



Programmmentwurf Roguelike-Spiel

von

Lara Langnau & Gerrit Grätz

Matrikelnummer, Kurs 6847513, 1623238, TINF22B2

Abgabedatum 09.06.2025

Inhaltsverzeichnis

1 Einführung	1
1.1 Übersicht über die Applikation	1
1.2 Wie startet man die Applikation?	2
1.3 Wie testet man die Applikation?	2
2 Clean Architecture	3
2.1 Was ist Clean Architecture?	3
2.2 Analyse der Dependency Rule	3
2.2.1 Positiv-Beispiel 1: Position	3
2.2.2 Positiv-Beispiel 2: ItemStore	4
2.3 Analyse der Schichten	5
2.3.1 Schicht: Domain	5
2.3.2 Schicht: Application	6
3 SOLID	8
3.1 Analyse Single-Responsibility-Principle (SRP)	8
3.1.1 Positiv-Beispiel	8
3.1.2 Negativ-Beispiel	8
3.2 Analyse Open-Closed-Principle (OCP)	10
3.2.1 Positiv-Beispiel	10
3.2.2 Negativ-Beispiel	11
3.3 Analyse Liskov-Substitution- (LSP), Interface-Segregation- (ISP), Dependency-Inversion-Principle (DIP)	12
3.3.1 Positiv-Beispiel	12
3.3.2 Negativ-Beispiel	13
4 Weitere Prinzipien	15
4.1 Analyse GRASP: Geringe Kopplung	15
4.1.1 Positiv-Beispiel	15
4.1.2 Negativ-Beispiel	16
4.2 Analyse GRASP: Hohe Kohäsion	17
4.3 Don't Repeat Yourself (DRY)	18
5 Unit-Tests	20
5.1 10 Unit-Tests	20
5.2 ATRIP: Automatic	21
5.3 ATRIP: Thorough	22
5.3.1 Positiv-Beispiel	22
5.3.2 Negativ-Beispiel	25
5.4 ATRIP: Professional	27
5.4.1 Positiv-Beispiel	27
5.4.2 Negativ-Beispiel	28
5.5 Code Coverage	28
5.6 Fakes und Mocks	30

5.6.1	Beispiel 1: ItemStoreTest	30
5.6.2	Beispiel 2: MonsterTest	31
6	Domain Driven Design	33
6.1	Ubiquitous Language	33
6.2	Entities	34
6.3	Value Objects	35
6.4	Repositories	37
6.5	Aggregates	37
7	Refactoring	39
7.1	Code Smells	39
7.1.1	Beispiel 1: Long Method und Code Comments	39
7.1.2	Beispiel 2: Large Class	43
7.2	2 Refactorings	48
7.2.1	Beispiel 1: Polymorphismus Refactoring	48
7.2.2	Beispiel 2: Extract Method	49
8	Entwurfsmuster	51
8.1	Entwurfsmuster: Factory-Muster	52
8.2	Entwurfsmuster: Strategie-Muster	52

1 Einführung

Unser Projekt ist zu finden unter:

<https://github.com/two-Gee/roguelike-game-clean-architecture>

1.1 Übersicht über die Applikation

Unsere Applikation ist ein Roguelike Spiel, in dem der Spieler durch einen Dungeon navigiert. Dabei muss er Monster bekämpfen und weitere Räume des Dungeons erforschen. Der Spieler kann sich einen Vorteil verschaffen, indem er Items einsammelt wie z.B. Heiltränke, die seine Lebenspunkte wiederherstellen oder eine Waffe, mit der er den Monstern mehr Schaden zufügen kann. Es gibt verschiedene Arten von Monstern, die sich unterschiedlich bewegen und unterschiedlich viele Leben haben und Schaden verursachen können.

Ziel des Spiels ist es alle Monster zu besiegen, ohne dabei selbst zu sterben. Der Spieler kann sich dabei frei im Dungeon bewegen. Zunächst sieht der Spieler nur den Raum, in dem er sich befindet. Sobald er sich in einen anderen Raum bewegt, wird dieser Raum auch dauerhaft sichtbar. Jedoch sind Monster nur sichtbar, wenn der Spieler sich in der Nähe von ihnen befindet. Es gibt außerdem unterschiedliche Schwierigkeitsgrade, die die Anzahl der Monster und deren Stärke sowie die Anzahl der Hilfsmittel beeinflussen. Anhand des ausgewählten Schwierigkeitsgrades wird für jedes Spiel ein neues Dungeon generiert, indem mehrere Räume zufällig platziert und miteinander verbunden werden. Auch die Monster und Items werden zufällig in den Räumen platziert. Die Ausgabe des Spiels erfolgt über Text in der Konsole, sodass der Spieler das Spiel über die Tastatur steuern kann. Die Steuerung wird in einem Startbildschirm erklärt, der beim Starten des Spiels angezeigt wird. Sie ist wie folgt:

- **W** für nach oben bewegen,
- **A** für nach links bewegen,
- **S** für nach unten bewegen,
- **D** für nach rechts bewegen,
- **E** zum Aufnehmen von Waffen (im Spiel mit W gekennzeichnet) oder konsumierbaren Items (im Spiel mit C gekennzeichnet),
- Jede Aktion muss mit der **Enter**-Taste bestätigt werden.

1.2 Wie startet man die Applikation?

Voraussetzung für das Starten der Applikation ist, dass Java und Maven installiert sind. Außerdem muss das Github Repository des Projekts geklont werden.

Um die Applikation zu starten, müssen folgende Schritte durchgeführt werden:

1. In das Root-Verzeichnis des Projekts navigieren,
2. `mvn clean install -pl 1-adapters -am` ausführen,
3. `mvn exec:java -pl 1-adapters` ausführen.

1.3 Wie testet man die Applikation?

Als Voraussetzung für das Testen der Applikation muss maximal Java 21 installiert sein, da Mockito noch nicht mit einer höheren Java Version kompatibel ist.

Um alle Tests der Applikation auszuführen, müssen folgende Schritte ausgeführt werden:

1. In das Root-Verzeichnis des Projekts navigieren,
2. `mvn test` ausführen.

Alternativ stehen in den einzelnen Schichten jeweils `MainTest`- oder `DomainMainTest`-Klassen bereit, die über die IDE ausgeführt werden können, um die Tests der jeweiligen Schicht auszuführen.

2 Clean Architecture

2.1 Was ist Clean Architecture?

Allgemeingültiger Code und Geschäftsobjekte, die sich über ein gesamtes Unternehmen oder eine Anwendung erstrecken, befinden sich in den inneren Schichten, während spezifische Implementierungen und Frameworks in den äußeren Schichten liegen. Das führt dazu, dass die Implementierungsdetails, die in den äußeren Schichten liegen, die inneren Schichten nicht beeinflussen und problemlos ausgetauscht werden können, ohne dass die inneren Schichten angepasst werden müssen.

2.2 Analyse der Dependency Rule

In unserem Projekt wurde die Dependency Rule umgesetzt, indem jede Schicht als einzelnes Projekt erstellt wurde. Dadurch können über Maven Abhängigkeiten zwischen den Projekten (Schichten) definiert werden, die die Dependency Rule einhalten.

2.2.1 Positiv-Beispiel 1: Position

Die Klasse `Position` ist ein positives Beispiel für die Dependency Rule, da sie nur von dem `Direction`-Enum abhängt und dieses in der gleichen Schicht liegt. Andere Klassen, die von `Position` abhängen sind `Dungeon`, `DungeonRoom`, `LivingEntity`, `Item` und `DungeonGenerator` und noch weitere. Alle diese Klassen können sich nur in der gleichen Schicht oder in einer äußeren Schicht befinden, da die Klasse `Position` sich in der Domain Schicht befindet und damit in der innersten Schicht. Aus diesem Grund ist die Dependency Rule eingehalten.

Position		
Position(int, int)		
X_POS		int
Y_POS		int
getAdjacentPosition(Direction)		Position?
equals(Object)		boolean
getX_POS()		int
toString()		String
isAdjacent(Position)		boolean
hashCode()		int
getY_POS()		int

2.2.2 Positiv-Beispiel 2: ItemStore

Ein weiteres positives Beispiel für die Dependency Rule ist die Klasse `ItemStore`. Diese Klasse hängt nur von der Klasse `Item` ab. Die Klasse `ItemStore` befindet sich in der Application Schicht und damit in einer weiter äußeren Schicht, als die Domain Schicht, in der sich die Klasse `Item` befindet, deshalb ist die Abhängigkeit nur von einer äußeren Schicht zu einer inneren Schicht gegeben. Die Klasse `ItemStore` wird von den Klassen `Main`, `DungeonRenderer`, `ItemInteractionService` und `MonsterService` genutzt. Alle diese Klassen befinden sich entweder in der Application Schicht oder in der Domain Schicht, was bedeutet, dass sie sich in einer äußeren Schicht befinden und damit die Dependency Rule eingehalten wird.

ItemStore		
ItemStore(List<Item>)		
items		List<Item>
findByRoomNumber(int)		List<Item>
remove(Item)		boolean
add(Item)		void
getItems()		List<Item>

2.3 Analyse der Schichten

2.3.1 Schicht: Domain








Die Klasse `Player` ist ein Beispiel für die Domain Schicht. Sie repräsentiert den Spieler im Spiel und enthält alle relevanten Informationen über den Spieler, wie z.B. Name, Lebenspunkte und Position. Die Klasse ist in der Domain Schicht angesiedelt, da der Player eine Entity nach dem Domain-Driven-Design ist und damit Teil der Domäne des Spiels sein muss. Die Klasse erbt von `LivingEntity`, die ebenfalls in der Domain Schicht angesiedelt ist und die grundlegenden Eigenschaften und Methoden für alle lebenden Entitäten im Spiel definiert. Die Klasse `Player` ist somit ein zentrales Element der Domäne und definiert die Eigenschaften eines Spielers und grundlegende Methoden, die Logik beinhalten, die sich nur auf den Spieler bezieht und allgemeingültig für das gesamte Spiel ist.



2.3.2 Schicht: Application

Die Klasse `ItemInteractionService` ist Teil der Application Schicht, da sie die Logik für die Interaktion des Spielers mit Items im Spiel verwaltet. Sie ermöglicht es dem Spieler, Items aufzusammeln und zu wenn möglich direkt zu konsumieren, und stellt sicher, dass die Interaktion mit den Items korrekt durchgeführt wird. Die Klasse ist dafür verantwortlich, die Logik für das Aufnehmen von Items und das Konsumieren von

Items zu implementieren, ohne dabei die Eigenschaften der Items selbst zu verändern. Die Klasse `ItemInteractionService` ist somit Teil der Application Schicht, da sie die Logik für die Interaktion des Spielers mit Items verwaltet und damit einen Use Case des Spiels repräsentiert. Durch eine Feature-Änderung des Spiels kann es sein, dass die Logik für die Interaktion mit Items angepasst werden muss, dabei bleiben aber die Eigenschaften der Items oder des Spielers selbst unverändert in der Domain Schicht.

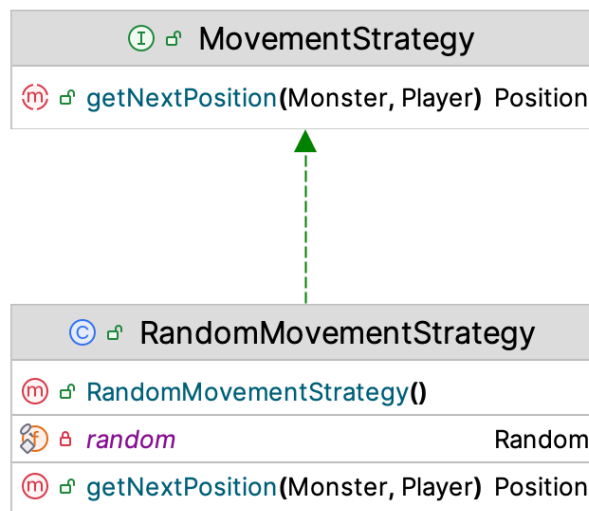
ItemInteractionService		
	<code>ItemInteractionService(DungeonRenderer, ItemStore, Player)</code>	
	<code>dungeonRenderer</code>	DungeonRenderer
	<code>itemStore</code>	ItemStore
	<code>player</code>	Player
	<code>itemsInCurrentRoom</code>	List<Item>
	<code>pickUpItem()</code>	void
	<code>getItemsInCurrentRoom()</code>	List<Item>

3 SOLID

3.1 Analyse Single-Responsibility-Principle (SRP)

3.1.1 Positiv-Beispiel

Die Klasse `RandomMovementStrategy` hat eine einzige, klar definierte Verantwortung: Sie bestimmt die nächste Position für ein Monster basierend auf einem zufälligen Bewegungsmuster und implementiert das Interface `MovementStrategy`. Sie erfüllt das Single-Responsibility-Prinzip, da die Methode `getNextPosition` ausschließlich darauf ausgerichtet ist, eine neue zufällige Position in der Nähe der aktuellen Position des Monsters zu berechnen.



3.1.2 Negativ-Beispiel

Die Klasse `MonsterMovement` übernimmt mehrere Verantwortlichkeiten:

1. Sie ruft die Methode `getNextPosition` der Bewegungsstrategie auf, um die nächste Position des Monsters zu berechnen.
2. Sie prüft, ob die neue Position von anderen Monstern belegt ist.
3. Sie prüft, ob sich an der neuen Position der Spieler befindet und deshalb die Bewegung nicht ausgeführt werden soll.
4. Sie prüft auf Kollision mit Items.
5. Sie prüft, ob das Feld an der neuen Position begehbar ist.
6. Sie führt die Bewegung des Monsters aus, wenn alle Bedingungen erfüllt sind.









 MonsterMovement	
 MonsterMovement	MonsterMovement (Monster, Player, Dungeon, List<Monster>, List
 player	Player
 otherMonsters	List<Monster>
 dungeon	Dungeon
 monster	Monster
 items	List<Item>
 move()	void

Abbildung 6: Vorher

Um das Single-Responsibility-Prinzip zu erfüllen, sollte die Klasse in mehrere Klassen aufgeteilt werden, die jeweils eine einzelne Verantwortung haben. Die Klasse `MonsterCollisionChecker` sollte für die Kollisionsprüfungen zuständig sein. Damit wäre die `MonsterMovement` Klasse nur noch für die Verwaltung der Bewegungslogik des Monsters verantwortlich, sie würde die neue Position des Monsters mithilfe der Bewegungsstrategie abrufen und die Bewegung ausführen, wenn die Kollisionprüfungen, die nun von der `MonsterCollisionChecker` Klasse durchgeführt werden, erfolgreich sind.

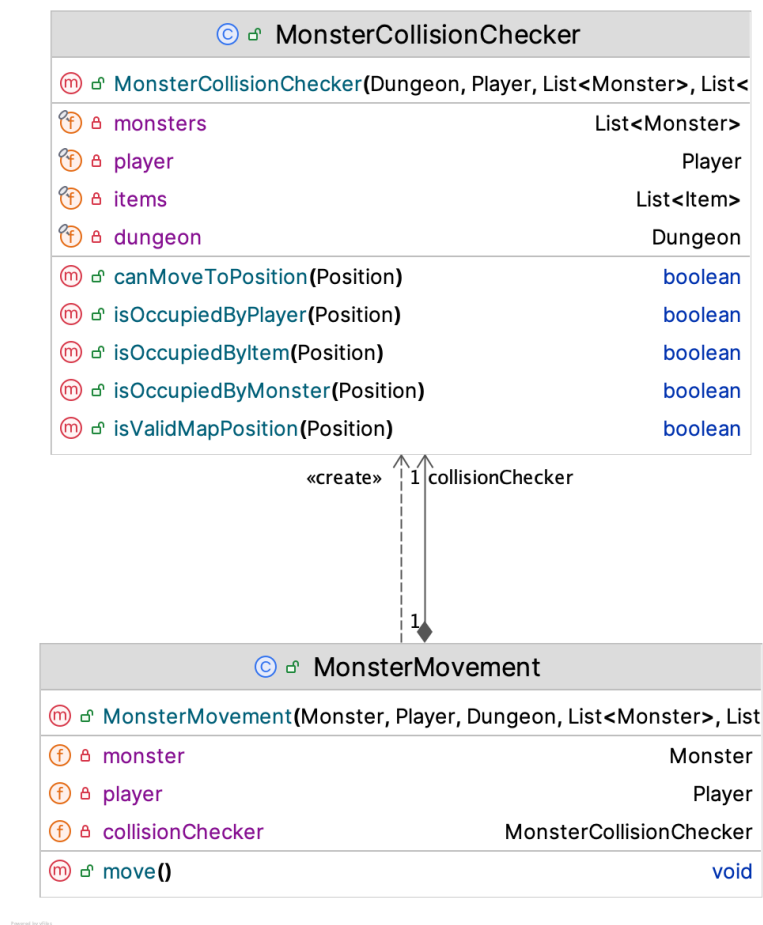


Abbildung 7: Nachher

3.2 Analyse Open-Closed-Principle (OCP)

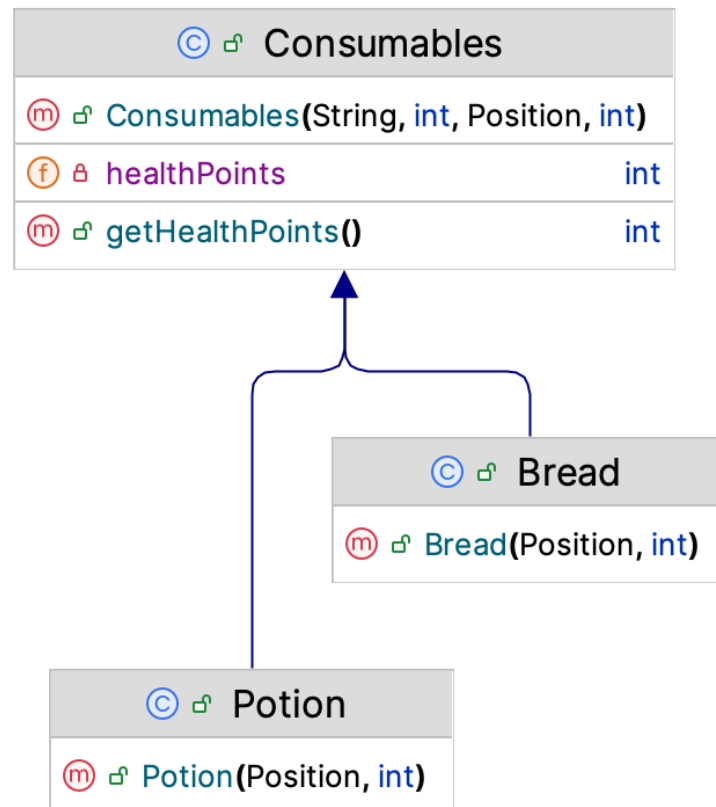
3.2.1 Positiv-Beispiel

Die Klasse `Consumables` ist ein gutes Beispiel dafür, wie das Open-Closed-Prinzip (OCP) in der Praxis angewendet werden kann. Dieses Prinzip besagt, dass eine Klasse für Erweiterungen offen, aber für Veränderungen geschlossen sein sollte. Im Fall von `Consumables` bedeutet das, dass neue Arten von konsumierbaren Items wie zum Beispiel `Potion` oder `Bread` einfach durch Vererbung hinzugefügt werden können, ohne dass der bestehende Code geändert werden muss.

Die Methode `getHealthPoints` stellt sicher, dass alle Unterklassen auf ein gemeinsames Verhalten zugreifen können, was die Wiederverwendbarkeit erhöht und den Code übersichtlich hält. Dadurch bleibt die Spiellogik, etwa die Heilfunktion oder das Inventar-

system, unverändert und funktioniert auch mit neuen `Consumables`-Typen problemlos weiter. So bleibt die Klasse `Consumables` erweiterbar und gleichzeitig geschlossen für Änderungen.

Die Erweiterbarkeit durch Vererbung und die Nutzung von Polymorphismus zeigen, wie sich OCP sinnvoll umsetzen lässt.



3.2.2 Negativ-Beispiel

Die Klasse `MonsterFactory` ist ein negatives Beispiel für OCP, da sie nicht offen für Erweiterungen ist. Wenn ein neues Monster hinzugefügt werden soll, muss die Methode `createMonster` der Klasse angepasst werden, um den neuen Monstertyp zu unterstützen. Die switch-Anweisung in der Methode `createMonster` muss um einen neuen Zweig erweitert werden, um den neuen Monstertyp zu berücksichtigen. Dies verstößt gegen das OCP, da Änderungen an der Factory-Klasse erforderlich sind, um neue Funktionalitäten hinzuzufügen.

MonsterFactory		
Ⓜ	MonsterFactory()	
Ⓜ	createRandomMonster(DungeonRoom, Set<Position>)	Monster
Ⓜ	createMonsters(int, Map<Integer, DungeonRoom>)	Map<UUID, Monster>
Ⓜ	createMonster(MonsterTypes, int, Position)	Monster
Ⓜ	createMonstersForRoom(int, DungeonRoom)	Map<UUID, Monster>

Abbildung 9: Vorher

Um OCP zu erfüllen, könnte die Factory-Klasse so umgestaltet werden, dass die `MonsterFactory` Klasse eine Map `monsterCreators` speichert. Die Map würde den Monstertypen als Schlüssel und eine Funktion, welche die Erstellung des Monsters übernimmt, als Wert enthalten. Über eine `register`-Methode könnte die Factory neue Monstertypen registrieren. Dadurch könnte die Factory um neue Monstertypen erweitert werden, ohne dass die Klasse selbst geändert werden muss. Die `registerMonsterCreator`-Methode kann dabei entweder in dem Konstruktor der Factory aufgerufen werden oder in einer separaten Initialisierungsmethode, die einmalig aufgerufen wird, um die Factory zu konfigurieren, somit ist die Factory offen für Erweiterungen, ohne, dass die `createMonster` Methode geändert werden muss.

MonsterFactory		
Ⓜ	MonsterFactory()	
⚙	monsterCreators	EnumMap<MonsterTypes, BiFunction<Integer, Position, Monster>>
Ⓜ	registerMonsterCreator(MonsterTypes, BiFunction<Integer, Position, Monster>)	void
Ⓜ	createMonster(MonsterTypes, int, Position)	Monster
Ⓜ	createMonsters(int, Map<Integer, DungeonRoom>)	Map<UUID, Monster>
Ⓜ	createRandomMonster(DungeonRoom, Set<Position>)	Monster
Ⓜ	createMonstersForRoom(int, DungeonRoom)	Map<UUID, Monster>

Abbildung 10: Nachher

3.3 Analyse Liskov-Substitution- (LSP), Interface-Segregation- (ISP), Dependency-Inversion-Principle (DIP)

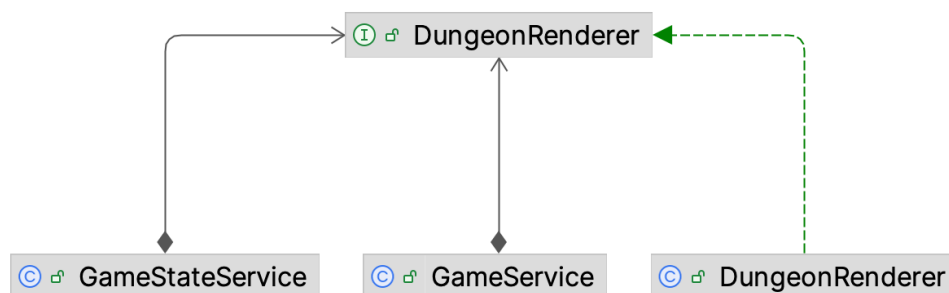
3.3.1 Positiv-Beispiel

Das Interface `DungeonRenderer` ist ein positives Beispiel für das Dependency-Inversion-Prinzip (DIP). Es definiert eine Abstraktion für die Darstellung des Dungeons, die über

die Klassen `GameService` und `GameStateService` von weiteren Klassen genutzt wird. Diese Klassen liegen in der Application-Schicht und stellen in diesem Fall die High-Level-Module dar.

Durch das Interface `DungeonRenderer` wird die Abhängigkeit von konkreten Implementierungen der Darstellung des Dungeons aufgelöst, also die High-Level-Module hängen nicht von Low-Level-Modulen ab, sondern nur von der Abstraktion `DungeonRenderer`.

Dadurch kann die Implementierung von `DungeonRenderer` geändert werden, ohne dass die Klassen der Application-Schicht angepasst werden müssen.



3.3.2 Negativ-Beispiel

Die Klasse `GameService` ist ein negatives Beispiel für DIP, da sie direkt Objekte der Klassen `MonsterService`, `PlayerService`, `ItemInteractionService`, `GameStateService` und `Player` speichert und ein Objekt der Klasse `fovCalculator` im Konstruktor erstellt. Dadurch ist die Klasse `GameService` stark von den konkreten Implementierungen dieser Klassen abhängig und DIP ist nicht erfüllt. Konkret heißt das, dass die Klassenvariablen der Klasse `GameService` als Typ nur Klassen haben, die konkrete Implementierungen sind, anstatt abstrakte Klassen oder Interfaces. Dies führt zu einer engen Kopplung zwischen den Klassen und erschwert die Testbarkeit und Wartbarkeit des Codes. Wenn beispielsweise eine neue Implementierung des `fovCalculator` benötigt wird, muss die Klasse `GameService` geändert werden, um die neue Implementierung zu verwenden.

© ↗ GameService		
Ⓜ ↗	GameService(Player, Dungeon, MonsterStore, ItemStore, Dungeon)	
Ⓜ ↗	gameStateService	GameStateService
Ⓜ ↗	playerService	PlayerService
Ⓜ ↗	itemInteractionService	ItemInteractionService
Ⓜ ↗	player	Player
Ⓜ ↗	monsterService	MonsterService
Ⓜ ↗	fovCalculator	FovCalculator
Ⓜ ↗	dungeonRenderer	DungeonRenderer
Ⓜ ↗	pickUpItem()	void
Ⓜ ↗	handlePlayer(Direction)	void
Ⓜ ↗	handleMonsters()	void
Ⓜ ↗	getDungeonRenderer()	DungeonRenderer
Ⓜ ↗	getGameStateService()	GameStateService
Ⓜ ↗	startMonsterMovementLoop(GameService)	void

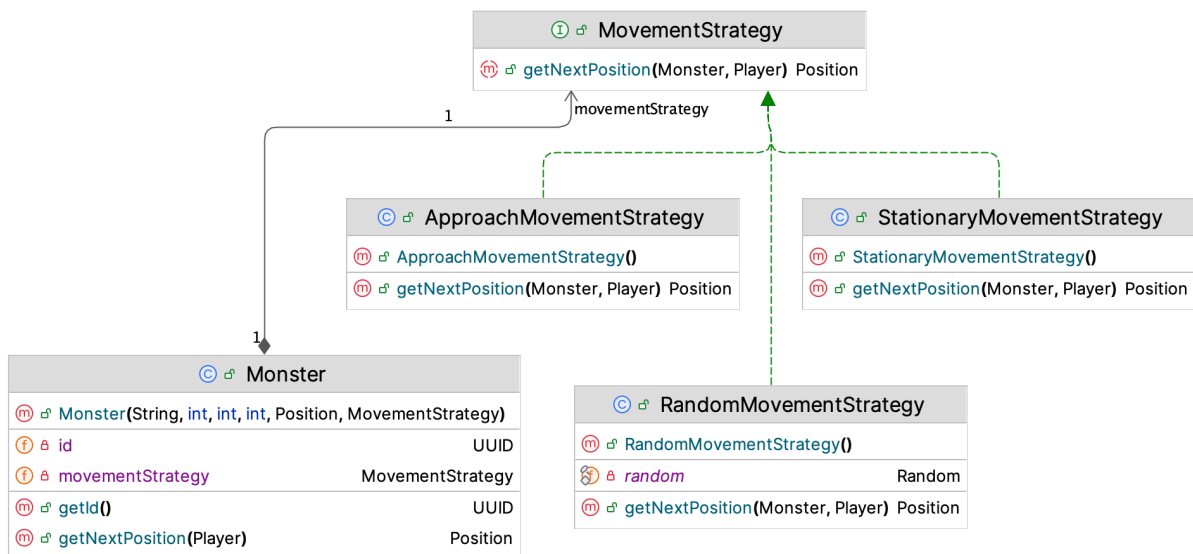
4 Weitere Prinzipien

4.1 Analyse GRASP: Geringe Kopplung

4.1.1 Positiv-Beispiel

Die Klasse `Monster` ist ein positives Beispiel für geringe Kopplung, da sie die Verantwortung für die Bewegungslogik an das `MovementStrategy`-Objekt delegiert.

Die Aufgabe der Klasse `Monster` ist es, als Implementierung des `LivingEntity`-Interfaces, die gemeinsamen Eigenschaften und grundlegenden Verhaltensweisen aller Monster im Spiel zu definieren und zu verwalten. Zudem ist sie dafür zuständig, den Monstern eine Identität zu geben, indem sie eine eindeutige ID für jedes Monster generiert. Als einige der wichtigsten Verhaltensweisen definiert die Klasse die Bewegung des Monsters zu einer neuen Position.



Die geringe Kopplung wird erreicht, indem die Klasse `Monster` nicht direkt für die Bewegungslogik verantwortlich ist, sondern stattdessen eine Instanz des Interfaces `MovementStrategy` verwendet. Die Klasse `Monster` hat ein Attribut `movementStrategy`, das eine Instanz des Interfaces `MovementStrategy` hält. Das Interface `MovementStrategy` definiert die Schnittstelle für verschiedene Bewegungsstrategien, die flexibel von der Klasse `Monster` genutzt werden können. Dadurch ist die Klasse `Monster` nicht direkt von einer konkreten Implementierung der Bewegungslogik abhängig und somit vollständig entkoppelt von der konkreten Implementierung der Bewegungsstrategien. Durch die

Delegation muss die Klasse `Monster` nicht geändert werden, wenn eine neue Bewegungsstrategie hinzugefügt wird.

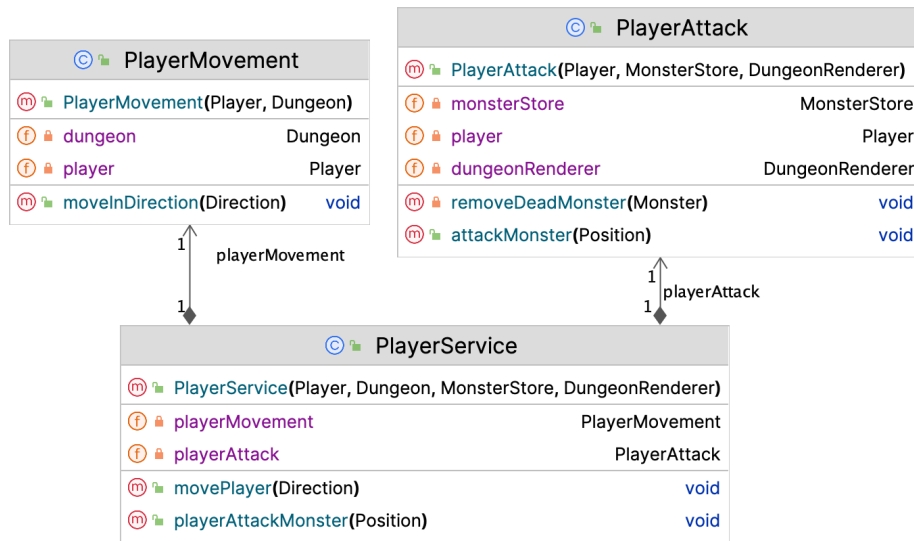
Die Kopplung zwischen der Klasse `Monster` und `Player` ist sehr gering, da das `Player`-Objekt nur in der Methode `getNextPosition` des Monsters an die Bewegungsstrategie durchgereicht wird, um die Bewegung des Monsters in Abhängigkeit von der Position des Spielers zu berechnen. Dadurch wird die Klasse `Monster` nicht direkt von der Klasse `Player` abhängig und kann unabhängig von der Implementierung des Spielers arbeiten.

4.1.2 Negativ-Beispiel

Die Klasse `PlayerService` ist ein Beispiel für hohe Kopplung, da sie stark von den Klassen `PlayerAttack` und `PlayerMovement` abhängt.

Die Klasse `PlayerService` ist für die Verwaltung der Spielerbewegung und des Angriffs auf Monster zuständig. Sie nutzt die Klassen `PlayerAttack` und `PlayerMovement`, um die Logik für den Angriff auf Monster und die Bewegung des Spielers zu implementieren.














Dadurch, dass die Klassen `PlayerAttack` und `PlayerMovement` direkt im Konstruktor der Klasse `PlayerService` instanziiert werden, entsteht eine starke Kopplung zwischen diesen Klassen. Wenn sich die Implementierung einer dieser Klassen ändert, muss auch die Klasse `PlayerService` angepasst werden. Zudem ist der Konstruktor der Klasse `PlayerService` überladen, da er mehrere Parameter entgegennimmt, die nur zur Instanziierung der Klassen `PlayerAttack` und `PlayerMovement` benötigt werden. Jede Änderung in den Parametern der Konstruktoren dieser beiden Klassen würde auch eine Änderung im Konstruktor der Klasse `PlayerService` erfordern. Das Diagramm zeigt die aktuelle Abhängigkeit der Klassen:



Die Kopplung kann aufgelöst werden, indem die internen Instanziierungen entfernt und mittels Dependency-Injection konkrete Instanzen der Klassen **PlayerAttack** und **PlayerMovement** übergeben werden. Dadurch wird die Klasse **PlayerService** unabhängiger von den konkreten Implementierungen der Klassen **PlayerAttack** und **PlayerMovement**. Dann müsste der Konstruktor der Klasse **PlayerService** nur noch diese Instanzen als Parameter entgegennehmen und nicht mehr seine aktuellen Parameter. Die hohe Kopplung könnte weiter aufgelöst werden, indem die Klassen **PlayerAttack** und **PlayerMovement** in Interfaces abstrahiert werden, sodass die Klasse **PlayerService** nur noch von den Interfaces abhängt und nicht mehr von den konkreten Implementierungen. Dadurch könnte die Klasse **PlayerService** bei Bedarf auch mit anderen Implementierungen der Angriffs- und Bewegungslogik arbeiten, ohne dass sie angepasst werden muss.

4.2 Analyse GRASP: Hohe Kohäsion

Die Klasse **PlayerAttack** ist ein Beispiel hoher Kohäsion, da sie einen einzigen, klar definierten Verantwortungsbereich hat. Sie behandelt die Logik für Angriffe des Spielers auf Monster und alles, was in der Klasse passiert, ist direkt mit dieser Aufgabe verbunden.

©  PlayerAttack		
 	PlayerAttack(Player, MonsterStore, DungeonRenderer)	
 	dungeonRenderer	DungeonRenderer
 	monsterStore	MonsterStore
 	player	Player
 	attackMonster(Position)	void
 	removeDeadMonster(Monster)	void

Sie ist fokussiert auf diese Aufgabe und enthält keine unnötigen oder irrelevanten Methoden oder Logik, die nicht direkt mit dem Angriff des Spielers zu tun haben, wie das Verwalten der Spielerbewegung. Stattdessen sind die einzigen Methoden, die in der Klasse enthalten sind, direkt auf den Angriff des Spielers auf Monster ausgerichtet oder darauf, die durch den Angriff besiegteten Monster zu entfernen. Die Methoden und Variablen innerhalb der Klasse sind logisch eng miteinander verbunden und tragen alle zur Aufgabe bei. Alle Attribute werden von der Klasse benötigt, um den Angriff des Spielers auf Monster zu verwalten und daher auch gemeinsam von allen Methoden verwendet.

Die Klasse übernimmt keine anderen Aufgaben, die nicht direkt mit dem Angriff und seinen Konsequenzen zu tun haben. Andere Aufgaben, wie das Laden der Monster im aktuellen Raum, das tatsächliche Entfernen der Monster aus dem Spiel oder das Rendern von Benachrichtigungen, sind in andere Klassen ausgelagert und werden an diese Klassen delegiert, besonders seit die Logik zum Laden der Monster im aktuellen Raum komplett in den `MonsterStore` ausgelagert wurde, wie in Abschnitt 4.3 beschrieben wird. So bleibt die Klasse `PlayerAttack` auf die Kernlogik des Angriffs fokussiert und erfüllt damit das Prinzip der hohen Kohäsion.

4.3 Don't Repeat Yourself (DRY)

In dem Commit `37526c691726a3f6fc9156926ecd7ae8283174c5` wurde duplizierter Code in den Klassen `MonsterService` und `PlayerAttack` aufgelöst. Vorher gab es in beiden Klassen die gleiche Methode, um alle Monster im aktuellen Raum aus dem `MonsterStore` zu laden, sofern es sich nicht um den Startraum handelt, da dort keine Monster vorhanden sein können. Diese Logik sah wie folgt aus:

```
private List<Monster> getMonstersInCurrentRoom() {
    if (player.getRoomNumber() != -1) {
        return monsterStore.findByRoomNumber(player.getRoomNumber());
    } else {
        return new ArrayList<>();
    }
}
```

Code 1: Vorher

In beiden Klassen wurde eine exakte Kopie dieser Logik verwendet, was zu dupliziertem Code führte. Da beide Klassen auf die `findByRoomNumber()`-Methode des `MonsterStore` zugreifen, wurde die Logik direkt in diese Methode des `MonsterStore` integriert. Dadurch wird der Code zentralisiert und duplizierter Code vermieden. Zusätzlich wird erreicht, dass die Logik, um zu prüfen, ob der Spieler im Startraum ist, direkt in der `findByRoomNumber()`-Methode behandelt wird, sodass keine eigene Methode benötigt wird, um diese Prüfung durchzuführen, bevor die `findByRoomNumber()`-Methode aufgerufen wird. Die neue Methode sieht wie folgt aus:

```
public List<Monster> findByRoomNumber(int roomNumber) {
    if (roomNumber < 0) {
        return new ArrayList<>();
    }
    return monsters.values().stream()
        .filter(monster -> monster.getRoomNumber() == roomNumber)
        .toList();
}
```

Code 2: Nachher

Sofern der Spieler in einem Gang ist (also `roomNumber = -1`) oder eine ungültige, negative Raumnummer (z.B. `-2`) übergeben wird, wird eine leere Liste zurückgegeben. Andernfalls werden die Monster im aktuellen Raum geladen.

Die Auswirkung dieser Änderung ist, dass der Code nun wartbarer und weniger fehleranfällig ist, da die Logik nur noch an einer Stelle gepflegt werden muss. Zudem wird die Lesbarkeit des Codes verbessert, da die Logik klarer strukturiert ist und gesammelt ist, sodass keine zusätzlichen Methoden benötigt werden, nur um eine if-Abfrage durchzuführen. Dadurch wird der Code insgesamt kompakter.

5 Unit-Tests

5.1 10 Unit-Tests

In der Tabelle sind 10 Unit-Tests aufgelistet, die verschiedene Aspekte der `MonsterStore` - und `ItemStore`-Klassen testen. Dabei wird das Schema `Klasse#Methode` verwendet, um die Tests zu identifizieren.

Unit-Test	Beschreibung
<code>MonsterStoreTest#testFindByRoomNumber_ExistingRoom</code>	Testet, ob die eine Liste mit der richtigen Anzahl und den richtigen Monster-Objekte für einen existierenden Raum zurückgegeben wird.
<code>MonsterStoreTest#testFindByRoomNumber_NonExistingRoom</code>	Testet, ob eine leere Liste zurückgegeben wird, wenn ein nicht existierender Raum abgefragt wird.
<code>MonsterStoreTest#testRemove_ExistingId</code>	Testet, ob ein Monster mit einer existierenden ID erfolgreich entfernt wird und die Liste entsprechend aktualisiert wird.
<code>MonsterStoreTest#testRemove_NonExistingId</code>	Testet, dass das Entfernen eines Monsters mit einer nicht existierenden ID fehlschlägt und die Liste unverändert bleibt.
<code>MonsterStoreTest#testGetMonsters_EmptyStore</code>	Testet, ob eine leere Liste zurückgegeben wird, wenn der Monster-Store leer ist.
<code>ItemStoreTest#testAdd_NewItem</code>	Testet, ob ein neues Item erfolgreich zum ItemStore hinzugefügt wird und die Liste entsprechend aktualisiert wird.

Unit-Test	Beschreibung
ItemStoreTest#testAdd_DuplicateItem	Testet, dass ein Item nicht doppelt hinzugefügt werden kann und die Liste unverändert bleibt.
ItemStoreTest#testAdd_NullItem	Testet, dass kein Null-Item hinzugefügt werden kann.
ItemStoreTest#testGetItems_PopulatedStore	Testet, ob eine Liste mit allen enthaltenen Items zurückgegeben wird, wenn der ItemStore nicht leer ist.
ItemStoreTest#testFindByRoomNumber_NegativeRoomNumber	Testet, ob eine leere Liste zurückgegeben wird, wenn eine negative Raumnummer abgefragt wird.

5.2 ATRIP: Automatic

Jede Methode, die als Testmethode ausgeführt werden soll, ist mit der Annotation `@Test` versehen. Diese Annotation ist Teil des JUnit-Frameworks und signalisiert dem Test-Runner, dass diese Methode ein Testfall ist und automatisch ausgeführt werden soll.

Mithilfe der `@Before`-Annotation wird die Methode `setUp()` vor jedem Testfall ausgeführt, um die Testumgebung vorzubereiten. Dies ist entscheidend für die Testisolation und das Zurücksetzen des Testzustands, ohne dass es manuell für jeden Testfall getan werden muss.

Da wir Maven nutzen, wird der JUnit Test Runner verwendet, um die Tests automatisch auszuführen. Dies geschieht in der Regel über den Befehl `mvn test`, der alle Testmethoden im Projekt findet und ausführt. Als Alternative kann auch jeweils die `MainTest`-Klasse einer Schicht über eine IDE ausgeführt werden, um die Tests dieser Schicht durch einen einzelnen Klick auszuführen.

Durch die Assertions in den Testmethoden wird automatisch überprüft, ob die erwarteten Ergebnisse mit den tatsächlichen Ergebnissen übereinstimmen. Wenn eine Assertion fehlschlägt, wird der Test als fehlgeschlagen markiert und gibt an, dass ein Fehler im Code vorliegt.

Das Prinzip Automatic wird insgesamt durch die Verwendung von JUnit-Annotations, Test Runner und Assertions realisiert. So wird sichergestellt, dass die Tests automatisch ausgeführt werden, ohne dass manuelle Eingriffe erforderlich sind.

5.3 ATRIP: Thorough

5.3.1 Positiv-Beispiel

Der `MonsterStoreTest` ist ein positives Beispiel für einen Test, der das Prinzip Thorough umsetzt. Er testet die `MonsterStore`-Klasse, die für das Laden und Speichern von Monstern im Spiel verantwortlich ist. Es ist ein positives Beispiel, da es versucht alle Aspekte der Klasse zu testen, einschließlich der Randbedingungen, Fehlerfälle und Immutabilität der zurückgegebenen Liste.

Jede öffentliche Methode der `MonsterStore`-Klasse wird durch mehrere Testfälle abgedeckt, wobei neben positiven Test, wie `testFindByRoomNumber_ExistingRoom()`, auch negative Tests und Randbedingungen, wie `testFindByRoomNumber_NonExistingRoom()` oder `testFindByRoomNumber_EmptyStore()`, enthalten sind. Diese Tests decken verschiedene Szenarien ab, um sicherzustellen, dass die Klasse in allen Fällen korrekt funktioniert.

Der Testfall `testGetMonsters_Immutability()` überprüft zusätzlich, dass die zurückgegebene Liste von Monstern unveränderlich ist, was wichtig ist, um die Integrität der Daten zu gewährleisten. Dies ist ein gutes Beispiel für Thoroughness, da es nicht nur die Funktionalität der Klasse testet, sondern auch sicherstellt, dass die Datenstruktur korrekt implementiert ist.

Zudem sind die Assertions mit klaren und präzisen Fehlermeldungen versehen, die im Falle eines Fehlers eine klare Diagnose ermöglichen.

Insgesamt ist der `MonsterStoreTest` also ein gutes Beispiel für Thoroughness, weil er versucht, alle Aspekte der `MonsterStore`-Klasse zu testen und sicherzustellen, dass sie in allen Szenarien korrekt funktioniert. Das folgende Code-Beispiel zeigt die Implementierung dieser Testklasse¹:

¹Da die Klasse sehr lang ist, muss das Code-Beispiel über mehrere Seiten aufgeteilt werden. Zudem wurde der Inhalt der `setUp()`-Methode entfernt, um Platz zu sparen.

```

public class MonsterStoreTest {
    private Map<UUID, Monster> monstersMap;
    private MonsterStore monsterStore;
    private Monster monster1;
    private Monster monster2;
    private Monster monster3;
    private Monster monster4;
    private MovementStrategy mockMovementStrategy;

    @Before
    public void setUp() {
        // Inhalt aus Beispiel entfernt um Platz zu sparen
    }

    @Test
    public void testFindByRoomNumber_ExistingRoom() {
        List<Monster> foundMonsters = monsterStore.findByRoomNumber(1);
        assertNotNull("List of monsters should not be null", foundMonsters);
        assertEquals("Should find 2 monsters in room 1", 2, foundMonsters.size());
        assertTrue("List should contain monster1", foundMonsters.contains(monster1));
        assertTrue("List should contain monster3", foundMonsters.contains(monster3));
        assertFalse("List should not contain monster2", foundMonsters.contains(monster2));
    }

    @Test
    public void testFindByRoomNumber_NonExistingRoom() {
        List<Monster> foundMonsters = monsterStore.findByRoomNumber(999);
        assertNotNull("List of monsters should not be null", foundMonsters);
        assertTrue("Should find no monsters in a non-existing room",
foundMonsters.isEmpty());
    }

    @Test
    public void testFindByRoomNumber_EmptyStore() {
        monsterStore = new MonsterStore(new HashMap<UUID, Monster>());
        List<Monster> foundMonsters = monsterStore.findByRoomNumber(1);
        assertNotNull("List of monsters should not be null", foundMonsters);
        assertTrue("Should find no monsters in an empty store", foundMonsters.isEmpty());
    }

    @Test
    public void testFindByRoomNumber_NegativeRoomNumber() {
        List<Monster> foundMonsters = monsterStore.findByRoomNumber(-1);
        assertNotNull("List of monsters should not be null", foundMonsters);
        assertTrue("Should return an empty list for negative room numbers",
foundMonsters.isEmpty());
    }
}
    
```

```

@Test
public void testRemove_ExistingId() {
    assertTrue("Should return true when removing an existing monster",
monsterStore.remove(monster1.getId()));
    assertFalse("Monster should be removed from the map",
monstersMap.containsKey(monster1.getId()));
    assertEquals("Map size should decrease by 1", 3, monstersMap.size());
}

@Test
public void testRemove_NonExistingId() {
    assertFalse("Should return false when removing a non-existing monster",
monsterStore.remove(UUID.randomUUID()));
    assertEquals("Map size should remain unchanged", 4, monstersMap.size());
}

@Test
public void testRemove_EmptyStore() {
    monsterStore = new MonsterStore(new HashMap<UUID, Monster>());
    assertFalse("Should return false when removing from an empty store",
monsterStore.remove(monster1.getId()));
    assertTrue("Map should remain empty", monsterStore.getMonsters().isEmpty());
}

@Test
public void testGetMonsters_PopulatedStore() {
    List<Monster> allMonsters = monsterStore.getMonsters();
    assertNotNull("List of monsters should not be null", allMonsters);
    assertEquals("Should return all 4 monsters", 4, allMonsters.size());
    assertTrue("List should contain monster1", allMonsters.contains(monster1));
    assertTrue("List should contain monster2", allMonsters.contains(monster2));
    assertTrue("List should contain monster3", allMonsters.contains(monster3));
    assertTrue("List should contain monster4", allMonsters.contains(monster4));
}

@Test
public void testGetMonsters_EmptyStore() {
    monsterStore = new MonsterStore(new HashMap<UUID, Monster>());
    List<Monster> allMonsters = monsterStore.getMonsters();
    assertNotNull("List of monsters should not be null", allMonsters);
    assertTrue("Should return an empty list for an empty store", allMonsters.isEmpty());
}

```

```

@Test
public void testGetMonsters_Immutability() {
    List<Monster> allMonsters = monsterStore.getMonsters();
    try {
        allMonsters.remove(0);
        fail("Expected UnsupportedOperationException, but none was thrown.");
    } catch (UnsupportedOperationException e) {
        // Expected exception, test passes
    } catch (IndexOutOfBoundsException e) {
        fail("Unexpected IndexOutOfBoundsException. The list was likely empty before
modification attempt.");
    }
}
}

```

5.3.2 Negativ-Beispiel

Die `PlayerServiceTest`-Klasse ist ein negatives Beispiel für Thoroughness, da sie nicht alle Aspekte der `PlayerService`-Klasse abdeckt. Sie bestätigt lediglich, dass die Methoden aufgerufen werden können, ohne sofort abzustürzen, aber es werden keine Assertions verwendet, um zu überprüfen, ob die Methoden tatsächlich das erwartete Verhalten zeigen.

Ein gründlicher Test müsste überprüfen, ob die Methoden `playerAttackMonster` und `movePlayer` des `PlayerService` die korrekten Methoden auf diesen delegierten Objekten aufrufen und ob die Argumente korrekt weitergeleitet werden. Es gibt jedoch keine `verify()`-Aufrufe von Mockito. Der Test stellt somit nicht sicher, dass die delegierten Methoden tatsächlich aufgerufen wurden.

Zudem gibt es keine Test, die negative Fälle oder Randbedingungen testen. Es werden nur positive Fälle mit gültigen (gemoekten) Eingaben getestet.

Ein weiterer Aspekt ist, dass für jede Methode nur ein einzelnes Szenario getestet wird. Zum Beispiel wird in `testPlayerAttackMonster` nur ein einzelnes `mockPosition`-Objekt verwendet, und in `testMovePlayer` wird nur eine Richtung (`Direction.EAST`) getestet. Ein gründlicher Test würde verschiedene Positionen oder Richtungen verwenden, um sicherzustellen, dass die Delegation für alle gültigen Eingaben konsistent funktioniert.

```

public class PlayerServiceTest {
    @Mock private Player mockPlayer;
    @Mock private Dungeon mockDungeon;
    @Mock private MonsterStore mockMonsterStore;
    @Mock private DungeonRenderer mockDungeonRenderer;
    @Mock private Position mockPlayerCurrentPosition;
    @Mock private Position mockNextPosition;

    private PlayerService playerService;

    @Before
    public void setUp() {
        MockitoAnnotations.initMocks(this);

        when(mockPlayer.getPosition()).thenReturn(mockPlayerCurrentPosition);

        when(mockPlayerCurrentPosition.getAdjacentPosition(any(Direction.class))).thenReturn(mockNextPosition);

        playerService = new PlayerService(mockPlayer, mockDungeon, mockMonsterStore,
        mockDungeonRenderer);
    }

    @Test
    public void testPlayerAttackMonster() {
        Position mockPosition = mock(Position.class);

        playerService.playerAttackMonster(mockPosition);
    }

    @Test
    public void testMovePlayer() {
        Direction mockDirection = Direction.EAST;

        playerService.movePlayer(mockDirection);
    }
}
    
```

5.4 ATRIP: Professional

5.4.1 Positiv-Beispiel

```

@Test
public void testEqualsWithSameValues() {
    Position pos1 = new Position(5, 10);
    Position pos2 = new Position(5, 10);
    assertTrue(pos1.equals(pos2));
}

@Test
public void testEqualsWithDifferentValues() {
    Position pos1 = new Position(5, 10);
    Position pos3 = new Position(6, 10);
    assertFalse(pos1.equals(pos3));
}

@Test
public void testEqualsWithNull() {
    Position pos1 = new Position(5, 10);
    assertFalse(pos1.equals(null));
}

@Test
public void testEqualsWithDifferentType() {
    Position pos1 = new Position(5, 10);
    assertFalse(pos1.equals("Not a position"));
}
    
```

In diesem Beispiel wird die `equals`-Methode der Klasse `Position` getestet. Die Tests decken verschiedene Szenarien ab, wie das Vergleichen von Positionen mit gleichen Werten, unterschiedlichen Werten, mit `null` und mit einem Objekt eines anderen Typs. Die Tests sind dabei sauber strukturiert und es wird sichergestellt, dass die `equals`-Methode korrekt funktioniert und robust gegen verschiedene Eingaben ist. Wie auch im Produktivcode sind die Tests aufgeteilt, dass jeder nur einen Fall überprüft, somit können die Tests leichter angepasst werden und sind übersichtlicher.

Die Namen der Tests sind klar beschrieben und geben an, was genau getestet wird. Dadurch ist es einfach zu verstehen, was jeder Test tut und welche Funktionalität er überprüft.

Durch die gute Strukturierung, die klare Benennung und die sinnvolle Aufgabe der Tests wird das Prinzip der Professionalität erfüllt.

5.4.2 Negativ-Beispiel

```
@Test
public void testGetWidth() {
    assertEquals(5, dungeon.getWidth());
}

@Test
public void testGetHeight() {
    assertEquals(3, dungeon.getHeight());
}

@Test
public void testGetDungeonRooms() {
    assertEquals(rooms, dungeon.getDungeonRooms());
}
```

In diesem Beispiel werden die Getter-Methoden der Klasse `Dungeon` getestet. Die Tests sind jedoch nicht professionell, da es unnötig ist, Getter-Methoden zu testen, die keine Logik enthalten. Getter-Methoden sollten in der Regel nicht getestet werden, da sie lediglich den Wert eines Attributs zurückgeben und keine Logik enthalten.

5.5 Code Coverage

Im Projekt werden lediglich die Application- und Domain-Schicht mit Unit-Tests abgedeckt, weshalb nur diese beiden Schichten in der Analyse der Code Coverage betrachtet werden. Die Adapter-Schicht wurde bewusst von den Unit-Tests ausgenommen, da sie primär als Konsolen-Schnittstelle dient und bereits während der Entwicklung manuell überprüft wird.

Die Code Coverage wurde mithilfe von IntelliJ erhoben. Die Abdeckung der Application- und Domain-Schicht ist wie folgt:

Element	Class, %	Method, %	Line, %	Branch, %
com.example.application	100% (23/23)	95% (70/73)	94% (336/355)	87% (168/191)
> monsterService	100% (5/5)	100% (8/8)	94% (54/57)	84% (27/32)
> factories	100% (5/5)	100% (11/11)	94% (51/54)	85% (18/21)
> map	100% (4/4)	100% (21/21)	98% (120/122)	91% (77/84)
> playerService	100% (3/3)	100% (8/8)	100% (29/29)	100% (12/12)
> stores	100% (2/2)	100% (9/9)	100% (18/18)	100% (12/12)
GameService	100% (1/1)	62% (5/8)	67% (19/28)	0% (0/2)
LevelSelection	100% (1/1)	100% (1/1)	100% (5/5)	75% (9/12)
GameStateService	100% (1/1)	100% (4/4)	100% (13/13)	100% (4/4)
ItemInteractionService	100% (1/1)	100% (3/3)	93% (27/29)	75% (9/12)
DungeonRenderer	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)

Abbildung 16: Application Coverage

Element	Class, %	Method, %	Line, %	Branch, %
all	52% (12/23)	72% (66/91)	80% (143/178)	77% (42/54)
com.example.domain	52% (12/23)	72% (66/91)	80% (143/178)	77% (42/54)
Position	100% (2/2)	88% (8/9)	90% (20/22)	96% (25/26)
Player	100% (1/1)	100% (5/5)	100% (17/17)	75% (3/4)
Dungeon	100% (1/1)	88% (8/9)	94% (17/18)	100% (4/4)
Direction	100% (1/1)	100% (2/2)	100% (5/5)	100% (0/0)
LivingEntity	100% (1/1)	92% (13/14)	95% (21/22)	100% (2/2)
MovementStrategy	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
> map	75% (3/4)	65% (17/26)	78% (43/55)	44% (8/18)
> item	25% (2/8)	55% (10/18)	55% (15/27)	100% (0/0)
> monster	20% (1/5)	37% (3/8)	41% (5/12)	100% (0/0)

Abbildung 17: Domain Coverage

Die Application-Schicht hat eine sehr hohe Klassen-, Methoden-, Zeilen- und Zweigabdeckung (insgesamt je $> 85\%$). Hierbei wird eine breite Abdeckung der Kernlogik des Projekts erreicht, einschließlich der Bewegungsstrategien, Interaktionen und Dungeon-Generierung. Der Grund für die hohe Abdeckung ist, dass die Application-Schicht die zentrale Logik des Spiels enthält und daher umfangreiche Tests erforderlich sind, um sicherzustellen, dass alle Use Cases abgedeckt sind. Eine hohe Abdeckung in der Application-Schicht ist besonders wichtig, da sie die Interaktion zwischen den verschiedenen Komponenten des Spiels steuert und somit die Grundlage für das gesamte Spielverhalten bildet. Die hohe Code-Coverage deutet darauf hin, dass die wichtigsten Logikpfade und Szenarien intensiv getestet werden, was das Vertrauen in die Stabilität der Kernfunktionen stärkt. Es ist jedoch wichtig zu betonen, dass Code-Coverage allein keine Aussage über die Qualität der Tests oder die korrekte Funktionalität der Anwendung macht. Sie zeigt lediglich, welcher Code von den Tests erreicht wird.

Die Domain-Schicht hat eine weniger hohe Klassenabdeckung (insgesamt nur 52%), dafür aber ebenfalls eine gute Methoden-, Zeilen- und Zweigabdeckung (insgesamt je $> 70\%$). Die Domain-Schicht enthält Klassen für die grundlegenden Spielobjekte wie `Position`, `Item`, `Monster` und `LivingEntity`. Viele dieser Klassen sind als reine Datenobjekte konzipiert, die hauptsächlich Attribute und einfache Getter-/Setter-Methoden enthalten. Diese Klassen enthalten in der Regel keine komplexe Geschäftslogik, die umfangreiche Tests erfordern würde. Die hohe Methoden-, Zeilen- und Zweigabdeckung für die getesteten Teile der Domain-Schicht deutet jedoch darauf hin, dass die vorhan-

dene Logik geprüft wird. Dies stellt sicher, dass die grundlegenden Verhaltensweisen und die Integrität der Datenobjekte gewährleistet sind.

Die Code Coverage lässt sich begründen, durch die Tatsache, dass die Application-Schicht die zentrale Logik des Spiels enthält und daher umfangreiche Tests erforderlich sind, um sicherzustellen, dass alle Use Cases abgedeckt sind. Während die Domain-Schicht hauptsächlich Datenobjekte enthält, die keine komplexe Logik haben, weshalb die Klassenabdeckung niedriger ist.

5.6 Fakes und Mocks

5.6.1 Beispiel 1: ItemStoreTest

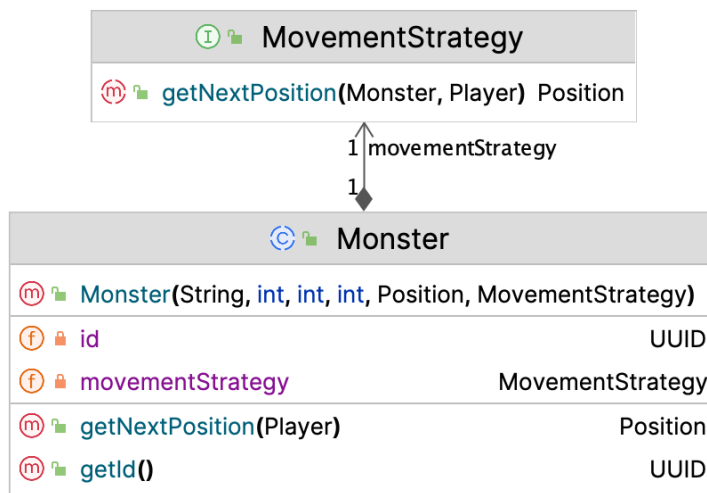
© ↗ ItemStore	
Ⓜ ↗ ItemStore(List<Item>)	
ⓕ ↗ items	List<Item>
Ⓜ ↗ findByRoomNumber(int)	List<Item>
Ⓜ ↗ remove(Item)	boolean
Ⓜ ↗ add(Item)	void
Ⓜ ↗ getItems()	List<Item>

Bei den Tests der Klasse `ItemStore` werden die zu verwaltenden Items als Mock-Objekt erstellt. Der Einsatz von Mock-Objekten in `ItemStoreTest` ist aus mehreren Gründen essenziell und vorteilhaft für die Qualität und Effizienz der Tests:

Der primäre Grund für das Mocking ist die Isolation des `ItemStore`. Ein Unit-Test soll genau eine Einheit, in diesem Fall die `ItemStore`-Klasse, prüfen, ohne dass andere Komponenten oder deren Verhalten die Testergebnisse verfälschen. Der `ItemStore` verwaltet `Item`-Objekte. Ohne Mocks müssten wir echte `Item`-Objekte mit all ihren Abhängigkeiten (wie Position) und potenziell komplexer Konstruktionslogik instanziiieren. Dies würde die Tests unnötig kompliziert machen und die Testausführung verlangsamen. Durch das Mocken von `Item` wird sichergestellt, dass ausschließlich die Logik des `ItemStore` (z.B. wie er Items hinzufügt, entfernt oder filtert) validieren wird und nicht Fehler in der `Item`-Klasse selbst den Tests beeinflussen.

Mocks bieten Kontrolle über das Verhalten der simulierten Abhängigkeiten. Dies erlaubt eine präzise Kontrolle über das Verhalten von Abhängigkeiten, indem spezifische Rückgabewerte (z.B. für `getRoomNumber`) definiert und so diverse Szenarien wie das Vorhandensein oder Fehlen von Items in bestimmten Räumen simuliert werden können. Gleichzeitig wird das Test-Setup vereinfacht, da `Item`-Objekte nicht vollständig instanziiert werden müssen, sondern es können nur die Attribute definiert werden welche von Bedeutung sind für den Test. Das macht den Code schlanker und lesbarer.

5.6.2 Beispiel 2: MonsterTest



In Unit-Tests der Klasse `Monster` kommt ein Mock-Objekt für das Interface `MovementStrategy` zum Einsatz. Der Grund dafür ist die interne Abhängigkeit: Die Methode `getNextPosition()` der Klasse `Monster` delegiert die Berechnung der nächsten Position an die Methode `getNextPosition()` ihrer `MovementStrategy`-Implementierung. Der Fokus des Tests der Klasse `Monster` liegt nicht darauf, wie eine Bewegung konkret berechnet wird, das ist die Aufgabe der Klasse `MovementStrategy` selbst.

Stattdessen soll der Test sicherstellen, dass die Klasse `Monster` ihre Bewegungslogik korrekt an die Klasse `MovementStrategy` delegiert und deren Ergebnis ordnungsgemäß verarbeitet. Durch die Übergabe eines Mock-Objekts der Klasse `MovementStrategy` im Konstruktor des Tests der Klasse `Monster` ergeben sich mehrere Vorteile.

Der Test kann exakt vorgeben, welche Position die Methode `getNextPosition()` des Mocks zurückgeben soll, was den Test deterministisch macht und die Überprüfung ermöglicht, ob die Klasse `Monster` diesen Wert korrekt übernimmt.

Zudem lässt sich mithilfe des Mocks verifizieren, ob die Methode `getNextPosition()` der Klasse `MovementStrategy` tatsächlich aufgerufen wurde und dies mit den erwarteten Argumenten geschah. So wird die korrekte Delegation der Aufgabe sichergestellt. Diese Vorgehensweise ermöglicht es, die Klasse `Monster` isoliert zu testen, ohne von der komplexen Logik konkreter Implementierungen der Klasse `MovementStrategy` abhängig zu sein.

6 Domain Driven Design

6.1 Ubiquitous Language

Bezeichnung	Bedeutung	Begründung
Dungeon	Beschreibt ein konkretes Level des Spiels. Es ist die zentrale Spielumgebung, in dem der Spieler agiert und besteht aus einem zweidimensionalen Raster von Tiles. Es enthält mehrere Räume, Korridore und andere Elemente, die die Spielwelt definieren.	Das Dungeon ist die Grundlage, auf dem das Spiel basiert und beschreibt für Experten und Entwickler die zentrale Spielumgebung. Es muss klar definiert sein, dass ein Dungeon ein Level des Spiels ist und nicht selbst mehrere Level enthält.
Dungeon Tile	Repräsentiert die kleinste Einheit, aus der die Spielwelt aufgebaut ist. Es ist ein einzelnes Element des zweidimensionalen Rasters des Dungeons und kann verschiedene Eigenschaften haben, wie z.B. ob es begehbar ist, ob es ein Floor- oder Wall-Tile ist, oder ob es andere Elemente wie Monster enthält.	Dungeon Tiles ermöglichen es, die Spielwelt in kleinere Einheiten zu unterteilen und die Struktur des Dungeons genauer zu beschreiben. Sie sind wichtig, um unter anderem die Begehrbarkeit, visuelle Darstellung, Platzierung von Elementen und Navigation im Dungeon darzustellen.
Dungeon Room	Ein abgegrenzter Bereich innerhalb des Dungeons, der vom Spieler betreten werden kann und Gegenstände und Monster enthalten kann. Ein Dungeon Room besteht aus mehreren Floor-Tiles und einer Abgrenzung von anderen Räumen und Korridoren durch Wall-Tiles. Ge-	Es ist ein zentraler Bestandteil des Dungeons und wird als Container für andere Spielinhalte und in verschiedenen Spielmechaniken (z.B. Generierung, Kampf, Gegenstände) verwendet. Ohne eine klare Definition von Dungeon Rooms wäre es schwierig, über die Struktur, Ge-

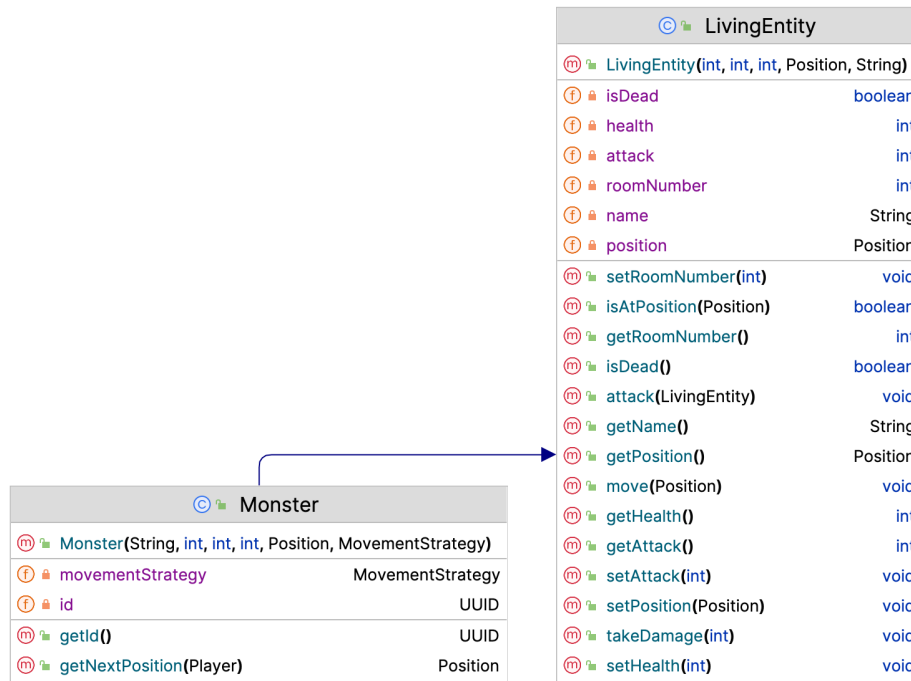
Bezeichnung	Bedeutung	Begründung
	meinsam mit den Korridoren beschreibt ein Dungeon Room den vom Spieler begehbaren Bereich im Dungeon.	nerierung und Inhalte eines Dungeons zu sprechen. Es ist wichtig, um zu verdeutlichen, welchen Bereich des Dungeons der Spieler betreten kann und wo welche anderen Elemente enthalten sein können.
Player	Repräsentiert den menschlichen Spieler, der die Hauptfigur im Spiel steuert und dessen Aktionen und Entscheidungen das Spiel beeinflussen. Der Player hat Attribute wie Position, Gesundheit und Angriffsstärke, die für die Spielmechanik und Interaktionen im Dungeon wichtig sind. Der Player kann sich innerhalb des Dungeons bewegen, angreifen und mit anderen Elementen interagieren.	Der Player ist die zentrale Entität, ohne die das Spiel nicht funktionieren würde. Diese Bezeichnung ist notwendig, um den menschlichen Spieler von anderen Entitäten im Spiel (wie Monstern) zu unterscheiden und um die Interaktionen des Spielers mit der Spielwelt zu definieren.

6.2 Entities

Die Entity **Monster** bezeichnet eine feindliche Entität im Spiel (einen Gegner), die den Spieler angreifen kann oder vom Spieler angegriffen werden kann. Ein Monster besitzt Attribute wie Position, Gesundheit und Angriffsstärke, die für die Spielmechanik wichtig sind. Jedes Monster gehört zu einem bestimmten Typ (z.B. Goblin, Troll), der seine Eigenschaften bestimmt. Dieser Typ wird dem Monster als Name zugewiesen. Zudem hat jedes Monster einen Bewegungstyp, der bestimmt, ob und wie es sich im Dungeon bewegt.

Die Entity **Monster** ist ein zentrales Konzept der Domäne und wird benötigt, um die Interaktionen zwischen dem Spieler und den feindlichen Entitäten zu definieren sowie die Schwierigkeit des Spiels zu beeinflussen. Die Klasse **Monster** ist mit ihren

Attributen und Methoden in der nachfolgenden Abbildung dargestellt. Die Abbildung zeigt ebenfalls die Klasse `LivingEntity`, von der `Monster` erbt, um auch die geerbten Attribute und Methoden der Klasse `Monster` abzubilden.



Die Klasse `Monster` wird als Entity modelliert, da ein Objekt dieser Klasse nicht durch seine Attributwerte definiert wird, sondern durch seine eindeutige Identität, die über eine UUID realisiert wird. Die Werte des Objekts können sich im Laufe des Spiels verändern, z.B. wenn das Monster Schaden nimmt oder sich bewegt und dadurch seine Position ändert. Die Identität des Monsters bleibt jedoch immer gleich und ermöglicht es, das Monster im Spiel eindeutig zu identifizieren und zu verfolgen. Wenn zwei Monster die gleichen Attribute haben, sind sie dennoch zwei verschiedene Entitäten, da sie unterschiedliche Identitäten (IDs) haben. Dadurch kann ein Monster gezielt aus dem Spiel entfernt oder verändert werden, sodass jedes Monster seinen eigenen Lebenszyklus hat. Demnach lässt sich die Klasse `Monster` eindeutig als Entity modellieren.

6.3 Value Objects

Die Klasse `Position` ist ein Value Object, das die Position eines Objekts im Dungeon beschreibt. Eine Position besteht aus einer x- und einer y-Koordinate, die die Position im zweidimensionalen Raster des Dungeons eindeutig definiert. Die Klasse `Position` wird












verwendet, um die Platzierung und Bewegung von Objekten wie Spielern, Monstern und Gegenständen oder von Dungeon Tiles und Dungeon Rooms zu beschreiben.

Die Position ist ein unveränderliches Objekt, das keiner eigenen Identität bedarf, da es nur durch seine Werte (x- und y-Koordinate) definiert ist. Zwei Positionen mit den gleichen Werten sind identisch und können nicht unterschieden werden. Die Position eines Objekts beschreibt einen Wert zu einem bestimmten Zeitpunkt. Im Laufe des Spiels kann sich die Position eines Objekts ändern, z.B. wenn sich ein Spieler oder ein Monster bewegt, aber die Objekte der Klasse `Position` können nicht verändert werden, sondern es wird immer eine neue Instanz erstellt, wenn eine Position verändert wird. Dadurch ist die Klasse `Position` ein typisches Beispiel für ein Value Object, das nur durch seine Werte definiert ist.

© <code>Position</code>	
<code>Position(int, int)</code>	
<code>Y_POS</code>	<code>int</code>
<code>X_POS</code>	<code>int</code>
<code>getAdjacentPosition(Direction)</code>	<code>Position?</code>
<code>equals(Object)</code>	<code>boolean</code>
<code>hashCode()</code>	<code>int</code>
<code>toString()</code>	<code>String</code>
<code>getY_POS()</code>	<code>int</code>
<code>isAdjacent(Position)</code>	<code>boolean</code>
<code>getX_POS()</code>	<code>int</code>

Die Abbildung zeigt die Klasse `Position` mit ihren Attributen und Methoden. Die Klasse selbst sowie die Attribute `x` und `y` sind als final deklariert, um Vererbung und Veränderung zu verhindern. Die Klasse `Position` implementiert die Methoden `equals` und `hashCode`, um die Gleichheit von Positionen basierend auf ihren Attributwerten zu überprüfen. Dies ist wichtig, um sicherzustellen, dass zwei Positionen mit den gleichen Werten als identisch betrachtet werden. Innerhalb des Konstruktors wird überprüft, ob die übergebenen Werte gültig sind, also ob sie nicht negativ sind, um sicherzustellen, dass nur gültige Positionen erstellt werden können.

6.4 Repositories

 MonsterStore	
 MonsterStore	 Map<UUID, Monster>
 monsters	 Map<UUID, Monster>
 getMonsters()	 List<Monster>
 findByRoomNumber(int)	 List<Monster>
 remove(UUID)	 boolean

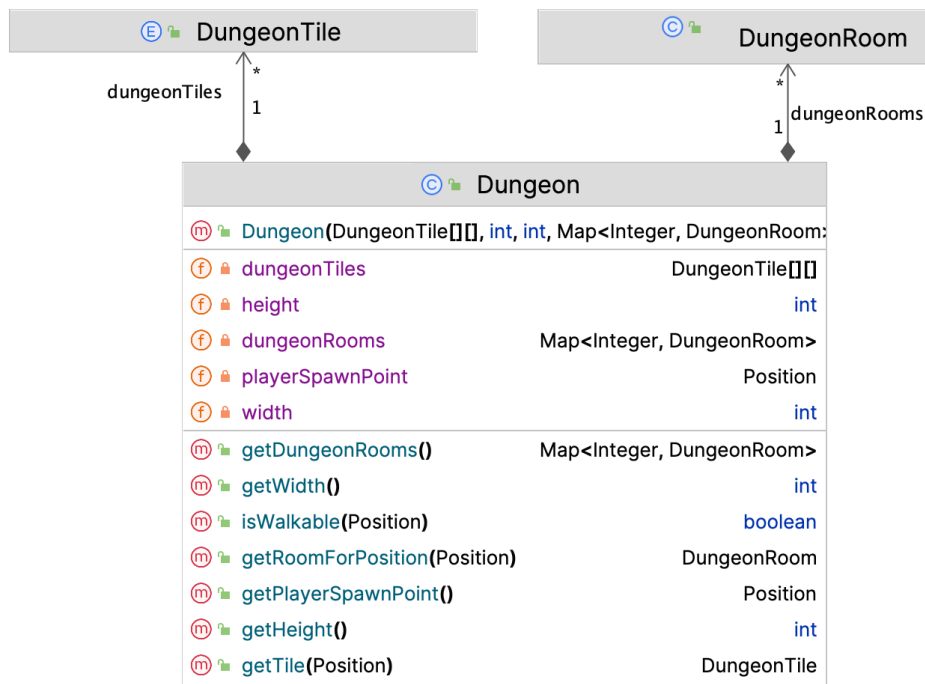
In der Abbildung ist die Klasse `MonsterStore` mit ihren Attributen und Methoden dargestellt. Die Klasse ist ein Beispiel für ein Repository des Projekts. Sie dient dazu, alle Objekte der Klasse `Monster` zu speichern und zu verwalten. Sie bietet die Möglichkeit, Monster über ihre Raumnummer zu suchen, einzelne Monster über ihre ID zu entfernen und alle Monster zu laden. Das Repository ist für die Verwaltung der Lebenszyklen der Monster verantwortlich und ermöglicht es, die Monster im Spiel zu verfolgen und zu aktualisieren.

Die Klasse `MonsterStore` ist als Repository modelliert, um dem Anwendungscode einen einfachen Zugriff auf die Monster zu ermöglichen, ohne dass der Anwendungscode sich um die Details der Speicherung und Verwaltung der Monster kümmern muss. Es kapselt die Logik zur Verwaltung der Monster und bietet eine klare Schnittstelle für den Zugriff auf die Monster, sodass Monster anhand ihrer wichtigsten Eigenschaften gesucht werden können.

6.5 Aggregates

Das Aggregat `Dungeon` ist ein zentrales Konzept des Projekts und stellt die gesamte Spielwelt dar, in der der Spieler agiert. Es besteht aus mehreren Dungeon Rooms, die durch Korridore miteinander verbunden sind. Das Aggregat `Dungeon` enthält alle Informationen über die Struktur des Dungeons, die Platzierung von Objekten wie Monstern, Gegenständen oder dem Spieler und die Interaktionen im Dungeon. `Dungeon` verwaltet die Dungeon Rooms und Dungeon Tiles als gemeinsame Einheiten, die zusammen eine konsistente Spielwelt bilden. Die gleichnamige Klasse `Dungeon` ist dabei das Aggregat-Root, das den Zugriff auf die Dungeon Rooms und Dungeon Tiles im Dungeon steuert.

Alle Zugriffe auf die Dungeon Rooms und Dungeon Tiles erfolgen über das Aggregat-Root **Dungeon**, um die Konsistenz des Aggregats zu gewährleisten.



Das Dungeon-Aggregat ist in der Abbildung dargestellt. Die Klasse **Dungeon** enthält die Attribute `dungeonRooms` und `dungeonTiles` und bietet Methoden zum Suchen von Dungeon Rooms und Dungeon Tiles. Das Aggregat wurde gewählt, da ein Raum oder Tile nicht unabhängig vom Dungeon existieren kann. Sie sind immer Teil eines Dungeons und können nicht ohne diesen existieren. Das Aggregat **Dungeon** stellt sicher, dass alle Dungeon Rooms und Dungeon Tiles konsistent und zusammenhängend sind und ermöglicht es, die Spielwelt als Ganzes zu verwalten.

7 Refactoring

7.1 Code Smells

7.1.1 Beispiel 1: Long Method und Code Comments

In der Klasse `DungeonGenerator` gibt es die Methode `generateRooms()`, die eine sehr lange Methode ist, da sie mehrere Aufgaben gleichzeitig erledigt. Die Methode hat folgende Verantwortlichkeiten:

1. Generieren der Raumanzahl
2. Generieren der Raumgrößen und -positionen
3. Den ersten Raum als Startpunkt initialisieren
4. Prüfen, ob die Räume sich überschneiden
5. Hinzufügen der Räume zum Dungeon
6. Generieren der Verbindungen zwischen den Räumen

Aufgrund der Länge der Methode ist es schwierig, den Überblick zu behalten und Änderungen vorzunehmen. Daher sind in der Methode mehrere Kommentare enthalten, die die einzelnen Schritte erklären. Diese Kommentare sind jedoch nicht ideal, da sie das eigentliche Problem nicht lösen. Die Kommentare sind ein Hinweis darauf, dass die Methode zu lang ist und in kleinere Methoden aufgeteilt werden sollte. Im folgenden Code-Beispiel ist die Methode `generateRooms()` in ihrer ursprünglichen Form zu sehen:

```

private void generateRooms() {
    int numRooms = rand.nextInt(dungeonConfiguration.getMaxRooms() -
dungeonConfiguration.getMinRooms() + 1) + dungeonConfiguration.getMinRooms();
    int count = 0;

    // Generate rooms with random sizes and positions
    while (count < numRooms) {
        // Room size
        int width = (count == 0) ? 4 : rand.nextInt(dungeonConfiguration.getMaxRoomSize()
- dungeonConfiguration.getMinRoomSize() + 1) + dungeonConfiguration.getMinRoomSize();
        int height = (count == 0) ? 4 : rand.nextInt(dungeonConfiguration.getMaxRoomSize()
- dungeonConfiguration.getMinRoomSize() + 1) + dungeonConfiguration.getMinRoomSize();
        // Room location

        int x = rand.nextInt(dungeonConfiguration.getWidth() - width - 2) + 1;
        int y = rand.nextInt(dungeonConfiguration.getHeight() - height - 2) + 1;

        DungeonRoom room = new DungeonRoom(x, y, width, height, count);

        // Check if room is first room
        if(dungeonRooms.isEmpty()) {
            // Set room center of first room as player spawn
            playerSpawnPoint = room.getRoomCenter();

            generateRoom(room, dungeonRooms);
            dungeonRooms.put(count, room);
            count++;
        }
        else {
            // Check if room intersects one of the previously generated rooms
            boolean intersects = false;
            for (DungeonRoom other : dungeonRooms.values()) {
                if(room.intersectsOtherRoom(other)) {
                    intersects = true;
                    break;
                }
            }

            if(!intersects) {
                generateRoom(room, dungeonRooms);
                dungeonRooms.put(count, room);
                count++;
            }
        }
    }
}
    
```

Code 9: Vorher

Ein möglicher Lösungsweg wäre, die Methode in kleinere Methoden aufzuteilen, die jeweils eine einzelne Aufgabe übernehmen. Dadurch wird die Lesbarkeit verbessert und die Wartbarkeit erhöht. Die Kommentare könnten dann durch aussagekräftige Methodennamen ersetzt werden, die den Zweck der jeweiligen Methode klarer machen. Der Code für diesen Lösungsweg könnte wie folgt aussehen²:

```
private void generateRooms() {
    int numRooms = calculateNumberOfRooms();
    int count = 0;

    while (count < numRooms) {
        DungeonRoom room = createRandomRoom(count);

        if (isFirstRoom(count)) {
            initializeFirstRoom(room);
            count++;
        } else if (!doesRoomIntersect(room)) {
            addRoomToDungeon(room);
            count++;
        }
    }
}
```

Code 10: Nachher (1/2)

²Da der Lösungsweg sehr lang ist, musste das Code-Beispiel über mehrere Seiten aufgeteilt werden. Es sollte dennoch als ein zusammenhängendes Beispiel betrachtet werden.

```

private DungeonRoom createRandomRoom(int count) {
    int width = (count == 0) ? 4 : rand.nextInt(dungeonConfiguration.getMaxRoomSize() -
dungeonConfiguration.getMinRoomSize() + 1)
        + dungeonConfiguration.getMinRoomSize();
    int height = (count == 0) ? 4 : rand.nextInt(dungeonConfiguration.getMaxRoomSize() -
dungeonConfiguration.getMinRoomSize() + 1)
        + dungeonConfiguration.getMinRoomSize();
    int x = rand.nextInt(dungeonConfiguration.getWidth() - width - 2) + 1;
    int y = rand.nextInt(dungeonConfiguration.getHeight() - height - 2) + 1;

    return new DungeonRoom(x, y, width, height, count);

private int calculateNumberOfRooms() {
    return rand.nextInt(dungeonConfiguration.getMaxRooms() -
dungeonConfiguration.getMinRooms() + 1)
        + dungeonConfiguration.getMinRooms();
}

private boolean isFirstRoom(int count) {
    return count == 0;
}

private void initializeFirstRoom(DungeonRoom room) {
    playerSpawnPoint = room.getRoomCenter();
    addRoomToDungeon(room);
}

private boolean doesRoomIntersect(DungeonRoom room) {
    for (DungeonRoom other : dungeonRooms.values()) {
        if (room.intersectsOtherRoom(other)) {
            return true;
        }
    }
    return false;
}

private void addRoomToDungeon(DungeonRoom room) {
    generateRoom(room, dungeonRooms);
    dungeonRooms.put(room.getId(), room);
}

```

Code 11: Nachher (2/2)

Hierbei fällt auf, dass die Methode `generateRooms()` nun deutlich kürzer und übersichtlicher ist. Jede einzelne Aufgabe ist in eine eigene Methode ausgelagert, die einen klaren Namen hat und somit den Zweck der Methode beschreibt. Somit sind auch keine Kommentare mehr notwendig, da der Code selbst selbsterklärend ist.

7.1.2 Beispiel 2: Large Class

Die Klasse `GameService` ist ein Beispiel für den Code Smell „Large Class“. Diese Klasse umfasst viele Instanzvariablen und Methoden, sodass sie zu viele Verantwortlichkeiten übernimmt. Dadurch wird die Klasse unübersichtlich und schwer wartbar. Die Klasse ruft die Methoden zur Spieler- und Monsterbewegung auf und kümmert sich dabei selbst um die Angriffslogik. Zudem kümmert sich die Klasse um das Aufheben und Konsumieren von Gegenständen und prüft, ob das Spiel gewonnen oder verloren wurde. Um den Code Smell „Large Class“ zu beheben wird die Klasse in mehrere Use Cases aufgeteilt, die jeweils eine einzelne Verantwortung übernehmen. Hierfür wurden die Klassen `PlayerService`, `MonsterService` eingeführt, auf die der `GameService` zugreift. Diese Klassen sammeln wiederum die Use Cases der Klassen `PlayerMovement`, `MonsterMovement` und die neuen Klassen `GameStateService`, `MonsterAttack` und `ItemInteractionService`, die jeweils eine einzelne Verantwortung übernehmen. Dadurch kümmert sich der `GameService` nur noch um die Spiellogik und die Interaktion zwischen den einzelnen Use Cases. Der `GameStateService` kümmert sich um den Zustand des Spiels und prüft, ob das Spiel gewonnen oder verloren wurde. Die `MonsterAttack`-Klasse kümmert sich um die Angriffslogik der Monster und die `ItemInteractionService`-Klasse kümmert sich um das Aufheben und Konsumieren von Gegenständen. Der `PlayerService` und `MonsterService` kombinieren jeweils die Use Cases der Spieler und Monster, also ihre Bewegungen und Angriffe.

Der Code für den Lösungsweg dieser Refactoring-Maßnahme ist sehr umfangreich, da er mehrere Klassen umfasst. Daher wird hier nicht der tatsächliche Quellcode im Dokument angegeben, sondern nur auf den Commit verwiesen, der die Refactoring-Maßnahme enthält: `377b5f83d251c305220c41c545cbf6dffb645faf2`. Um einen Überblick zu erhalten, wie der `GameService` vor dem Refactoring aussah, ist im Folgenden der Code der Klasse `GameService` vor dem Refactoring zu sehen³. Eine übersichtlichere Darstellung kann durch das Auschecken des Commits `35f7f59542677d556cfb14b574afff107f6816d2` erhalten werden, da es der letzte Commit vor der Refactoring-Maßnahme war.

³Da die Klasse sehr lang war, musste das Code-Beispiel über mehrere Seiten aufgeteilt werden. Es sollte dennoch als ein zusammenhängendes Beispiel betrachtet werden.

```

public class GameService {
    private FovCalculator fovCalculator;
    private Player player;
    private Dungeon dungeon;
    private MonsterStore monsterStore;
    private ItemStore itemStore;
    private DungeonRenderer dungeonRenderer;
    private boolean gameOver;

    public GameService(Player player, Dungeon dungeon, MonsterStore monsterStore, ItemStore
itemStore, DungeonRenderer dungeonRenderer, FovCache fovCache) {
        this.player = player;
        this.dungeon = dungeon;
        this.monsterStore = monsterStore;
        this.itemStore = itemStore;
        this.dungeonRenderer = dungeonRenderer;
        gameOver=false;
        this.fovCalculator = new FovCalculator(dungeon, fovCache);
        Position spawnPoint = dungeon.getPlayerSpawnPoint();
        this.fovCalculator.calculateFov(spawnPoint.getX_POS(), spawnPoint.getY_POS(), 5);
    }

    public void movePlayer(Direction direction){
        List<Monster> monstersInCurrentRoom;
        List<Item> itemsInCurrentRoom;
        if(player.getRoomNumber() != -1){ monstersInCurrentRoom =
monsterStore.findByRoomNumber(player.getRoomNumber()); }
        else{ monstersInCurrentRoom = new ArrayList<>(); }
        PlayerMovement playerMovement = new PlayerMovement(player, dungeon,
monstersInCurrentRoom);
        Position newPos = player.getPosition().getAdjacentPosition(direction);
        for(Monster monster : monstersInCurrentRoom){
            if(monster.getPosition().equals(newPos)){
                player.attack(monster);
                dungeonRenderer.renderNotification(player.getName() + " attacks " +
monster.getName() + "! " + monster.getName() + " took " + player.getAttack() + "
damage.");
                if(monster.isDead()){
                    monsterStore.remove(monster.getId());
                    dungeonRenderer.renderNotification(player.getName() + " killed " +
monster.getName() + "!");
                }
                checkForWin();
                return;
            }
        }
        playerMovement.moveInDirection(direction);
        fovCalculator.calculateFov(player.getPosition().getX_POS(),
player.getPosition().getY_POS(), 5);
        dungeonRenderer.renderDungeon();
    }
}
    
```

```

public void movePlayer(Direction direction){
    List<Monster> monstersInCurrentRoom;
    List<Item> itemsInCurrentRoom;
    if(player.getRoomNumber() != -1){
        monstersInCurrentRoom = monsterStore.findByRoomNumber(player.getRoomNumber());
    }else{
        monstersInCurrentRoom = new ArrayList<>();
    }
    PlayerMovement playerMovement = new PlayerMovement(player, dungeon,
monstersInCurrentRoom);
    Position newPos = player.getPosition().getAdjacentPosition(direction);

    for(Monster monster : monstersInCurrentRoom){
        if(monster.getPosition().equals(newPos)){
            player.attack(monster);
            dungeonRenderer.renderNotification(player.getName() + " attacks " +
monster.getName() + "! " + monster.getName() + " took " + player.getAttack() + "
damage.");
            if(monster.isDead()){
                monsterStore.remove(monster.getId());
                dungeonRenderer.renderNotification(player.getName() + " killed " +
monster.getName() + "!");
            }
            checkForWin();
            return;
        }
    }
    playerMovement.moveInDirection(direction);
    fovCalculator.calculateFov(player.getPosition().getX_POS(),
player.getPosition().getY_POS(), 5);

    dungeonRenderer.renderDungeon();
}
    
```



```

public void pickUpItem(){
    List<Item> itemsInCurrentRoom;
    if(player.getRoomNumber() != -1){
        itemsInCurrentRoom = itemStore.findByRoomNumber(player.getRoomNumber());
    }else{
        itemsInCurrentRoom = new ArrayList<>();
    }
    for(Item item : itemsInCurrentRoom){
        if(item.getPosition().isAdjacent(player.getPosition())){
            if(item instanceof Weapon weaponNew){
                if(player.getEquippedWeapon() != null){
                    Weapon weapon = player.unEquipWeapon();
                    weapon.setPosition(item.getPosition());
                    weapon.setRoomNumber(item.getRoomNumber());
                    itemStore.add(weapon);
                    dungeonRenderer.renderNotification("Player switched weapon! " +
weaponNew.getName() + " adds " + weaponNew.getAttack() + " attack damage.");
                }else{
                    dungeonRenderer.renderNotification("Player picked up a weapon! " +
weaponNew.getName() + " adds " + weaponNew.getAttack() + " attack damage.");
                }
                player.equipWeapon(weaponNew);
                itemStore.remove(weaponNew);
            }else if(item instanceof Consumables consumables){
                player.heal(consumables.getHealthPoints());
                itemStore.remove(item);
                dungeonRenderer.renderNotification("Player used a consumable! " +
consumables.getName() + " heals " + consumables.getHealthPoints() + " health.");
            }
            return;
        }
    }
    dungeonRenderer.renderNotification("No item to pick up");
}

```

```

    public void moveMonsters(){
        if(player.getRoomNumber() != -1){
            boolean renderDungeon = false;
            for (Monster monster : monsterStore.findByRoomNumber(player.getRoomNumber())){
                if(monster.getPosition().isAdjacent(player.getPosition())){
                    monster.attack(player);
                    dungeonRenderer.renderNotification(monster.getName() + " attacks " +
player.getName() + "! " + player.getName() + " took " + monster.getAttack() + " damage.");
                    if(checkGameLost(player)) return;
                }else{
                    MonsterMovement monsterMovement = new MonsterMovement(monster, player,
dungeon, monsterStore.findByRoomNumber(player.getRoomNumber()),
itemStore.findByRoomNumber(player.getRoomNumber()));
                    monsterMovement.move();
                    renderDungeon= true;
                }
            }
            if(renderDungeon) dungeonRenderer.renderDungeon();
        }
    }
    private boolean checkGameLost(Player player){
        if(player.isDead()){
            gameOver=true;
            dungeonRenderer.renderGameLost();
            return true;
        }
        return false;
    }
    private void checkForWin(){
        if(monsterStore.getMonsters().isEmpty()){
            dungeonRenderer.renderWin();
            gameOver=true;
        }
    }
    public boolean isGameOver() {
        return gameOver;
    }

    public DungeonRenderer getDungeonRenderer() {
        return dungeonRenderer;
    }

```

```

public static void startMonsterMovementLoop(GameService gameService){
    Runnable monsterMovementLoop = () -> {
        while (!gameService.isGameOver()) {
            gameService.moveMonsters();
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    };
    new Thread(monsterMovementLoop).start();
}
    
```

7.2 2 Refactorings

7.2.1 Beispiel 1: Polymorphismus Refactoring

Bei verschiedenen `Monster`-Typen wird das Polymorphismus Refactoring angewendet, um einzelne `Monster`-Typen in eigene Klassen zu kapseln. Dadurch werden die Eigenschaften der `Monster`-Typen in den jeweiligen Unterklassen definiert. Dadurch wird die Übersichtlichkeit und Wartbarkeit des Codes verbessert, da die Eigenschaften der `Monster`-Typen klar von der Erstellung der `Monster` Objekte in der `MonsterFactory` getrennt sind.

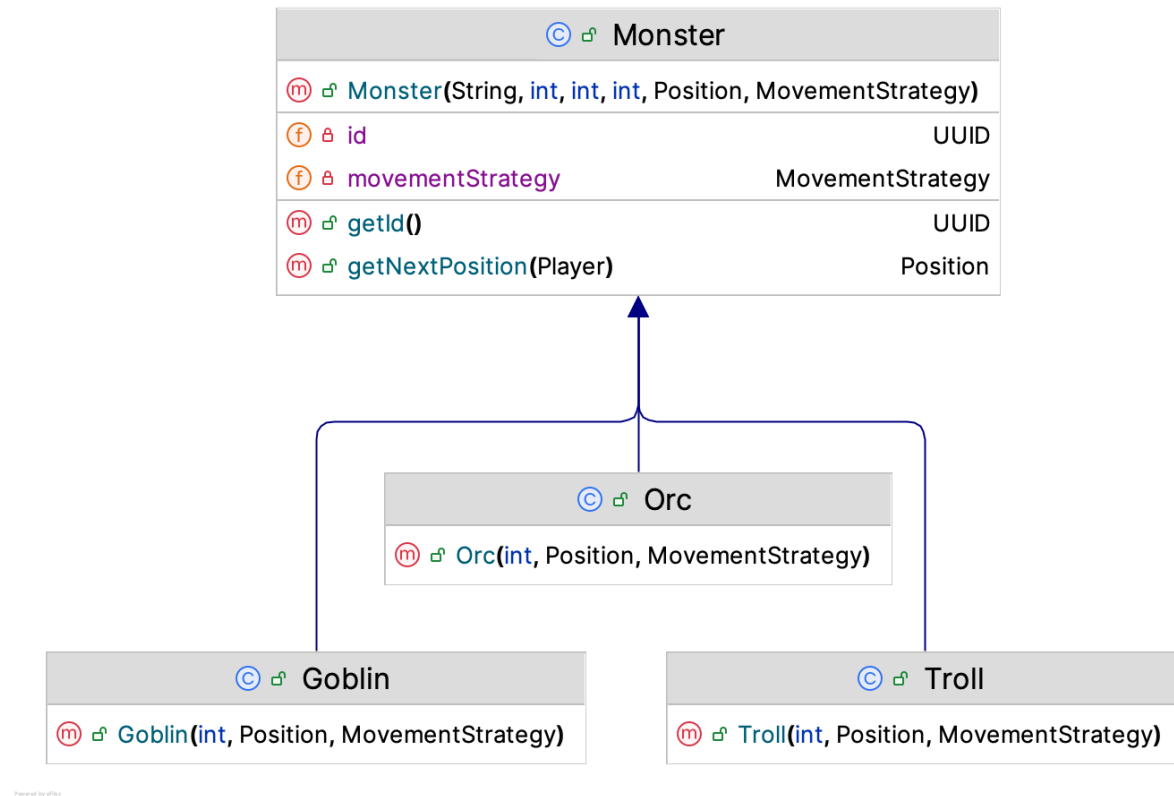
Vorher

Commit : 7913893f3131d21121cac37020df855e55be4c4a

© Monster		
Ⓜ	Monster(String, int, int, int, Position, MovementStrategy)	
f	id	UUID
f	movementStrategy	MovementStrategy
Ⓜ	getId()	UUID
Ⓜ	getNextPosition(Player)	Position

Nachher

Commit : `fdaaaebfd4849cc2defeb81f4a55598b7f6419bf`



```

public static Monster createMonster(MonsterTypes type, int roomID, Position position){
    return switch (type) {
        case GOBLIN -> new Goblin(roomID, position, new ApproachMovementStrategy());
        case ORC -> new Orc(roomID, position, new RandomMovementStrategy());
        case TROLL -> new Troll(roomID, position, new StationaryMovementStrategy());
    };
}

```











7.2.2 Beispiel 2: Extract Method











Es wird eine Methode aus der `MonsterFactory` und aus der `ItemFactory` extrahiert, um eine zufällige `Position` in einem Raum zu generieren, die noch nicht von einem anderen Objekt belegt ist. Die Logik wird sowohl in der `MonsterFactory` als auch in der `ItemFactory` verwendet, da sowohl beim Erstellen der Monster als auch beim Erstellen der Items eine zufällige Position benötigt wird, an der das Objekt platziert werden soll. Durch die Extraktion der Methode wird die Duplizierung von Code vermieden und die

Lesbarkeit des Codes verbessert, da die Logik zur Generierung der Position nun an einem zentralen Ort definiert ist.

Vorher

Commit: 90e695d37807da53a07d44862db85cacf10abd39

ItemFactory		
	 <code>ItemFactory()</code>	
	 <code>createItems(int, List<DungeonRoom>)</code>	List<Item>
	 <code>createItem(ItemTypes, Position, int)</code>	Item
	 <code>createRandomItem(DungeonRoom, Set<Position>)</code>	Item
	 <code>createItemsForRoom(int, DungeonRoom)</code>	List<Item>

MonsterFactory		
	 <code>MonsterFactory()</code>	
	 <code>createRandomMonster(DungeonRoom, Set<Position>)</code>	Monster
	 <code>createMonster(MonsterTypes, int, Position)</code>	Monster
	 <code>createMonstersForRoom(int, DungeonRoom)</code>	Map<UUID, Monster>
	 <code>createMonsters(int, Map<Integer, DungeonRoom>)</code>	Map<UUID, Monster>

Powered by IntelliJ

Nachher

Commit: fc0e43a6578beeb99bf77c9f717b36473dc4ae788

© ↗ ItemFactory		
Ⓜ ↗	ItemFactory()	
Ⓜ ↗	createItems(int, List<DungeonRoom>)	List<Item>
Ⓜ ↗	createItem(ItemTypes, Position, int)	Item
Ⓜ ↗	createRandomItem(DungeonRoom, Set<Position>)	Item
Ⓜ ↗	createItemsForRoom(int, DungeonRoom)	List<Item>

© ↗ PositionGenerator		
Ⓜ ↗	PositionGenerator()	
Ⓜ ↗	generateRandomPosition(DungeonRoom, Set<Position>)	Position

© ↗ MonsterFactory		
Ⓜ ↗	MonsterFactory()	
Ⓜ ↗	createRandomMonster(DungeonRoom, Set<Position>)	Monster
Ⓜ ↗	createMonster(MonsterTypes, int, Position)	Monster
Ⓜ ↗	createMonstersForRoom(int, DungeonRoom)	Map<UUID, Monster>
Ⓜ ↗	createMonsters(int, Map<Integer, DungeonRoom>)	Map<UUID, Monster>












Powered by gfm4s

8 Entwurfsmuster

8.1 Entwurfsmuster: Factory-Muster

Das Factory-Muster ist ein Entwurfsmuster, das die Instanziierung von Objekten kapselt und es ermöglicht, Objekte über einen Methodenaufruf zu erstellen, ohne zu definieren, welche Klasse instanziiert wird. Es fördert die Entkopplung von Klassen und ermöglicht eine flexible Erweiterbarkeit.

Im Kontext des Roguelike-Spiels wird das Factory-Muster verwendet, um verschiedene Monster zu erstellen. Die Klasse `MonsterFactory` ist dafür verantwortlich, die Instanziierung von Monstern zu kapseln. Sie bietet eine Methode `createMonsters`, die alle Monster für das Spiel erstellt. Dafür werden die Methoden `createMonstersForRoom`, `createRandomMonster` und `createMonster` verwendet. Gleichmaßen ist auch die Klasse `ItemFactory` für die Erstellung von Items zuständig. Mithilfe dieses Musters wird die Logik zur Erstellung von Monstern und Items von der Logik des Spiels getrennt. Falls nun ein neuer Monstertyp oder ein neuer Itemtyp hinzugefügt werden soll, muss lediglich die Factory-Klasse angepasst werden, um den neuen Typ zu unterstützen. Die Logik des Spiels bleibt dabei unverändert.

©  MonsterFactory		
 	<code>MonsterFactory()</code>	
 	<code>createRandomMonster(DungeonRoom, Set<Position>)</code>	<code>Monster</code>
 	<code>createMonsters(int, Map<Integer, DungeonRoom>)</code>	<code>Map<UUID, Monster></code>
 	<code>createMonster(MonsterTypes, int, Position)</code>	<code>Monster</code>
 	<code>createMonstersForRoom(int, DungeonRoom)</code>	<code>Map<UUID, Monster></code>

8.2 Entwurfsmuster: Strategie-Muster

Das Strategie-Muster ist ein Entwurfsmuster, das eine Familie von Algorithmen definiert, jeden in einer separaten Klasse kapselt und sie austauschbar macht. Das Muster ermöglicht, dass der Algorithmus unabhängig von den Clients variieren kann, die ihn verwenden.

Im Kontext des Roguelike-Spiels könnten verschiedene Monster unterschiedliche Bewegungsmuster haben. Einige könnten sich dem Spieler nähern, andere könnten sich

zufällig bewegen und einige könnten stationär bleiben. Das Strategie-Muster wird verwendet, um diese verschiedenen Bewegungsverhaltensweisen zu kapseln, indem für jede Bewegungsstrategie eine eigene Klasse erstellt wird, die das Interface `MovementStrategy` implementiert. Dabei sind die Bewegungsstrategien austauschbar, sodass das Monster jederzeit seine Bewegungsstrategie ändern kann, ohne dass die `Monster`-Klasse oder die `MonsterMovement`-Klasse geändert werden müssen.

Außerdem könnten weitere Bewegungsstrategien hinzugefügt werden, ohne die bestehende Logik zu verändern, und mehrere Monster können die gleiche Bewegungsstrategie verwenden, ohne dass die Logik dupliziert wird. Dies fördert die Erweiterbarkeit und Wartbarkeit des Codes.

