DHBW
Duale Hochschule
Baden-Württemberg
Karlsruhe

Plan on Rocks

–

Programmentwurf

der Vorlesung Advanced Software Engineering

an der

Dualen Hochschule Baden-Württemberg Karlsruhe

von

Lucie Weber

Abgabedatum	17.05.2022
Matrikelnummer	5691176
Kurs	tinf19B4
Gutachter der Studienakademie	Mirko Dostmann

Inhaltsverzeichnis

Kapitel 1

Projektdefinition

1.1 Start Bemerkungen

Zu Beginn des Projekts wurde neben der Umsetzung des Backends auch die Umsetzung eines Frontends geplant. Nach Aufsetzen und ersten Implementierungen im Frontend ist schnell klar geworden, dass dieses für die Anwendung der zu lernenden Konzepte des Softwareengineerings nicht geeignet ist. Daher konzentriert sich dieser Programmentwurf auf die Implementierung und Analyse des Backends. Zur Umsetzung der Anwendung kann zukünftig das Frontend erweitert werden. Aufgrund dieses Fokusses sind im letzten Use Case „Wettericon in Ausflugstipss anzeigen“ nicht alle Anforderungen umgesetzt worden, da diese in größeren Teilen frontendseitige Überprüfungen enthalten.
Der Vollständigkeit halber sollen beide Links zum Backend- und Frontend-Repository zur Verfügung gestellt werden. Zum Nachvollziehen der theoretischen Betrachtungen dieser Arbeit ist nur das Ausführen des Backends erforderlich. Dazu liegt eine Readme-Datei im Repository vor.

- PlanOnRocks-Backend
- PlanOnRocks-Frontend

1.2 Use Cases

Neuen Kletterfels anlegen
Ein Kletterfels hat einen Namen, einen Standort (GPS-Koordinate), einen Schwierigkeitsgrad (einfach, mittel, schwer), einen Absicherungsgrad (schlecht, okay, gut, sehr gut). Der Standort muss eindeutig sein. Existiert der Standort bereits wird die Anlage verweigert.

Kletterfels labeln
Der aktuelle Standort des Gerätes des Benutzers wird automatisch ermittelt und damit die Entfernung zum Kletterfelsen berechnet. Über die berechnete Entfernung werden die Kletterfelsen in vier Kategorien unterteilt:

- $<$ 50km: Halbtagesausflug
- $<$ 140km: Tagesausflug
- $<$ 250km: Wochenendausflug
- $>=$ 250 km: Urlaub

Ausflug anfragen
Ein Ausflug kann erst angefragt werden, wenn bereits 5 Kletterfelsen angelegt sind. Bei der Anfrage muss der Benutzer die Zeit angeben (Datum/Zeitspanne) und das Teilnehmer-Können (Anfänger, Fortgeschritten, Profi). Ist die Zeitangabe nur ein Datum, wird eine weitere Abfrage generiert, ob es sich um einen Halbtages- oder Tagesausflug handelt.

Ausflugstipp erstellen
Anhand von der Datumangabe werden die entsprechenden, bereits angelegten Kletterfelsen durchsucht:

- 1 Tag + Checkbox Halbtagesausflug: Halbtagesausflug
- 1 Tag + Checkbox Tagesausflug: Tagesausflug
- 2-3 Tage: Wochenendausflug
- Länger als 3 Tage: Urlaub

Zusätzlich wird die entsprechende Liste der Kletterfelsen gefiltert nach dem angegebenen Teilnehmer-Können:

- Anfänger: Kletterfelsen mit Schwierigkeitsgrad einfach und Absicherung gut oder sehr gut
- Fortgeschritten: Schwierigkeitsgrad einfach/mittel; Absicherung okay/gut/sehr gut
- Profi: Schwierigkeitsgrad einfach/mittel/schwer; Absicherung schlecht/okay/gut/sehr gut

Wettericon in Ausflugstipps anzeigen
Liegt das Ausflugs Datum maximal 8 Tage in der Zukunft, soll in den Ausflugstipps, ein Wettericon (Sonne/Wolke/Regen/Schnee) das Wetter für das angegebene Datum und den Standort des jeweiligen Felsen anzeigen. Regnet es in über drei Felsstandorten (falls es mehr als drei Felsstandorte gibt; sonst gilt: regnet es an allen Felsstandorten) kommt eine Warnmeldung: "Willst du wirklich an diesem Tag klettern gehen?"

1.3 Technologie Stack

Frontend

- Framework: Angular 2.0
- Sprache: TypeScript
- IDE: WebStorm

Backend

- Framework: Spring
- Sprache: Java
- IDE: IntellJ
- Testen: JUnit4
- Datenbank: H2 Datenbank

Projektmanagement
Versions Kontrolle: Git and GitHub

Kapitel 2

Domain Driven Design

2.1 Analyse der Ubiquitous Language

Als einheitliche Übersetzung und Verwendung der Domänen Begriffe im Code wird folgendes Glossar verwendet:

2.1.1 Glossar

Absicherungsgrad
Der Absicherungsgrad gibt an wie gut ein Kletterfels abgesichert ist. Das beinhaltet sowohl nach wie vielen Metern ein Bohrhaken kommt, als auch wie sinnvoll die Bohrhaken gesetzt sind. Der Absicherungsgrad kann schlecht, okay, gut oder sehr gut sein.
=> im Domänencode: bolting (bad, okay, good, very good)

Ausflug
Ein Ausflug hat ein Datum oder eine Zeitspanne. Er beinhaltet die Angabe zur Teilnehmererfahrung und einer Ausflugskategorie. Liegt das Ausflugsdatum oder das Startdatum der Zeitspanne weniger als acht Tage in der Zukunft, enthälte er auf Informationen über das Wetter.
=> im Domänencode: trip

Ausflugskategorie
Die Ausflugskategorie wird anhand der Entfernung des Benutzergerätes zum Kletterfelsen bestimmt. Dabei werden vier Kategorien anhand der berechneten Entfernung unterschieden:

- $<$ 50km: Halbtagesausflug

- $<$ 140km: Tagesausflug
- $<$ 250km: Wochenendausflug
- $>=$ 250 km: Urlaub

=> im Domänencode: trip category (half-day trip, day-trip, weekend trip, vacation)

Ausflugsziel
Ein Ausflugsziel ist ein Kletterfels, der die Voraussetzungen eines Ausflugs erfüllt. Das heißt die gewünschte Ausflugskategorie besitzt und dem Teilnehmererfahrung entspricht. Ein Ausflugsziel existiert in der Domäne ist technisch gesehen allerdings ein Kletterfels.
=> im Domänencode: trip destination

Benutzergerät
Das Benutzergerät hat einen Standort. Damit wird die Entfernung zum Kletterfels berechnet. Der Standort kann sich verändern und wird beim Öffnen der Anwendung neu abgefragt.
=> im Domänencode: user device

Entfernung
Die Entfernung ist als Entfernung zwischen dem Standort des Kletterfelsens und dem Standort des Benutzergerätes definiert.
=> im Domänencode: distance

Kletterfels
Ein Kletterfels wird vom Benutzenden angelegt und hat einen Standort, ein Schwierigkeitsgrad, ein Absicherungsgrad und kann einer Ausflugskategorie zugeordnet werden.
=> im Domänencode: climbing rock

Schwierigkeitsgrad
Der Schwierigkeitsgrad ist eine Eigenschaft des Kletterfelsens. Er gibt an wie schwer die Routen an dem Kletterfels im Durchschnitt sind. Er kann als einfach, mittel oder schwer angegeben sein.
=> im Domänencode: difficulty (easy, medium, hard)

Standort

Sowohl ein Kletterfels als auch ein Benutzergerät besitzt ein Standort. Der Standort wird in Form einer GPS-Koordinate angegeben und ist somit einzigartig.

=> im Domänencode: location

Teilnehmererfahrung

Mit der Teilnehmererfahrung soll es möglich sein anzugeben wie viel Können und Wissen die Teilnehmenden eines Ausflugs mitbringen. Dabei wird in Anfänger, Fortgeschritten und Profi unterschieden.

=> im Domänencode: participant experience (beginner, advanced, professional)

Wetter

Das Wetter ist eine Angabe für Ausflüge, um entscheiden zu können, ob es sinnvoll ist an dem ausgewählten Zeitpunkt klettern zu gehen. Das Wetter wird vereinfacht in vier Kategorien aufgeteilt: Sonne, Wolke, Regen, Schnee.

=> im Domänencode: weather (sun, cloud, rain, snow)

2.2 Analyse und Begründung der verwendeten taktischen Muster

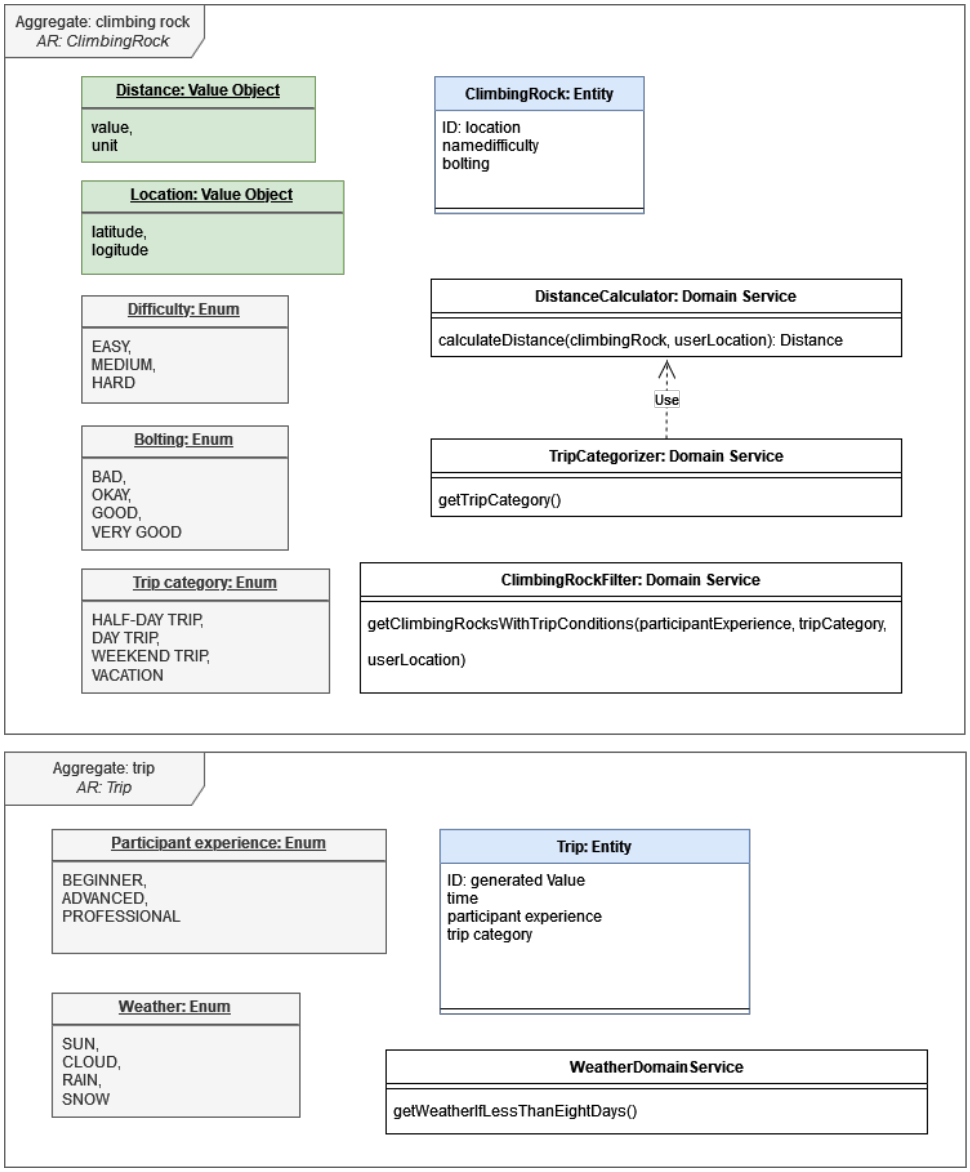

Abbildung 1: Planung des Domain Driven Designs

2.2.1 Value Objects

- Location
- Distance

Bei der Umsetzung der *Location* und der *Distance* wurden Value Objects verwendet. Die *Distance* ist eine klassisches Beispiel für die Kapselung eines gleichbleibenden Wertekonzepts und wurde implementiert, damit der Wert und die Einheit der *Distance* nicht getrennt werden können.
Die *Location* ist ebenfalls ein gleichbleibendes Wertekonzept aus *Latitude* und *Longitude*. Sie wurde zuerst als Embeddable in der *ClimbingRock-Entity* implementiert, danach aber nochmal rausgezogen, damit sie mehrfach verwendet werden kann. Denn die *userLocation* soll ebenfalls über das Value Object *Location* abgebildet werden können.

2.2.2 Aggregates

Das Projekt enthält zwei Entitäten und damit auch zwei Aggregates (vgl. Abbildung 1). Das erste Aggregate ist das *ClimbingRock Aggregate*. Es enthält neben der AggregateRoot, dem *ClimbingRock Entity*, die zur Entity gehörenden Value Objects, Enums und Domain Services. Das zweite Aggregate ist das *Trip Aggregate* mit der *Trip Entity* als Aggregate Root.

2.2.3 Entitäten

- Climbing Rock
- Trip

Der *ClimbingRock* ist als Entität umgesetzt, damit er gespeichert werden kann und der Anwendung nach dem Speichern immer wieder zur Verfügung steht. Das Gleiche gilt für den *Trip*. Bei beiden wird eine ID mittels JPA-Methoden automatisch generiert.

2.2.4 Repositories

- ClimbingRockRepository
- TripRepository

Die Repositories wurden implementiert, um eine Schnittstelle zur H2-Datenbank zur Verfügung zu haben. Über diese Schnittstelle können die oben genannten Enitities gespeichert und verwaltet werden.

2.2.5 Domain Services

- DistanceCalculator
- TripCategorizer
- ClimbingRockFilter
- Weather

Der *DistanceCalculator* hat die Aufgabe die komplexe Berechnung der Entfernung zweier Koordinaten-Paare durchzuführen. Da er keiner bestimmten Entity zugeordnet ist, sondern zur Berechung zum Einen die *ClimbingRockLocation* und zum Anderen die *UserLocation* benötigt, ist er als Domain Service umgesetzt. Ein weitere Grund für dafür ist, dass die komplexe Entfernungsberechnung nicht das Domänenmodell mit unnötiger Komplexität überlastet. Der *TripCategorizer* verwendet den *DistanceCalculator*, um dessen Ergebnis auf die passende *TripCategory* zu mappen. Die anderen Domain Services wurden aus ähnlichen Gründen wie der *DistanceCalculator* als Domain Service umgesetzt.

Kapitel 3

Clean Architecture

3.1 Planung und Begründung der Schichten

Das Projekt beinhaltet die Implementierung von vier Schichten:

- Domain (innerste Schicht)
- Application
- Adapters
- Plugins (äußerste Schicht)

Es enthält zur möglichen Erweiterung bereits als fünfte Schicht, den Abstraction Code. In dieser Schicht sind bis jetzt allerdings keine Klassen implementiert. Jede Schicht ist im Projekt als einzelnes Maven Modul umgesetzt. Dies soll die Unabhängigkeit voneinander gewährleisten beziehungweise ermöglicht die Dependency Rule einzuhalten. Die Dependency Rule besagt, dass Abhängigkeiten von außen nach innen zeigen. Diese Regel wurde ebenfalls mit der Hilfe der Maven Module umgesetzt. Jedes Maven Modul einer Schicht bekommt immer nur die Abhängigkeit der Schicht eins tiefer zugewiesen.

3.1.1 Was ist in den Schichten umgesetzt?

Domain
In dieser Schicht wird die Domäne abgebildet, d.h. das komplette *ClimbingRock Aggregate* und das komplette *Trip Aggregate*.

Application

Auf dieser Schicht werden die Use Cases anhand von Application Services implementiert. Zusätzlich erfolgt hier die Implementierung der DomainService Interfaces aus der Domänen-Schicht.

Adapters

Auf der Adapter Schicht wird das Mapping zwischen den Daten Objekten aus dem Frontend und den Entities aus dem Backend umgesetzt.

Plugins

Die Plugin-Schicht enthält die Implementierungen der Repositories, die eine starke Abhängigkeit zu externen Frameworks haben und die Implementierung der RestController als Schnittstelle zum Frontend.

Kapitel 4

Programming Principles

4.1 SOLID

4.1.1 Single Responsibility Principle

Das Single Responsibility Principle ist in allen taktischen Entwurfsmustern umgesetzt. Beispielsweise ist die *ClimbingRock Entity* nur zur Modellierung der Eigenschaften des Kletterfelsens verantworlich.

4.1.2 Open Close Principle

Das Open Close Principle ist beispielsweise im WeatherService beachtet worden. Dieser ist offen für Veränderungen, z.B. das Hinzufügen einer neuen Wetter-Kategorie im *Weather Enum*. Auf der anderen Seite ist er geschlossen gegenüber Veränderungen durch den Standard-Return Wert *Weather.NO_FORECAST* der *getWeather*-Methode.

4.1.3 Liskov Substitution Principle

Das LSP wird durch den Einsatz von Interfaces im Projekt angewendet, z.B. dem Interface des ClimbingRockFilterDomainServices. Eine Klasse, die von diesem Interface erbt, ist durch andere Klassen, die ebenfalls von diesem Interface erben, ersetzbar.

4.1.4 Interface Segregation Principle

Anstatt ein großes Repository Interface zu implementieren, wird im Projekt für jede Entity ein eigenes Interface implementiert.

4.1.5 Dependency Inversion Principle

Die DIP wird durch den Einsatz von Clean Architecture im Projekt umgesetzt.

4.2 GRASP

GRASP ist die Abkürzung für General Responsibility Assingment Software Patterns.

4.2.1 Information Expert

Im Projekt stellte sich beispielsweise die Frage, wer zur Bestimmung des Wetters zuständig ist. Auswahl war zwischen dem *ClimbingRockService* und dem *TripService*. Da der *TripService* bereits das Wissen über den *Trip* enthält und damit mehr für den Use Case relevantes Wissen als der *ClimbingRockService*, wurde diese Methode im *TripService* implementiert.

4.2.2 Creator

Creator sind die Mapper auf der Adapterschicht. Sie sind ganz klar dafür zuständig entweder ein Entity-Objekt oder ein DTO-Objekt zu erstellen.

4.2.3 Controller

Im Projekt gibt es zwei Controller (siehe Liste unterhalb), die die REST-Anfragen entgegennehmen und über die Services an die Domänen-Schicht weiterleiten.

- ClimbingRockController
- TripController

4.2.4 Indirection

Als Vermittler zwischen Client und Server dienen die Mapper in der Adapter-Schicht, die die DTOs in Entities umwandeln und umgekehrt.

4.2.5 Niedrige Kopplung

Ein Beispiel für niedrige Kopplung ist der DistanceCalculatorService. Dieser stellt Methoden bereit und ist dabei unabhängig von anderen Services oder Entities.

4.2.6 Hohe Kohäsion

Hohe Kohäsion ist auch beim *DistanceCalculator* gegeben. Er weißt nur, dass er ein Domain-Service ist und was er berechnen muss, nicht aber, in welchem Zusammenhang er verwendet wird.

4.2.7 Polymorphismus

Polymorphismus wird im Projekt nicht angewendet, da dies technisch nicht notwendig ist, um die Use Cases sinnvoll umzusetzen.

4.2.8 Protected Variation

Auch hier soll wieder auf den *DistanceCalculator* eingegangen werden. Über das Interface *DistanceCalculatorDomainService* wird nur die Implementierung der *getDistance*-Methode gefordert. Dadurch wird die konkrete Implementierung der *calculateDistance*-Methode versteckt. Diese ist private und andere Services, die vom Interface *DistanceCalculatorDomainService* erben, wissen nichts von dieser Methode.

4.2.9 Pure Fabrication

Eine Pure Fabrication ist beispielsweise der LocationConverter. Er existiert in der Problemdomäne nicht, wird allerdings für die technische Umsetzung des Value Objects *Location* benötigt.

4.3 DRY

DRY wird im Projekt unter anderem in dem Unit Test ClimbingRockFilterServiceTest umgesetzt. Anstelle, dass das zu vergleichende Objekt zum Start des Tests immer neu erstellt wird, wird es in eine *@BeforeEach*-Methode ausgelagert.

Kapitel 5

Refactorings

5.1 Codesmell - keine Wiederverwendbarkeit

- Commit: 4c98a7f60fd969d633b8b56f64e77e68e17350ae

In diesem Refactoring wurde das Mapping der Location aus dem *ClimbingRockMapper* in ein eigenen *LocationMapper* ausgelagert, um die Wiederverwendbarkeit dieses Mappings zu ermöglichen.

5.2 Codesmell - zu lange Methode

- Commit: a77abfbe299214bb75c659f29febf2f46dc103d5

In diesem Refactoring wurde die *mapToLocation*-Methode angepasst, sodass sie nun übersichtlicher und leichter verständlich ist.

Kapitel 6

Entwurfsmuster

Aufgrund der geringen Vorschriften an die Business-Logik sind viele Entwurfsmuster nicht für den Einsatz im Projekt geeignet. Das Builder-Pattern hat keine sinnvolle Anwendung, da die Entitäten *ClimbingRock* und *Trip* beide nur Attribut-Felder enthalten, die nicht optional sind. Daher ist die Möglichkeit unterschiedlicher Konstruktionsprozesse hier nicht notwendig und wäre nur eine Aufblähung der Komplexität. Auch das Observer-Pattern ist nicht geeignet, da sich die Zustände der Entitäten nicht verändern. Was sich verändert ist die *userLocation*, die beispielsweise eine Änderung der *TripCategory* eines Kletterfelsens bewirkt. Dieses Attribut wurde aus diesen Gründen von Beginn an nicht in der *ClimbingRock*-Entität als Feld aufgenommen, sondern wird immer dynamisch bestimmt. Dadurch hat es keinen Bedarf an einem Observer. Aus diesen Gründen soll im Folgenden nur das verwendet Bridge-Pattern beschrieben werden.

6.1 Bridge

Das Bridge-Pattern wird in diesem Projekt für die Implementierung der Repositories verwendet, um unabhängig von Spring Data und JPA zu sein. Dadurch ist es in Zukunft möglich, das Persistenzframework zu wechseln, ohne den restlichen Code verändern zu müssen.
Die Bridge verknüpft dabei den Domänencode mit den Methoden des Persistenzframeworks.

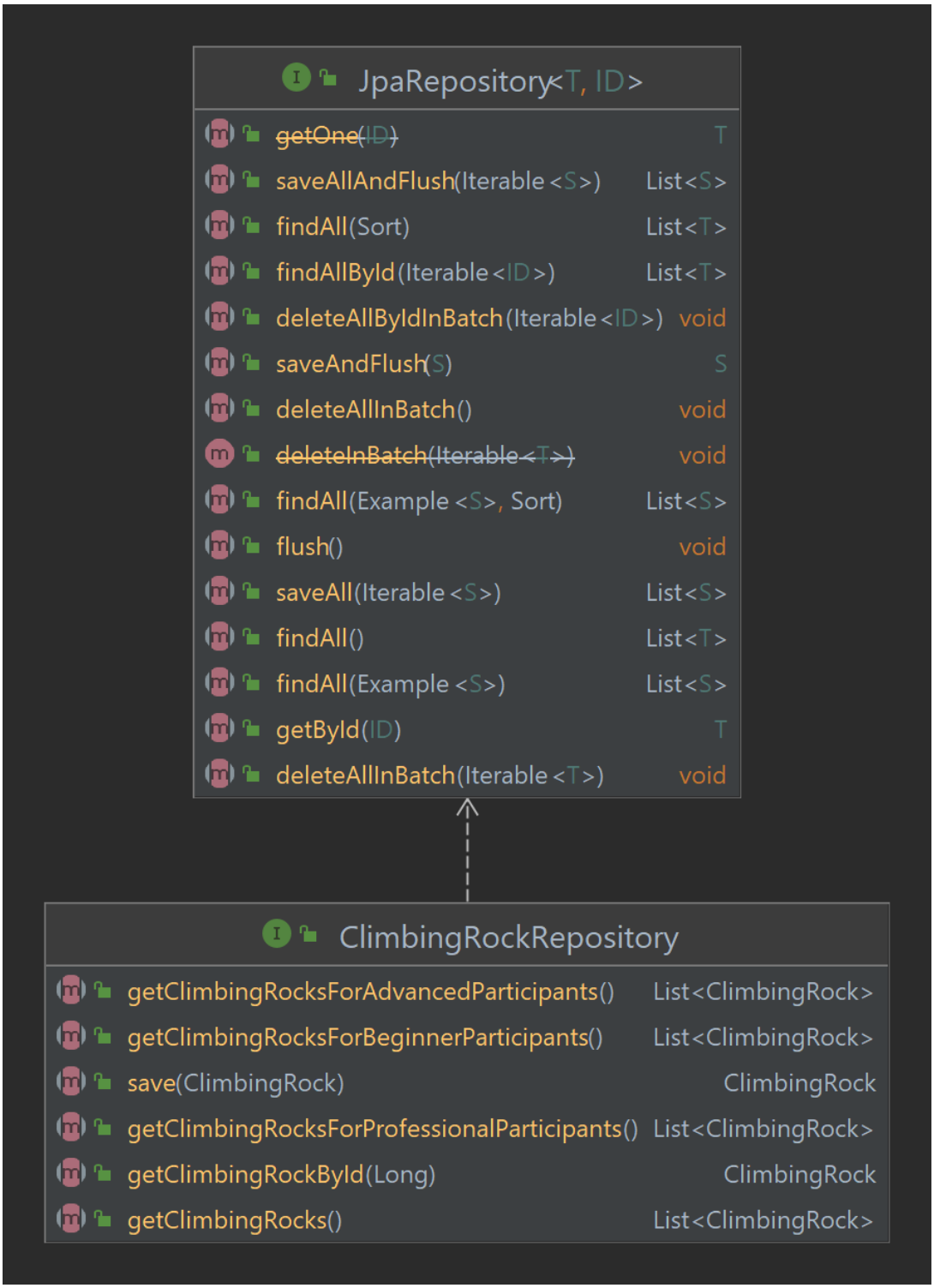

Abbildung 2: Repository Implementierung vor dem Bridge-Pattern

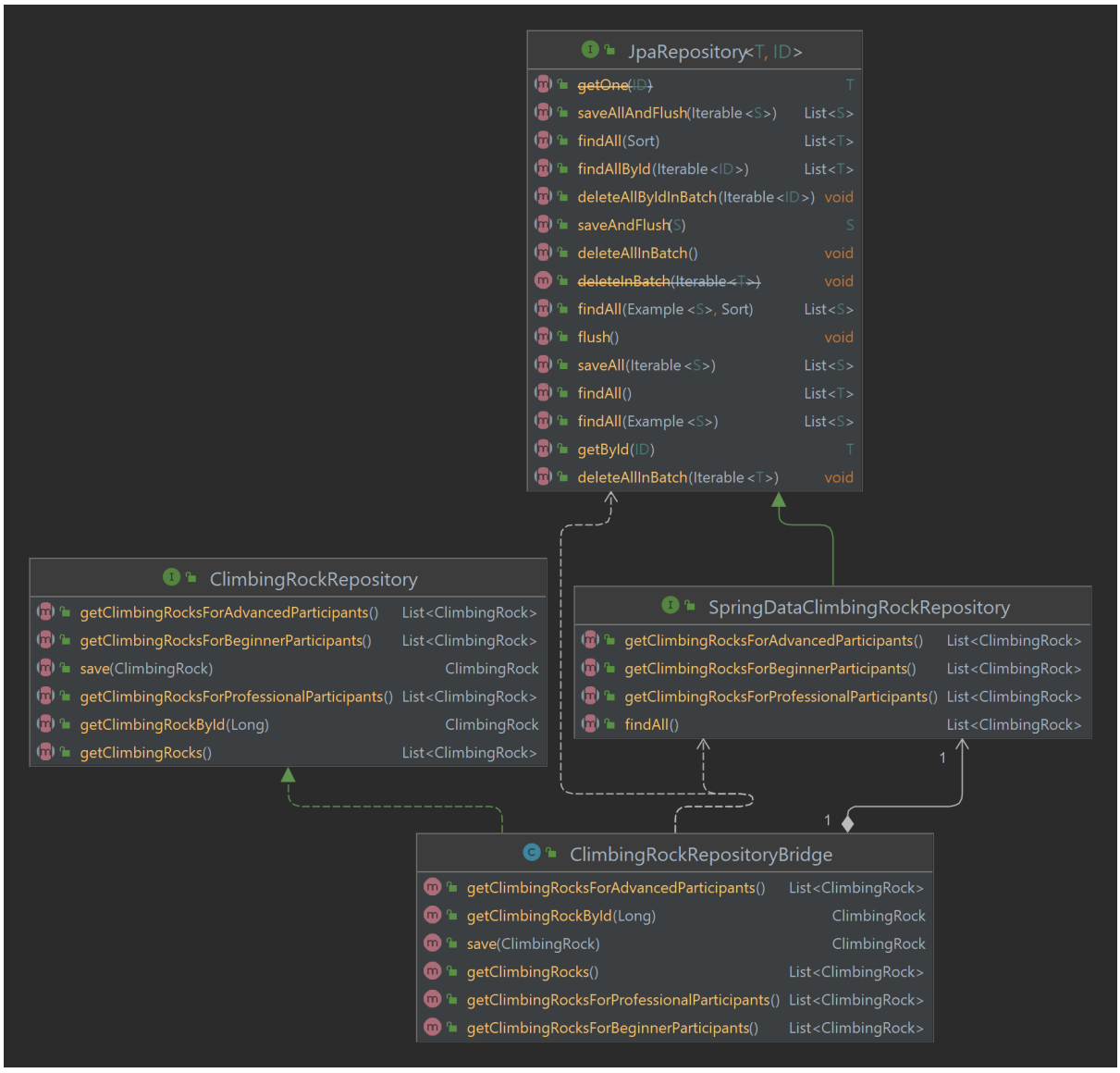

Abbildung 3: Repository Implementierung mit dem Bridge-Pattern

Kapitel 7

Unit Testing

7.1 Implementierte Unit Tests

Beim Implementieren folgender Use Cases wurde auf das Einhalten der ATRIP Regeln geachtet und Mocks verwendet:

- ClimbingRockServiceTest (1)
- DistanceCalculatorTest (1)
- TripCategorizerServiceTest (1)
- TripServiceTest(2)
- WeatherServiceTest (2)
- ClimbingRockFilterServiceTest (3)