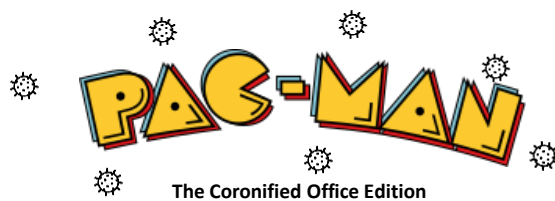


Advanced Software Engineering – Dokumentation



Dozent	Mirko Dostmann
Datum	31. Mai 2021
Student	Jeanne Helm
Matrikelnummer	6786416

Inhaltsverzeichnis

Clean Architecture	4
Domain Driven Design (DDD)	6
<i>Ubiquitous Language (UL)</i>	6
<i>Taktisches DDD</i>	7
Value Objects (VO)	7
Entities.....	7
Aggregate und Repositories.....	8
Domain Services	9
Programming Principles	10
<i>SOLID</i>	10
Single Responsibility.....	10
Open Closed Principle	10
Liskov Substitution Principle	11
Interface Segregation Principle	11
Dependency Inversion Principle	12
<i>GRASP</i>	13
Low Coupling.....	13
High Cohesion	14
Polymorphismus	15
<i>Don't Repeat Yourself! (DRY)</i>	17
Refactoring	20
<i>Large Class</i>	20
<i>Long Method</i>	21
<i>Code Comments</i>	21
Entwurfsmuster	23
<i>Iterator</i>	23
<i>Bridge</i>	24
<i>Builder</i>	25

Eine Webapplikation soll entwickelt werden, die auf der Idee von Pac-Man basiert. Es soll ein bis zwei Karten geben, auf denen sich der Spieler bewegen kann. Die Karten basieren auf einem Grundriss eines Bürogebäudes. Zwischen den Räumen laufen „infizierte“ Kollegen auf vordefinierten Wegen hin und her. Der Spieler muss zu zufällig eingestellten Zielen laufen und darf dabei nicht selbst „angesteckt“ werden, sonst verliert er ein Leben. Er wird angesteckt, sobald er in einem vordefinierten Radius zu einem Kollegen steht. Auf der Karte kann eine „Impfung“ erscheinen, die dem Spieler nach dem Einsammeln ein Extra Leben gibt. Hat der Spieler kein Leben mehr, ist das Spiel beendet und seine Punktezahl wird ihm angezeigt.

Technologie

Das Backend basiert auf Spring Boot (Programmiersprache Java, Dependency Management via Maven). Das Frontend basiert auf VueJs. Als Controller werden sowohl REST als auch WebSockets genutzt.

Anforderungen

- Initialisierung
 - Karte(n) in Koordinaten aufteilen
 - Erlaubte Wege/Koordinaten festlegen
 - Kollegen positionieren und Weg definieren
- Spielsystem
 - Spiel erstellen
 - Synchrone Bewegung/Koordination
 - Spieler: manuell mit Eingabe
 - Kollege(n): vordefinierter (= automatischer) Weg
 - Effekte
 - Zufällige Koordinaten für Ziel(e) und Impfungen
 - Anstecken (mit einer Bedingung)
- Abfragen und Sichern der Spielstände in Datenbank
- Grafische Darstellung

Der vollständige Code kann unter folgendem öffentlichen Repository auf dem Master-Branch abgerufen und eingesehen werden: https://github.com/jhelmcodes/advanced_swe

Clean Architecture

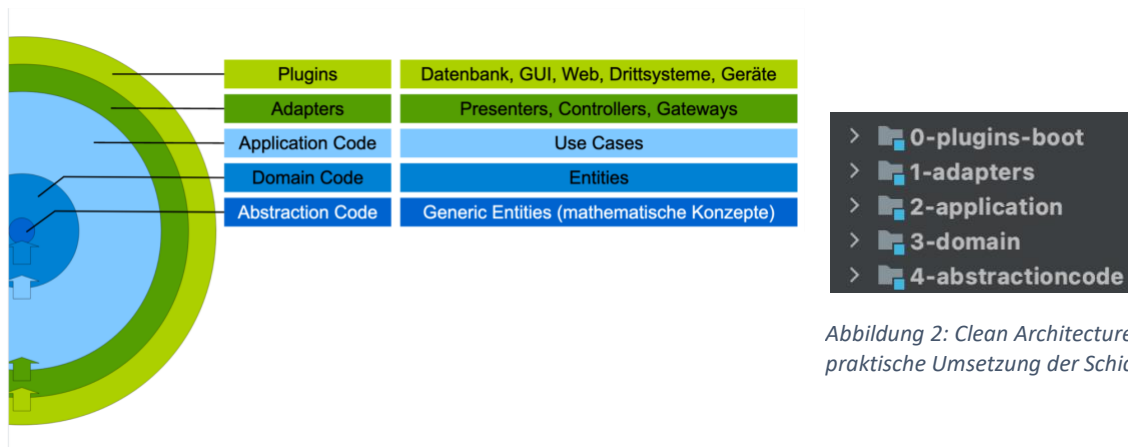


Abbildung 2: Clean Architecture - praktische Umsetzung der Schichten

Abbildung 1: Clean Architecture - Schichten (Theorie)

Die Applikation ist in die fünf Schichten Abstraction Code, Domain Code, Application Code, Adapters und Plugins der Clean Architecture unterteilt worden. Das ist in den oberen Abbildungen dargestellt. Bei der Umsetzung von Schichten in der Clean Architecture zeigen die Abhängigkeiten stets in innere Schichten bzw. den Kern der Anwendung. Dieses Prinzip wird auch Dependency Inversion genannt.

Da die komplette Applikation auf Spring und Maven basiert, wurde für jede Schicht ein eigenes Maven Modul erstellt. Die Module der inneren Schichten der Clean Architecture definieren die Schnittstellen, mit denen sie arbeiten. Die äußeren Schichten implementieren ebendiese Schnittstellen.

```
<modules>
  <module>4-abstractioncode</module>
  <module>3-domain</module>
  <module>2-application</module>
  <module>1-adapters</module>
  <module>0-plugins-boot</module>
</modules>
```

Abbildung 3: Clean Architecture - Maven POM (1)

```
<dependencies>
  <dependency>
    <groupId>de.dhbw</groupId>
    <artifactId>4-abstractioncode</artifactId>
    <version>1.0-SNAPSHOT</version>
  </dependency>
</dependencies>
```

Abbildung 4: Clean Architecture - Maven POM (2)

Um dies technisch zu realisieren, wurden innersten Schichten als Dependencies in den äußeren Schichten eingefügt. Dazu ist in Abbildung 3 die Maven-spezifischen Konfigurationsdatei POM dargestellt. Dort werden alle Module der Schichten zu einem gesamten Projekt verbunden. In Abbildung 4 ist beispielsweise die Abhängigkeit der Domain-Schicht auf die Abstraction-Code-Schicht dargestellt.

Der Kern des Projekts liegt der Abstraction Code. Hier liegen unter anderem Bibliotheken die in allen äußeren Schichten auch benötigt werden, deswegen sollten diese im Kern zentral definiert werden. Es gibt beispielsweise Abhängigkeiten zum Spring Context, um im gesamten Projekt eine Dependency Injection zu ermöglichen, aber auch zu Lombok zum Generieren von Konstruktoren und Getter-Methoden.

Die modellierten Objekte nach dem Domain Driven Design der Applikation befinden sich in der nächstäußeren Schicht: die Domain. Die Domainschicht gibt den Rahmen der Applikation vor. Etwaige benötigte Objekte oder Services werden hier vorgegeben. Welche das sind, werden gemeinsam mit der Ubiquitous Language im nächsten Kapitel zu DDD genauer erläutert.

Die eigentliche Implementierung der Problemdomäne befindet sich in der Schicht Application. Dort werden die Use Cases des Spiels realisiert.

Die Implementierung der Application Schicht baut auf den Vorgaben der Domainschicht auf. In der Application-Schicht werden zeitlich eher Änderungen durchgeführt, als am Kern der Domainschicht oder im Abstraction Code.

Zur Application-Schicht gehören folgende Funktionen:

- Karte(n) in Koordinaten aufteilen (**BoardService**)
 - Erlaubte Wege/Koordinaten festlegen
 - Kollegen positionieren und Weg definieren sowie starten
 - Zufällige Koordinaten für Ziel(e) und Impfungen
- **PlayerService**
 - Spielstand berechnen
 - Position des Spielers setzen
- Spielsystem (**GameService**)
 - Spiel erstellen
 - Aktionen für Spielobjekte ausführen
 - Anstecken
 - Bewegen
 - Arbeiten
 - Synchrone Bewegung/Koordination
 - Spieler: manuell mit Eingabe
- **HighscoreService**
 - Abfragen und Sichern der Spielstände in Datenbank

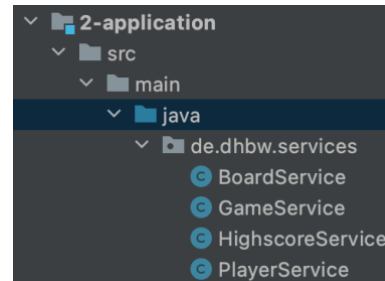


Abbildung 5: Clean Architecture - Application

In der Adapter-Schicht werden letztendlich zwischen den Daten der Application und der Visualisierung durch das Frontend vermittelt. Die Aufgabe ist es, zwischen den Objekten der Applikation und Domainschicht Transferobjekte (= Data Transfer Object (DTO)) bereitzustellen. DTOs fassen komplexere Objekte der inneren Schichten zusammen und ermöglichen die Serialisierung dieser Daten via JSON als Transportformat. Die Konvertierung übernehmen dazugehörige Mapper Klassen.

In der äußersten Schicht befinden sich die Plugins. Diese nutzen die Logik der inneren Schichten und knüpfen direkt an die Adapter an. Plugins können jederzeit durch andere bzw. neuere Technologien ausgetauscht werden und sind kurzlebig. Die Pluginschicht dieses Projektes beinhalten eine REST und WebSocket Schnittstelle zur Spiellogik, eine Datenbankbindung sowie das Starten eines Webserver unter dem Port 8080 mittels Spring Boot. Hierbei handelt es sich um technische Details, die in keine innere Schicht gehören.

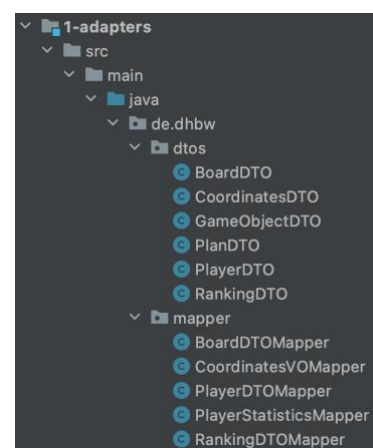


Abbildung 7: Clean Architecture - Adapter

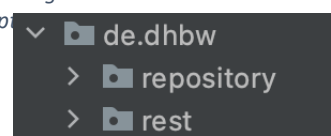


Abbildung 6: Clean Architecture - Plugin

Als Datenbank-Anbindung wird *Hibernate JPA* genutzt sowie die InMemory-Datenbank *H2*. Hier befindet sich entsprechend auch die Implementierung der Repositories aus der Domain Schicht. Diese werden über den Spring Context automatisch an alle inneren Schichten eingefügt (injected).

Das Frontend ist eine eigene Anwendung in sich und ist nicht Teil der Clean Architecture des Backends. Im Rahmen dieses Programmentwurfes wurde nur das Backend für die Erfüllung der Anforderungen betrachtet. Das Frontend dient lediglich der Visualisierung des Spiels. Es wird hier als Technologie VueJs verwendet sowie einen eigenen Webserver zum Starten und Rendern von HTML-Seiten für die Visualisierung des Spiels im Browser. Dieses ist erreichbar unter dem Port 8081. Das Frontend nutzt dabei die REST-API und die Websocket Schnittstellen der Plugin-Schicht vom Backend.

Domain Driven Design (DDD)

Die Objekte und Services nach dem Prinzip des Domain Driven Designs (DDD) befinden sich in der Domain Schicht und Modul dieser Applikation. Diese definieren die gegenseitigen Abhängigkeiten von Objekten sowie die Validität ihrer Zustände. Die Implementierung hängt von der Vorgabe der Domain Schicht ab, denn diese beschreibt den Problembereich der Anwendung.

Der Weg zum Verständnis einer Domäne führt über die in dieser Domäne gesprochene Sprache. DDD legt daher größten Wert auf das Vokabular, dass zwischen Domänenexperten und Entwicklern verwendet wird und führt dafür ein eigenes Konzept ein: die sog. Ubiquitous Language (UL). Diese wird zuerst beschrieben und anhand dieser die eigentlichen Objekte der Domänenschicht abgeleitet.

Ubiquitous Language (UL)

Die Kerndomäne der Applikation ist das Spiel, hierbei gibt es auch keine weiteren unterstützende Domänen, die untereinander kommunizieren müssten. Man könnte evtl. das Frontend als weitere, unterstützende Domäne sehen. Da das Frontend weniger komplex ist als die Spielmechanik, wird dort allerdings kein DDD angewendet (*das per Definition der Vorlesung so auch gestattet ist*). Entsprechend ist auch kein Bounded Context aus mehreren Domänen notwendig.

Die geplante Anwendung hat das Ziel Pacman zu implementieren. Hier muss es also einen Spieler (**Player**), einen bzw. mehrere Gegner (**Colleagues**) geben, einen bzw mehrere **GameObjects** sowie **Ranking** Einträge geben. Game Objects sind wiederum unterteilt in Impfungen (**Vaccination**) als Leben, **Infections** sowie **Work Items**. Berührt ein Player eine Vaccination, so wird seine gespeicherte Anzahl (**GameObjectCount**) der Vaccination erhöht. Die gesammelte Anzahl pro GameObjects sichert sich ein Player zur Laufzeit innerhalb seiner Statistiken (**PlayerStatistics**). GameObjects, Player und Colleagues werden auf einem **Board** platziert. Ein Board hat ein Layout (**BoardLayout**) bestehend aus Koordinaten (**Coordinates**), einer Fläche (**Plan**) sowie Hindernissen (**Obstacles**). Die Werte dürfen nicht negativ sein. Jedes Objekt auf dem Board hat eine aktuelle Coordinate und diese müssen sich auf dem Board befinden. Colleagues bewegen (**ColleagueMovement**) sich auf einem vordefinierten Pfad vor und zurück. Für den Plan müssen Breite und Höhe definiert sein. Ein Gegner hat einen positiven **Radius**, in dem sich ein Player infizieren kann. Ob sich dieser tatsächlich infiziert hat,

hängt von einer Wahrscheinlichkeit (**Probability**) mit einem festgelegten prozentualen Wert im Intervall]0;1[ab. Radius und Probability sind Teil der Konfiguration von einem jeweiligen Board und werden als **BoardConfiguration** zusammengefasst. Ein Player kann sich auf dem Board bewegen, darf allerdings keine Obstacles überspringen. Läuft ein Player durch die Coordinates eines GameObjects, so wird eine **GameAction** ausgeführt. Hat der Player genauso viele Infections wie Vaccinations eingesammelt (sprich hat er nicht genügend Leben mehr) so wird das Spiel beendet und sein Ranking abgespeichert, sofern es sich in den Top 10 (Highscore bestehend aus Ranking Einträgen) mit den höchsten Werten befindet.

Taktisches DDD

Aus der oben beschriebenen UL werden nun echte Objekte abgeleitet.

Value Objects (VO)

In Java müssen VOs als Klassen final deklariert werden, ebenso ihre Felder. Es gibt keine Setter. Werte in einem VO können nur über einen Konstruktor gesetzt werden und müssen nach dieser valide sein. VOs sind gleich, wenn sie in allen ihren Felder übereinstimmen. Andernfalls muss ein VO neu kreiert werden. Hierbei wurden auch die equals- und hashCode-Methoden überschrieben.

- CoordinatesVO: beschreiben positive x und y Koordinaten
- GameObjectCountVO: speichert eine positive Anzahl an GameObjects
- PlanVO: speichert eine positive Breite und Länge
- ProbabilityVO: speichert eine Wahrscheinlichkeit im Intervall]0;1[
- RadiusVO: speichert einen positiven Radius

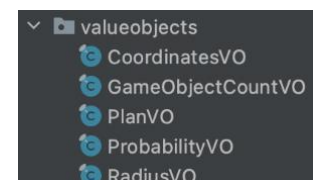


Abbildung 8: DDD - Value Object

Da die hier dargestellten Objekte alle einen bis maximal zwei primitive Datentypen beinhalten, die zudem unveränderlich sein sollen und einen Wert ohne eigene Identität beschreiben, wurden diese als Value Object implementiert. So sind beispielsweise Coordinates gleich, wenn ihre x- und y-Werte übereinstimmen. Ändern sich die Coordinates, so müssen diese neu erzeugt werden.

Entities

Entities sind wie VOs, allerdings ist hier nur ein Identifier unveränderlich (immutable). Die equals- und hashCode Methoden vergleichen entsprechend nur die Identifier. Weitere Felder dürfen ihren Zustand ändern.

- GameObjectEntity: definiert generelle Verhaltensweisen für GameObjects. Diese haben eine Position (Coordinates) sowie eine Entity Id. Spezielle Verhaltensweisen werden für eine Infection, Vaccination und WorkItem implementiert.
- PlayerStatisticsEntity: speichert eine UUID als Identifier. Darin enthalten sind alle gesammelten GameObjects sowie die genaue Anzahl der GameObjectCount eines Spielers

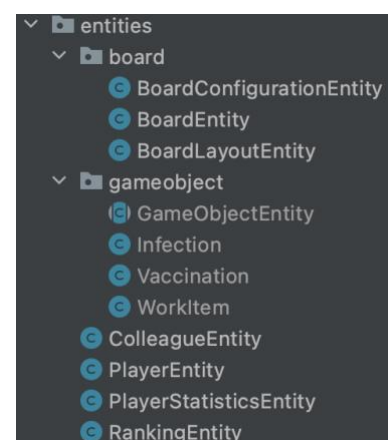


Abbildung 9: DDD - Entity

- **PlayerEntity**: beschreibt einen Player. Besitzt einen Namen als Identifier, eine Position (Coordinates) sowie PlayerStatistics mit Informationen über seine gesammelten GameObjects
- **ColleagueEntity**: beschreibt einen Colleague und besitzt als Identifier einen Namen. Ein Colleague bewegt sich auf einem vordefinierten Pfad (ColleagueMovement). Entsprechend besitzt dieser eine Liste an Coordinates
- **RankingEntity**: beschreibt ein Ranking und besitzt eine UUID als Identifier für die Entität. Teil eines Rankings sind ein Name, die gesammelten Punkte, die gesammelten PlayerStatistics und das Datum des Spiels.
- **BoardLayoutEntity**: besitzt eine Id, einen Plan sowie eine Liste mit Obstacles (Coordinates)
- **BoardConfigurationEntity**: besitzt eine Id, einen Infektionsradius sowie eine Wahrscheinlichkeit, angesteckt zu werden
- **BoardEntity**: beschreibt das Board des Spiels. Besitzt einen einzigartigen Namen als Id, ein BoardLayout, BoardConfiguration sowie eine Liste mit Colleagues

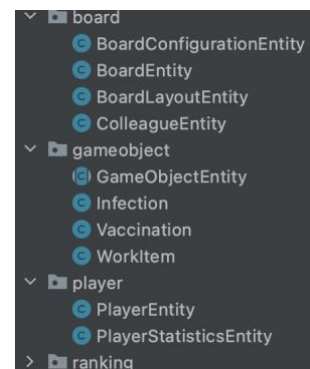
Die hier implementierten Entities haben alle einen Lebenszyklus und können ihre Eigenschaften ändern. Ein Player hat beispielsweise wechselnde Koordinaten, aber einen immer gleichen Namen als Identifier. Zwei Player sind gleich, wenn sie denselben Namen haben. Als weitere Beispiele wechseln GameObjects bei Berührung des Spielers zur Laufzeit ebenfalls ihre Koordinaten oder es werden PlayerStatistics zur Spiellaufzeit erweitert und neu berechnet.

Aggregate und Repositories

Aggregate gruppieren die Entities und VO

```
public class BoardEntity implements AggregateRoot
```

zu gemeinsam verwalteten Einheiten, selbst wenn das Aggregat nur aus einer Entity besteht. Eine Entity gruppiert hierbei in der Rolle eines sogenannten Aggregat Root (AR) ein Set an weiteren VOs und Entities, die auch (wenn nötig) nur durch das AR aus einer Datenbank gespeichert oder abgerufen werden dürfen. Um Entitäten als Aggregate hervorzuheben, wurden diese in Packages gruppiert, wie in der rechten Abbildung zu sehen ist.



Die AR implementieren das Interface „Aggregate Root“, um ihre besondere Rolle hervorzuheben. Da beispielsweise der Zugriff auf eine BoardConfigurationEntity, ein BoardLayout oder eine ColleagueEntity nur über ein BoardEntity als „Türsteher“ (AR) möglich ist, wurden diese also als Aggregat gruppiert. Zudem gibt es ein BoardRepository, um BoardEntities aus dem Persistenzspeicher zu lesen oder in diesen abzulegen. Das Repository greift stets über das AR BoardEntity auf den Speicher zu.

Als weiteres Beispiel umfasst das Aggregat für Rankings nur die RankingEntity. Diese ist ein alleiniges Aggregat und nimmt auch selbst die Rolle des AR ein, da dies einen alleinigen logischen Zusammenhang repräsentiert. Da auch Rankings persistiert werden sollen, werden diese ebenfalls über ein RankingRepository aus dem Persistenzspeicher geladen bzw. in diesem abgelegt.

Die weiteren Aggregate werden nur zur Spiellaufzeit genutzt und nicht persistiert. Entsprechend existiert für diese kein Repository.

Repositories der Domainschicht werden als Interfaces implementiert. Sie geben die notwendigen Methoden vor, um auf eine Datenbank zugreifen zu können.

```
public interface BoardRepository {  
    BoardEntity getBoardByName(String name);  
  
    void save(BoardEntity boardEntity);  
}
```

Abbildung 10: DDD - Repository

Domain Services

Domain Services beschreiben Services, die als kleine Helfer, komplexes Verhalten abbilden, die nicht eindeutig einem Repository, einer Entität oder einem VO zugeordnet werden können. Die Domain Services geben die zu implementierenden Use Cases dieser Applikation vor. Die technischen Details befinden sich aber in den zugrundeliegenden Schichten. In der Domain Schicht liegen also Interfaces der Services mit definierten Methoden sowie Entities oder VOs als Ein- und Ausgabeparameter.

Der GameDomainService, PlayerDomainService, HighscoreDomainService sowie das BoardDomainService stellen Methoden für die Spiellogik bereit. Sie implementieren teilweise auch Hilfsservices, die besondere Verhaltensweisen, wie die Initialisierung oder das Zurücksetzen eines Games, Players oder allgemeinen Objektes separat gruppieren. Auch GameActions vermitteln die Interaktion mehrerer Entitäten untereinander.

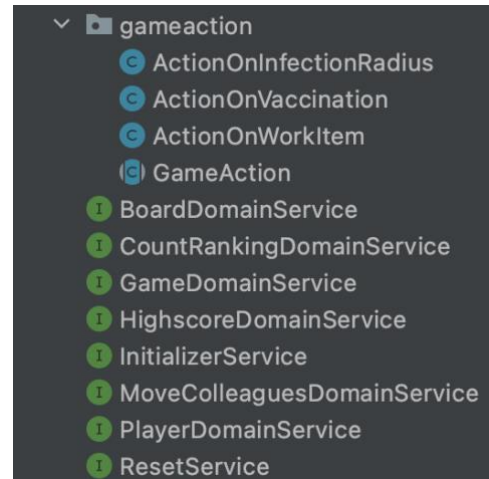


Abbildung 11: DDD - Domain Service

Als Beispiel gibt der GameDomainService Verhaltensweisen vor, um das Spiel zu starten, den aktuellen Zustand des Players zurückzubekommen oder den Spieler auf dem Board zu bewegen. Da dieses Verhalten sowohl das BoardRepository benötigt, um das Board aus dem Persistenzspeicher zu laden, als auch Informationen einer PlayerEntity und dem Board für seine Berechnungen benötigt, wurde dieses Verhalten in einem Domain Service definiert.

Programming Principles

SOLID

Single Responsibility

Als Beispiel für die Anwendung dieses Prinzips werden Repositories betrachtet. Diese haben die Aufgaben das Kreieren, Lesen, Aktualisieren und Löschen (CRUD) von Daten im Persistenzspeicher vorzugeben. Sie sind damit die Schnittstelle(n) zur Datenbank und müssen sich nur darum kümmern. Ihr Aufgabenbereich ist also klar definiert.

Open Closed Principle

Das Spiel gestattet es beispielsweise die GameActions und GameObjects jederzeit zu erweitern, nicht aber zu ändern, wie es das Open Closed Principle vorgibt. Bewegt sich ein Spieler auf eine neue Koordinate, werden dynamisch die verfügbaren GameActions und -Objects mit dieser Koordinate verglichen und die zugrundeliegende Logik ausgeführt. Vor dem Anwenden des Open Closed Principles wurden Spielobjekte nur anhand ihrer Benennung unterschieden, die durch sich wiederholende If-Conditions zudem auch noch duplizierten Code produziert haben. Die Conditions hätten so in der Methode selbst statisch erweitert werden müssen. Der Vorher-Nachher-Vergleich ist in der nachfolgenden Abbildung dargestellt.

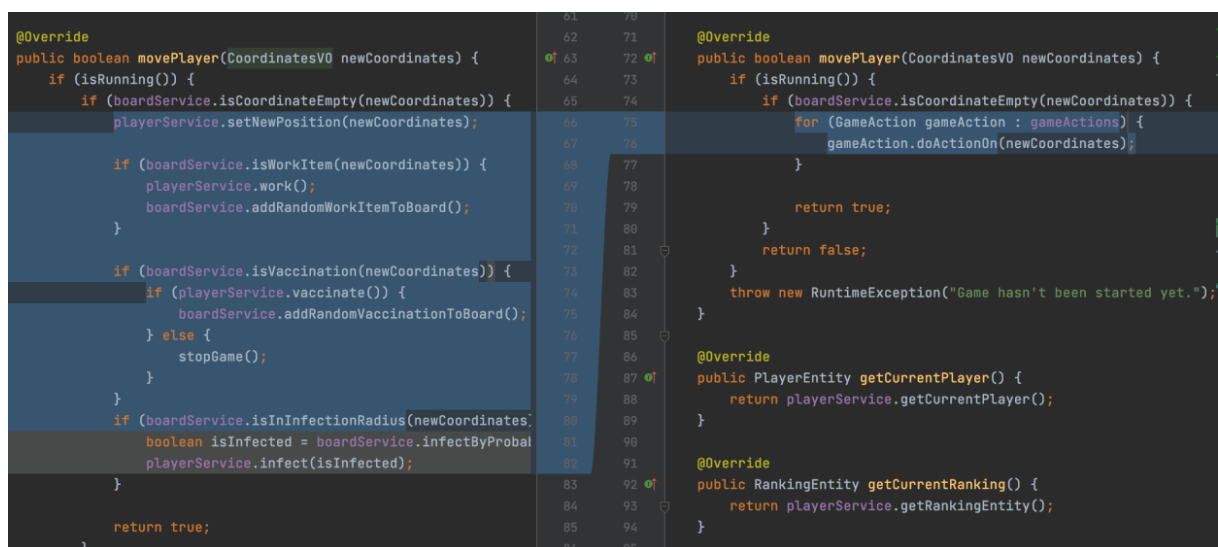


Abbildung 12: Open Closed - GameService – vorher hart codierte Conditions und nachher polymorph erweiterbare Lösung

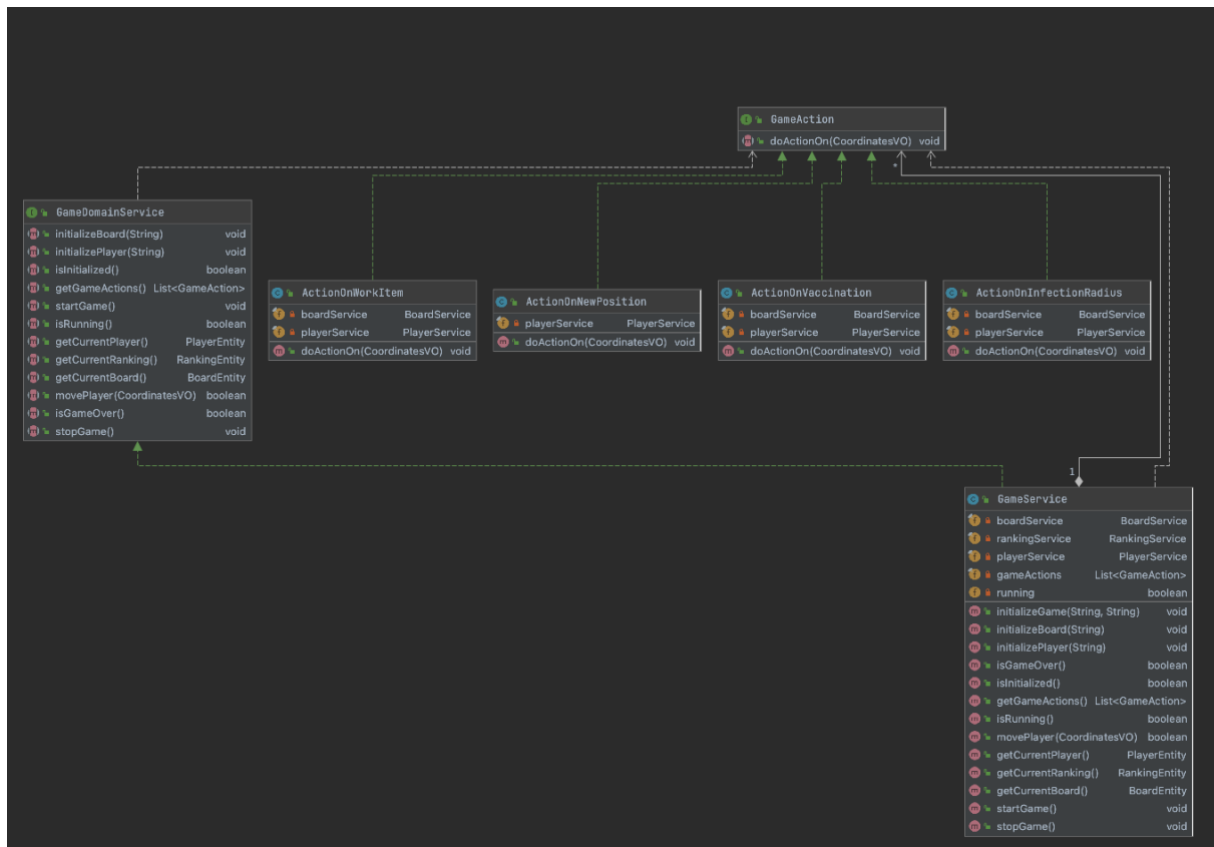


Abbildung 13: Open Closed - GameService – Darstellung der GameActions

Liskov Substitution Principle

Das Liskov Substitution Principle (LSP) besagt, dass abgeleitete Typen (z.B. beim Überschreiben von Methoden) schwächere Vorbedingungen haben müssen, aber stärkere Nachbedingungen. In diesem Zusammenhang werden die **GameObjectEntities** betrachtet. Die Subklassen **Vaccination**, **WorkItem** und **Infection** überschreiben je nach Bedarf die Vorbedingungen der allgemeinen **GameObjectEntity**. Anhand des zugrundeliegenden Codes wurde das LSP nicht verletzt. Die allgemeinen Aussagen, Eigenschaften oder Verhaltensweisen der **GameObjectEntity** gelten auch für die abgeleiteten Typen. Alle **GameObjects** haben eine Koordinate, können einen Wert für das Ranking zugeteilt bekommen, haben eine default Anzahl und können auf einen Player angewendet werden.

Sollten neben den drei Standard-GameObjects (**Vaccination**, **Infection** und **WorkItem**) des Pacman-Spiels irgendwann weitere Objekte ergänzt und erweitert werden, die unter Umständen auch die Standard-GameObjects überschreiben, könnte eine Verletzung des LSP eintreten. Wird beispielsweise eine neue Form einer Infizierung ergänzt und die Klasse **Infection** überschrieben, kann es sein, dass ein Spieler nun nicht mehr über die Distanz angesteckt wird, sondern durch eine Berührung, Tröpfcheninfektion, Mücken oder durch andere Formen. In diesem Fall wäre die „ist-ein“-Beziehung der Vererbung dann nicht mehr gültig.

Interface Segregation Principle

Das Interface Segregation Principle wurde für die **DomainServices** angewendet. Diese hatten viele Methoden und mehrere Verantwortlichkeitsbereiche. Die schweren Interfaces (fat Interfaces) wurden nun nach ihren Aufgabenbereichen aufgeteilt und verkleinert. Vor dem

Anwenden dieses Prinzips gab es in den Domain Services sich wiederholende Methoden, wie beispielsweise das Initialisieren oder Zurücksetzen von Komponenten im Spiel (unter anderem für die PlayerEntity im PlayerDomainService oder die BoardEntity im BoardDomainService). Nun können die kleineren Interfaces je nach Service wiederverwendet werden und tragen in ihrem Namen den klar definierten Aufgabenbereich.

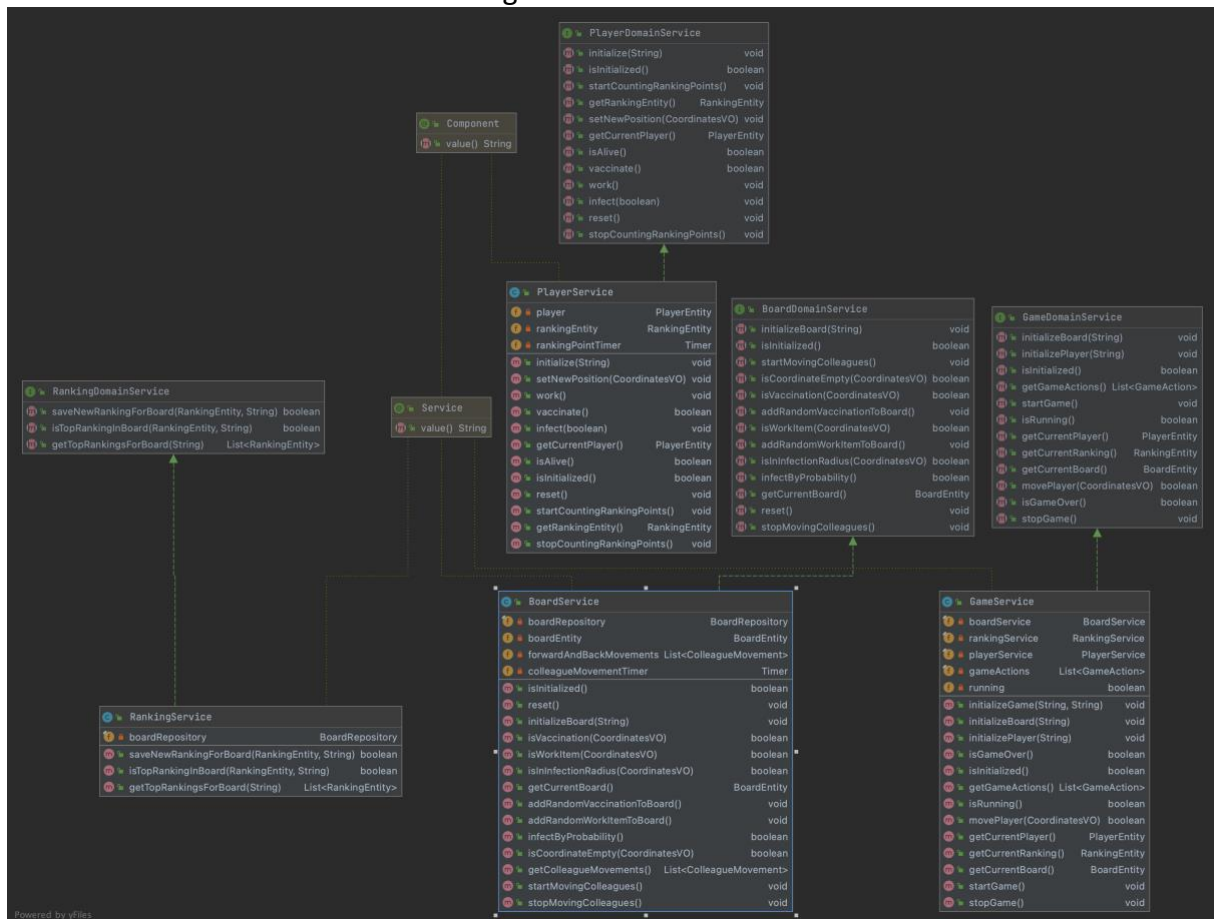


Abbildung 14: Interface Segregation - Domain Services - vorher

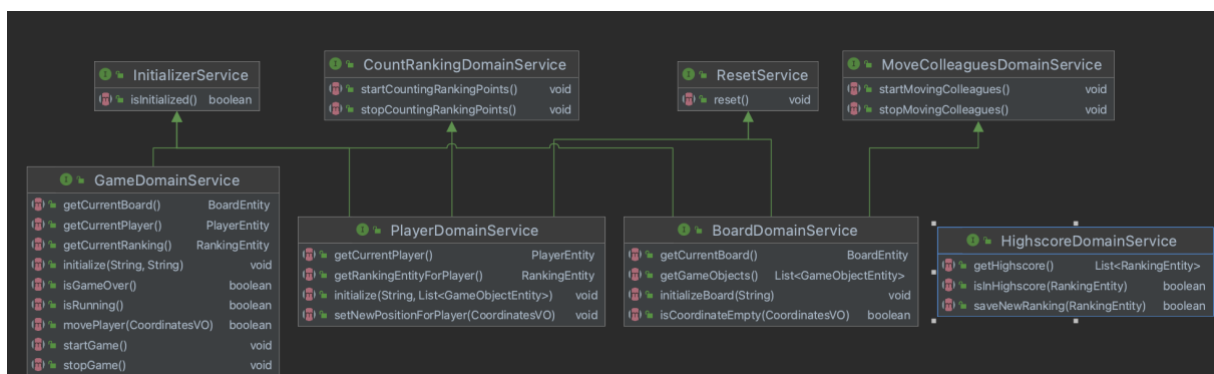


Abbildung 15: Interface Segregation - Domain Services - nachher

Dependency Inversion Principle

Beim Dependency Inversion Principle dürfen High Level Module nicht von Low Level Modulen abhängen. Diese Bedingung wurde beispielsweise durch die Clean Architecture umgesetzt. Abstraktionen sollten nicht von Details abhängig sein. Die Low Level Module (also die äußeren Schichten der Clean Architecture) implementieren dann die Regeln der High Level Module (die inneren Schichten). Die Module der inneren Schichten definieren die Schnittstellen, mit denen

sie arbeiten. Die äußeren Schichten realisieren diese Schnittstellen. Ohne diese klare Anordnung, kann es zu zyklischen Abhängigkeiten kommen, Module werden stärker gekoppelt und Änderungen im Code können komplex werden.

GRASP

Low Coupling

Bei dem Programmierprinzip Low Coupling geht es darum, nicht zu viele Abhängigkeiten zu anderen Codeteilen zu haben. Einige Stellen im Code mussten refactored werden, um das Programmierprinzip zu erfüllen.

Der PlayerService benötigt beispielsweise kein Wissen über das dazugehörige Board des Spiels. In seiner „Welt“ gibt es lediglich den Player sowie die Berechnung seines aktuellen Rankings.

```
@Component
public class PlayerService implements PlayerDomainService {

    private PlayerEntity player;
    private RankingEntity rankingEntity;
    private Timer rankingPointTimer;
```

Abbildung 16: Low Coupling - PlayerService

Auch der HighscoreService benötigt lediglich Zugriff auf das RankingRepository, um den aktuellen Highscore eines Spielers zu berechnen oder zu aktualisieren.

```
@Service
public class HighscoreService implements HighscoreDomainService {

    private final RankingRepository rankingRepository;
    private final int NUMBER_OF_RANKINGS = 10;
```

Abbildung 17: Low Coupling - HighscoreService

High Cohesion

Unter High Cohesion wird als weiteres Programmierprinzip die Zusammenhängigkeit von Objekten verstanden. Eine Klasse sollte eine Aufgabe haben und nicht viele voneinander unabhängige Aufgaben haben (vgl. Single Responsibility Principle). Hierbei werden Informationen entsprechend ihrer Art gruppiert.

Im Projekt wurde dieses Prinzip auf verschiedene Arten umgesetzt. Repositories haben beispielsweise nur Kenntnisse von Ihrer Anbindung an den Persistenzspeicher und haben die Aufgabe mit diesem zu interagieren. Sie brauchen ansonsten keine Kenntnisse zu anderen Objekten und erfüllen die High Cohesion.

Als weiteres Beispiel sind in der nachfolgenden Abbildung ein Ausschnitt aus dem BoardDomainService und der BoardEntity dargestellt. Dort ist zu sehen, dass Methoden je nach Aufgabe in ein eigenes Interface ausgelagert wurden, bzw. Attribute noch entsprechend ihrer Zugehörigkeit und ihres Kontextes in eigene Klassen zusammengefasst wurden. Nun hat die BoardEntity als Aufgabe lediglich ihre eigenen Attribute zu koordinieren und zu setzen, während die Validierung dieser intern und entkoppelt von anderen Attributen der BoardEntity stattfindet.

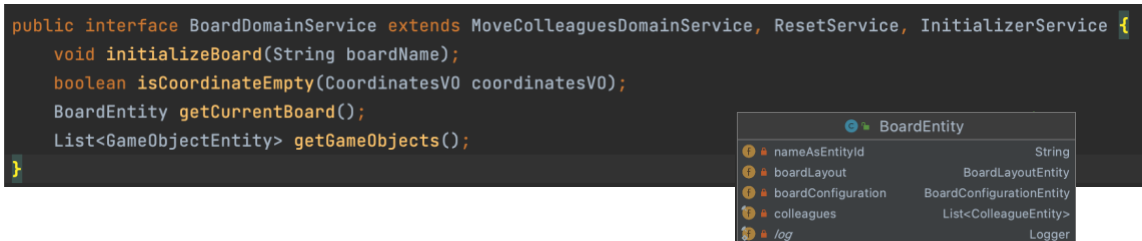


Abbildung 18: High Cohesion - Beispiele anhand von Attributen und Methoden

Polymorphismus

Bei Polymorphismus werden Alternativen abhängig von ihrem konkreten Typ anders ausgeführt. Dieses Programmierprinzip wurde beispielsweise bei den Game Actions eingesetzt. Je nachdem welches Spielfeld sich auf einer Coordinate befindet, wird eine andere Handlung ausgeführt. Die abstrakte Klasse der GameAction ist in der folgenden Abbildung dargestellt. Der Aufruf ist in der wiederum nachfolgenden Abbildung grün markiert. Ohne Polymorphismus gab es eine lange Abfolge von If-Conditions, die unter anderem statisch im Code hinterlegt waren, die nicht erweiterbar waren oder die Codeabfolgen duplizierten. (vgl. auch das Open-Closed Principle)

```
@Service
public abstract class GameAction<T extends GameObjectEntity> {
    private final Class<T> type;

    @Autowired
    private PlayerDomainService playerService;
    @Autowired
    private BoardDomainService boardDomainService;

    public GameAction(Class<T> type) { this.type = type; }

    public Class<T> getType() { return this.type; }

    public void doActionOn(T object) {
        object.doToPlayer(playerService.getCurrentPlayer());
        if (object.needsNewCoordinateAfterAction()) {
            object.setNewCoordinate(boardDomainService.getCurrentBoard().getBoardLayout().getRandomCoordinate());
        }
    }
}
```

Abbildung 19: Umsetzung des Polymorphismus anhand Game Action

```
@Override
public boolean movePlayer(CoordinatesVO newCoordinates) {
    if (isRunning()) {
        if (!playerService.getCurrentPlayer().isAlive()) {
            stopGame();
            return false;
        }
        if (boardService.isCoordinateEmpty(newCoordinates)) {
            playerService.setNewPositionForPlayer(newCoordinates);
            doAction(newCoordinates);
        }
        return true;
    }
    return false;
}

throw new RuntimeException("Game hasn't been started yet.");
}
```

Abbildung 20: Polymorphismus - Ausschnitt aus GameService

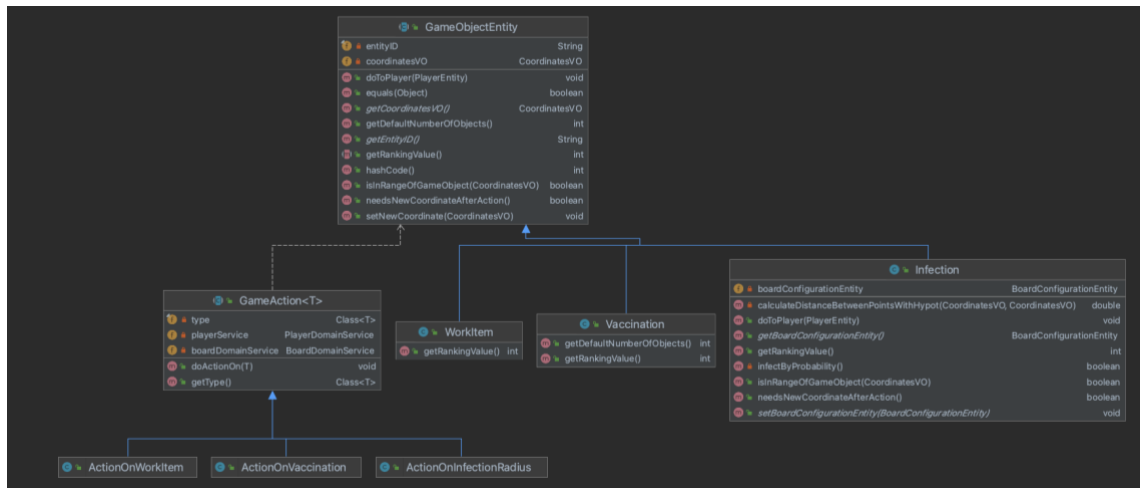


Abbildung 21: Umsetzung des Polymorphismus von GameAction und GameObject veranschaulicht als UML

Don't Repeat Yourself! (DRY)

Die Durchführung des Programmierprinzips DRY lassen sich in mehreren Arten in dem Projekt wiederfinden. Zum einen wurden die Services der Domain-Schicht verkleinert. Wichtige Funktionen, die mehrfach verpflichtend hätten implementiert werden müssen, wurden in eigene Interfaces ausgelagert. Die Services, die die eigentlichen Use Cases implementieren (für Game, Player und Board), erben von diesen entsprechend. In diesem Zusammenhang wurde auch das Interface Segregation Principle angewendet, da die Interfaces reduziert wurden und nun nur noch einen Aufgabenbereich haben. Die Umsetzung ist in dem folgenden Ausschnitt dargestellt.

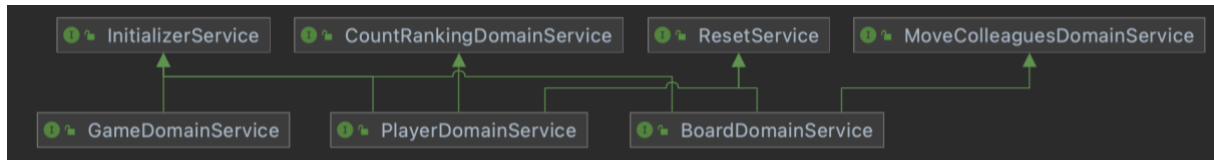


Abbildung 22: DRY und Interface Segregation anhand der Domain Services

An einigen Stellen im Code gab es weiteren duplizierten Code, wie beispielsweise im BoardDomainService (siehe nachfolgenden Ausschnitt). Dort wurde für eine Vaccination und ein WorkItem dieselbe Abfolge zum Setzen einer neuen Coordinate aufgerufen. Die Methoden wurden durch ein größeres Refactoring nun in die BoardLayoutEntity ausgelagert. Denn um neue Coordinates zu berechnen, sind lediglich Kenntnisse über das Board und die darauf befindlichen Hindernisse (Obstacles) notwendig (siehe den nachfolgenden Ausschnitt des Klassendiagrammes von BoardLayoutEntity).

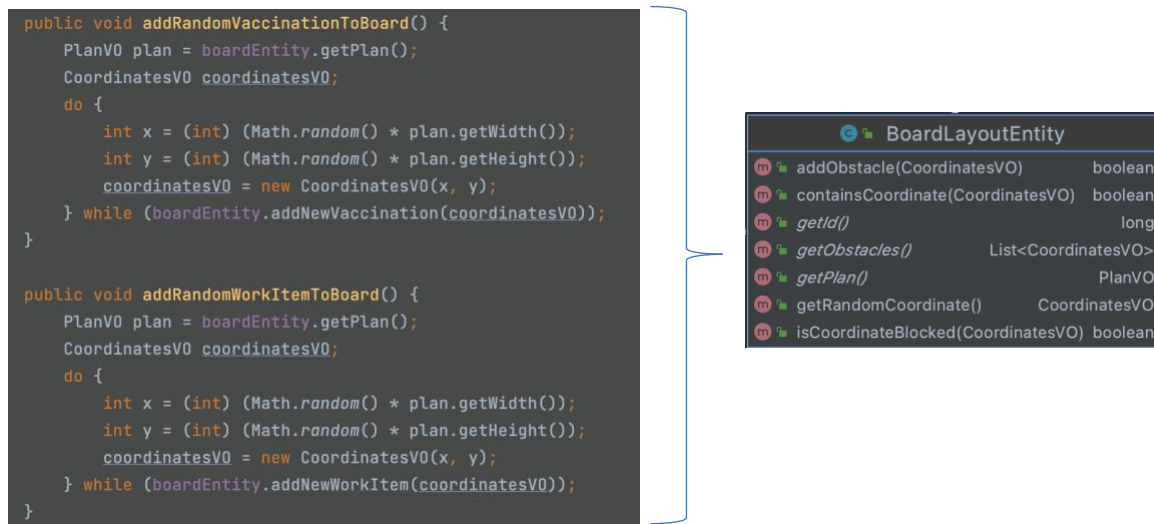


Abbildung 23: DRY – sich wiederholende Codestellen innerhalb der BoardEntity

Damit GameObjects mehrfach hinzugefügt werden können und nicht nur anhand des Namens unterschieden werden, wurden diese in eigenen Objekten polymorph zusammengefasst. Diese werden nun auch in einer Liste gesammelt, dadurch ist es auch möglich von jedem Typ mehrere Objekte auf einem Board abzusichern. Diese werden nun auch zur Laufzeit auf ein Board gesetzt bzw. von dort entnommen, d.h. nicht mehr in der Datenbank gesichert. Die aktuellen GameObjectEntities werden vom BoardService zwischengespeichert.

```

@Entity
public class BoardEntity implements AggregateRoot {

    @NotNull
    @Id
    private String entityID;

    @NotNull
    @Column
    private String name;

    @NotNull
    @OneToMany(cascade = CascadeType.ALL)
    @LazyCollection(LazyCollectionOption.FALSE)
    private final List<CoordinatesVO> obstacles = new ArrayList<>();

    @OneToOne(cascade = CascadeType.ALL)
    private CoordinatesVO vaccination;

    @OneToOne(cascade = CascadeType.ALL)
    private CoordinatesVO workItem;
}

```

```

@Service
public class BoardService implements BoardDomainService {

    private final BoardRepository boardRepository;
    private BoardEntity boardEntity;
    private List<ColleagueMovement> forwardAndBackMovements;
    private Timer colleagueMovementTimer;
    private final List<GameObjectEntity> gameObjectEntities;
}

```

Abbildung 24: DRY - Gruppierung der GameObjects

Werden weitere Typen an GameObjects irgendwann zum Spiel ergänzt, müssen diese lediglich ein GameObject sowie eine dazugehörige GameAction implementieren (Siehe auch Open Closed Principle)

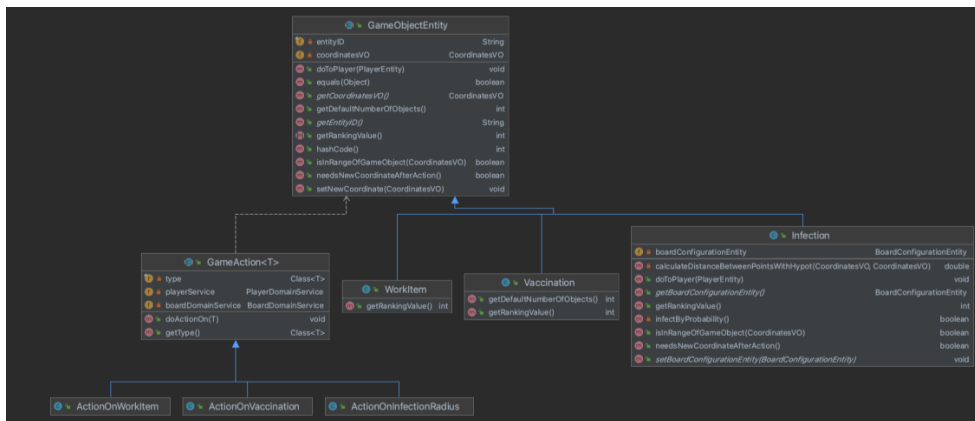


Abbildung 25: DRY - UML der GameObjects und GameActions

Alle existierenden GameActions werden via Dependency Injection von Spring Boot automatisch zur Laufzeit in einer Liste innerhalb des GameServices (diese sind nämlich Teil der Spielelogik) gesammelt. Für GameObjects sammelt Spring ebenfalls derzeit alle existierenden Objekte einmalig in einer Liste (diese liegen im BoardService, da sie sich auf dem Board befinden). Alle GameObjects bzw. GameActions sind generisch mit einer Koordinate eines Spielers zuordnungsbar. Die Logik wird dann entsprechend der speziellen Implementierung eines GameObjects bzw. einer GameAction ausgeführt.

```

@Autowired
public GameService(BoardService boardService, HighscoreService highscoreService, PlayerService playerService,
    List<GameAction> gameActions) {
    this.boardService = boardService;
    this.highscoreService = highscoreService;
    this.playerService = playerService;
    this.gameActions = gameActions;
}

```

Abbildung 26: DRY - Dependency Injection aller existierenden GameActions mittels der Autowired Annotation

Die initialisierte Liste der GameObjects lässt sich jedoch auch überschreiben, indem entweder in der Application-Schicht der GameService oder aber in einer noch weiter darunter liegenden Schicht eine eigene Implementierung beispielsweise die Initialisierungsmethode überschreibt und neue Game Objekte dort händisch hinzufügt.

Diese Flexibilität ermöglicht hier keine Notwendigkeit, GameObjekte als Koordinaten in einem Objekt zu vervielfachen und händisch zum Spiel ergänzen zu müssen.

```
private GameObjectEntity get(CoordinatesV0 coordinate) {
    for (GameObjectEntity gameObjectEntity : boardService.getGameObjects()) {
        if (gameObjectEntity.isInRangeOfGameObject(coordinate)) {
            return gameObjectEntity;
        }
    }
    return null;
}

private GameAction<GameObjectEntity> getActionFor(GameObjectEntity gameObjectEntity) {
    for (GameAction<GameObjectEntity> gameAction : gameActions) {
        if (gameAction.getType().equals(gameObjectEntity.getClass())) {
            return gameAction;
        }
    }
    return null;
}

@Override
public boolean movePlayer(CoordinatesV0 newCoordinates) {
    if (isRunning()) {
        if (!playerService.getCurrentPlayer().isAlive()) {
            stopGame();
            return false;
        }
        if (boardService.isCoordinateEmpty(newCoordinates)) {
            playerService.setNewPositionForPlayer(newCoordinates);
            doAction(newCoordinates);
            return true;
        }
        return false;
    }
    throw new RuntimeException("Game hasn't been started yet.");
}
```

Abbildung 27: Ausschnitt des generischen Aufrufs von GameAction und GameObjects im GameService

Refactoring

Large Class

Große Klassen sind häufig Indizien für zu viele Instanzvariablen oder Methoden bzw. ein Hinweis darauf, dass die Klasse an sich zu viel Verantwortung hat.

Diese Art „Code Smell“ wurde beispielsweise in der Klasse GameDomainService festgestellt. In mehreren Schritten wurden zusammengehörende Teilfunktionen in weitere Service Klassen ausgelagert und aus dem GameDomainService entfernt. Dadurch werden die einzelnen Klassen übersichtlicher und haben eine eigene Verantwortung.

<pre>public interface GameDomainService { void initialize(PlayerEntity player); void initializeBoard(String boardName); void initializeRanking(); void initializeDate(); boolean isInitialized(); void startGame(); void startCountingRankingPointsForPlayer(); boolean isRunning(); void stopGame(); void stopCountingRankingPointsForPlayer(); boolean isGameOver(); boolean movePlayer(CoordinatesV0 newCoordinates); void playerHasWorked(); void vaccinatePlayer(); void infectPlayer(); int getLastRankingPointsForPlayer(); }</pre>	<div>5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28</div>	<pre>public interface GameDomainService { void initializeBoard(String boardName); void initializePlayer(String playerName); void initializeRanking(); void initializeDate(); boolean isInitialized(); void startGame(); void startCountingRankingPointsForPlayer(); boolean isRunning(); PlayerEntity getCurrentPlayer(); BoardAggregate getCurrentBoard(); void stopGame(); void stopCountingRankingPointsForPlayer(); boolean isGameOver(); boolean movePlayer(CoordinatesV0 newCoordinates); int getLastRankingPointsForPlayer(); }</pre>
---	---	---

Abbildung 28: Ausschnitt aus einem Refactoring Prozess von GameDomainService

<pre>package de.dhbw.domain.service; import de.dhbw.aggregates.BoardAggregate; import de.dhbw.entities.PlayerEntity; import de.dhbw.valueobjects.CoordinatesV0; public interface GameDomainService { void initializeBoard(String boardName); void initializePlayer(String playerName); void initializeRanking(); void initializeDate(); boolean isInitialized(); void startGame(); void startCountingRankingPointsForPlayer(); boolean isRunning(); PlayerEntity getCurrentPlayer(); BoardAggregate getCurrentBoard(); void stopGame(); void stopCountingRankingPointsForPlayer(); boolean isGameOver(); boolean movePlayer(CoordinatesV0 newCoordinates); int getLastRankingPointsForPlayer(); }</pre>	<div>1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27</div>	<pre>package de.dhbw.domain.service; import de.dhbw.aggregates.BoardAggregate; import de.dhbw.entities.PlayerEntity; import de.dhbw.entities.RankingEntity; import de.dhbw.valueobjects.CoordinatesV0; public interface GameDomainService { void initializeBoard(String boardName); void initializePlayer(String playerName); boolean isInitialized(); void startGame(); boolean isRunning(); PlayerEntity getCurrentPlayer(); RankingEntity getCurrentRanking(); BoardAggregate getCurrentBoard(); boolean movePlayer(CoordinatesV0 newCoordinates); boolean isGameOver(); void stopGame(); }</pre>
---	--	---

Abbildung 29: weiterer Ausschnitt aus einem Refactoring Prozess von GameDomainService

Long Method

Die Methode `movePlayer()` in `GameServices` hatte zu viele Bedingungen und war zudem zu unübersichtlich und lang. Desweiteren konnten hier keine weiteren `GameActions` eingefügt werden, wenn ein Spieler ein neues Feld betritt. Beispielsweise können Aktionen also nicht verändert werden und auch nicht neue Sonderfelder eingefügt werden. Hier wurde gleichzeitig das Open-Closed-Principle, der Polymorphismus und auch DRY angewendet, um die Methode auf wenige Zeilen zu reduzieren und die Abfolgen verständlicher zu schreiben.

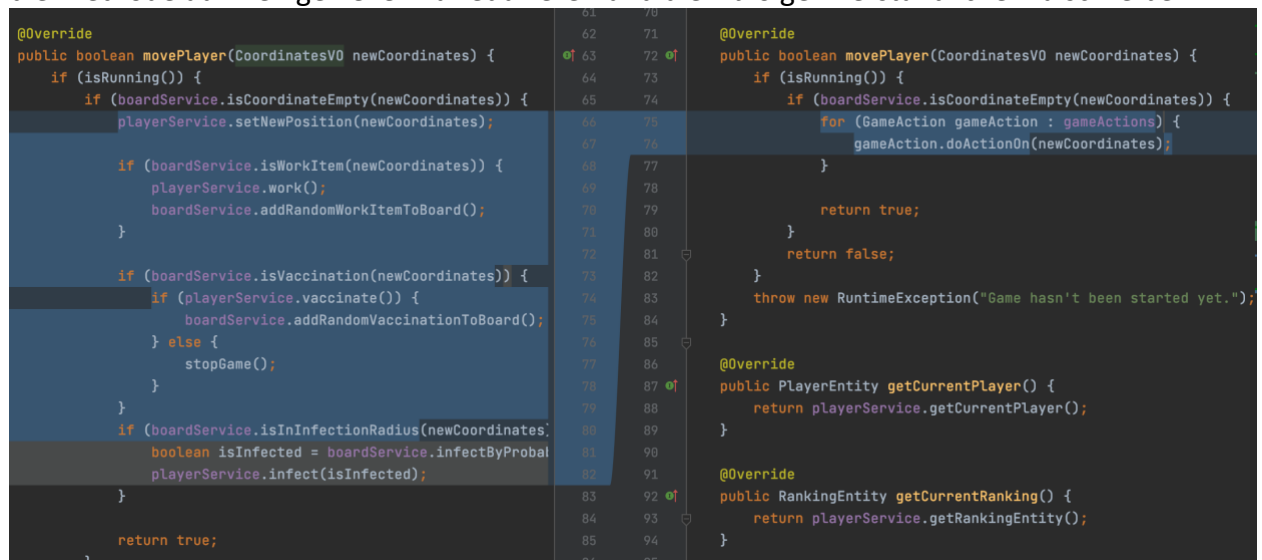
The image shows a side-by-side comparison of a Java method `movePlayer` before and after refactoring. The left side shows the original, long method with multiple nested `if` statements checking various conditions like `boardService.isCoordinateEmpty`, `boardService.isWorkItem`, `boardService.isVaccination`, and `boardService.isInInfectionRadius`. The right side shows the refactored version, which is much shorter. It uses a `for` loop to iterate over `gameActions` and calls `gameAction.doActionOn`. The refactored method also includes `@Override` annotations for `getCurrentPlayer` and `getCurrentRanking`.

Abbildung 30: Long Method - Verkleinern der Methode `movePlayer` in `GameService`

Code Comments

Während der Entwicklung wurden mindestens zwei Code Smells „Code Comments“ entdeckt und behoben.

In der `BoardEntity` befand sich noch eine alte Coding Anmerkung, die darauf hinwies, die Entity noch als Aggregat zu kennzeichnen. Diese Leiche wurde entsprechend entfernt. Eine Auslagerung/Kennzeichnung wie im Kommentar beschrieben fand bereits statt.

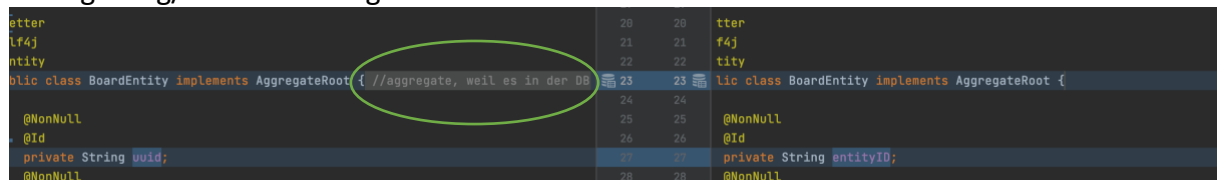
The image shows a side-by-side comparison of the `BoardEntity` class. The left side shows the original code with a comment `//aggregate, weil es in der DB` next to the `AggregateRoot` interface. The right side shows the refactored code where this comment has been removed. The class `BoardEntity` implements `AggregateRoot` and has attributes `uuid` and `entityID`.

Abbildung 31: Code Comments – Entfernen eines alten Kommentars in `BoardEntity`

In der `Ranking Entity` sollte das Attribut `UUID` eine Entity Id darstellen. Besser ist es, das Attribut direkt entsprechend umzubenennen. Dadurch muss die Bedeutung von diesem auch nicht erst an der entsprechenden Stelle in der Klasse nachgelesen werden. Sollte sich zudem die Rolle des Attributs irgendwann ändern, so kann die Information, die nun im Namen enthalten ist, nicht übersehen werden.

@Getter	15	15	@Getter
@AllArgsConstructor	16	16	@AllArgsConstructor
@NoArgsConstructor	17	17	@NoArgsConstructor
@Entity	18	18	@Entity
public class RankingEntity {	19	19	public class RankingEntity {
	20	20	
@NotNull	21	21	@NotNull
@Id	22	22	@Id
private String uuid;//entity id	23	23	private String entityID;

Abbildung 32: Code Comments - Verschmelzen eines Kommentares mit einem Attributnamen in RankingEntity

Entwurfsmuster

Iterator

Vorher

Abspeichern der Positionen und Kontrolle über die Bewegungen innerhalb des Colleagues. (vorheriger Name war „ColleagueAggregate“)

Nachher

Einführen des Iterators ColleagueMovement, das die Operationen `getCurrentPosition()` und `nextPosition()` bereitstellt. Dabei verwaltet es zudem die aktuelle Position in der Struktur selber. Der Iterator gibt immer ein Element zurück und setzt seine interne Positionsmarke dabei weiter. Ausgehend von dieser Position bestimmt er das nächste zurückgebende Element (entsprechend der Traversierungsstrategie vor und zurück). Der Colleague (nun auch „ColleagueEntity“ genannt) wird dabei von den Aufgaben der Traversierung befreit. Der Colleague definiert eine Factory-Methode zur Erzeugung des konkreten (passenden) Iterators. Die Verwendung des Iterators hat den Vorteil, dass der Colleague schlank gehalten werden kann. Zudem könnten verschiedene Traversierungsvarianten durch unterschiedliche Colleagues umgesetzt werden. Die Bewegung ist zudem unabhängig von Colleagues und kann jederzeit um weitere Implementierungen erweitert werden.

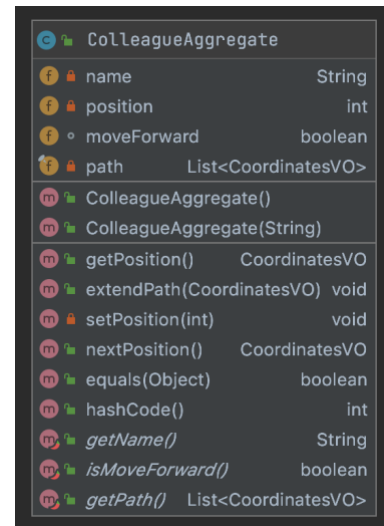


Abbildung 33: Iterator – UML des Colleague (vorher)

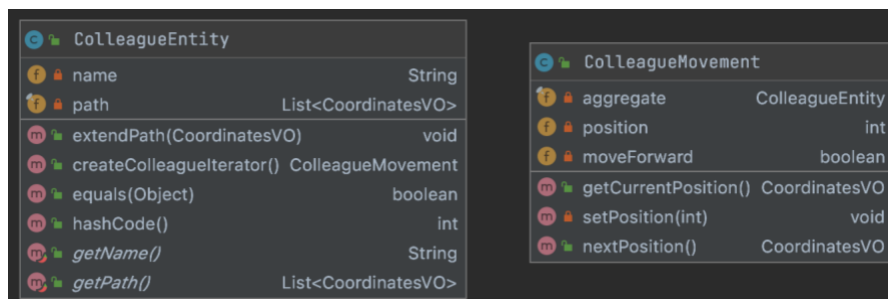


Abbildung 34: Iterator – UML des Colleague (nachher)

Bridge

Das Bridge-Pattern wurde in der Plugin-Schicht für die Implementierung der Repositories angewendet. Das Pattern bringt in diesem Zusammenhang den Vorteil, dass die technischen Details von Hibernate mit den erforderlichen Methoden aus der Domain-Schicht miteinander verknüpft werden können, ohne Änderungen an der Implementierung der Domain-Schicht vornehmen zu müssen. Die Implementierung ist erweiterbar und austauschbar. Die Anwendung ist in der folgenden Abbildung am Beispiel des RankingRepositories dargestellt.

Die Highscore Repository Bridge nutzt die technische Umsetzung der Datenbankbindung von Hibernate, um die erforderlichen Methoden des RankingRepository zu füllen.

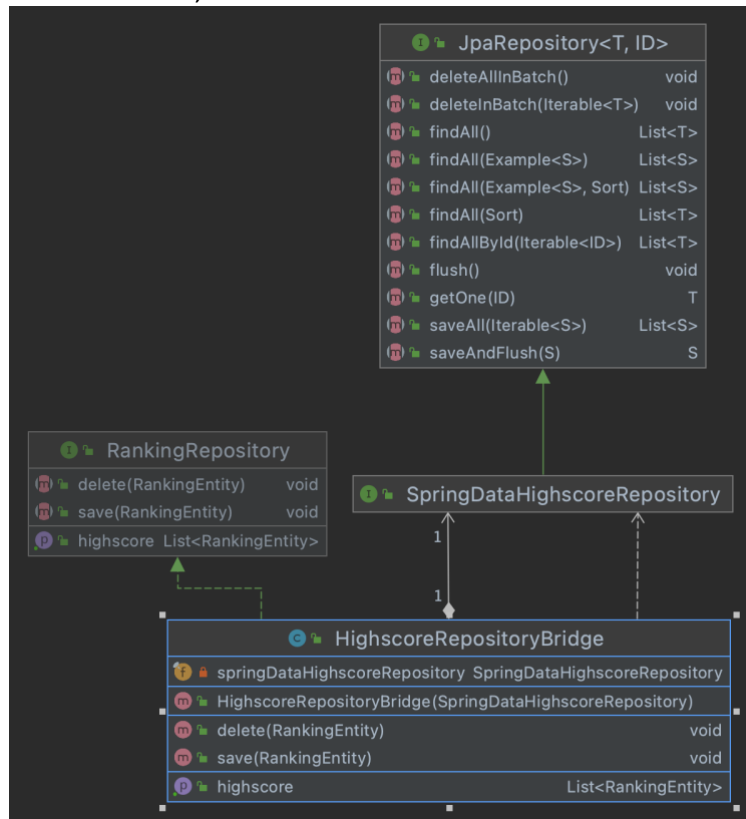


Abbildung 36: Anwendung des Bridge Patterns für Repositories (UML)

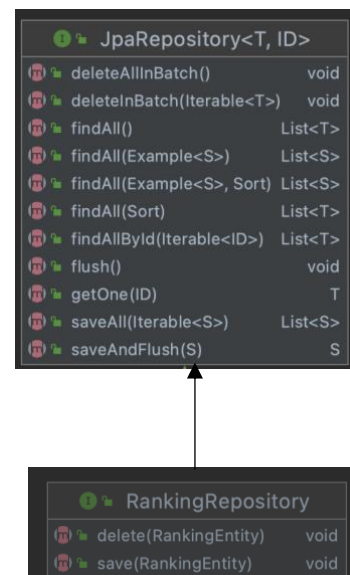


Abbildung 35: Alternative zum Bridge-Pattern (UML)

Das Bridge-Pattern wurde direkt in der Applikation angewendet, wodurch es keinen echten vorher-nachher-Vergleich gibt. Ohne Anwendung der Clean-Architecture hätte das Ranking Repository auch direkt Hibernate integrieren können, ohne das Bridge-Pattern zu nutzen. Dadurch wären aber Abstraktion und Implementierung (also die technischen Details) nicht voneinander entkoppelt gewesen. Das dazugehörige UML hätte dann nur das RankingRepository beinhaltet, das dann wiederum nur von einem Hibernate-Repository hätte erben müssen. Dieses wäre allerdings nur schwer austauschbar gewesen.

Builder

Beim Builder wird die Erzeugung komplexer Objekte vereinfacht, indem der Konstruktionsprozess in eine spezielle Klasse verlagert wird. Das Builder Pattern wird für die DTOs in der Adapter Schicht angewendet.

Vorher

Vor der Verwendung des Builders muss ein Objekt mit einem langen Konstruktor erzeugt werden oder für jedes Feld ein gesonderter Setter implementiert werden. DTOs sollten eigentlich nur die Aufgabe haben, Daten zu halten.

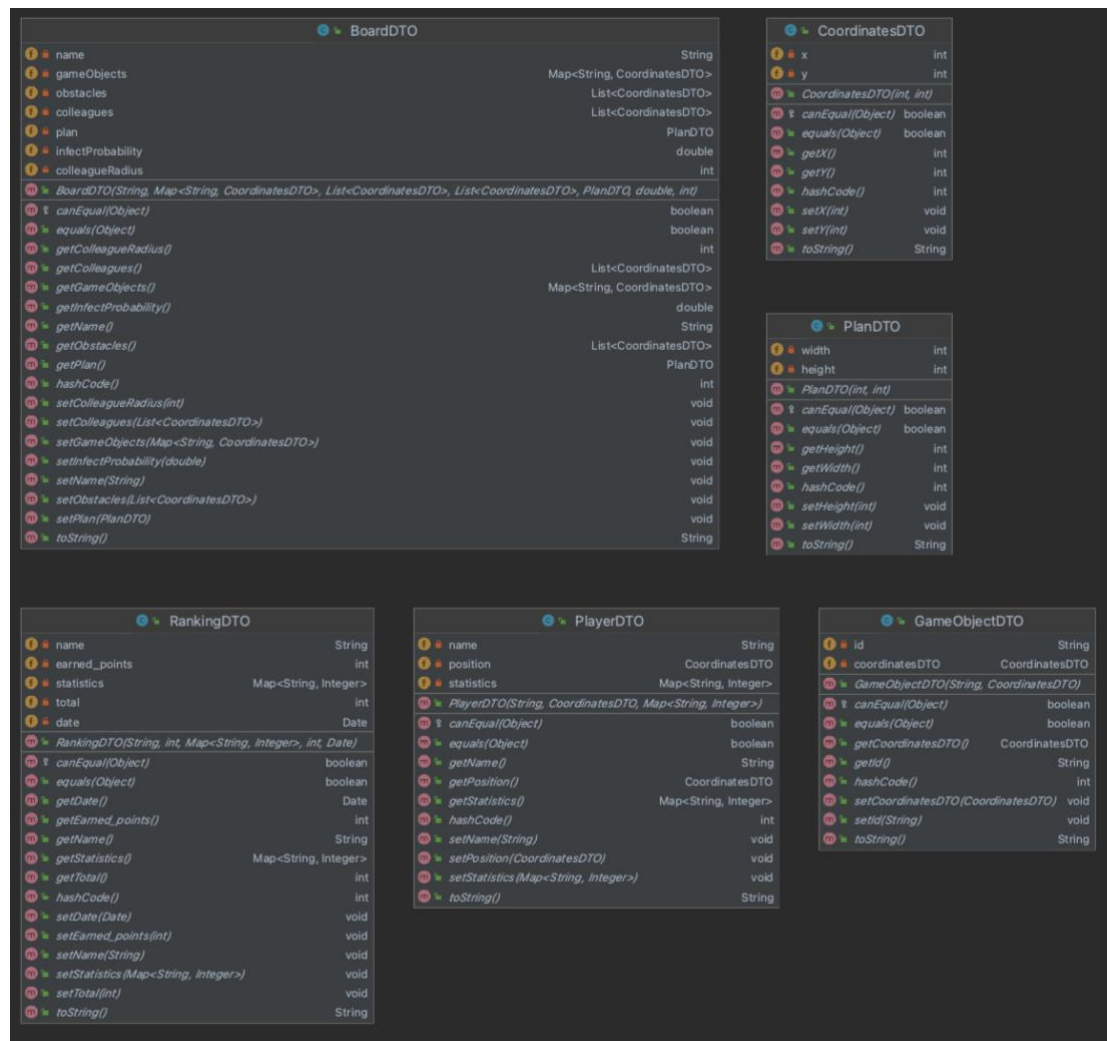


Abbildung 37: Builder – DTOs in der Adapterschicht (UML vorher)

Nachher

Der Client instanziiert zunächst ein passendes konkretes Builder-Objekt mit einer builder-Methode. Innerhalb dieser Operation werden nun die Methoden des konkreten Builders aufgerufen. Bei Erfolg jedes Feldes kann sich der Client an diesem Objekt mit einer build-Methode das fertige Objekt abholen. Das hat den Vorteil, dass der Konstruktionsprozess für das Objekt von Detailscheidungen getrennt wird. Die DTOs sind nur Datenhalter und sind zudem deutlich übersichtlicher. Da die Daten innerhalb der DTOs nur einmal gesetzt und nicht verändert werden, können die Attribute zudem als final deklariert werden und sind damit unveränderlich nach der Konstruktion eines Objekts.

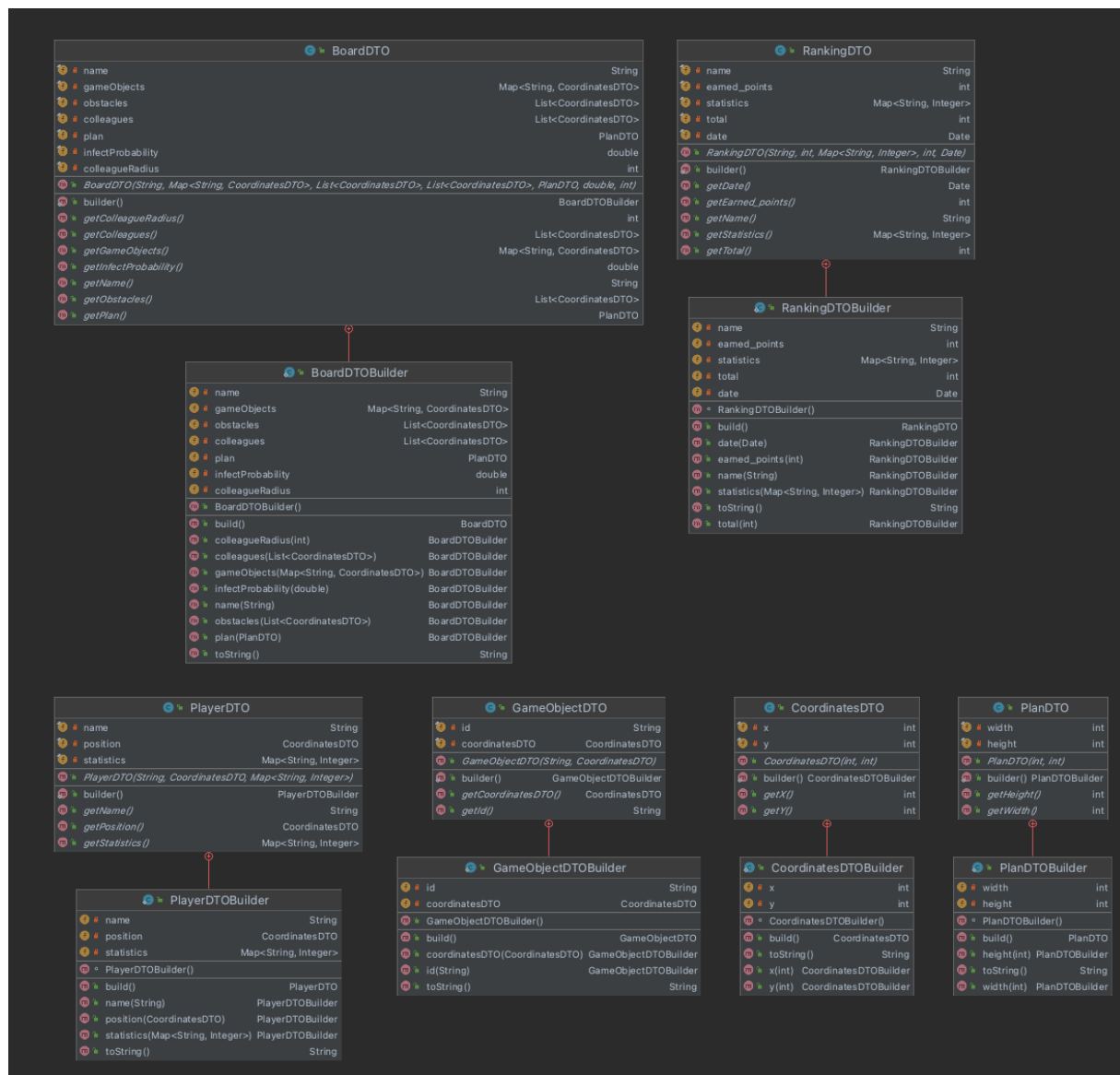


Abbildung 38: Builder - Umsetzung für alle DTOs in der Adapter-Schicht (UML nachher)

Umsetzung am Beispiel des BoardDTOs

Über das BoardDTO wird ein BoardDTOBuilder instanziiert.

```
public static BoardDTO.BoardDTOBuilder builder() { return new BoardDTO.BoardDTOBuilder(); }
```

Jede Konstruktionsmethode kann vor dem Sichern eines Datums in einem Attribut vorab eine Validierung durchführen oder eben diese direkt setzen.

```

public BoardDTO.BoardDT0Builder name(String name) {
    this.name = name;
    return this;
}

public BoardDTO.BoardDT0Builder vaccination(CoordinatesDTO vaccination) {
    this.vaccination = vaccination;
    return this;
}

public BoardDTO.BoardDT0Builder workItem(CoordinatesDTO workItem) {
    this.workItem = workItem;
    return this;
}

```

Das fertige Objekt kann durch den Builder über die build-Methode ausgegeben werden.

```

public BoardDTO build() {
    return new BoardDTO(this.name, this.vaccination, this.workItem, this.obstacles,
        this.colleagues, this.plan, this.infectProbability,
        this.colleagueRadius);
}

```