

Musikvereinsverwaltung

PROGRAMMENTWURF

der Vorlesung „Advanced Software Engineering“

an der

Dualen Hochschule Baden-Württemberg Karlsruhe

von

Elisabeth Kletsko

Abgabedatum 16.05.2022

Kurs
Bearbeitungszeitrum
Gutachter der Studienakademie


TINF19B4
5. & 6. Semester
Mirko Dostmann

Inhaltsverzeichnis





Inhaltsverzeichnis	ii
Abbildungsverzeichnis	iii
Codeverzeichnis	iv
Abkürzungsverzeichnis	v

1	Allgemeines zum Projekt	1
2	Domain Driven Design (DDD)	2
2.1	Analyse der Ubiquitous Language	2
2.2	Analyse und Begründung taktischer Muster	2
3	Clean Architecture	5
3.1	Schichtenarchitektur	5
4	Programming Principles	7
4.1	SOLID	7
4.2	GRASP – General Responsibility Assignment Software Patterns	8
4.3	DRY – Don’t Repeat Yourself	11
5	Refactoring	12
5.1	Code Smell – Zu viele <i>if-Bedingungen</i>	12
5.2	Code Smell – Duplicate Code	12
5.3	Code Smell – Long Method	13
5.4	Code Smell – Dead Code	13
6	Entwurfsmuster	14
6.1	Specification Pattern	14
7	Unit Testing	15
7.1	Beachtung der ATRIP-Regeln	15

Abbildungsverzeichnis

6.1	<i>Specification Pattern UMLs vorher und nachher</i> 	14
-----	--	----

Liste der Algorithmen

4.1	Low Coupling Beispiel; Klasse: <i>GetAllInstrumentRentalEntries</i> 	9
4.2	Low Coupling Beispiel; Klasse: <i>CreateNewMember</i> 	9
4.3	High Cohesion Beispiel; Klasse: <i>MembersRepositoryImpl</i> 	10
4.4	Don't Repeat Yourself; Internal Object: <i>MemberControllerProperties</i> 	11

Abkürzungsverzeichnis

DDD	Domain Driven Design	ii
DIP	Dependency Inversion Principle	8
DTO	Data Transfer Object	6
ISP	Interface Segregation Principle	8
JSON	JavaScript Object Notation	6
LSP	Liskov Substitution Principle	7
OCP	Open Closed Principle	7
SRP	Single Responsibility Principle	7

1. Allgemeines zum Projekt

Icons

- Alle mit 🔗 gekennzeichneten Elemente sind Hyperlinks
- Alle mit 💡 gekennzeichneten Absätze sind Definitionen oder Erläuterung von Begriffen, welche in der Vorlesung behandelt worden sind
- Alle mit </> gekennzeichneten Absätze sind die Umsetzung im Projekt und Anwendung der Prinzipien

Tatsächliche Umsetzung der geplanten Use Cases

Es wurde eine Anwendung entwickelt, welche eine Musikvereinsverwaltung mit eingeschränkter Funktion darstellt. Folgende UseCases wurden aus der *Themenmitteilung* 🔗 für einen *generischen* Musikverein umgesetzt:

- Erstellen eines Mitglieds
- Erstellen eines Instruments
- Verleihen eines Leihinstruments

Die initial geplanten UseCases „Gruppierung erstellen können“ und „Musiker einer Gruppierung hinzufügen können“ wurden aufgrund es ursprünglich angeforderten Umfangs von 1500 Zeilen *nicht* umgesetzt.

Vollständiger Code: *GitHub Repository* [liza-kl/ase-project](#) 🔗

Verwendete Technologie

- **Datenbank:** H2
- **ORM Framework:** Exposed
- **Backend Framework:** Ktor
- **Frontend Framework:** React

2. DDD

2.1 Analyse der Ubiquitous Language

💡 Unter der *Ubiquitous Language* versteht man die Sprache, die zwischen Domänenexperten und Entwicklern gesprochen wird. Das Vokabular dieser Sprache ist essentiell zum Verstehen der Domäne.

🔗 Die *Fachdomäne* (Musik)verein ist einfach gehalten. Es ist jedoch zu beachten, dass Begriffe wie *Instrument* nicht das *handwerkliche* Instrument meinen.

1. *Member* Ein Musikvereinsmitglied
2. *Instrument* Ein Musikinstrument und kein *handwerkliches* Instrument
3. *Instrument Rental Entry* Ein „Verleih“-Eintrag, welches Mitglied und verliehenes Instrument zusammenfasst
4. *Rental Instrument* Ein Instrument, welches im Verein vorliegt **und** als Verleihinstrument genutzt werden kann
5. *Instrument Type* Gibt den Instrument Typ an (bspw. eine Oboe)



2.2 Analyse und Begründung taktischer Muster

Aus der oben erarbeiteten Ubiquitous Language werden nun Objekte für die Applikation abgeleitet mit verschiedenen taktischen Muster des Domain Driven Designs.

Entities

💡 Eine *Entity* ist in einer Domäne *eindeutig* identifizierbar, bei unterschiedlichen IDs handelt es sich um unterschiedliche Entitäten. Darüber hinaus hat eine Entity einen *Lebenszyklus*, welcher sich verändert.

🔗 Das Projekt umfasst *zwei* Entitäten. Deren Attribute können verändert werden, sie können persistiert werden und haben eine *individuelle* Identität. Beispielsweise ist das Mitglied *Max Mustermann* mit der *memberId* = 1 nicht dasselbe Mitglied wie *Max Mustermann* mit der *memberId* = 3.

1. *Member*  – ein Musikvereinsmitglied, durch seine *memberId* unterscheidbar
2. *Instrument*  – durch seine *InstrumentIdentification* unterscheidbar.

Value Objects

💡 Im Gegensatz zu zu Entities besitzen Value Objects kein *eindeutiges* Zugehörigkeitsmerkmal, sie werden nur nach ihrem *Wert* bewertet. Value Objects sind identisch, wenn sie in allen ihren Werten übereinstimmen. Wird der Wert eines Value Objects verändert, so muss ein neues Value Object erzeugt werden.



🔗 Nachfolgend werden die implementierten Value Objects erläutert. Es sind Value Objects, da wie oben beschrieben sie bei gleichen Werten identisch zu behandeln sind.

1. *InstrumentCategory* Die *InstrumentCategory* prüft, um welche *Art* von Instrument es sich handelt (Blech-, Streichinstrument usw.). Es ist ein Value Object, da es kein Identifizierungsmerkmal (bspw. eine *InstrumentCategory-ID*) besitzt.
2. *InstrumentIdentification* Eine *InstrumentIdentification* wird für die Identifizierung eines Instruments benötigt. Die einzelnen Bestandteile dürfen nicht leer sein. Liegen zwei identische *InstrumentIdentifications* vor, so kann man von einem gleichen Instrumenten-**Modell** ausgehen.
3. *RentalRequest* Beinhaltet die benötigten Informationen, um ein Instrument auszuleihen.
4. *RentalRequestResult* Gibt in Kombination mit dem *Specification Pattern* zurück, ob ein *RentalRequest* bewilligt wird und falls nicht, die Gründe dafür.
5. *MemberName* Besteht aus einem *firstName* und *lastName*, welche beide nicht leer sein dürfen.
6. *MemberStatus* Der Status eines Mitglieds, kann derzeit die Ausprägung „ACTIVE“ oder „PASSIVE“ haben. Mit einem Status gehen bestimmte Rechte einher.

Aggregates

💡 Aggregates erlauben Entities und Value Objects als logische und verwaltbare Einheit zu definieren.

🔗 Das Projekt umfasst *zwei* Aggregate:

1. *InstrumentRentalEntry*  – stellt einen *Ausleiheintrag* dar. Verknüpft somit die Entitäten *Instrument* und *Member*
2. *RentalInstrument*  – stellt Instrumente im Musikvereinsinventar dar, welche ausleihbar sind. Jedoch nur solange die *quantity* > 0 ist.

Repositories

💡 Repositories geben notwendige Methoden vor, um Entitäten zu persistieren. In der Domain-Schicht werden sie als Interfaces definiert. *Implementiert* werden sie jedoch erst in der *Plugins*-Schicht. Sie helfen somit die Persistierung von Objekten in die äußere Schicht zu *abstrahieren*.

🔗 Das Projekt umfasst *vier* Repositories.

1. *MemberRepository* 🌀 – definiert Methoden, um mit der Member-Entity zu arbeiten.
2. *InstrumentRepository* 🌀 – definiert Methoden, um mit der Instrument-Entity zu arbeiten.
3. *InstrumentRentalEntryRepository* 🌀 – definiert Methoden, um mit dem InstrumentRentalEntry-Aggregat zu arbeiten.
4. *RentalInstrumentRepository* 🌀 – definiert Methoden, um mit dem RentalInstrument-Aggregat zu arbeiten.

Domain Service

💡 Domain Services enthalten *Geschäftslogik*, welche für die Erstellung weder in ein ValueObject sollen noch in eine Entität – diese Logik bzw. Regeln werden in sogenannte *Domain Services ausgelagert*.

🔗 Das Projekt umfasst *einen* Domain Service, den *RentalRequestService* 🌀. Dieser hat die Aufgabe zu überprüfen ob das Ausleihen eines Instruments genehmigt wird.

Der Vorteil durch die Auslagerung in einen eigenen Service ist, anstatt alle Regeln im UseCase *RentInstrument* 🌀 abzufragen und wenn mehr *Domänenregeln* dazukommen.

3. Clean Architecture

3.1 Schichtenarchitektur

Technische Umsetzung im Projekt

◄► Das Projekt ist in vier Schichten unterteilt (von innen nach außen):

- *domain*
- *use-cases*
- *adapters*
- *plugins*

Für jede Schicht wurde hierbei ein eigenes *Modul* erstellt mit dem dazugehörigen Code. Die Abhängigkeiten zeigen dabei von *innen* nach *außen*. Dabei verfügt die *plugins*-Schicht über das meiste „Wissen“, die *domain*-Schicht über das geringste.

Die beiden „inneren“ Schichten (*domain* und *use-cases*) definieren die Schnittstellen für die beiden „äußeren“ Schichten *adapters* und *plugins*. Diese Schnittstellen werden dann von den äußeren Schichten *implementiert*.

In den äußeren Schichten werden jeweils die inneren Schichten als *Dependencies* implementiert, jedoch keineswegs umgekehrt (siehe Dependency Inversion Principle 4.1) Umgesetzt durch einzelne *build-gradle.kts*-Dateien.

- *domain/build.gradle.kts* 🔗
- *use-cases/build.gradle.kts* 🔗
- *adapters/build.gradle.kts* 🔗
- *plugins/build.gradle.kts* 🔗

Die Module für das gesamte Projekt werden dabei über eine *settings.gradle.kts* 🔗 zusammengeführt.

Planung

Domain-Schicht

💡 Die Domainschicht gibt den *Rahmen* für die Anwendung vor.

⚡ Die modellierten Objekte und benötigte Services (in Kapitel 2 erläutert) befinden sich hier.

Use-Cases-Schicht

💡 In der Use-Cases Schicht (auch *Application* Schicht genannt) befindet sich die Business-Logik bzw. Applikationslogik. Diese basiert auf der Domain-Schicht, welche Business-Regeln vorgibt (ein *MemberName* darf beispielsweise nicht leer sein).

⚡ Ein Beispiel dafür ist der UseCase *RentInstrument* 🌀. Hier ist „Business“-Logik verankert wie, dass ein Instrument nur von einem *Member* mit dem Status *ACTIVE* ausgeliehen werden kann, das *RentalInstrument* muss vorhanden sein (*quantity* > 0) und das Instrument muss in der Instrumentenliste der geführt werden.

! Es gibt auch die Möglichkeit, sogenannte zusammenhängende *Services* zu erstellen, bei welchen alle benötigten Use Cases von einer Entität in eine Klasse geschrieben werden. In diesem Projekt wurde sich dagegen entschieden, da es übersichtlicher ist die Use Cases in eigene Klassen auszulagern. Dies hält die Klassen kleiner und übersichtlicher.

Adapter Schicht

💡 Die Adapter Schicht steht als „Vermittler“ zwischen der *äußeren* Schicht *plugins* und der inneren Schichten (*use-cases* und *domain*).

⚡ Dies wird durch *Data Transfer Object (DTO)s* 🌀 erreicht. Da in diesem Fall eine Webanwendung mit einem React Frontend vorliegt, werden in den DTOs die komplexen inneren Objekte vereinfacht und eine Serialisierung in JavaScript Object Notation (JSON) ermöglicht (durch die *kotlinx.serialization* Library).

Die dazugehörigen *Mapper* befinden sich als sogenanntes *Companion-Object* (in Java mit *statischen* Methoden vergleichbar) in den jeweiligen DTO-Klassen.

Plugins Schicht

💡 / ⚡ Die Plugins Schicht ist die Schicht, welche der größten Veränderung unterliegt. Hier finden die konkreten Implementierung statt (bspw. die Wahl des Frameworks fürs Backend).

Darüber hinaus befindet sich hier die REST Schnittstelle, die Datenbankanbindung, eine Funktion zur Starten des Webservers. Diese konkreten technischen *Details* gehören nicht in die inneren Schichten.

4. Programming Principles

4.1 SOLID

Single Responsibility Principle (SRP)

💡 Das SRP sagt aus, dass jede Klasse / Modul / Funktion für genau *einen* Aufgabenbereich zuständig ist.

🔗 Als Beispiel im Projekt kann man dafür die *Use Case* 🔗 Klassen nehmen. Ihre einzige Aufgabe ist es, die Applikationslogik für einen Use Case zu prüfen – und nicht noch die Art der Persistierung vorzugeben bspw. Dies führt zu einer besseren Wartbarkeit und Verständlichkeit des Codes.

Open Closed Principle (OCP)

💡 Das OCP sagt, dass Module sowohl offen (für Erweiterungen) sollten als auch verschlossen (für Modifikationen) sein sollen. Veränderungen an einer Stelle führen zwangsläufig zu Veränderungen an anderen Stellen, diese gilt es zu minimieren.

🔗 Im Projekt wurde das OCP unter anderem im RentalRequestService befolgt. Hier ist es immer möglich, neue *Regeln* 🔗 zu definieren, ohne eine *if-Bedingung* im UseCase *RentInstrument* 🔗 hinzuzufügen. Die einzige Änderung, die vorgenommen werden muss, ist das Hinzufügen der Regel in der Liste vom *RentalRequestService* 🔗.

Liskov Substitution Principle (LSP)

💡 Das LSP besagt, dass eine Klasse vom Typ T durch andere Klassen von Typ T ersetzbar sein sollte, ohne dass ein logischer Bruch in der Anwendung entsteht.

🔗 Ein Beispiel für das LSP ist die Tatsache, dass in der Anwendung eine beliebige DataSource verwendet werden kann, ohne dass ein logischer Bruch entsteht (bspw. In-Memory Datenbank, lokaler Speicher über eine Liste etc).

Beispiel 🔗, wo der Storage von MutableListStorage zu H2Storage im MemberController geändert worden ist. Die Storages verhalten sich gleich, auch wenn sie vom gleichen Typ / Interface sind (*MemberStorage* 🔗).

Interface Segregation Principle (ISP)

💡 Das ISP besagt, dass es besser ist, viele kleinere und spezifischere Interfaces zu haben als ein großes „generisches“ Interface.

Dadurch erreicht man unter anderem eine bessere Modularität, Wartbarkeit und klarere Aufgabenverteilung.

🔗 Im Projekt wurden in der Domain Schicht für benötigten Repository-Interfaces nach Entitäten bzw. Aggregaten aufgeteilt *repository package* 🌀. Somit ist der Client nicht gezwungen, ein großes Repository Interfaces mit allen (für die Anwendung) benötigten Methoden zu implementieren.

Dependency Inversion Principle (DIP)

💡 Das DIP sagt aus, dass *High-Level-Module* nicht von *Low-Level-Modulen* abhängen dürfen. Umformuliert: Abstraktionen dürfen nicht von Details abhängen. Durch diese strikte Trennung vermeidet man ein steigendes Maß an Komplexität und beugt *zyklische* Abhängigkeiten vor.

🔗 Dieses Prinzip wurde durch den Einsatz der *Clean Architecture* in diesem Projekt umgesetzt. Die inneren Schichten *Domain* und *Use Cases* definieren Schnittstellen, mit welchen die äußeren Schichten *Adapters* und *Plugins* arbeiten können und diese *realisieren*.

4.2 GRASP – General Responsibility Assignment Software Patterns

💡 Die GRASP-Prinzipien helfen, gutes Object-Oriented Design zu erreichen.

Information Expert

💡 Für eine Aufgabe soll derjenige zuständig sein, der das meiste Wissen hat. Dadurch fördert man unter anderem hohe Kohäsion.

🔗 Soll geprüft werden, ob ein *RentalRequest* angenommen oder verworfen wird, so nimmt man die Klasse *RentalRequestService* 🌀 anstatt des *UseCases RentInstrument*. *RentalRequestService* enthält alle Informationen, um dies zu entscheiden.


Creator

💡 Das Creator Pattern gibt vor wer für Erzeugung einer Instanz zuständig sein soll sein.

🔗 Als Beispiel für das Creator Pattern soll die *MemberStorageFactory* 🌀 genommen werden. Diese ist eine Factory und somit für das Creator Pattern geeignet.


Controller

💡 Der *Controller* ist die erste Schnittstelle nach der GUI, nimmt die erhaltenen Requests entgegen und delegiert an dazugehörige Module weiter.

✎ Im Projekt gibt es ein *controller*  package, welches für die Entitäten die Events annimmt und die dazugehörigen UseCases aufruft.

Indirection

💡 Das *Indirection* Pattern unterstützt niedrige Kopplung, in dem es einen „Vermittler“ zwischen *Client* und *Server einführt*. Dies fördert eine niedrige Kopplung unter den Modulen und ist flexibler als Vererbung.

✎ Die Vermittler in diesem Fall sind teilweise die DTOs auf der *adapter* -Schicht des Projekts. Man könnte theoretisch direkt auf die Entities zugreifen von der Plugins Schicht. Durch den Adapter ist man jedoch flexibler bei Veränderung an der Entität.

Low Coupling – Niedrige Kopplung


💡 Bei dem Prinzip der niedrigen Kopplung geht es darum, dass Klassen möglichst *wenig* Abhängigkeiten zu anderen aufweisen.

✎ Das beiden aufgeführten Beispiele 4.1 und 4.2 zeigen jeweils einen Use Case (liegen in der *use-case*-Schicht). Beide Use Cases benötigen ein Repository von einem bestimmten Typ.

In beiden Fällen wird als Argument der Funktion *keine* konkrete Implementierung des Repositories übergeben, sondern lediglich ein Interface vom benötigten Typ. Die Kopplung zu einer *bestimmten* Implementierung wird somit ausgeschlossen (dieses „Detail“ gehört auch nicht in die Applikationslogik). Die konkrete Implementierung des Repositories liegt in der *plugins*-Schicht. Somit ist die niedrige Kopplung gegeben.

```

1 package de.dhbw.ka.instrumentrental
2
3 class GetAllInstrumentRentalEntries(private val instrumentRentalEntryRepository :
4     InstrumentRentalEntryRepository) {
5     fun execute() : List<InstrumentRentalEntry> {
6         return instrumentRentalEntryRepository.getAllRentalEntries()
7     }
8 }
```

Algorithmus 4.1: Low Coupling Beispiel; Klasse: *GetAllInstrumentRentalEntries* 

```

1 package de.dhbw.ka.members
2
3 class CreateNewMember(private val memberRepository: MemberRepository) {
4     fun execute(memberData: Member) : Boolean {
5         return memberRepository.create(memberData)
6     }
7 }
```

Algorithmus 4.2: Low Coupling Beispiel; Klasse: *CreateNewMember* 

High Cohesion – Hohe Kohäsion

💡 Das Maß der „Kohäsion“ in einem Projekt sagt aus, wie viel *logischer* Zusammenhang in den einzelnen Klassen besteht.

🔗 Als Beispiel lässt sich hierfür die Implementierung eines Repositories nehmen, vgl. Code Listing 4.3.

In diesem Fall weiß die konkrete Implementierung nur, dass sie mit einem `MemberStorage` interagieren muss, sonst nichts. Darüber hinaus kümmert sich das `MembersRepositoryImpl` 🔗 nicht um Aufrufe im `InstrumentStorage` o.Ä.

```

1 package de.dhbw.ka.repository
2
3 class MembersRepositoryImpl(private val memberStorage: MemberStorage) :
4     MemberRepository {
5     override fun findAll(): List<Member> {
6         val result = memberStorage.findAll()
7         return result.map { toMember(it) }
8     }
9 }
```

Algorithmus 4.3: High Cohesion Beispiel; Klasse: `MembersRepositoryImpl` 🔗

Polymorphismus

💡 Der Polymorphismus (dt. die *Vielgestaltigkeit*) sagt aus, dass beim Zugriff auf Methoden mit *identischer* Signatur diese unterschiedliche Ergebnisse liefern. Sprich, Methoden mit gleichem Namen werden unterschiedlich realisiert.

🔗 Ein Beispiel im Projekt sind dafür die *Storage Interfaces* 🔗.

Hierbei können die Methoden der einzelnen Storages in einer konkreten Implementierung (bspw. *H2 Implementierungen* 🔗) überschrieben und passend gemacht werden.

In einer früheren Version des Projekts wurde zum Beispiel ein *LocalStorage* 🔗 verwendet, der aus einer „Liste“ bestand.

Welche Implementierung verwendet wird, wird im *Controller* 🔗 in den Repositories festgelegt.

Protected Variation

💡 Interfaces sollen immer verschiedene konkrete Implementierungen verstecken. Man nutzt also Polymorphismus und Delegation, um zwischen den Implementierungen zu wechseln. Dadurch kann das restliche System vor den Auswirkungen eines Wechsels der Implementierung geschützt werden.

🔗 Ein Beispiel dafür sind die Storage Interfaces, welche unterschiedliche Implementierungen von Speicher ermöglichen ohne dabei andere Objekte zu tangieren und zu beeinflussen.

Pure Fabrication

💡 Pure Fabrications stellen meistens Hilfsklassen dar, die so nicht in der Problemdomäne existieren. Sie trennt Technologiewissen von Expertenwissen.

🔗 Repositories in der Domäne stellen beispielsweise „reine Erfindungen“ dar. Ohne diese Interfaces hätte man die Option, „Persistenzfunktionen“ in den Entitäten zu definieren. Dies würde jedoch die Kohäsion mindern, deshalb lagert man diese „Aufgabe“ in Repositories aus und wahrt somit das Maß der Kohäsion.

4.3 DRY – Don't Repeat Yourself

💡 Dieses Prinzip soll helfen, Redundanzen zu vermeiden, in dem man sich nicht wiederholt.

🔗 Im *MemberController* wurde dies dadurch erreicht, in dem man ein sogenanntes *internal Object* implementiert hat, wo man den Typ des Persistenzspeichers angeben kann (in diesem Fall, da könnte aber jede andere Speicherform möglich sein). Dies gibt man einmal an und kann die Variable durch den ganzen Controller hindurch aufrufen kann. Bei Änderung der Speicherform muss man diese nur *einmal* ändern.

```

1 internal object MemberControllerProperties {
2     private val memberStorageFactory = StandardMemberStorageFactory()
3     private val memberStorage = memberStorageFactory.createMemberStorageFromType(
4         "h2")
5     val memberRepository: MemberRepository = MembersRepositoryImpl(
6         memberStorage = memberStorage
7     )
8 }

```

Algorithmus 4.4: Don't Repeat Yourself; Internal Object: *MemberControllerProperties* 🔗

5. Refactoring

5.1 Code Smell – Zu viele *if-Bedingungen*

Commits:

1. [7ebd605885582043c6b230cb8fcf3ea8910f1583](#) 🔗
2. *Commit History für RentInstrument Use Case* 🔗

🔗 In der *use-case* Schicht befand sich ein *RentInstrument* 🔗 Use Case welcher drei verschiedene Bedingungen geprüft hat bevor ein Instrument ausgeliehen werden konnte.

Problematisch wird es jedoch, wenn mehr Bedingungen hinzukommen (bspw. die Prüfung eines Status, vlt ist ein Mitglied aktiv, darf aber keine Instrumente ausleihen aus bestimmten Gründen usw.). Für jede neue Bedingung müsste man eine neue *if-Abfrage* hinzufügen in der UseCase-Klasse. Dies würde unter anderem das OCP verletzen.

5.1.1 Fix

💡 Dieser Code Smell lässt sich mittels eines Specification Patterns 6.1 lösen. Das Specification Pattern erlaubt uns, Objekte gegen bestimmte Kriterien zu prüfen. Diese *Specifications* lassen sich aneinanderreihen und somit werden komplexere Abfragen möglich.

🔗 In der Domain Schicht wurden hierfür *Rules* 🔗 definiert, welche in einem *RentalRequestService* 🔗 geprüft werden. Diese können bei Bedarf erweitert oder entfernt werden.

5.2 Code Smell – Duplicate Code

Commits:

1. [119184607e0771586c5e425ace18ff2914b0e617](#) 🔗
2. [fbd08fe800f57ffac91240a0dd44935938c95526](#) 🔗

In der *plugins* Schicht befand sich in den einzelnen controller Dateien mehrere Code Duplikate, wo sich herausstellte, dass man diese mit einem *internal* Objekt lösen können. Der commit ist hierbei

Begründung

Dieser Code Smell wurde behoben um eine „Single Source of Truth“ zu haben. Da das Entfernen des Code Duplikats keine Unleserlichkeit bzw. nicht die Verständlichkeit des Codes mindert, war dieses Refactoring in Ordnung (vgl. Refactoring Guru – Duplicate Code 🐛)

Fix

Gefixt wurde der Code Smell mit einem sogenannten *internal*-Objekt von Kotlin. Hierbei wurden die einzelnen Variablen definiert von der Storage-Implementierung und der Repository-Implementierung.

Dies hat nun den Vorteil, dass bei einer Veränderung der Implementierung, diese nur einmal in dem *internal* Objekt verändert werden muss. Somit wird auch das **DRY** Principle befolgt.

5.3 Code Smell – Long Method

1. `674e2c4e96f55fe7973f58da128e617e5a81c9bc` 🐛

Fix und Begründung

Anstatt alles in die *execute()* Methode der Klasse zu packen, wurden die Logikabfragen in eine eigene Methode ausgelagert. Dies fördert die Lesbarkeit des Codes.

5.4 Code Smell – Dead Code

Commits:

1. `317c062a369e8dde0bf34a29bd75bf22b8acbe15` 🐛

Fix und Begründung

Da der Code nicht gebraucht wird, sollte er nicht unnötig in der Codebasis vorliegen und wurde somit gelöscht. Sollte er dennoch gebraucht werden, gibt es die Möglichkeit der Wiederherstellung durch die Versionsverwaltung mittels *git*.

6. Entwurfsmuster

6.1 Specification Pattern

💡 Das Specification Pattern wird verwendet, um Business-Regeln aneinanderreihen zu können und somit *komplexe Abfragen* zu erleichtern. Hierbei müssen Regeln einmal definiert werden und können wiederverwendet werden.

🔗 Im Projekt wurde dieses Entwurfsmuster eingesetzt, um eine Kaskade an if-Bedingungen zu vermeiden (siehe Refactoring 5.1). Vorher wurden nämlich die Bedingungen für einen Rental-Request im UseCase RentInstrument geprüft. Nach dem Einsatz des Entwurfsmusters wurden designierte Klassen für die Regeln definiert und in einem RentalRequestService verarbeitet.

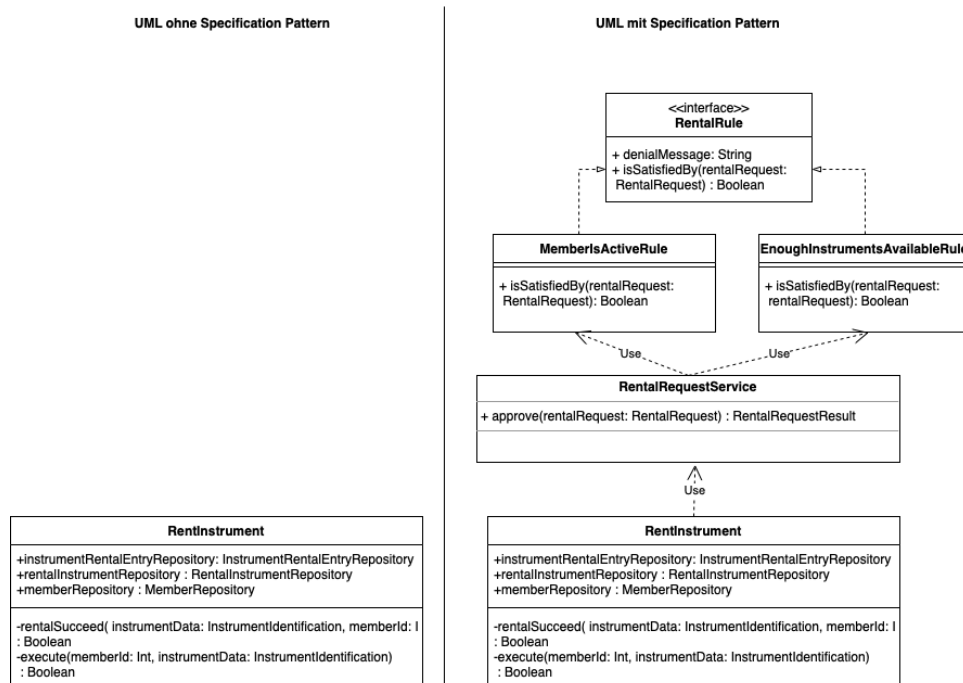


Abbildung 6.1: *Specification Pattern UMLs vorher und nachher* 🌀

7. Unit Testing

Testklassen:

1. *ControllerTests* (1) 🔗
2. *EntitiesTestClass* (3) 🔗
3. *VOTestClass* (3) 🔗
4. *UseCaseTests* (4) 🔗

7.1 Beachtung der ATRIP-Regeln

- Automatic → **Gegeben**, da alle Tests mit einem Gradle Befehl `./gradlew test` ausgeführt werden können
- Thorough → **Gegeben**, die vorhandenen Tests überprüfen die benötigte Funktionalität
- Repeatable → **Gegeben**, da alle Tests beliebig oft wiederholbar sind und immer das gleiche Ergebnis liefern
- Independent → **Gegeben**, die gegebenen Tests sind nicht von anderen Tests abhängig
- Professional → Es wurden keine unnötigen Tests oder Code geschrieben (bspw. für *Getter* oder *Setter*)

Einsatz von Mocks

Für die Testung der *Use Cases* 🔗 wurden Mocks der benötigten Repository Klassen verwendet, um Unabhängigkeit zu erreichen (die Bibliothek hierfür war *mockk.io* 🔗).