

Programmmentwurf

Textbasiertes RPG

“Exam Escape”

Name: Monika Grigorova

Matrikelnummer: 2571611

Abgabedatum: 27.05.2023

1. Einführung

1.1. Übersicht über die Applikation

Die Applikation ist ein Rogue-like-RPG-Spiel, bei dem der Spieler aus drei Klassen wählen und gegen verschiedene Feinde in verschiedenen Levels kämpfen kann. Die Feinde sind von anderen Spielen stark inspiriert und das Spiel bietet eine textbasierte Spielerfahrung.

Die Funktionsweise der Applikation besteht darin, dass der Spieler eine Klasse wählt und dann in rundenbasierten Kämpfen gegen Feinde antritt. Jeder Feind hat einzigartige Fähigkeiten und Angriffe. Der Spieler kann sowohl normale Angriffe als auch spezielle Angriffe einsetzen, um den Feind zu besiegen.

Es gibt drei verschiedene Level mit zufällig generierten Feinden. Jedes Mal, wenn der Spieler ein neues Level betritt, werden die Feinde zufällig ausgewählt.

Die Würfel im Spiel werden verwendet, um zufällige Zahlen zu generieren und Entscheidungen im Spiel zu treffen. Zum Beispiel kann mit einem Würfelwurf entschieden werden, ob ein Feind seinen Spezialangriff einsetzt oder einen normalen Angriff durchführt. Darüber hinaus wird mithilfe der Würfel auch der Schaden ermittelt, den ein Angriff verursacht. Durch die Verwendung von Würfeln wird eine gewisse Unvorhersehbarkeit und Variation in den Spielverlauf gebracht.

Die Applikation löst das Problem der Langeweile, indem sie eine unterhaltsame und zugängliche Spielerfahrung bietet. Das Spiel ist relativ einfach gehalten und wurde für Spaß und zur Unterhaltung entwickelt. Für Testzwecke und zur Einfachheit wurden die Schwierigkeit der Feinde und der Levels bewusst niedrig gehalten.

1.2. Wie startet man die Applikation?

Um die Anwendung zu starten, benötigt man IntelliJ IDEA und Maven. Die folgende Schritt-für-Schritt-Anleitung beschreibt den Installationsprozess:

[GitHub Repository für das Projekt](#)

1. Kopieren Sie das Projekt aus dem GitHub-Repository oder entpacken Sie die ZIP-Datei auf Ihrem Computer.
2. Öffnen Sie das Projekt in IntelliJ IDEA.
3. Führen Sie einen Maven-Build des Projekts durch. Dabei werden alle Abhängigkeiten heruntergeladen und alle Tests durchgelaufen.

4. Navigieren Sie im Projektbaum zur Klasse "StartGame" in dem Modul "x-global-ase-project" (Pfad: `/IdeaProjects/ase-textbased-rpg/x-global-ase-project/src/main/java/ase/project/StartGame.java`).
5. Klicken Sie mit der rechten Maustaste auf die Klasse "StartGame" und wählen Sie "Run", um das Spiel zu starten.

Das Spiel wird im Terminal gestartet und man erhält Anweisungen zur Steuerung des Spiels.

1.3. Wie testet man die Applikation?

Um die Anwendung zu testen, benötigen Sie IntelliJ IDEA und Maven. Die folgende Schritt-für-Schritt-Anleitung beschreibt den Testprozess:

1. Öffnen Sie das Projekt in IntelliJ IDEA.
2. Führen Sie einen Maven-Build des Projekts durch, um sicherzustellen, dass alle Abhängigkeiten korrekt heruntergeladen wurden und dass alle Tests erfolgreich durchgeführt werden.

Falls separate Module getestet werden müssen, werden folgende Schritte benötigt:

1. Navigieren Sie im Projektbaum zu dem Modul, das Sie testen möchten.
2. Klicken Sie mit der rechten Maustaste auf das Modul und wählen Sie "Run Tests" oder "Debug Tests", um die Tests des Moduls auszuführen.
3. Überprüfen Sie die Testergebnisse in der "Run" oder "Debug" Konsole, um sicherzustellen, dass alle Tests erfolgreich durchgeführt wurden.

Bitte beachten Sie, dass das Projekt so konfiguriert ist, dass alle Tests beim Build mit Maven automatisch durchgeführt werden. Wenn der Build ohne Fehler abgeschlossen wird, können Sie davon ausgehen, dass die Anwendung ordnungsgemäß funktioniert.

2. Clean Architecture

2.1. Was ist Clean Architecture

Clean Architecture ist ein Softwarearchitekturkonzept, das darauf abzielt, die Abhängigkeiten zwischen den Komponenten einer Anwendung zu minimieren und Qualität des Codes, Wartbarkeit und Testbarkeit zu verbessern.

Die Clean Architecture besteht aus vier Schichten. Die Domain-Schicht enthält die Kernlogik, Geschäftsregeln und -modelle. Sie ist unabhängig von anderen Schichten und enthält keine technischen Details. In die Applikation-Schicht werden die Anwendungsfälle (use cases) und

die Interaktion zwischen den Benutzern und der Domain-Schicht implementiert. Die Adapter-Schicht enthält die Implementierung von Schnittstellen und Datenzugriff für externe Komponenten wie Datenbanken, Webdienste und UI-Frameworks. In diesem Projekt wurde das File-System als externe Komponente betrachtet. Die Adapter-Schicht ermöglicht die Kommunikation zwischen der Applikation-Schicht und den externen Systemen. Die Plugin-Schicht enthält spezifische Erweiterungen für externe Bibliotheken. Sie ist optional und wird verwendet, um zusätzliche Funktionen anzubieten.

Die Vorteile der Clean Architecture liegen in ihrer Flexibilität, Testbarkeit und Wartbarkeit. Durch diese Trennung der Verantwortlichkeiten und die Minimierung von Abhängigkeiten wird der Code einfacher zu verstehen, zu testen und zu ändern. Die Anwendung wird auch weniger anfällig für Änderungen in externen Systemen, da die Abhängigkeiten durch Schnittstellen definiert sind. Dies ist besonders anfällig bei schnellen Wechseln und der Erweiterung der verwendeten Technologien. Dadurch wird eine langfristige Skalierbarkeit und Erweiterbarkeit der Anwendung ermöglicht.

2.2. Analyse der Dependency Rule

Positiv-Beispiel 1: Dependency Rule

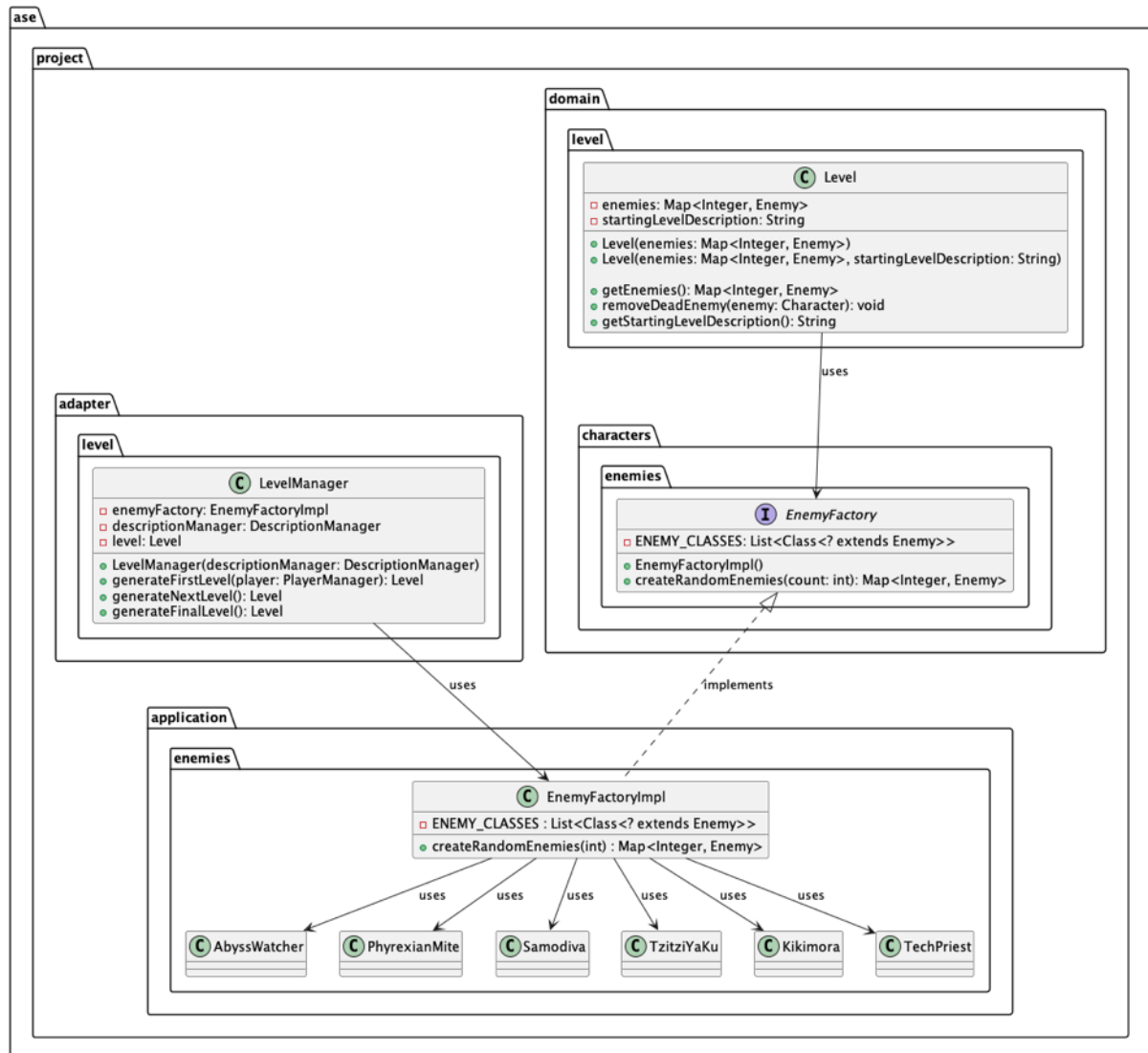


Abbildung 1. Dependency Rule - Positiv-Beispiel 1

Für das Beispiel wurde die Klasse `EnemyFactoryImpl` (Application) gewählt. Das oben dargestellte Diagramm zeigt die Struktur und die Beziehungen zwischen den gewählten Klassen in der Anwendung. Die `LevelManager` Klasse in der Adapter-Schicht verwendet `EnemyFactoryImpl` aus dem Applikations-Modul, das wiederum das `EnemyFactory` Interface aus dem Domain-Modul implementiert. Die Klasse `Level` im Domain-Modul verwendet auch das `EnemyFactory` Interface.

In Bezug auf die Dependency Rule zeigt dieses Diagramm, dass äußere Schichten (wie Adapter und Applikation) von den inneren Schichten (wie Domain) abhängen, aber nicht umgekehrt.

Dies fördert eine Entkopplung und erhöht die Modularität der Software. Dasselbe gilt für die Adapter-Schicht, die von der Applikation-Schicht abhängt.

Positiv-Beispiel 2: Dependency Rule

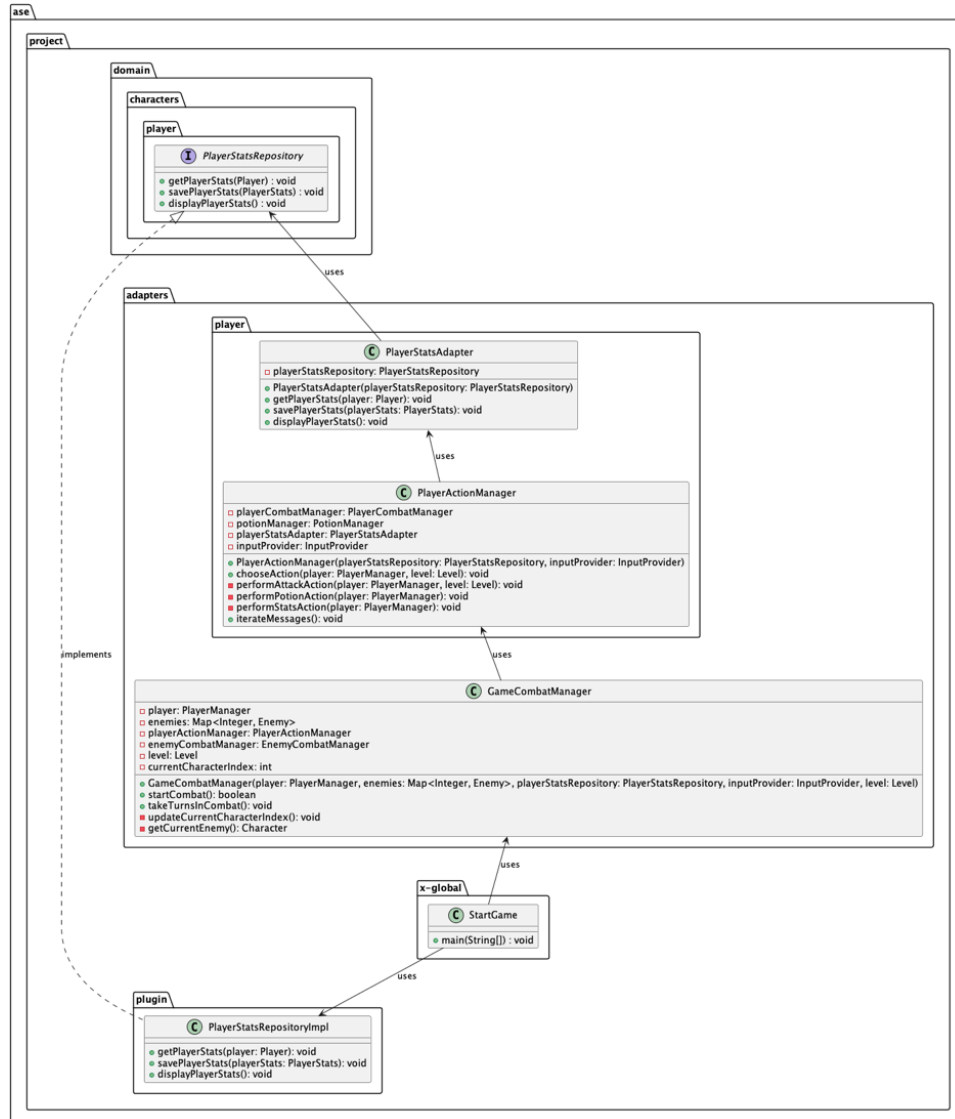


Abbildung 2. Dependency Rule - Positiv-Beispiel 2

In Abbildung 2 implementiert die Klasse **PlayerStatsRepositoryImpl** (Plugin) das Interface **PlayerStatsRepository** (Domain), was der Dependency Rule entspricht, da äußere Schichten von inneren Schichten abhängig sein sollen. Zudem wird **PlayerStatsRepositoryImpl** erst in der **StartGame**-Klasse genutzt, sodass höhere Ebenen nicht direkt von Details der niedrigeren Ebene abhängig sind. Damit hält das Design die Dependency Rule ein. Die Klasse **StartGame** befindet sich im Modul **x-global-ase-project**, das die Abhängigkeiten der vier Schichten enthält.

2.3. Schicht: Domain

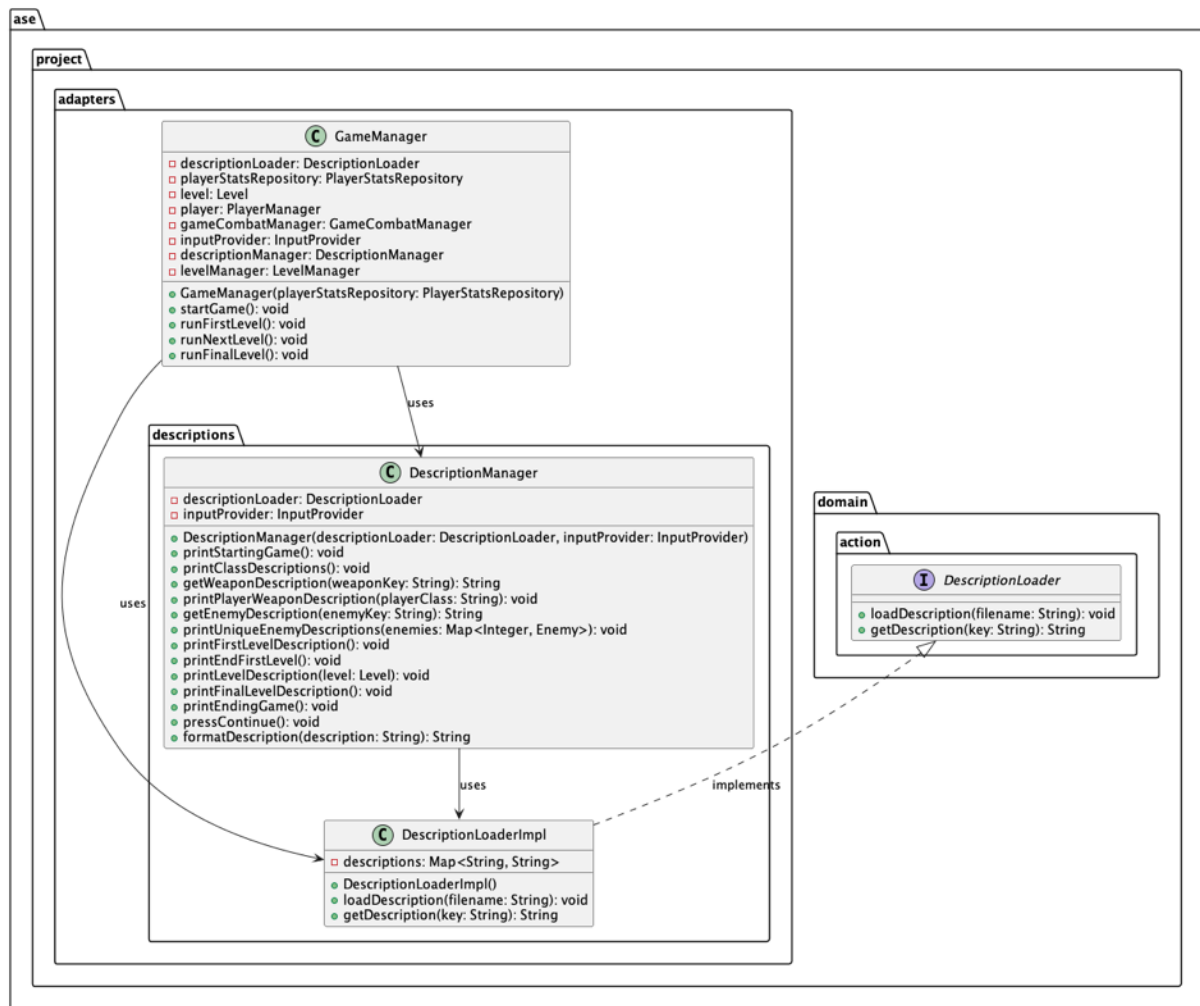


Abbildung 3. UML-Diagramm der DescriptionLoader in der Domain und ihre Implementierung in dem Adapter

Die Aufgabe der Domain-Schicht in der Clean-Architecture besteht darin, die zentrale Geschäftslogik und die Konzepte der Domäne zu repräsentieren. Sie enthält die Schnittstellen und abstrakten Klassen, die die Kernfunktionalität des Systems definieren, unabhängig von den externen Details. In diesem konkreten Fall wurde die Schnittstelle `DescriptionLoader` gewählt, die in der Adapter-Schicht implementiert wird, um sämtliche Beschreibungen zu laden und abzurufen. Die Domain-Schicht bietet somit eine isolierte und unabhängige Darstellung der Beschreibungslogik des Spiels und kann von anderen Schichten, wie der Adapter-Schicht, verwendet werden, um die konkrete Implementierung bereitzustellen und mit externen Ressourcen zu interagieren.

2.4. Schicht: Applikation

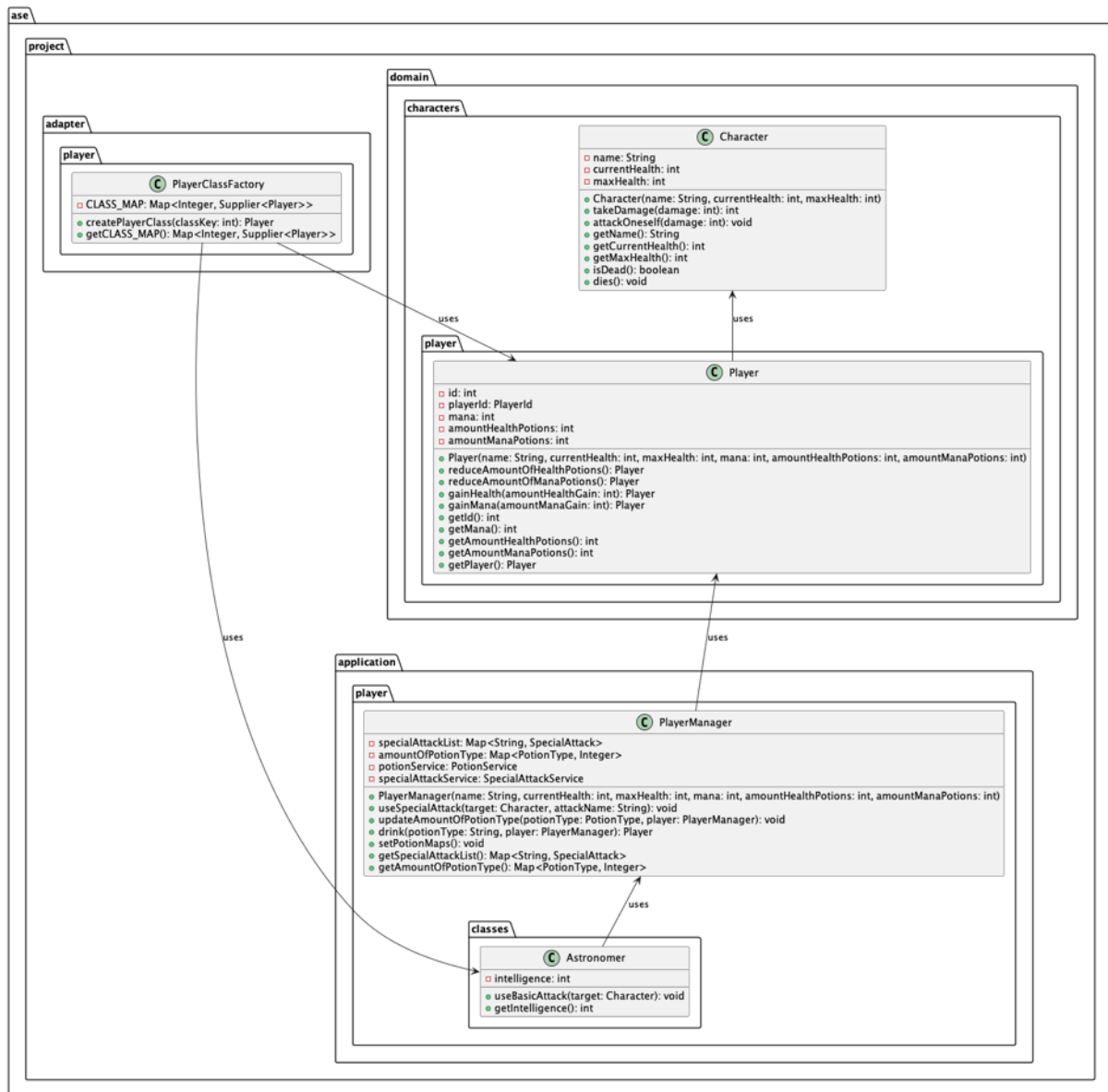


Abbildung 4. UML-Diagramm der Klasse Astronomer und ihr Zusammenspiel mit anderen Klassen aus anderen Schichten

Die Aufgabe der Applikationsschicht in der Clean-Architecture besteht darin, die Anwendungsfälle zu implementieren. Sie enthält die konkreten Implementierungen von Use Cases und Interaktionen, die die Funktionalität des Systems ermöglichen. In diesem Fall erbt die Klasse **Astronomer** (Applikation) von den Klassen **Character** und **Player** (Domain), um spezifische Eigenschaften und Fähigkeiten des **Astronomer** zu definieren. Anschließend wird **Astronomer** in der Adapter-Schicht verwendet, um dem Spieler die Auswahl einer Spielerklasse zu ermöglichen. Diese Einordnung ermöglicht eine klare Trennung zwischen den Konzepten in der Domain-Schicht und der konkreten Anwendungsumsetzung in der

Applikationsschicht. Damit werden die Wiederverwendbarkeit und Testbarkeit des Codes gefördert.

3. SOLID

3.1. SRP: Analyse Single-Responsibility-Principle

Positiv-Beispiel: SRP

Die Klasse `SpecialAttackService` ist ein Beispiel für die Erfüllung der SRP, da sie sich auf eine spezifische Aufgabe konzentriert: die Auswahl von speziellen Angriffen und die Behandlung von Ausnahmen. Diese Aufgabe ist gut abgegrenzt und hat eine klare Verantwortung. Dadurch wird der Code übersichtlicher und wartbarer, da die Funktionalität der speziellen Angriffe an einer Stelle zusammengefasst ist und von anderen Teilen des Systems unabhängig ist. Es wird vermieden, dass die Logik zur Auswahl von Angriffen und zur Behandlung von Ausnahmen in anderen Klassen dupliziert wird, was zu einem konsistenteren Code führt.

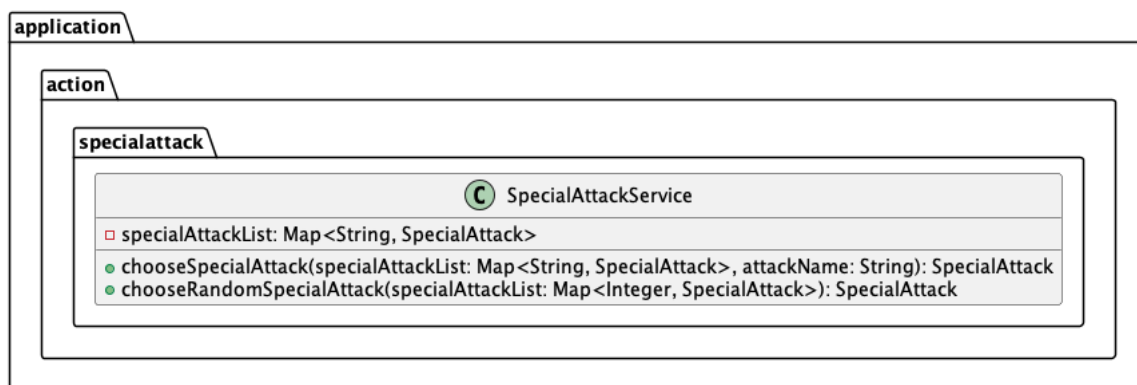


Abbildung 5. Die Klasse "SpecialAttackService" als Positiv-Beispiel für SRP

Negativ-Beispiel: SRP

Das zweite Beispiel mit der Klasse `GameCombatManager` erfüllt im Gegensatz dem SRP nicht. Die Klasse hat mehrere Verantwortlichkeiten, einschließlich der Durchführung des Kampfablaufs zwischen Spielern und Feinden, der Verwaltung des Spielerzugs und des Feindzugs sowie der Aktualisierung des aktuellen Charakterindex. Dies führt zu einer Verletzung des SRP, da die Klasse mehr als eine einzige Verantwortung hat.

Das aktuelle Design des `GameCombatManager` könnte problematisch sein, wenn die Klasse aufgrund zukünftiger Anforderungen weiterwachsen muss. Eine mögliche Lösung für das negative Beispiel wäre die Aufteilung der Verantwortlichkeiten in separate Klassen. Zum Beispiel könnten separate Klassen für den Spielerzug, den Feindzug und die Verwaltung des

Kampfablaufs erstellt werden. Dadurch wird die Klasse GameCombatManager auf ihre wichtigste Verantwortlichkeit reduziert, nämlich die Koordination und Steuerung des Kampfablaufs. Dies würde zu einer besseren Einhaltung des SRP führen.



Abbildung 6. Die Klasse GameCombatManager als Negativ-Beispiel für SRP

3.2. OCP: Analyse Open-Closed-Principle

Positiv-Beispiel: OCP

Die Klasse DescriptionLoaderImpl verstößt nicht gegen das Open-Closed-Prinzip (OCP). Sie implementiert das DescriptionLoader-Interface, das die Abstraktion für das Laden von Beschreibungen darstellt. Dadurch wird die Klasse offen für Erweiterungen, da neue Implementierungen des DescriptionLoader-Interfaces erstellt werden können, ohne den vorhandenen Code zu ändern. Das OCP wird erfüllt, da die DescriptionLoaderImpl-Klasse die vorgegebene Abstraktion verwendet und keine direkten Abhängigkeiten von konkreten Implementierungen hat. Dadurch kann die Klasse durch eine andere Implementierung des DescriptionLoader-Interfaces ersetzt werden, wenn sich die Anforderungen ändern oder eine alternative Implementierung benötigt wird.

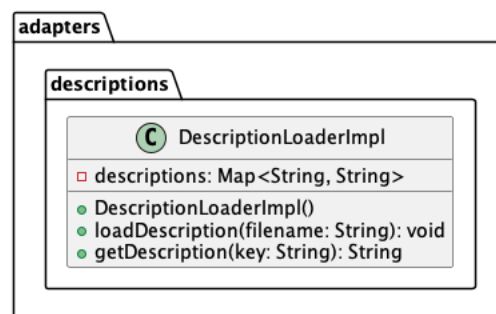


Abbildung 7. Die Klasse DescriptionLoaderImpl als Positiv-Beispiel für OCP

Negativ-Beispiel: OCP

Die Klasse EnemyManager erweitert direkt die Klasse Enemy und fügt Verhalten im Zusammenhang mit Spezialangriffen hinzu. Dies verletzt das OCP. Wenn wir einen neuen Typ

von Gegner einführen möchten, der keine Spezialangriffe verwendet, müssen wir die Klasse ändern. Somit ist sie nicht für Modifikationen geschlossen.

Ein besseres Vorgehen wäre es, das Verhalten der Spezialangriffe in eine separate Klassenhierarchie auszulagern. Dadurch können wir problemlos neue Arten von Gegnern mit unterschiedlichem Verhalten einführen, ohne den vorhandenen Code zu ändern. Dies fördert das OCP, indem Erweiterungen ermöglicht werden, ohne Modifikationen vornehmen zu müssen.

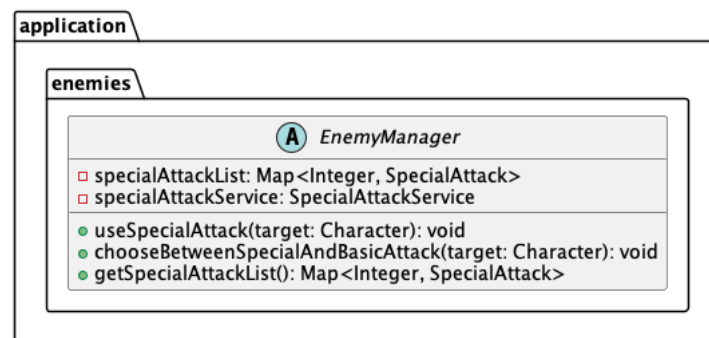


Abbildung 8. Die Klasse EnemyManager als Negativ-Beispiel für OCP

3.3. ISP: Analyse Interface-Segregation

Positiv-Beispiel: ISP

Abbildung 9 stellt ein positives Beispiel für das Interface-Segregation-Prinzip (ISP) dar, da das Interface PlayerStatsRepository spezifische Methoden bereitstellt, die sich auf die Verwaltung von Spielerstatistiken beziehen. Dadurch implementieren diese Methoden nur diejenigen Klassen oder Module, die tatsächlich Spielerstatistiken benötigen. Dadurch bleibt das Interface nur auf die relevanten Funktionen fokussiert.

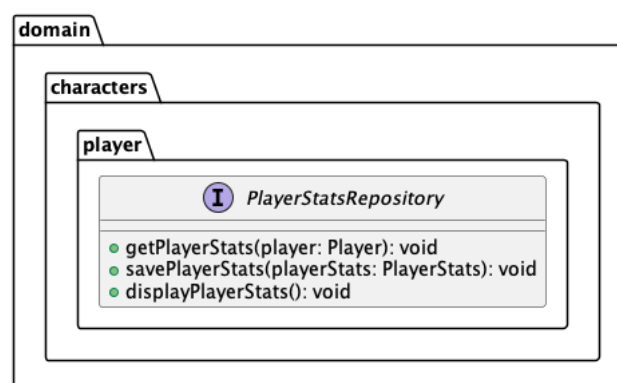


Abbildung 9. Das PlayerStatsrepository-Interface als Positiv-Beispiel für ISP

Negativ-Beispiel: ISP

In diesem Beispiel wird das Interface `SpecialAttack` sowohl von Gegnern als auch von Spielern implementiert, um einen speziellen Angriff auszuführen. Allerdings haben nur Spieler Mana, daher ist die Methode `getManaCost` für Gegner überflüssig. Dies führt dazu, dass das Interface nicht das Interface-Segregation-Prinzip (ISP) erfüllt, da Gegner eine Methode implementieren müssen, die für sie nicht relevant ist.

Um das ISP zu erfüllen, könnte das `SpecialAttack`-Interface in kleinere Teil-Interfaces aufgeteilt werden, die jeweils spezifische Funktionen repräsentieren. Zum Beispiel könnte man ein Interface `PlayerSpecialAttack` erstellen, das die speziellen Angriffe spezifisch für Spieler definiert, und ein separates Interface `EnemySpecialAttack` für spezielle Angriffe, die nur für Gegner relevant sind. Dadurch könnten Spieler und Gegner nur von denjenigen Interfaces abhängig sein, die für sie relevant sind, und das ISP würde erfüllt werden.

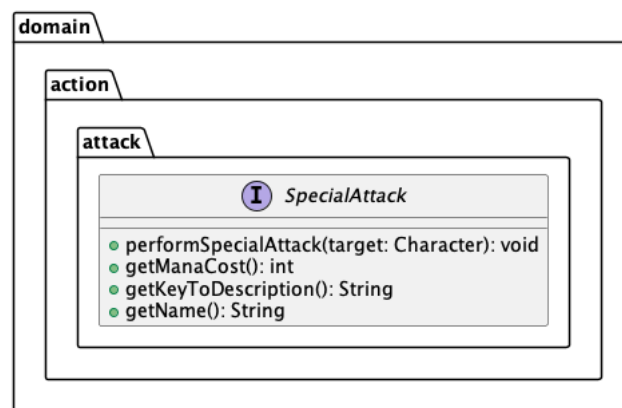


Abbildung 10. Das `SpecialAttack`-Interface als Negativ-Beispiel für ISP

4. Weitere Prinzipien

4.1. Analyse GRASP: Geringe Kopplung

Positiv-Beispiel: Geringe Kopplung

Die Klasse `LevelBuilder` hat eine niedrige Kopplung. Sie ist verantwortlich für den Aufbau eines Level-Objekts und arbeitet dabei mit einer Map von Feinden zusammen. Die Klasse hat nur wenige direkte Abhängigkeiten und nimmt ihre Daten hauptsächlich als Parameter entgegen. Dadurch ist sie flexibel und wiederverwendbar. Die niedrige Kopplung ermöglicht es auch, die `LevelBuilder`-Klasse unabhängig von anderen Klassen zu testen und zu ändern, ohne dass dies Auswirkungen auf den Rest der Anwendung hat.

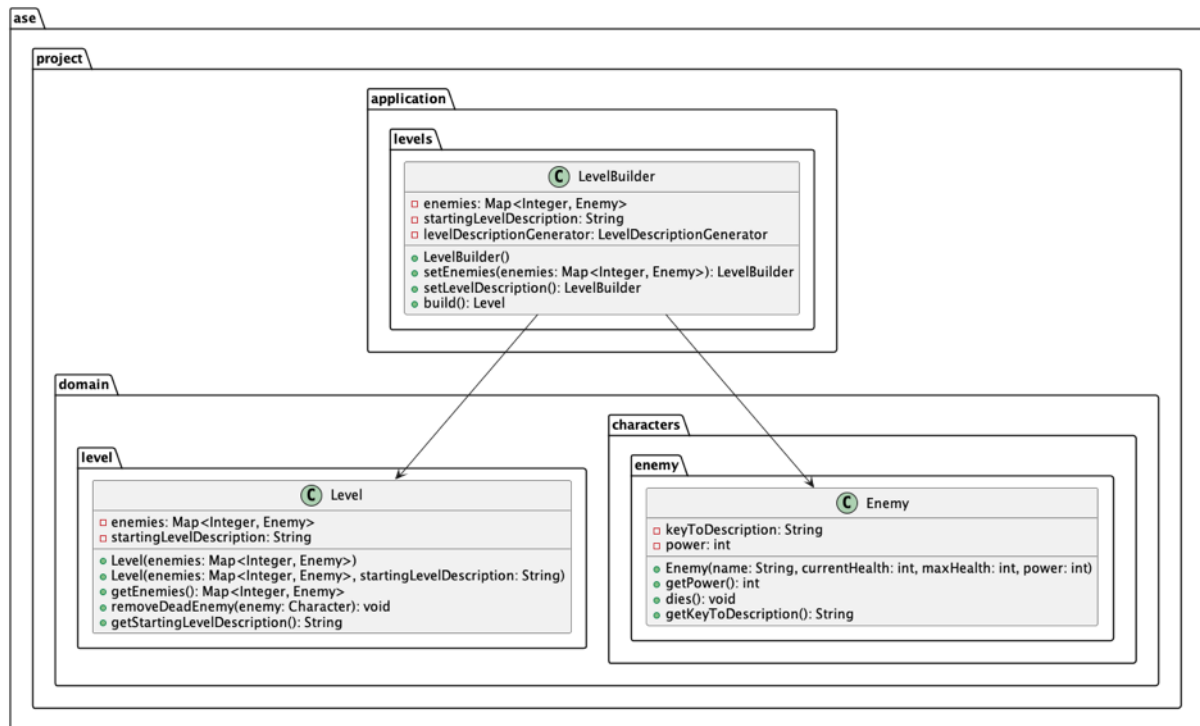


Abbildung 11. Die Klasse LevelBuilder als Positiv-Beispiel für eine relativ niedrige Kopplung

Negativ-Beispiel: Geringe Kopplung

Die GameManager-Klasse ist verantwortlich für die Steuerung des Spiels und interagiert mit vielen anderen Klassen wie PlayerManager, GameCombatManager, LevelManager, DescriptionManager usw.

Die hohe Kopplung ergibt sich aus den direkten Abhängigkeiten zu den genannten Klassen. Der GameManager kennt und verwendet diese Klassen direkt, was zu einer starken Kopplung führt. Dadurch entsteht eine starke Abhängigkeit und Änderungen in einer dieser Klassen können sich auf den GameManager auswirken.

Eine Möglichkeit, die Kopplung zu reduzieren, besteht darin, die Verantwortlichkeiten auf verschiedene spezifischere Klassen aufzuteilen und die Abhängigkeiten zwischen den Klassen zu verringern. Dies kann durch die Verwendung von Schnittstellen und Abstraktionen erreicht werden. Anstatt direkt auf konkrete Implementierungen wie PlayerManager, GameCombatManager usw. zuzugreifen, könnten abstrakte Schnittstellen verwendet werden, um die Interaktion zu kapseln. Dadurch wird die Kopplung gelockert und die Flexibilität und Erweiterbarkeit des Systems verbessert.

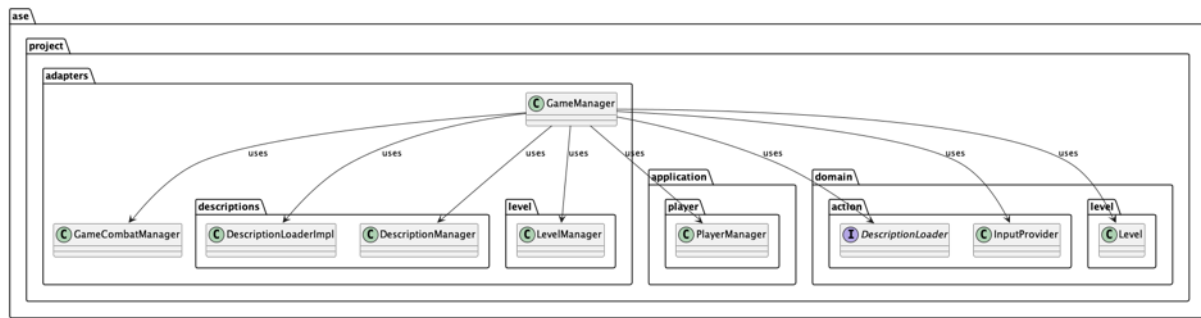


Abbildung 12. Die Klasse GameManager als Negativ-Beispiel mit einer sehr hohen Kopplung

4.2. Analyse GRASP: Hohe Kohäsion

Positiv-Beispiel: Hohe Kohäsion

Die Klasse DeathObserverManager ist für die Verwaltung des Todes von Charakteren zuständig. Sie enthält Methoden, um den Tod eines Charakters zu überprüfen und den zugehörigen DeathObserver zu benachrichtigen. Die Klasse hat hohe Kohäsion, da sie eine zusammenhängende Verantwortlichkeit hat, nämlich den Umgang mit dem Tod von Charakteren. Sie enthält spezifische Methoden, um den Tod zu überprüfen und den entsprechenden Observer zu benachrichtigen.

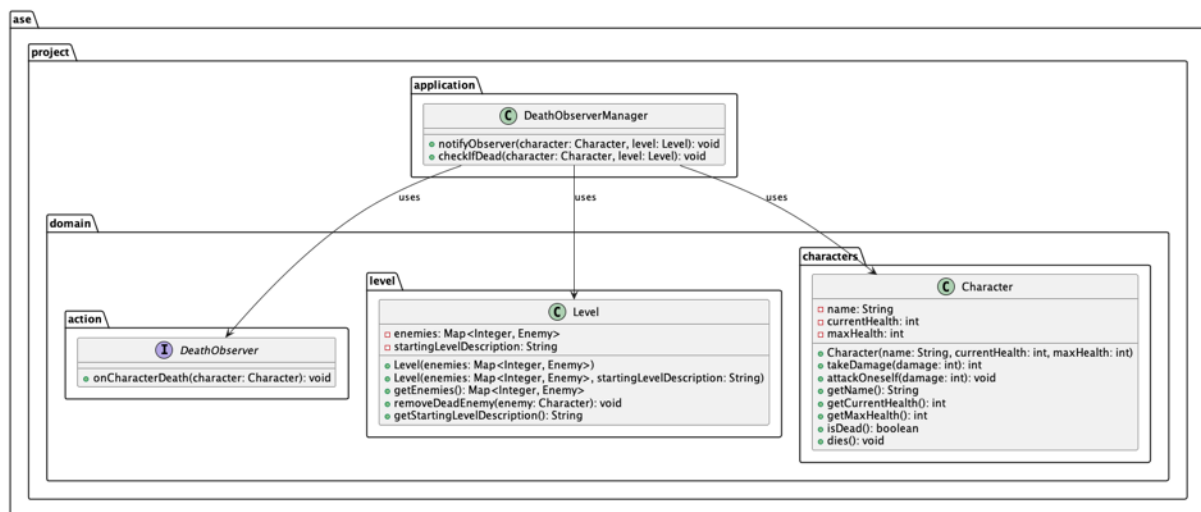


Abbildung 13. Die Klasse DeathObserverManager als Positiv-Beispiel für eine hohe Kohäsion

Negativ-Beispiel: Hohe Kohäsion

Die Klasse GameCombatManager hat eine moderate Kohäsion. Sie ist für das Management des Kampfes im Spiel verantwortlich und enthält Methoden zum Durchführen von Kampfzügen, Aktualisieren des aktuellen Charakterindex und Starten des Kampfes.

Allerdings hängt sie auch von anderen Klassen wie PlayerActionManager, EnemyCombatManager und DeathObserverManager ab, um ihre Funktionen auszuführen.

Dies deutet darauf hin, dass die Klasse für mehrere Aufgaben verantwortlich ist und Abhängigkeiten von verschiedenen Komponenten hat. Um die Kohäsion zu verbessern, könnte die Klasse so umstrukturiert werden, dass sie sich speziell auf die Verwaltung der Kampflogik konzentriert und andere Verantwortlichkeiten an separate Klassen delegiert. Dadurch können die Verantwortlichkeiten klarer verteilt und die Kohäsion der Klasse erhöht werden.



Abbildung 14. Die Klasse GameCombatManager als Negativ-Beispiel mit einer mittleren Kohäsion

4.3. DRY: Don't Repeat Yourself

Commits: [Commit von GitHub-Repo: UseSpecialAttack - Klasse](#)

[Commit von GitHub-Repo: PlayerManager - Klasse](#)

Dieser Commit bezieht sich auf das DRY-Prinzip (Don't Repeat Yourself), da die Methode useSpecialAttack in den Klassen Astronomer, Busker und Gladiator redundant implementiert war. Um diese Redundanz zu vermeiden, wurde die Methode in die separate Klasse ChooseSpecialAttack extrahiert und kann nun von allen betreffenden Klassen verwendet werden. Dazu wurde später auch eine weitere Klasse PlayerManager implementiert. Die Klassen Astronomer, Busker und Gladiator erben von der Klasse PlayerManager und können somit den Spezialangriff verwenden (Methode: useSpecialAttack). Die Code-Beispiele für die Beseitigung der Redundanz sind im Commit zu finden.

Die Verbesserung besteht darin, dass der Code für die Auswahl und Ausführung des Spezialangriffs nur an einer Stelle implementiert ist und von mehreren Klassen verwendet werden kann. Dadurch wird der Code lesbarer, kürzer und leichter zu warten, da Änderungen nur an einer Stelle vorgenommen werden müssen. Somit wird die Wartung des Codes erleichtert, da die Logik für die Spezialangriffe zentralisiert ist. Außerdem wird die Codequalität verbessert, da Redundanzen vermieden werden und der Code besser wiederverwendet werden kann. Dadurch wird die Effizienz des Entwicklungsprozesses gesteigert und die Wahrscheinlichkeit von Fehlern reduziert.

Vor dem Refactoring:

```
public class Astronomer extends Character {

    @Override
    public void useSpecialAttack(Enemy target, String attackName, int mana) {
        try {
            SpecialAttack specialAttack = specialAttackList.get(attackName);
            if (specialAttack != null) {
                specialAttack.useSpecialAttack(target, attackName, mana);
            } else {
                throw new InvalidAttackException("Invalid attack: " + attackName);
            }
        } catch (InvalidAttackException e) {
            System.out.println(e.getMessage());
        }
        UseSpecialAttack.useSpecialAttack(target, attackName, mana, specialAttackList);
    }
}
```

Nach dem Refactoring 1:

```
public class UseSpecialAttack {

    public static void useSpecialAttack(Enemy target, String attackName, int mana, Map<String,
    SpecialAttack> specialAttackList) {
        try {
            SpecialAttack specialAttack = specialAttackList.get(attackName);
            if (specialAttack != null) {
                specialAttack.useSpecialAttack(target, attackName, mana);
            } else {
                throw new InvalidAttackException("Invalid attack: " + attackName);
            }
        } catch (InvalidAttackException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

```
public class Astronomer extends Character {

    @Override
    public void useSpecialAttack(Enemy target, String attackName, int mana) {
        UseSpecialAttack.useSpecialAttack(target, attackName, mana, specialAttackList);
    }
}
```

Nach dem Refactoring 2:

```
public abstract class PlayerClass extends Player {
    public void useSpecialAttack(Character target, String attackName) throws
    InvalidAttackException {
        SpecialAttack specialAttack =
        ChooseSpecialAttack.chooseSpecialAttack(specialAttackList, attackName);
        try {
            ManaService.checkMana(mana, specialAttack.getManaCost());
            specialAttack.performSpecialAttack(target, attackName);
            mana = ManaService.useMana(mana, specialAttack.getManaCost());
        } catch (InvalidManaException manaException) {
            System.out.println(manaException.getMessage());
        }
    }
}
```


5. Unit Tests

5.1. 10 Unit Tests

Unit Test	Beschreibung
EnemyManagerTest # chooseBasicAttack	In diesem Testfall wird überprüft, ob der Angriff (useBasicAttack) ausgeführt wird, wenn der Würfelwurf größer als 3 ist. Dafür wird ein Objekt PhyrexianMite (spy) erstellt und die Methode chooseBetweenSpecialAndBasicAttack mit einem simulierten Würfelwurf von 4 aufgerufen. Anschließend wird überprüft, ob die Methode useBasicAttack auf dem gespionierten Objekt aufgerufen wurde.
PlayerManagerTest # updateAmountOfPotions	In diesem Testfall wird überprüft, ob die Methode updateAmountOfPotionType korrekt funktioniert, indem sie die Anzahl der verfügbaren Gesundheitstränke (/health potions) aktualisiert. Dazu wird ein Astronomer-Objekt mit bestimmten Attributen erstellt und die Methode setPotionMaps aufgerufen. Anschließend wird die Methode updateAmountOfPotionType mit dem Potion-Typ HEALTH aufgerufen, um die Anzahl der Gesundheitstränke zu aktualisieren.
PotionTypeServiceTest # testChoosePotionType	In diesem Test wird überprüft, ob die Methode choosePotionType des PotionTypeService korrekt funktioniert, indem sie basierend auf dem übergebenen Potion-Typ eine entsprechende Potion zurückgibt. Es wird überprüft, ob die zurückgegebene Potion eine Instanz der Klasse HealthPotion ist.
PotionServiceTest # testUsePotion_Mana	Hier wird überprüft, ob die Methode drink der Klasse PlayerManager korrekt funktioniert, wenn eine Mana Potion verwendet wird. Es wird überprüft, ob nach dem Verbrauch der Potion der Spieler korrekt aktualisiert wird, indem die Anzahl der Mana Potions reduziert wird und der Mana-Wert entsprechend erhöht wird.
ManaServiceTest # checkMana_NotEnough Mana	In diesem Test wird überprüft, ob die Methode checkMana der Klasse ManaService korrekt eine InsufficientManaException wirft, wenn der vorhandene Mana-Wert nicht ausreicht, um die Kosten für eine Aktion zu decken.
DrinkPotionServiceTest # testUsePotion	Dabei wird überprüft, ob die Methode drinkPotion der Klasse DrinkPotionService korrekt die drink-Methode des übergebenen Potion-Objekts aufruft und das aktualisierte Player-Objekt zurückgibt. Es wird also getestet, ob die Methode das korrekte Ergebnis liefert, wenn eine Potion verwendet wird.
PotionManagerTest # testChoosePotionType	In diesem Testfall wird überprüft, ob die Methode choosePotionType des PotionManager korrekt die Auswahl einer Potion durch den Spieler ermöglicht.
PlayerCombatManagerTest # testChooseTarget	In diesem Test wird überprüft, ob die Methode chooseTarget des PlayerCombatManager korrekt die Auswahl eines Gegners (target) ermöglicht. Es wird ein Level erstellt, das eine Map von Feinden enthält, und der Spieler wird aufgefordert, einen Gegner auszuwählen. Dabei wird überprüft, ob der ausgewählte Gegner korrekt zurückgegeben wird.
PlayerClassManagerTest testChooseClass	In diesem Test wird überprüft, ob die Methode chooseClass des PlayerClassManager korrekt die Auswahl einer Spielerklasse ermöglicht. Es wird ein InputProvider vorbereitet, der eine bestimmte Eingabe simuliert, und dann wird die Methode chooseClass aufgerufen. Es wird überprüft, ob der zurückgegebene Spieler eine Instanz der Klasse "Astronomer" ist.

LevelBuilderTest # testRemoveDeadEnemy	Hierbei wird überprüft, ob die Methode removeDeadEnemy des Level-Objekts korrekt tote Feinde aus der Feindesliste entfernt. Es wird eine Testumgebung erstellt, die aus einer Map mit zwei Feinden besteht, von denen einer bereits tot ist. Die Methode removeDeadEnemy wird aufgerufen und es wird überprüft, ob der tote Feind erfolgreich aus der Liste entfernt wurde und ob die Anzahl der Feinde korrekt aktualisiert wurde.
---	---

5.2. ATRIP: Automatic

Automatic (von ATRIP) wurde durch die Integration von Maven realisiert. Durch das Ausführen des Build-Prozesses werden automatisch alle Tests im Projekt durchgeführt. Die Ergebnisse der Tests werden erfasst und als „Tests run“, „Failures“, „Errors“, „Skipped“ ausgegeben. Dabei werden alle Tests aller Module im Projekt durchlaufen und ihre Ergebnisse ausgegeben werden. Wenn einer der Tests fehlschlägt, wird der Build-Prozess abgebrochen und eine entsprechende Fehlermeldung wird angezeigt. Dadurch wird sichergestellt, dass das Projekt nur dann erfolgreich gebaut wird, wenn alle Tests erfolgreich bestanden wurden.

```
[INFO] -----
[INFO]  T E S T S
[INFO] -----
[INFO] Running ase.project.application.action.ManaServiceTest
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.013 s - in ase.project.application.action.ManaServiceTest
[INFO] Running ase.project.application.action.PotionAmountServiceTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0 s - in ase.project.application.action.PotionAmountServiceTest
[INFO] Running ase.project.application.action.PotionTypeServiceTest
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0 s - in ase.project.application.action.PotionTypeServiceTest
[INFO] Running ase.project.application.action.DrinkPotionServiceTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.527 s - in ase.project.application.action.DrinkPotionServiceTest
[INFO] Running ase.project.application.action.PotionServiceTest
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.026 s - in ase.project.application.action.PotionServiceTest
[INFO] Running ase.project.application.item.potion.HealthPotionTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0 s - in ase.project.application.item.potion.HealthPotionTest
```

Abbildung 15. Durchlauf der Tests beim Build-Prozess von Maven

5.3. ATRIP: Thorough

Positiv-Beispiel:

Die Testklasse GladiatorTest kann als Thorough betrachtet werden, da sie verschiedene Aspekte der Funktionalität der Klasse Gladiator vollständig testet.

Die Methode testUseBasicAttack testet, ob der grundlegende Angriff des Gladiators den richtigen Schaden verursacht und ob das Ziel den Schaden erleidet. Die Methode testSpecialAttackListInitialization überprüft, ob die Spezialangriffe des Gladiators korrekt initialisiert werden und ob die Spezialangriffsliste die erwartete Größe und Schlüssel enthält.

Die Methoden testUseSpecialAttack_performsCorrectAttack_GladiatorGambit und

testUseSpecialAttack_performsCorrectAttack_SpinToWin testen, ob die speziellen Angriffe des Gladiators die erwarteten Schäden verursachen und ob das Ziel den Schaden erleidet. Die Methoden testUseSpecialAttack_reducesManaByCorrectAmount_GladiatorGambit und testUseSpecialAttack_reducesManaByCorrectAmount_SpinToWin überprüfen, ob der Einsatz eines speziellen Angriffs die Mana-Kosten richtig reduziert. Die Methode testUseSpecialAttack_throwsInvalidAttackException testet, ob eine ungültige Angriffsbezeichnung eine InvalidAttackException auslöst.

Diese Tests decken das Verhalten der Klasse Gladiator ab und stellen sicher, dass die Funktionalität korrekt implementiert ist. Durch das Testen der grundlegenden Angriffe, der speziellen Angriffe, der Mana-Reduzierung und des Umgangs mit ungültigen Angriffen wird eine umfassende Abdeckung erreicht. Daher kann die Testklasse als Thorough betrachtet werden.

```
@ExtendWith(MockitoExtension.class)
class GladiatorTest {
    private Gladiator player;
    private Enemy target;

    @Mock
    private Enemy mockTarget;

    @BeforeEach
    public void setUp() {
        player = new Gladiator("Gladiator Test", 60, 60, 40, 3, 2, 10);
        target = new PhyrexianMite("Test Mite", 25, 30, 10);
    }

    @Test
    void testUseBasicAttack() {
        final int diceRoll = 4;

        try (MockedStatic<DiceRoller> diceRollerMock =
            mockStatic(DiceRoller.class)) {
            diceRollerMock.when(() ->
                DiceRoller.rollDice(6)).thenReturn(diceRoll);
            int expectedDamage = diceRoll + player.getEndurance();

            player.useBasicAttack(target);
            player.useBasicAttack(mockTarget);

            assertEquals(11, target.getCurrentHealth());
            verify(mockTarget).takeDamage(expectedDamage);
        }
    }

    @Test
    void testSpecialAttackListInitialization() {
        Map<String, SpecialAttack> specialAttackMap =
            player.getSpecialAttackList();

        assertEquals(2, specialAttackMap.size());
        assertTrue(specialAttackMap.containsKey("Gladiator's Gambit"));
        assertTrue(specialAttackMap.containsKey("Spin To Win"));
    }
}
```

```

@Test
void testUseSpecialAttack_performsCorrectAttack_GladiatorGambit()
    throws InvalidAttackException, InsufficientManaException {
    String attackName = "Gladiator's Gambit";
    int diceRoll = 20;

    try (MockedStatic<DiceRoller> diceRollerMock =
        mockStatic(DiceRoller.class)) {
        diceRollerMock.when(() ->
            DiceRoller.rollDice(20)).thenReturn(diceRoll);

        int expectedHealth = target.getCurrentHealth() - diceRoll;

        player.useSpecialAttack(target, attackName);
        player.useSpecialAttack(mockTarget, attackName);

        assertEquals(expectedHealth, target.getCurrentHealth());
        verify(mockTarget).takeDamage(diceRoll);
    }
}

@Test
void testUseSpecialAttack_performsCorrectAttack_SpinToWin()
    throws InvalidAttackException, InsufficientManaException {
    String attackName = "Spin To Win";
    int diceRoll = 20;

    try (MockedStatic<DiceRoller> diceRollerMock =
        mockStatic(DiceRoller.class)) {
        diceRollerMock.when(() ->
            DiceRoller.rollDice(20)).thenReturn(diceRoll);

        int expectedHealth = target.getCurrentHealth() - diceRoll;

        player.useSpecialAttack(target, attackName);
        player.useSpecialAttack(mockTarget, attackName);

        assertEquals(expectedHealth, target.getCurrentHealth());
        verify(mockTarget).takeDamage(diceRoll);
    }
}

@Test
void testUseSpecialAttack_reducesManaByCorrectAmount_GladiatorGambit()
    throws InvalidAttackException, InsufficientManaException {
    String attackName = "Gladiator's Gambit";

    player.useSpecialAttack(mockTarget, attackName);

    assertEquals(35, player.getMana());
}

@Test
void testUseSpecialAttack_reducesManaByCorrectAmount_SpinToWin()
    throws InvalidAttackException, InsufficientManaException {
    String attackName = "Spin To Win";

    player.useSpecialAttack(mockTarget, attackName);

    assertEquals(30, player.getMana());
}

@Test
void testUseSpecialAttack_throwsInvalidAttackException() {
    String attackName = "Wrong attack name";

    assertThrows(InvalidAttackException.class, () -> {
        player.useSpecialAttack(mockTarget, attackName);
    });
}

```

```
}  
}
```

Negativ-Beispiel

Die Testklasse `PlayerClassManagerTest` kann nicht als Thorough betrachtet werden, da sie nicht alle Stellen der Klasse `PlayerClassManager` testet. Die Testklasse testet nur, ob das Ergebnis der `chooseClass()`-Methode eine Instanz der Klasse `Astronomer` ist. Es fehlen jedoch Tests für andere mögliche Klassen, die von der `chooseClass`-Methode erstellt werden können. Um die Testklasse als Thorough zu betrachten, müssten zusätzliche Tests hinzugefügt werden, um sicherzustellen, dass die `chooseClass`-Methode korrekt funktioniert und verschiedene Klassenobjekte zurückgibt, basierend auf den Eingaben des Benutzers.

```
class PlayerClassManagerTest {  
  
    private PlayerClassManager playerClassManager;  
  
    @BeforeEach  
    void setUp() {  
        InputProvider inputProvider = mock(InputProvider.class);  
        when(inputProvider.readInt()).thenReturn(1);  
        playerClassManager = new PlayerClassManager(inputProvider);  
    }  
    @Test  
    void testChooseClass() {  
        Player player = playerClassManager.chooseClass();  
        assertTrue(player instanceof Astronomer);  
    }  
}
```

5.4. ATRIP: Professional

Positiv-Beispiel

Die Testklasse `PotionServiceTest` kann als „professionell“ betrachtet werden. Sie verwendet Testframeworks wie Mockito und enthält aussagekräftige Assertions, um sicherzustellen, dass die erwarteten Ergebnisse erzielt werden. Die Verwendung von Mocks und das Definieren von Testdaten sind gängige Praktiken in Testklassen, um die Isolation von zu testenden Komponenten zu gewährleisten.

Die Testklasse überprüft das Verhalten der `PotionService`-Klasse und stellt sicher, dass sie korrekt funktioniert. Dabei könnten Fehler in den Testklassen zu falschen Schlussfolgerungen führen und die Zuverlässigkeit des getesteten Codes beeinträchtigen.

Des Weiteren testet die Testklassenspezifische Funktionen der `PotionService`-Klasse, nämlich die Verwendung von Tränken für Mana und Gesundheit (Potions). Somit werden keine redundanten oder überflüssigen Tests durchgeführt.

Durch das Lesen der Testklasse kann man verstehen, wie die PotionService-Klasse verwendet wird und welche Ergebnisse erwartet werden.

```
@ExtendWith(MockitoExtension.class)
class PotionServiceTest {
    @Mock
    private PlayerManager mockPlayer;

    @BeforeEach
    public void setMockPlayer() {
        mockPlayer = new Astronomer("AstroTest", 60, 60, 40, 3, 2, 10);
    }

    @Test
    void testUsePotion_Mana() {
        PotionType potionType = PotionType.MANA;

        Player result = mockPlayer.drink(potionType.toString(), mockPlayer);

        assertEquals(mockPlayer, result);
        assertEquals(1, result.getAmountManaPotions());
        assertEquals(50, result.getMana());
    }

    @Test
    void testUsePotions_Health() {
        PotionType potionType = PotionType.HEALTH;

        Player result = mockPlayer.drink(potionType.toString(), mockPlayer);

        assertEquals(mockPlayer, result);
        assertEquals(2, result.getAmountHealthPotions());
        assertEquals(80, result.getCurrentHealth());
    }
}
```

Negativ-Beispiel

Die Testklasse PotionTypeServiceTest kann als nicht „professionell“ betrachtet werden. Die Tests unterliegen nicht den gleichen Qualitätsstandards wie der Produktivcode. Die Testklasse enthält keine aussagekräftigen Assertions, die die korrekte Funktionalität der PotionTypeService-Klasse überprüfen. Stattdessen wird nur überprüft, ob die zurückgegebenen Instanzen bestimmte Klassen sind.

Die Testklasse erstellt eine neue Instanz der PotionTypeService-Klasse, anstatt eine vorhandene Instanz zu verwenden oder Mock-Objekte zu verwenden. Des Weiteren enthält die Testklasse redundante Tests, die die gleiche Überprüfung durchführen, indem sie nur unterschiedliche Eingabewerte verwenden (Bsp. testHealthStringToPotion).

```
class PotionTypeServiceTest {

    private PotionTypeService potionTypeService = new PotionTypeService();

    @Test
    void testChoosePotionType() {
        Potion potion =
            potionTypeService.choosePotionType(PotionType.HEALTH);
    }
}
```

```

        assertTrue(potion instanceof HealthPotion);
    }

    @Test
    void testHealthStringToPotionType() {
        String typeString = "HEALTH";
        PotionType type = PotionType.valueOf(typeString.toUpperCase());
        Potion expectedPotion = potionTypeService.choosePotionType(type);

        Potion actualPotion = new HealthPotion();

        assertEquals(expectedPotion.getClass(), actualPotion.getClass());
    }

    @Test
    void testManaStringToPotionType() {
        String typeString = "mana";
        PotionType type = PotionType.valueOf(typeString.toUpperCase());
        Potion expectedPotion = potionTypeService.choosePotionType(type);

        Potion actualPotion = new ManaPotion();

        assertEquals(expectedPotion.getClass(), actualPotion.getClass());
    }
}

```

5.5. Code Coverage

Um die Codeabdeckung (Code Coverage) im Projekt zu analysieren, betrachtet man die Prozentwerte für Class, Method und Line. Hier sind die Werte für das gesamte Projekt sowie für jedes Modul separat dargestellt (Abbildung 16).

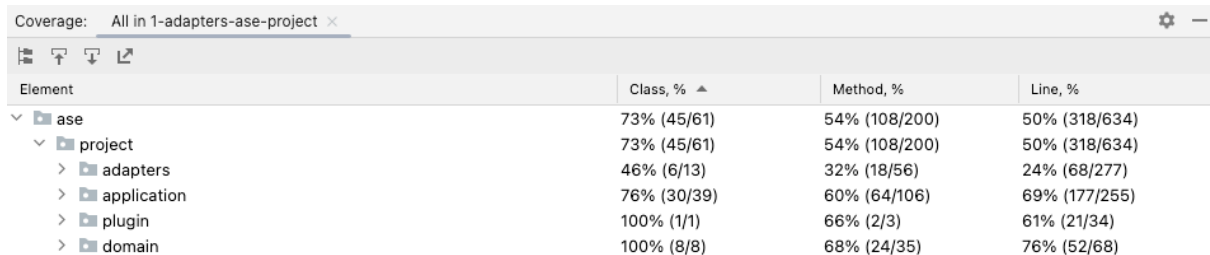
Die Codeabdeckung gibt an, wie viel vom Quellcode durch die Tests abgedeckt wird. Eine höhere Codeabdeckung bedeutet, dass ein größerer Prozentsatz des Codes während der Tests ausgeführt wurde.

Das Gesamtprojekt weist eine Class Coverage von 73%. Das bedeutet, dass 73% der Klassen im Code durch Tests abgedeckt wurden. Somit wurde die Mehrheit der Klassen im Projekt getestet, was ein gewisses Maß an Vertrauen in die Funktionalität bietet.

Eine Method Coverage von 54% im gesamten Projekt bedeutet, dass 54% der Methoden im Code durch Tests abgedeckt wurden. Dies zeigt, dass einige Methoden nicht ausreichend getestet wurden und möglicherweise Lücken in der Testabdeckung aufweisen. Eine niedrigere Methodenabdeckung kann darauf hinweisen, dass einige Methoden nicht oder nur teilweise getestet wurden, was das Risiko von Fehlern erhöhen könnte.

Eine Line Coverage von 50% im gesamten Projekt bedeutet, dass 50% der Codezeilen durch Tests ausgeführt wurden. Dies zeigt, dass die Tests nicht alle Zeilen des Codes erreichen und bestimmte Szenarien möglicherweise nicht ausreichend abgedeckt sind. Eine niedrigere Zeilenabdeckung deutet auf nicht getesteten Code hin, der potenzielle Fehler verbergen könnte.

Insgesamt ist die Codeabdeckung im Projekt nicht optimal, insbesondere in Bezug auf Methoden- und Zeilenabdeckung. Eine höhere Codeabdeckung bietet eine größere Sicherheit, dass der Code korrekt funktioniert und mögliche Fehler abdeckt. Es ist notwendig, die Testabdeckung zu verbessern, indem fehlende Tests für Methoden und Codezeilen hinzugefügt werden, um die Zuverlässigkeit und Stabilität des Systems zu erhöhen.



Element	Class, %	Method, %	Line, %
ase	73% (45/61)	54% (108/200)	50% (318/634)
project	73% (45/61)	54% (108/200)	50% (318/634)
adapters	46% (6/13)	32% (18/56)	24% (68/277)
application	76% (30/39)	60% (64/106)	69% (177/255)
plugin	100% (1/1)	66% (2/3)	61% (21/34)
domain	100% (8/8)	68% (24/35)	76% (52/68)

Abbildung 16. Code Coverage des Projekts und einzelner Module

5.6. Fakes and Mocks

Mock: DiceRoller

Die Klasse DiceRoller ist dafür verantwortlich, Zufallszahlen zu generieren, die einem Würfelwurf entsprechen. Sie enthält eine Methode namens rollDice, die einen Parameter sides (Anzahl der Seiten des Würfels) akzeptiert und eine zufällige Zahl zwischen 1 und der Anzahl der Seiten zurückgibt. Die Klasse DiceRoller wurde in mehreren Tests als Mock verwendet, da sie einen wichtigen Teil der Anwendung repräsentiert. Sie wird verwendet, um den Schaden (damage) zu ermitteln und ermöglicht es den Feinden, ihre Angriffe mithilfe der rollDice-Methode auszuwählen.

In den Tests, in denen ein Mock der Klasse DiceRoller verwendet wird, wird das Verhalten der rollDice-Methode simuliert, um vorhersehbare Ergebnisse zu erzielen. Dies ermöglicht es, spezifische Testfälle zu erstellen und das Verhalten des Codes unter verschiedenen Bedingungen zu überprüfen. Außerdem wird der Code von externen Abhängigkeiten isoliert, wie dem tatsächlichen Zufallsgenerator. Dies gewährleistet, dass die Tests deterministisch sind und unabhängig von äußeren Einflüssen immer das gleiche Verhalten zeigen. Dadurch können bestimmte Szenarien gezielt getestet werden, z. B. den Schaden eines Angriffs, der auf dem Würfelwurf basiert.

Somit ermöglicht der Einsatz von Mocks der Klasse DiceRoller, spezifische Szenarien zu testen und das Verhalten des Codes genau zu überprüfen, ohne von echtem Zufall abhängig zu sein. Es erleichtert die Erstellung von reproduzierbaren Tests.

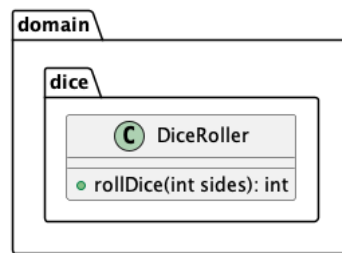


Abbildung 17. UML-Diagramm der Klasse DiceRoller

Mock: Enemy

Die Klasse Enemy wurde auch mehrmals gemockt. Dabei wird das Verhalten des Feindes während der Auswahl des Ziels (Target) zu simulieren. Der Player-Character kann in den Tests eine spezifische Instanz des Feindes als Ziel auswählen und überprüfen, ob die richtigen Aktionen auf das Ziel angewendet werden kann, indem ein Mock (mockTarget) verwendet wird.

Durch den Mock der Enemy-Klasse können verschiedene Szenarien und Verhaltensweisen des Feindes getestet werden, ohne auf tatsächliche Instanzen der Feind-Klasse angewiesen zu sein. Dadurch wird die Testbarkeit verbessert und potenzielle Abhängigkeiten oder Komplexitäten bei der Erzeugung und Verwaltung realer Feindobjekte vermieden. Der Mock ermöglicht es, den Fokus auf die Testlogik zu legen und die Interaktionen zwischen dem Player-Character und dem Feind gezielt zu überprüfen.

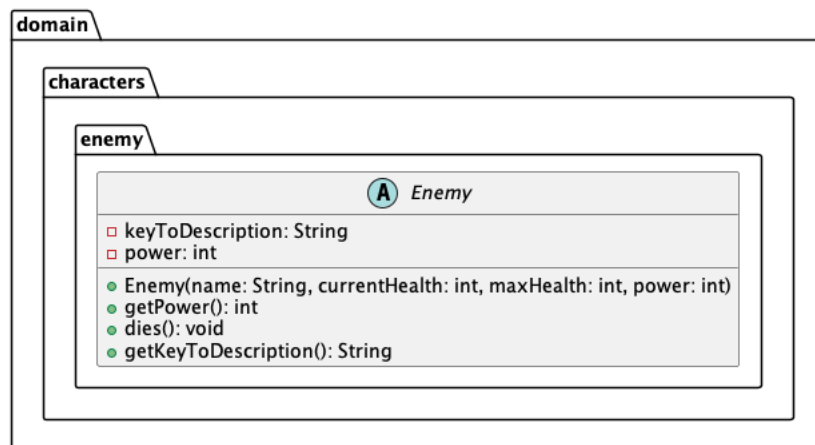


Abbildung 18. UML-Diagramm der Klasse Enemy

6. Domain-Driven Design (DDD)

6.1. Ubiquitous Language

Bezeichnung	Bedeutung	Begründung
Character	Spielfigur im Spiel	Die Bezeichnung "Character" wird verwendet, um eine Spielfigur zu beschreiben, unabhängig von ihrer spezifischen Rolle oder Klasse.
Health	Der aktuelle Gesundheitszustand einer Spielfigur	Der Begriff "Health" ist eine gängige Bezeichnung für den Gesundheitszustand einer Spielfigur in vielen Spielen. Er beschreibt die Menge an Lebenspunkten oder Energie, die eine Figur noch besitzt.
Bezeichnung	Bedeutung	Begründung
Enemy	Eine feindliche Spielfigur	Der Begriff "Enemy" wird verwendet, um feindliche Wesen oder Spielfiguren zu bezeichnen, gegen die der Spieler kämpfen muss. Es wird verwendet, um die feindlichen Charaktere eindeutig von den verbündeten oder neutralen Charakteren zu unterscheiden.
Level	Eine Stufe im Spiel, der bestimmte Herausforderungen oder Ziele bietet	Der Begriff "Level" ist ein gebräuchlicher Begriff in der Gaming-Branche und wird verwendet, um den Fortschritt des Spielers durch das Spiel zu kennzeichnen

6.2. Entities

Die Player-Klasse kann als Entity (DDD) betrachtet werden. Die Klasse repräsentiert den Spieler im Spiel. Sie erbt von der abstrakten Character-Klasse und erweitert sie um zusätzliche Eigenschaften und Verhaltensweisen, die spezifisch für einen Spieler sind. Die Player-Entität enthält Attribute wie die Spieler-ID, den aktuellen Mana-Wert, die Anzahl der verbleibenden Health-Potions und Mana-Potions usw. Sie enthält auch Methoden, um den Zustand des Spielers zu ändern, wie das Verringern der Anzahl der Tränke, das Erhöhen von Gesundheit oder Mana usw.

Die Player-Klasse besitzt eine eindeutige Identität (durch die Spieler-ID), über die gesamte Lebensdauer des Spielers bestehen bleibt. Dadurch können Spielerobjekte eindeutig identifiziert und referenziert werden, was insbesondere bei der Persistenz, dem Datenzugriff und der Interaktion mit anderen Domänenobjekten von Vorteil ist. Sie ist durch die oben

genannten Attribute und Verhalten definiert, die spezifisch für einen Spieler sind. Das Konzept des Spielers ist auch in der Ubiquitous Language des Spiels vorhanden und hat eine klar definierte Rolle und Verantwortlichkeiten im Spiel. Daher ist die Player-Klasse ein gutes Beispiel für eine Entity im Domain-Driven Design.

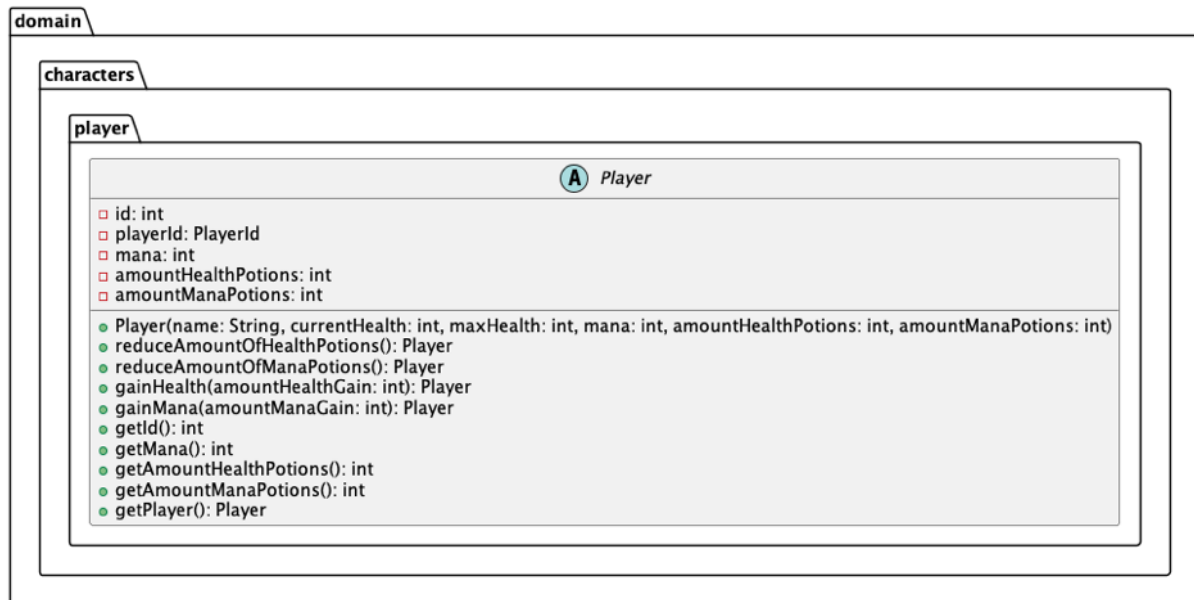


Abbildung 19. Die Klasse Player als Beispiel für eine Entity (DDD)

6.3. Value Objects

Das Value Object PlayerId wird verwendet, um eine eindeutige Spieler-ID zu repräsentieren. Es wird als Value Object implementiert, da es unveränderlich ist und seine Identität ausschließlich auf dem enthaltenen Wert basiert. Durch die Verwendung eines Value Objects wird eine klare Trennung zwischen Identität und Zustand erreicht.

Die PlayerId ermöglicht es die einfache Identitätsprüfung, da die Gleichheit auf dem Wert der ID basiert. Die Klasse unterstützt die Unveränderlichkeit, da die ID nicht geändert werden kann, sobald sie erstellt wurde. Dadurch wird die Konsistenz und Integrität des Systems gewährleistet. Insgesamt bietet die Verwendung von Value Objects wie PlayerId eine strukturierte und konsistente Möglichkeit, Identitäten in der Domäne zu repräsentieren und unterstützt die Wartbarkeit und Erweiterbarkeit des Codes.

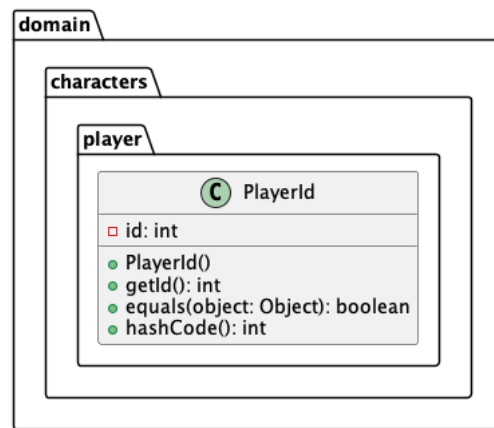


Abbildung 20. Die Klasse *PlayerId* als Value Object (DDD)

6.4. Repositories

Die Klasse `PlayerStatsRepository` dient als Schnittstelle für den Zugriff auf Spielerstatistiken. Die Methode `getPlayerStats` erstellt eine `PlayerStats`-Instanz und speichert sie, während die Methode `savePlayerStats` die Spielerstatistiken in eine Datei schreibt. Die Methode `displayPlayerStats` liest die gespeicherten Spielerstatistiken aus der Datei und gibt sie aus. Im Fall der Anwendung stellt das Ordnersystem die Persistenzschicht dar.

Ein Repository in diesem Kontext ermöglicht eine klare Trennung zwischen der Geschäftslogik und der Persistenzschicht. Es abstrahiert den Datenzugriff und ermöglicht eine einheitliche und konsistente Schnittstelle zum Lesen, Schreiben und Abrufen von Spielerstatistiken. Repositories vereinfachen Tests durch die Möglichkeit, Mock-Implementierungen für das Repository zu verwenden. Insgesamt fördert dies die Wartbarkeit, Erweiterbarkeit und Testbarkeit des Codes.

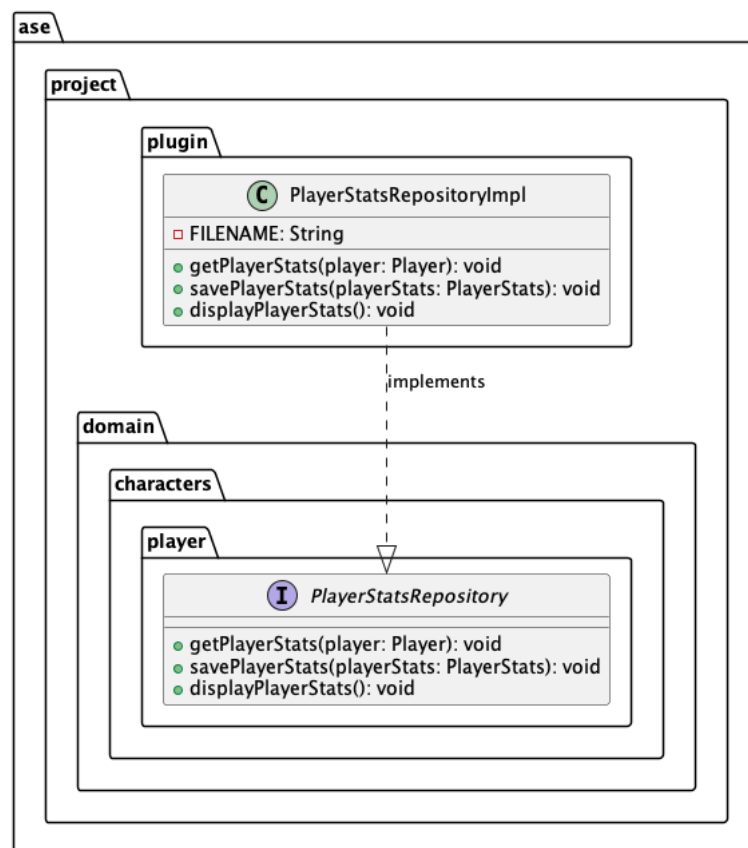


Abbildung 21. `PlayerStatsRepository` als Beispiel für eine Repository (DDD)

6.5. Aggregate

Die Klasse `GameCombatManager` dient als Aggregat (DDD) und ist für die Verwaltung und Durchführung von Kämpfen im Spiel verantwortlich. Sie arbeitet eng mit anderen Klassen wie `PlayerManager`, `EnemyManager`, `PlayerActionManager` und `EnemyCombatManager`.

zusammen, um die Aktionen des Spielers und der Feinde zu koordinieren. Sie steuert den Ablauf der Runden im Kampf und überwacht den Status von Spielern und Feinden.

Der Einsatz von GameCombatManager als Aggregat liegt darin, dass er mehrere verwandte Klassen zusammenführt, um eine komplexe Funktion im Spiel zu erfüllen. Als zentraler Koordinator vereint er die verschiedenen Aspekte des Kampfes und ermöglicht eine effiziente Steuerung und Interaktion zwischen den beteiligten Komponenten.

Die Vorteile von GameCombatManager als Aggregat in DDD liegen in der klaren Abgrenzung und Zusammenfassung der Kampffunktionen in einer einzigen Klasse. Dadurch wird die Codeorganisation und -wartung verbessert, da alle relevanten Funktionen und Verantwortlichkeiten an einem zentralen Ort gebündelt sind.



Abbildung 22. GameCombatManager als Beispiele für einen Aggregat (DDD)

7. Refactoring

7.1. Code Smell: Duplicated Code

Die Methode `useSpecialAttack` wurde redundant in jeder Player-Klasse (Astronomer, Busker, Gladiator) implementiert, obwohl sie dieselbe Aufgabe erfüllte. Daher wurde ein Refactoring durchgeführt, bei dem die Methode in eine neue Klasse extrahiert und von dort aus aufgerufen wird. Dadurch können alle Player-Klassen (Astronomer usw.) die Methode redundanzfrei verwenden (Refactoring 2).

Die Code-Beispiele für die vorgenommenen Verbesserungen sind unten zu finden und die Commits für die Refactorings sind im nächsten Kapitel dargestellt.

Vor dem Refactoring:

```
public class Astronomer extends Character {  
  
    @Override  
    public void useSpecialAttack(Enemy target, String attackName, int mana) {  
        try {  
            SpecialAttack specialAttack = specialAttackList.get(attackName);  
            if (specialAttack != null) {  
                specialAttack.useSpecialAttack(target, attackName, mana);  
            } else {  
                throw new InvalidAttackException("Invalid attack: " + attackName);  
            }  
        } catch (InvalidAttackException e) {  

```

```

        System.out.println(e.getMessage());
    }
    UseSpecialAttack.useSpecialAttack(target, attackName, mana, specialAttackList);
}
}

```

Nach dem Refactoring 1:

```

public class UseSpecialAttack {

    public static void useSpecialAttack(Enemy target, String attackName, int mana, Map<String,
SpecialAttack> specialAttackList) {
        try {
            SpecialAttack specialAttack = specialAttackList.get(attackName);
            if (specialAttack != null) {
                specialAttack.useSpecialAttack(target, attackName, mana);
            } else {
                throw new InvalidAttackException("Invalid attack: " + attackName);
            }
        } catch (InvalidAttackException e) {
            System.out.println(e.getMessage());
        }
    }
}

public class Astronomer extends Character {

    @Override
    public void useSpecialAttack(Enemy target, String attackName, int mana) {
        UseSpecialAttack.useSpecialAttack(target, attackName, mana, specialAttackList);
    }
}

```

Nach dem Refactoring 2:

```

public abstract class PlayerClass extends Player {
    public void useSpecialAttack(Character target, String attackName) throws
InvalidAttackException {
        SpecialAttack specialAttack =
ChooseSpecialAttack.chooseSpecialAttack(specialAttackList, attackName);
        try {
            ManaService.checkMana(mana, specialAttack.getManaCost());
            specialAttack.performSpecialAttack(target, attackName);
            mana = ManaService.useMana(mana, specialAttack.getManaCost());
        } catch (InvalidManaException manaException) {
            System.out.println(manaException.getMessage());
        }
    }
}

```

7.2. Refactoring: Extract Method

Commit Refactoring1: [Commit von GitHub-Repo: UseSpecialAttack - Klasse](#)

Commit Refactoring2: [Commit von GitHub-Repo: PlayerManager - Klasse](#)

Die Methode `useSpecialAttack` wurde aus den einzelnen Klassen (Astronomer, Busker, Gladiator) extrahiert und in eine neue Klasse `UseSpecialAttack` verschoben (Refactoring 1). Durch die Extraktion dieser Methode wurde die Redundanz beseitigt und der Code entspricht dem DRY--Prinzip (Don't Repeat Yourself), was zu einer besseren Wartbarkeit und Lesbarkeit des Codes führt. Durch die Platzierung der `useSpecialAttack`-Methode in der neuen Klasse

können nun alle Player-Klassen (Astronomer, Busker, Gladiator) diese Methode verwenden. Dadurch wird der Code vereinfacht und die Funktionalität der speziellen Angriffe kann effizienter und einheitlicher implementiert werden. Dieses Refactoring ermöglicht eine einfachere Skalierbarkeit und Erweiterbarkeit des Codes, da zukünftige Änderungen oder Ergänzungen der speziellen Angriffe nur an einer Stelle vorgenommen werden müssen, anstatt in jeder einzelnen Player-Klasse.

Die Klassen wurden anschließend angepasst, damit die PlayerManager-Klasse diese Methode implementiert. Dadurch können alle Player-Klassen (Astronomer usw.) die Methode erben und redundanzfrei verwenden (Refactoring 2).

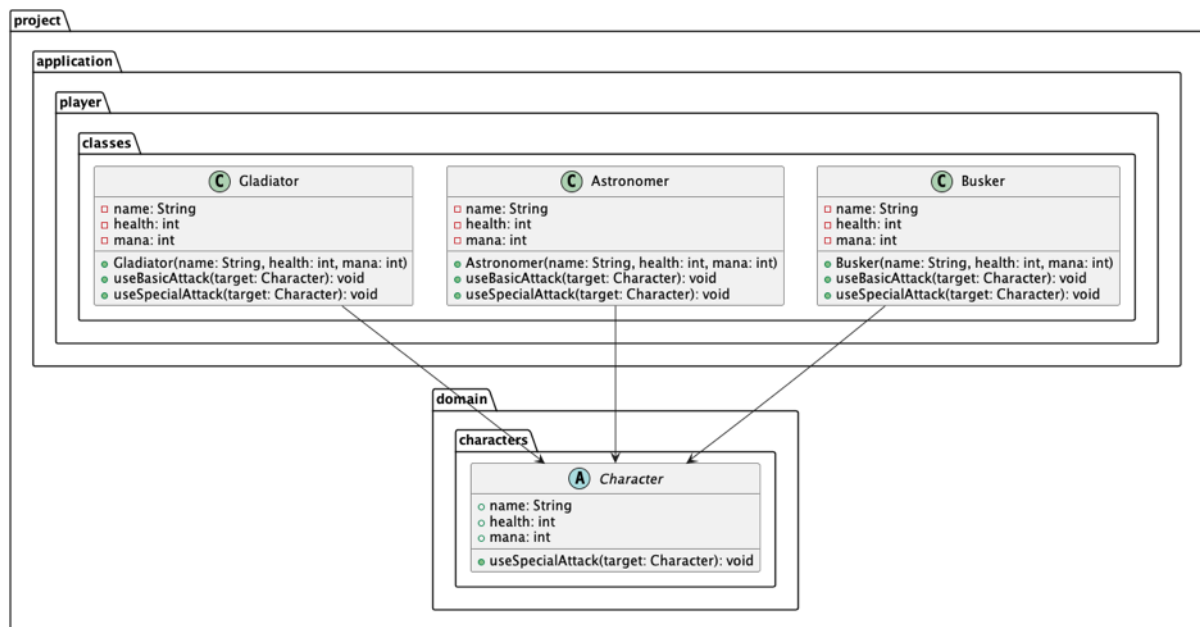


Abbildung 23. **Vor dem Refactoring** - die 3 Klassen haben alle die redundante Methode useSpecialAttack

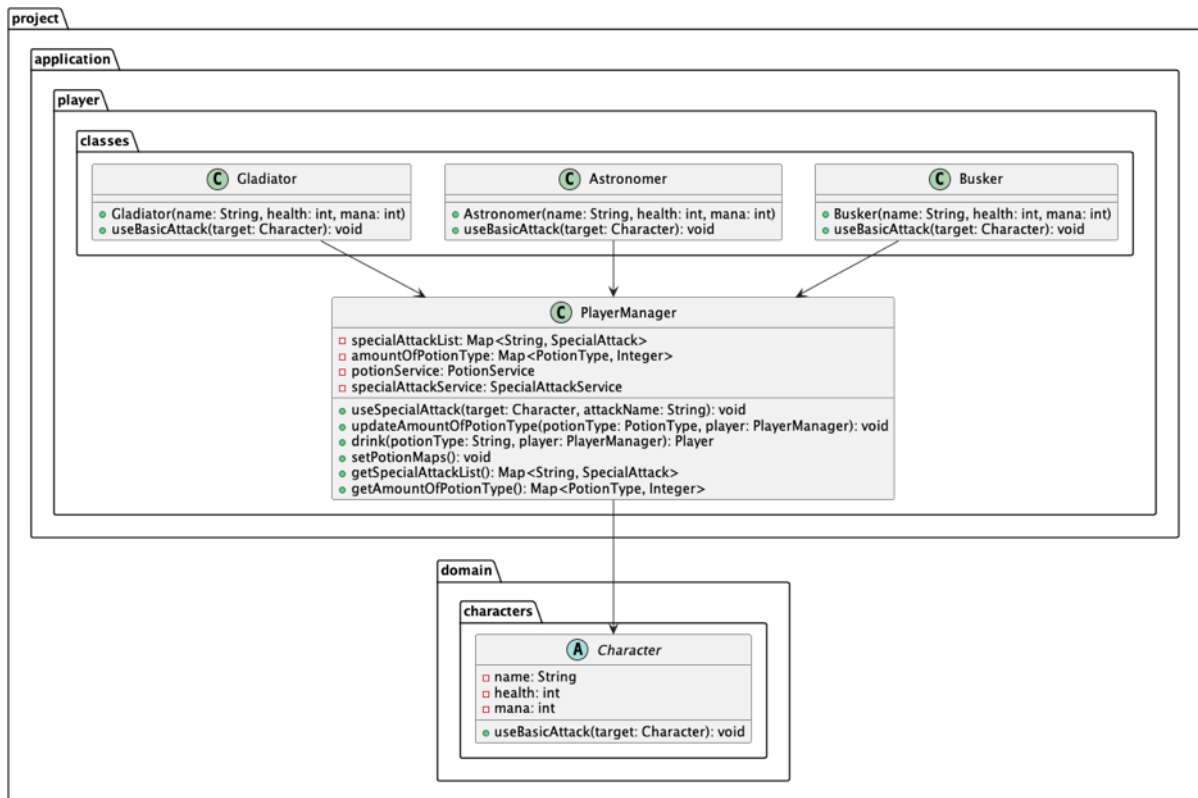


Abbildung 24. **Nach dem Refactoring** – nur die Klasse PlayerManager implementiert useSpecialAttack

7.3. Code-Smell: Long Method

Die Methode `takeTurnsInCombat` in der Klasse `GameCombatManager` stellt den Code-Smell "Long Method" dar, da sie zu lang und zu umfangreich ist. Dies führt zu einer schlechten Lesbarkeit und erschwert das Verständnis des Codes. Außerdem erschwert es die Wartbarkeit und Erweiterbarkeit der Methode.

Code Smell: Long Method

```
public void takeTurnsInCombat() throws InvalidAttackException {
    boolean playerTurn = true;

    while (!enemies.isEmpty() && !player.isDead()) {
        Character currentPlayer;

        if (playerTurn) {
            playerActionManager.chooseAction(player, level);
        } else {
            if (enemies.size() == 1) {
                currentPlayer = enemies.get(1);
            } else {
                currentPlayer = getCurrentEnemy();
            }

            if (currentPlayer != null && !currentPlayer.isDead()) {
                enemyCombatManager.chooseAction((EnemyManager) currentPlayer, player, level);
            }
        }

        playerTurn = !playerTurn;
        updateCurrentCharacterIndex();
    }
}
```

Um die Methode zu verbessern, kann sie aufgeteilt werden, um die Verantwortlichkeiten besser zu trennen. Eine mögliche Lösung besteht darin, separate Methoden für die Aktionen des Spielers und der Gegner zu erstellen. Dadurch wird der Code in überschaubare Teile aufgeteilt und jeder Teil ist besser lesbar und verständlich. Dies verbessert die Wartbarkeit und erleichtert es, die Methode zu erweitern oder anzupassen, da Änderungen an einem Teil des Codes isoliert vorgenommen werden können, ohne den gesamten Ablauf der Methode zu beeinflussen.

Verbesserung des Code Smells

```
public void takeTurnsInCombat() throws InvalidAttackException {
    while (shouldContinueCombat()) {
        if (isPlayerTurn()) {
            playerActionManager.chooseAction(player, level);
        } else {
            Character currentPlayer = getCurrentEnemy();
            if (currentPlayer != null && !currentPlayer.isDead()) {
                enemyCombatManager.chooseAction((EnemyManager) currentPlayer, player, level);
            }
        }
        switchTurn();
    }
}

private boolean shouldContinueCombat() {
    return !enemies.isEmpty() && !player.isDead();
}

private boolean isPlayerTurn() {
    return playerTurn;
}

private void switchTurn() {
    playerTurn = !playerTurn;
}
```

7.4. Refactoring: Replace Conditional with Polymorphism

Commit: [Commit - GitHub Repo](#)

Mit der abstrakten Klasse "Character", die von "Enemy" und "Player" geerbt wird, können die Methoden, die von "Enemy" und "Player" gemeinsam genutzt werden, in der abstrakten Klasse definiert werden. Es müssen keine bedingten Überprüfungen durchgeführt werden, ob das Ziel ein "Enemy" oder ein "Player" ist, weil die abstrakte Klasse "Character" als gemeinsame Basis verwendet wird. Stattdessen kann einfach eine Instanz von "Character" übergeben werden, und die gemeinsamen Methoden können aufgerufen werden, ohne dass eine explizite Typüberprüfung erforderlich ist.

Dies nutzt das Prinzip des Polymorphismus aus, da verschiedene Objekte denselben Code aufrufen können, der in der übergeordneten abstrakten Klasse definiert ist. Durch die Anwendung von Polymorphismus wird der Code vereinfacht und die Lesbarkeit verbessert, da die Logik für den Umgang mit "Enemy" und "Player" zentralisiert wird. Es ermöglicht auch eine

einfachere Erweiterbarkeit, da neue Klassen, die von "Character" erben, automatisch Zugriff auf die gemeinsamen Methoden haben und das Verhalten an ihre spezifischen Bedürfnisse anpassen können. Dadurch wird der Code flexibler, besser strukturiert und leichter zu warten. Abbildung 25 stellt das UML-Diagramm vor dem Refactoring dar, wo die Klasse Player noch nicht implementiert wurde. Dabei wurde die Klasse Character zur Darstellung des Players verwendet.

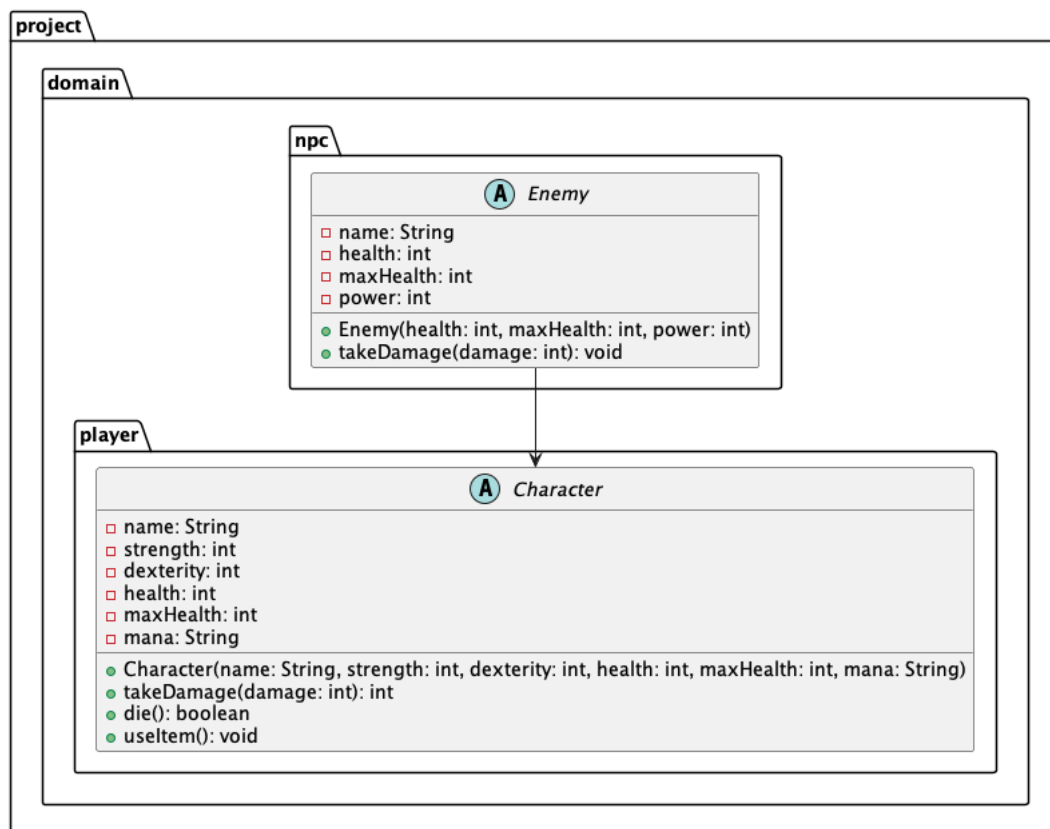


Abbildung 25. **Vor dem Refactoring** – Klasse Enemy erbt von der Klasse Character

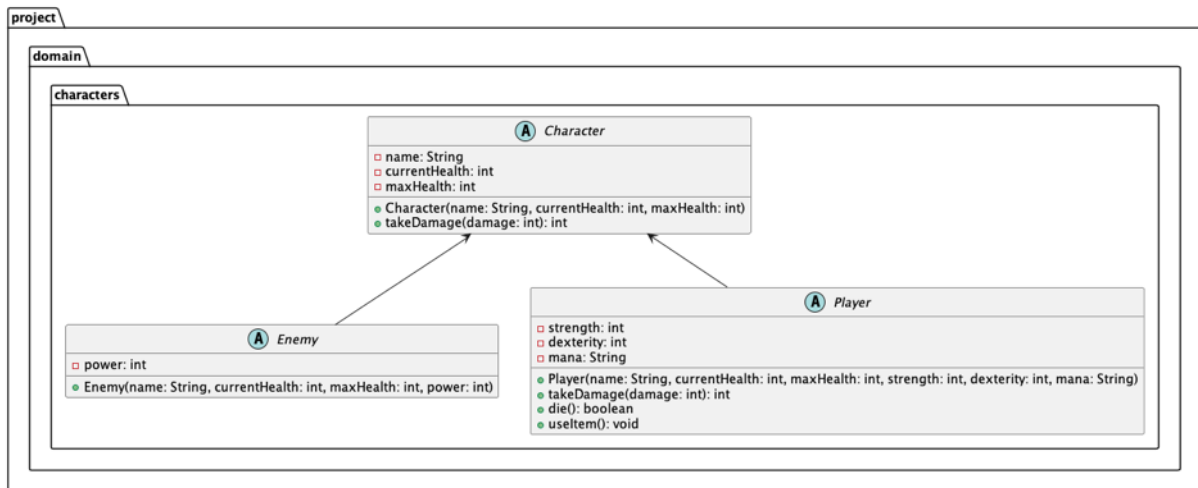


Abbildung 26. **Nach dem Refactoring** – Klassen *Player* und *Enemy* erben von *Character*

8. Entwurfsmuster

8.1. Erbauer (Builder): LevelBuilder

Die Klasse *LevelBuilder* ermöglicht das Erstellen eines *Level*-Objekts mit einer festgelegten Konfiguration. Sie enthält Methoden zum Festlegen der Feinde und der Levelbeschreibung sowie eine *Build*-Methode, um das endgültige *Level*-Objekt zu erstellen.

Die Implementierung des Erbauers (Builder) in Form des *LevelBuilders* ist sinnvoll, da er eine flexible und erweiterbare Möglichkeit bietet, *Level*-Objekte zu erstellen. Durch die Verwendung des Builders können verschiedene Parameter des Levels schrittweise festgelegt werden, ohne den Konstruktor der *Level*-Klasse mit einer Vielzahl von Parametern zu überladen. Dadurch wird der Code lesbarer, einfacher zu warten und ermöglicht die klare Trennung der Verantwortlichkeiten für das Erstellen eines Levels. Der Builder ermöglicht es, die Konfiguration des Levels Schritt für Schritt anzupassen und erleichtert so die Anpassung an verschiedene Anforderungen. Darüber hinaus bietet der Builder eine klare Schnittstelle zum Erstellen von *Level*-Objekten. Das erleichtert die Erstellung von Levels und erhöht die Wiederverwendbarkeit des Codes.

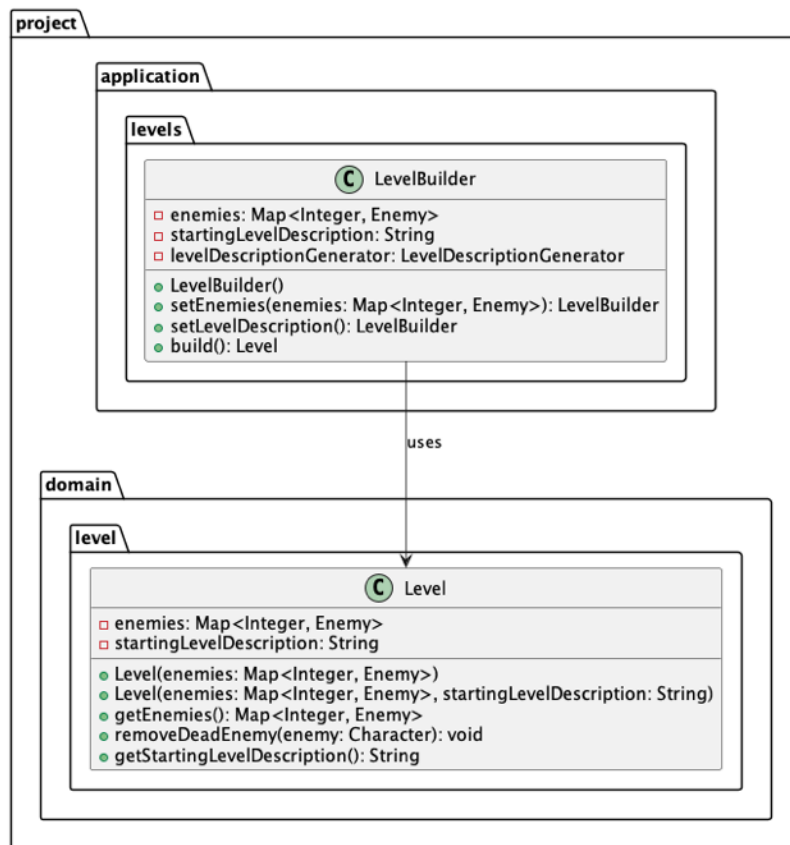


Abbildung 27. UML-Diagramm des Builder Pattern - Bsp. LevelBuilder

8.2. Beobachter (Observer): DeathObserver

Die Klasse **DeathObserver** ist ein Interface, das die Methode `onCharacterDeath` definiert, die aufgerufen wird, wenn ein Charakter stirbt. Der **DeathObserverManager** verwaltet und benachrichtigt die Observer über den Tod eines Charakters. Die Klasse **Death** implementiert das **DeathObserver**-Interface und handhabt den Tod von Charakteren.

Je nachdem, ob der Charakter ein Spieler oder ein Feind ist, wird die entsprechende Methode `onPlayerDeath` oder `onEnemyDeath` aufgerufen.

Die Implementierung des Beobachter-Entwurfsmusters ist sinnvoll, um über den Tod von Charakteren zu informieren und entsprechende Aktionen auszuführen. Dadurch können verschiedene Teile des Systems auf den Tod eines Charakters reagieren, ohne dass eine direkte Abhängigkeit zwischen ihnen besteht. Ein Vorteil dieser Implementierung ist die Entkopplung von Klassen, die den Tod eines Charakters beobachten, und den Klassen, die den Tod verursachen. Dadurch wird eine bessere Modularität und Wiederverwendbarkeit erreicht. Zudem ermöglicht dieses Entwurfsmuster eine einfache Skalierbarkeit, da neue Beobachter hinzugefügt werden können, um spezifische Reaktionen auf den Tod von Charakteren zu implementieren, ohne den bestehenden Code zu ändern.

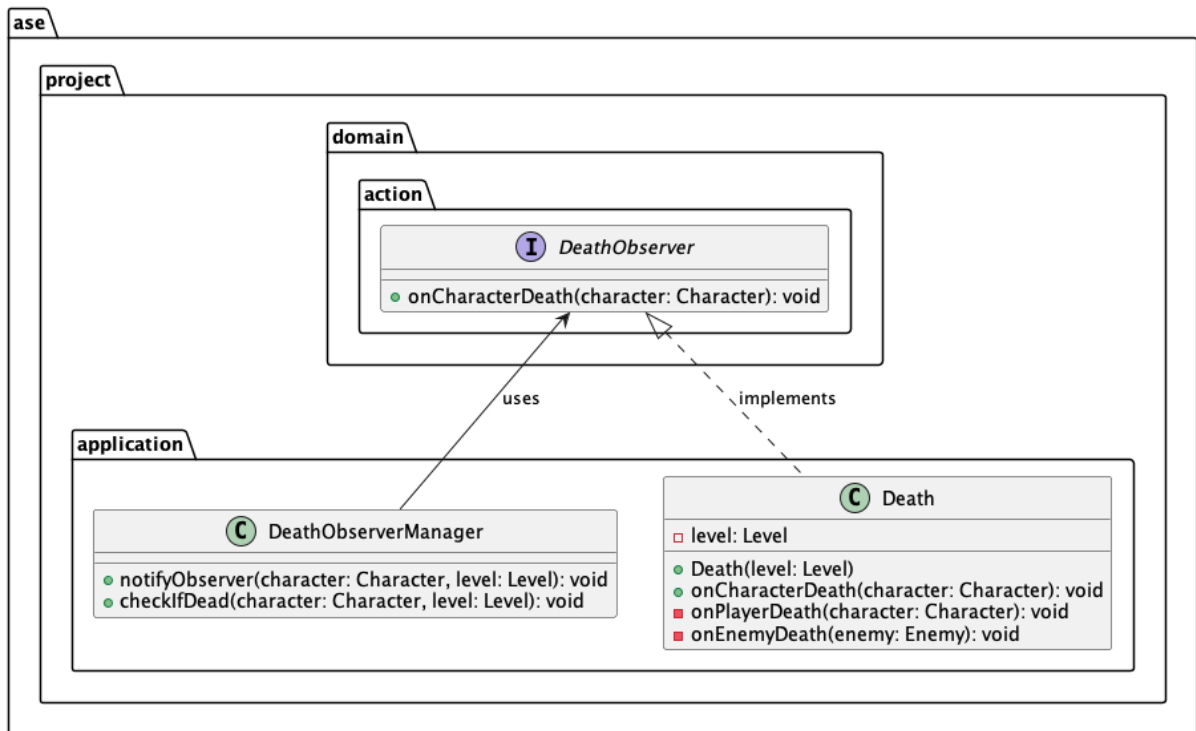


Abbildung 28. Observer-Pattern - Bsp. DeathObserver-Interface und ihre Beziehung mit anderen Klassen