzungszeit festgelegt und sind danach nicht mehr veränderbar. Bekannte Vertreter sind hierbei

- Adapter (engl. adapter / wrapper)

#### 5.1.5 Objektmuster

Objektmuster ordnen Objekte in eine Struktur ein. Diese Strukturierung ist in der Laufzeit veränderbar. Bekannte Vertreter sind hierbei

- Adapter (engl. adapter / wrapper)
- Bridge (engl. bridge)
- Stellvertreter (engl. surrogate)
- Dekorierer (engl. decorator)
- ...

#### 5.1.6 Verhaltensmuster

Verhaltensmuster beschreiben die Interaktion zwischen Objekten und komplexen Kontrollflüssen.

#### 5.1.7 Klassenmuster

Klassenmuster teilen die Kontrolle auf verschiedene Klassen auf. Bekannte Vertreter sind hierbei

- Adapter (engl. adapter / Wrapper)

#### 5.1.8 Objektmuster

Objektmuster nutzen hierfür die Komposition statt der Vererbung. Bekannte Vertreter sind hierbei

- Beobachter (engl. vbserver)
- Besucher (engl. visitor)
- Iterator (engl. iterator)
- Vermittler (engl. mediator)
- ...

## 5.2 Analyse der verwendeten Muster

### 5.2.1 Bridge Pattern

Das Bridge Pattern ist in der Klasse der Strukturmuster, in der Sektion Objektmuster angesiedelt. Es befasst sich somit mit der Struktur im Projekt. Hierbei werden Objekte zu größeren Strukturen zusammengefasst. Konkret erfolgt die Anwendung dieses Muster im Modul »0 - Plugins« angewendet. Hierbei werden die eigentlichen „JPA - Repository“ über eine Bridge Klasse an die vorhandenen Repositories im Modul »3 - Domain« angebunden. Diese Bridge Klassen implementieren so mit Hilfe der JPA-Repositories die Interfaces, welche durch die Domain vorgegeben werden.

Dadurch wird die Implementierung von der Abstraktion entkoppelt. Konkret bedeutet dass, eine Entkopplung der eigentlichen Geschäftslogik von der Persistierung erfolgt. Hierbei kann die Persistierungslogik leicht ausgetauscht. Hierzu müsste die neue Lösung nur die gegebenen Methoden der Interfaces aus den Interfaces des Modul »3 - Domain« implementieren.

### UML Vergleich

Zur Verdeutlichung sind die Unified Modeling Language (UML)-Diagramme vor (Anhang A.11) und nach (Anhang A.12) der Anwendung des Entwurfsmuster angehängt.

### 5.2.2 Builder Pattern

Das Builder Pattern ist in der Klasse der Erzeugungsmuster, in der Sektion Objektmuster angesiedelt. Es befasst sich somit mit der Generierung und Erschaffung von anderen Objekte. Hierzu wird eine »Builder« entwickelt, welche im Konstruktor die erforderlichen Attribute annimmt, um ein Projekt zu erschaffen.

```
1 public BottleBuilder(String label) {  
2     this.label = label;  
3 }
```

Algorithmus 5.1: Beispiel eines Constructors des BottleBuildes

Im weiteren werden die nicht zwingend erforderlichen Attribute mit weiteren Methoden ergänzt. Wichtig ist, dass hierbei immer die aktuelle Instanz zurückgegeben wird.

```
1 public BottleBuilder price(double price) {  
2     this.price = price;  
3     return this;  
4 }
```

Algorithmus 5.2: Beispiel einer Builder Methode

Schlussendlich muss der Erbauer noch ein Objekt des jeweiligen Types erschaffen. Hierzu muss in der Zielentität ein Konstruktor vorhanden sein, der einen Builder als Übergabeparameter akzeptiert.

```

1  public Bottle(BottleBuilder bottleBuilder) {
2      this(bottleBuilder.getUuid(),
3          bottleBuilder.getLabel(),
4          bottleBuilder.getPrice(),
5          bottleBuilder.getYearOfManufacture(),
6          bottleBuilder.getManufacturer(),
7          bottleBuilder.isForSale(),
8          bottleBuilder.isFavorite(),
9          bottleBuilder.isUnsaleable(),
10         bottleBuilder.getSeries());
11 }

```

Algorithmus 5.3: Beispiel einer Builder Methode

Schlussendlich wird das Zielobjekt erschaffen, wobei evtl. noch eine Validierung der Instanz stattfindet.

```

1  public Bottle build() {
2      Bottle bottle = new Bottle(this);
3      //Validation
4      return bottle;
5  }

```

Algorithmus 5.4: Beispiel einer build() Methode eines Builders

Die schlussendliche Erschaffung eines Objektes erfolgt nun im Baukasten - System. Hier am Beispiel der Umwandlung einer BottleDTO und einer Bottle-Entität

```

1  private Bottle map(BottleDTO bottleDTO) {
2      BottleBuilder newBottle = new BottleBuilder(bottleDTO.getLabel());
3      newBottle.uuid(bottleDTO.getUuid());
4      .price(bottleDTO.getPrice());
5      .yearOfManufacture(bottleDTO.getYearOfManufacture());
6      .manufacturer(manufacturerRepository.getManufacturerByUuid(bottleDTO.getManufacturer().getUuid()))
7  )
8      .forSale(bottleDTO.getForSale());
9      .favorite(bottleDTO.getFavorite());
10     .unsaleable(bottleDTO.getUnsaleable());
11     if(bottleDTO.getSeries() != null){
12         newBottle.series(seriesRepository.getSeriesByUuid(bottleDTO.getSeries().getUuid()));
13     } else {
14         newBottle.series(null);
15     }
16     return newBottle.build();
17 }

```

Algorithmus 5.5: Beispiel einer build() Methode eines Builders

Durch die Verwendung dieses Muster wird die Erschaffung und Repräsentation eines beliebigen Objektes getrennt. Hierbei wird die Erschaffungslogik in einer dedizierten Klasse gekapselt, die bei späteren Änderungen einfach angepasst werden kann.

## UML Vergleich

Hierbei wird auf den Vergleich mit Hilfe von UML-Diagrammen verzichtet, da hier lediglich die Klasse selbst (before), bzw. eine Verbindung von der Klasse zum Builder zu sehen wäre. Außerdem wurde dieses Muster Schemenhaft am Beispiel der Bottle-Entität im Modul »3 - Domain« entwickelt.

Die Java Library Lombok vereinfacht diesen Prozess, indem die Klassen mit der Annotation „@Builder“ annotiert werden. So muss keine konkrete Entwicklung des Builders erfolgen

# A. Anhang

## A.1 Aggregate Visualisierung

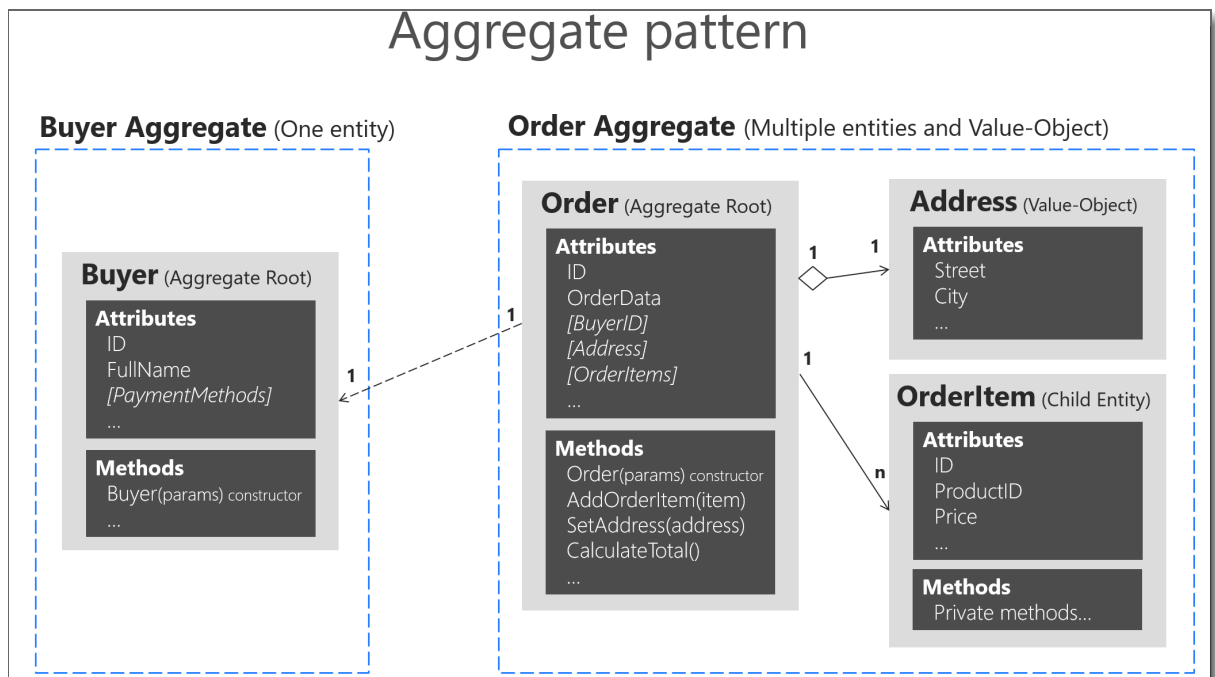


Abbildung A.1: Aggregate Visualisierung [ANIL 2021]

## A.2 Domain Driven Design Visualisierung

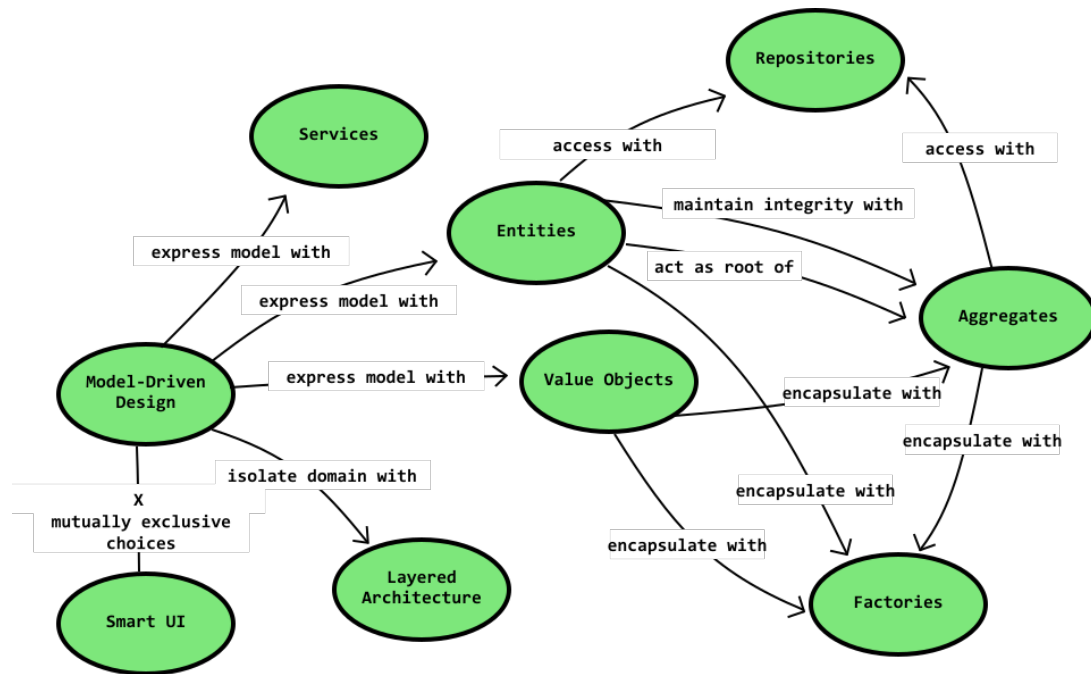


Abbildung A.2: Domain Driven Design Visualisierung [STEMMLER 2019]



### A.3 Clean Architecuture in Spring Boot

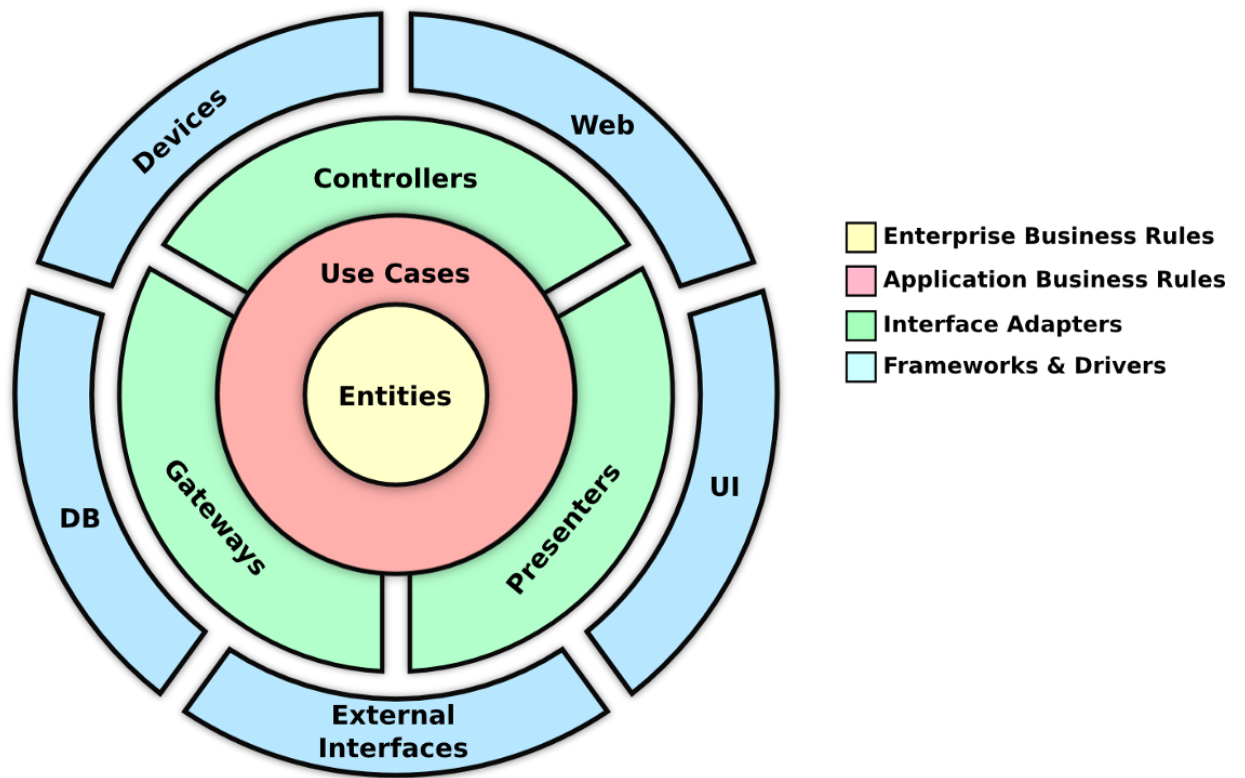


Abbildung A.3: Clean Architecuture in Spring Boot [HEWAGAMAGE 2020]

## A.4 Code Smell - Backend - 1 - Before

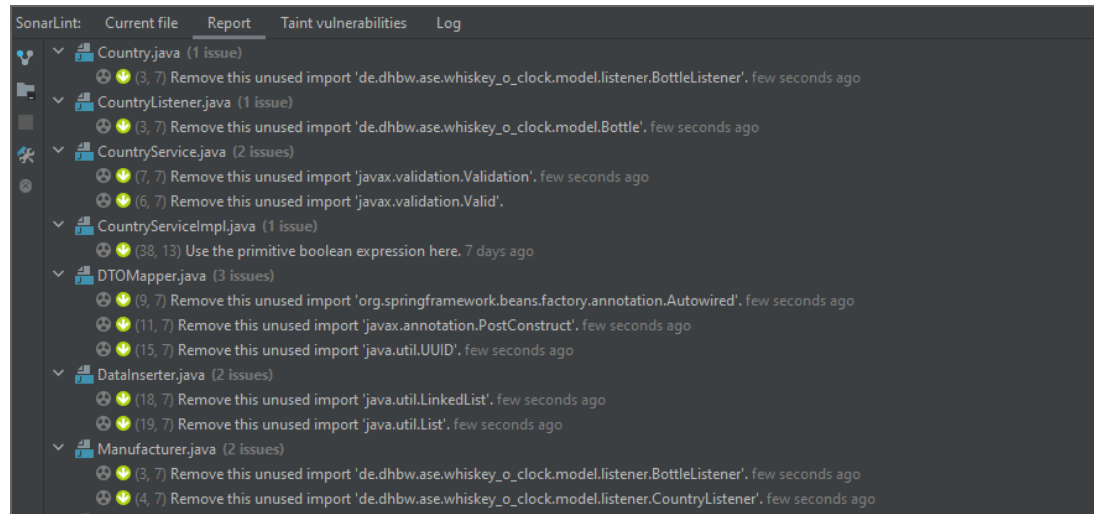


Abbildung A.4: Code Smell - Backend - 1 - Before

## A.5 Code Smell - Backend - 1 - Fix

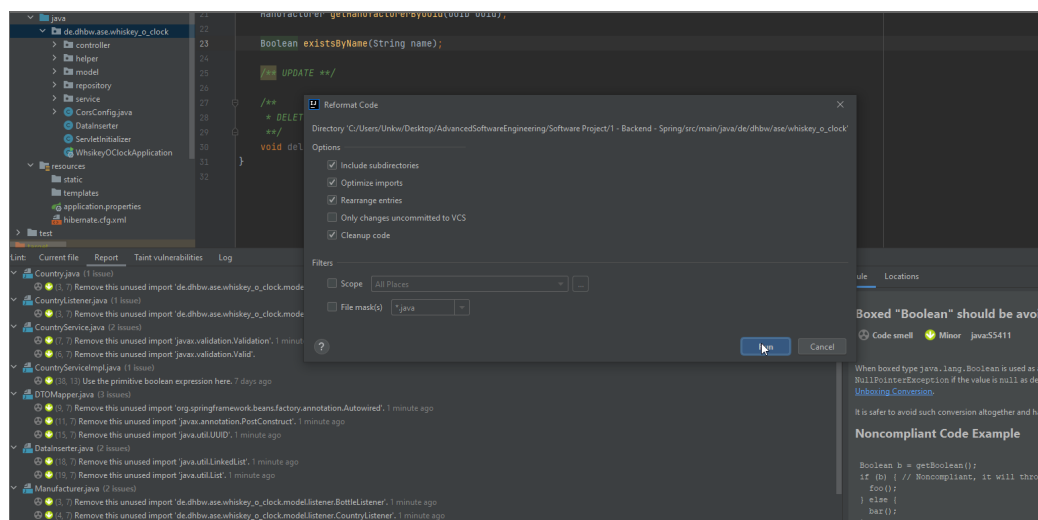


Abbildung A.5: Code Smell - Backend - 1 - Fix

## A.6 Code Smell - Backend - 1 - After

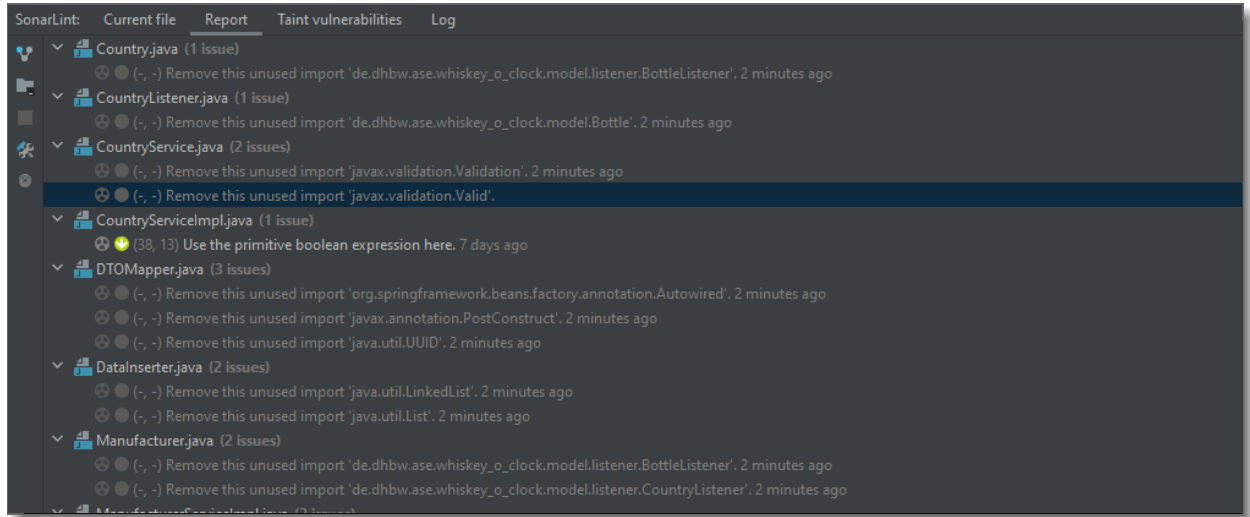


Abbildung A.6: Code Smell - Backend - 1 - After

## A.7 Code Smell - Backend - 2 - Before

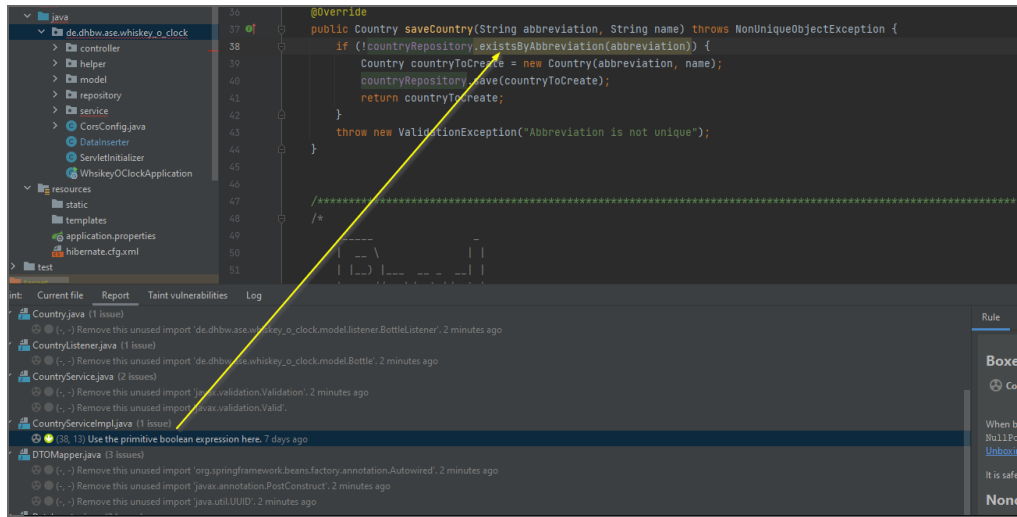


Abbildung A.7: Code Smell - Backend - 2 - Before

## A.8 Code Smell - Backend - 2 - After



Abbildung A.8: Code Smell - Backend - 2 - After

## A.9 Code Smell - Frontend - 2 - Before

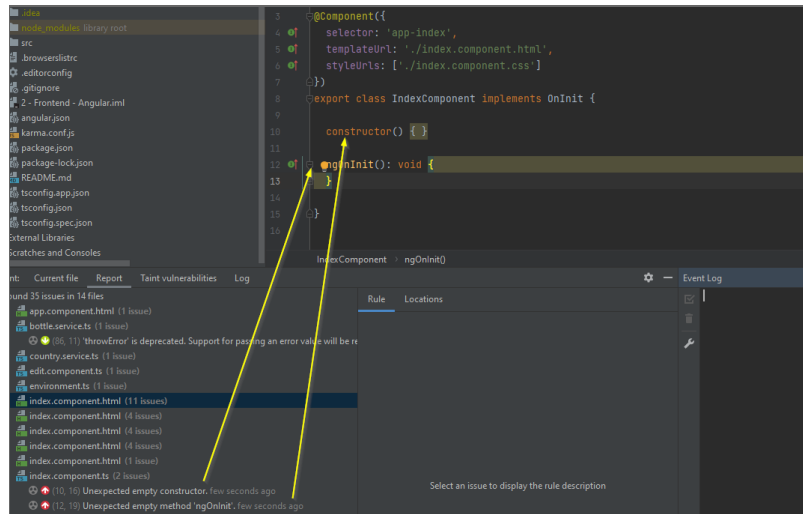


Abbildung A.9: Code Smell - Backend - 2 - Before

## A.10 Code Smell - Frontend - 2 - After

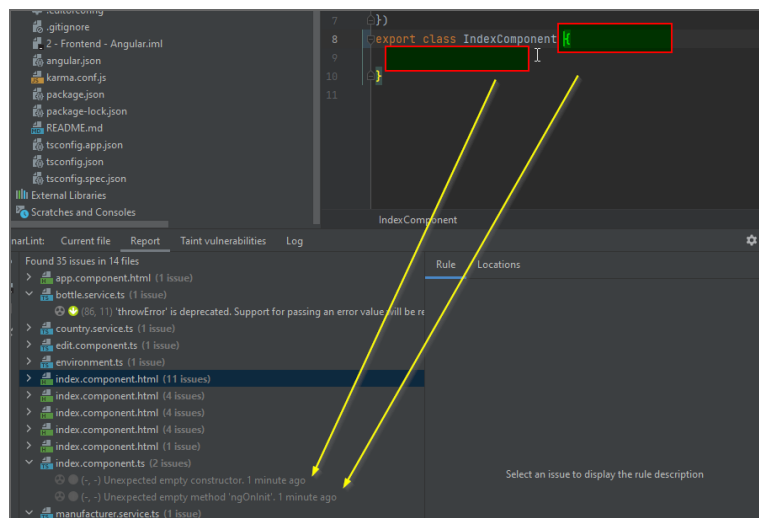


Abbildung A.10: Code Smell - Backend - 2 - Fix

## A.11 Entwurfsmuster - Bridge - Before

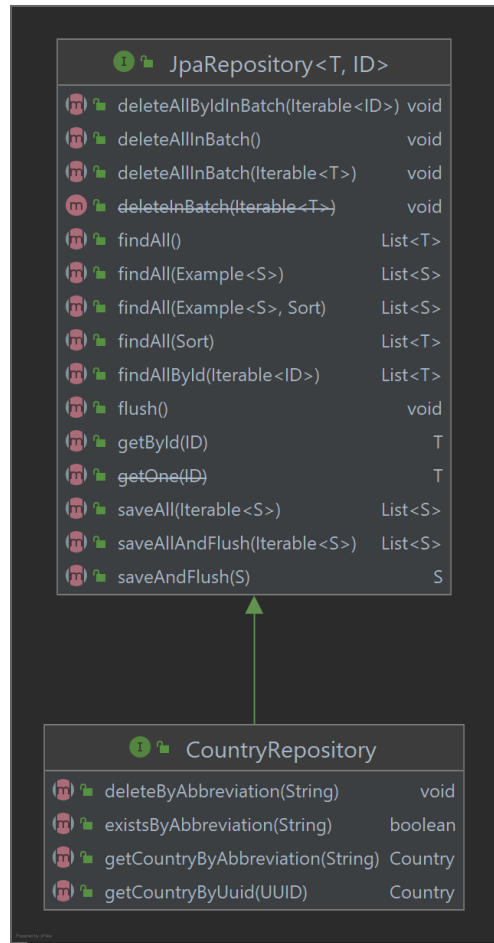


Abbildung A.11: Entwurfsmuster - Bridge - Before

## A.12 Entwurfsmuster - Bridge - After

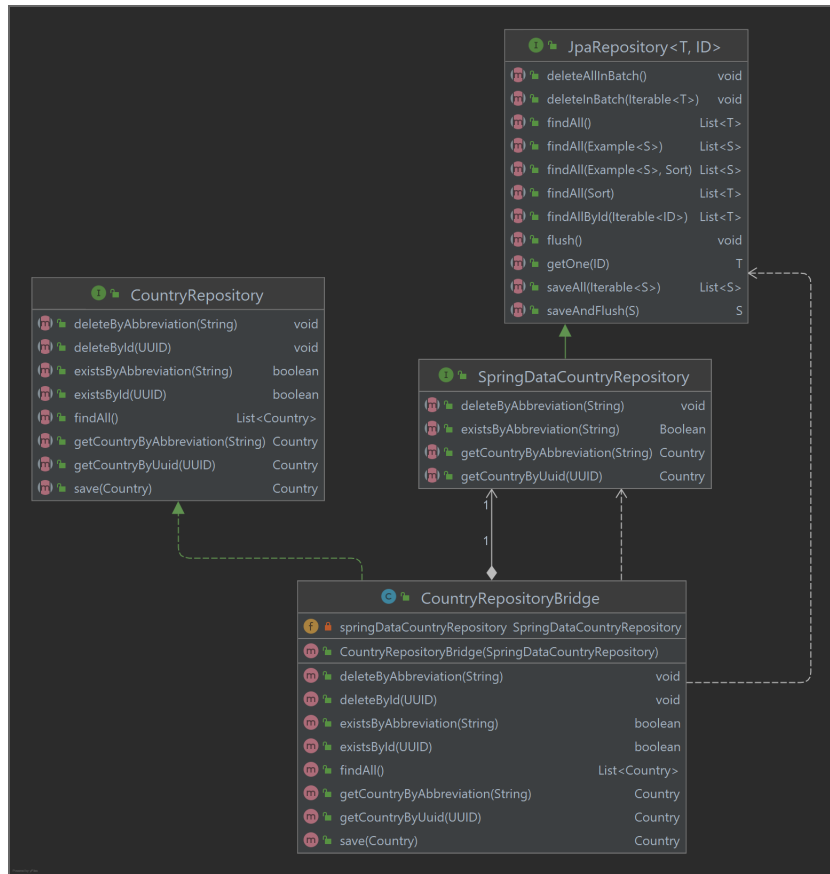


Abbildung A.12: Entwurfsmuster - Bridge - After

# Literaturverzeichnis

Alle Quellen sind zusätzlich im Ordner Quellensicherung gespeichert !

- ANIL, Nish [2021]. *Entwerfen eines Microservicedomänenmodells*. URL: <https://docs.microsoft.com/de-de/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/microservice-domain-model> [besucht am 21.02.2022] [siehe S. 2, I].
- BATISTA, Felipe De Freitas [2019]. *Developing the ubiquitous language*. URL: <https://medium.com/@felipefreitasbatista/developing-the-ubiquitous-language-1382b720bb8c> [besucht am 21.02.2022] [siehe S. 2].
- CNEUDE, Matthieu [2022]. *The DRY Principle: Benefits and Costs with Examples*. URL: <https://thevaluable.dev/dry-principle-cost-benefit-example/> [besucht am 25.02.2022] [siehe S. 9].
- EVANS, Eric [2004]. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley [siehe S. 1, 3].
- GAMMA, Erich u. a. [1994]. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1. Aufl. Addison-Wesley Professional. ISBN: 0201633612. URL: [http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612/ref=ntt\\_at\\_ep\\_dpi\\_1](http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612/ref=ntt_at_ep_dpi_1) [siehe S. 13].
- GORODINSKI, Leo [2012]. *Services in Domain-Driven Design (DDD)*. URL: <http://gorodinski.com/blog/2012/04/14/services-in-domain-driven-design-ddd/> [besucht am 21.02.2022] [siehe S. 3].
- HEWAGAMAGE, Pasindu [2020]. *Clean Architecture on Spring Boot*. URL: <https://medium.com/@pasinduhewagamage/clean-architecture-on-spring-boot-da3ff7bdcc7> [besucht am 22.02.2022] [siehe S. III].
- HOLZHÄUSER, Nico [15. Nov. 2021]. »Projektvorschlag ASE«. Projektantrag. DHBW Karlsruhe [siehe S. 1].
- ŠIMARA, Svafa [2020]. *Domain-Driven Design, part 5 — Repository*. URL: <https://docs.microsoft.com/de-de/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/microservice-domain-model> [besucht am 21.02.2022] [siehe S. 3].
- MILIAN, Paul [2019]. *Value Objects to the rescue!* URL: <https://medium.com/swlh/value-objects-to-the-rescue-28c563ad97c6> [besucht am 21.02.2022] [siehe S. 2].
- PLÖD, Michael [2022]. *Qualitätssicherung*. URL: <https://entwickler.de/software-architektur/warum-ist-sprache-so-wichtig-001> [besucht am 21.02.2022] [siehe S. 1].



- SEVINC, Harun [2022]. *Die SOLID-Design-Prinzipien*. URL: <https://www.adesso.de/de/news/blog/die-solid-design-prinzipien.jsp> [besucht am 25.02.2022] [siehe S. 6].
- SITES, Google [2022]. *Kohäsion und Kopplung*. URL: <https://sites.google.com/site/koesterprogramming/home/softwareentwicklung/kohaesion-und-kopplung> [besucht am 25.02.2022] [siehe S. 8].
- SONAR [2022a]. *Boxed "Boolean" should be avoided in boolean expressions*. URL: <https://rules.sonarsource.com/java/RSPEC-5411> [besucht am 25.02.2022] [siehe S. 10].
- [2022b]. *Functions should not be empty*. URL: <https://rules.sonarsource.com/javascript/RSPEC-1186> [besucht am 25.02.2022] [siehe S. 11].
- [2022c]. *Unnecessary imports should be removed*. URL: <https://rules.sonarsource.com/java/RSPEC-1128> [besucht am 25.02.2022] [siehe S. 10].
- STEMMLER, Khalil [2019]. *Understanding Domain Entities [with Examples] -DDD w/ TypeScript*. URL: <https://khalilstemmler.com/articles/typescript-domain-driven-design/entities/> [besucht am 21.02.2022] [siehe S. II].