Programmentwurf
DnD-Character Manager

Name:	Leon Knorr
Matrikelnummer:	9800840
Abgabedatum:	01. Februar 2018

Inhaltsverzeichnis

1 Einführung

1.1 REPO Link

https://github.com/lkno0705/DnD-CharacterManager

1.2 Übersicht über die Applikation

Die Abgegebene Applikation, „DnD-CharacterManager“, ermöglicht das erstellen und verwalten von Dungeons & Dragons Charakteren nach dem 5e Regelwerk. Somit soll sie den bekannten Papiercharakterbogen ablösen und durch eine digitale Version ersetzen. Neben dem persistenten Halten von Daten nach dem In-Memory Prinzip moderner Datenbanken, annulliert sie ausserdem das anstrengende Kopfrechnen bei verschiedenen Checks und stellt einen Character Creation Wizard bereit, der den Nutzer durch den Charaktererstellungsprozess führt.

1.3 Wie startet man die Applikation?

Die Applikation kann man starten, in dem man die bereitgestellte *.jar Datei in einem Terminalemulator mit dem Befehl
`java -jar DnDCharacterManager-jar-with-dependencies.jar` ausführt. Voraussetzung dafür ist eine valide funktionierende Java Installation. Das Projekt wurde mittels des `openjdk-18` und einem „language level“ von 17 erstellt. Bitte verwenden Sie zum ausführen eine ähnliche Version des JDKs, da es ansonsten zu Problemen und Inkompatibilität kommen kann.

1.4 Wie testet man die Applikation?

Die Unit Tests der Applikation können über Maven mit Maven test, package und install ausgeführt werden. Die Applikation selbst lässt sich in der Kommandozeile bedienen.

2 Clean Architecture

2.1 Was ist Clean Architecture?

Die Clean Architecture ist eine Richtlinie für das Programmieren von Software. Dabei wird die Codebasis in 4 Schichten aufgeteilt:

- **Plugin Schicht:** Diese Schicht enthält sämtlichen Framework oder Gerät abhängigen Code
- **Adapter Schicht:** Diese Schicht enthält jeglichen Code, der Daten / Methoden der Plugin Schicht, zur Nutzung der Unteren Schichten umformt. Sie stellt also sicher, das die unteren Schichten fehlerfrei verwendet werden können.
- **Application Schicht:** Die Applikationsschicht enthält alle Use-Cases der Applikation und repräsentiert somit alle möglichen und verfügbaren Aktionen die die Applikation abbildet. Sie enthält ausschließlich Business Logik und weiß nicht wer den Code Ausführt, noch wie er am Ende präsentiert werden soll.
- **Domain Schicht:** Diese Schicht enthält alle Objekte die die Anwendungsdomäne repräsentieren.

Wichtig ist, das eine äußere Schicht abhängig von einer inneren Schicht sein darf, eine innere Schicht allerdings nicht von einer äußeren Schicht. Das ist die sogennante „Dependency Rule“. Damit wird der primäre Zweck der Clean Architecture sichergestellt: Das ordnen von Code nach zeitlicher Relevanz und das ermöglichen von einfachem Austauschen von kurzfristigem Code. Durch diese Architektur, lassen sich Frameworks oder Persistierungsimplementation jederzeit schnell und einfach austauschen, während der gültige Domaincode nicht verändert werden muss. Somit muss der Kern einer Applikation nur einmal geschrieben werden, während man sie auf beliebige Art und Weise bereitstellen kann.

2.2 Analyse der Dependency Rule

[(1 Klasse, die die Dependency Rule einhält und eine Klasse, die die Dependency Rule verletzt); jeweils UML der Klasse und Analyse der Abhängigkeiten in beide Richtungen (d.h., von wem hängt die Klasse ab und wer hängt von der Klasse ab) in Bezug auf die Dependency Rule]

2.2.1 Positiv-Beispiel: Dependency Rule

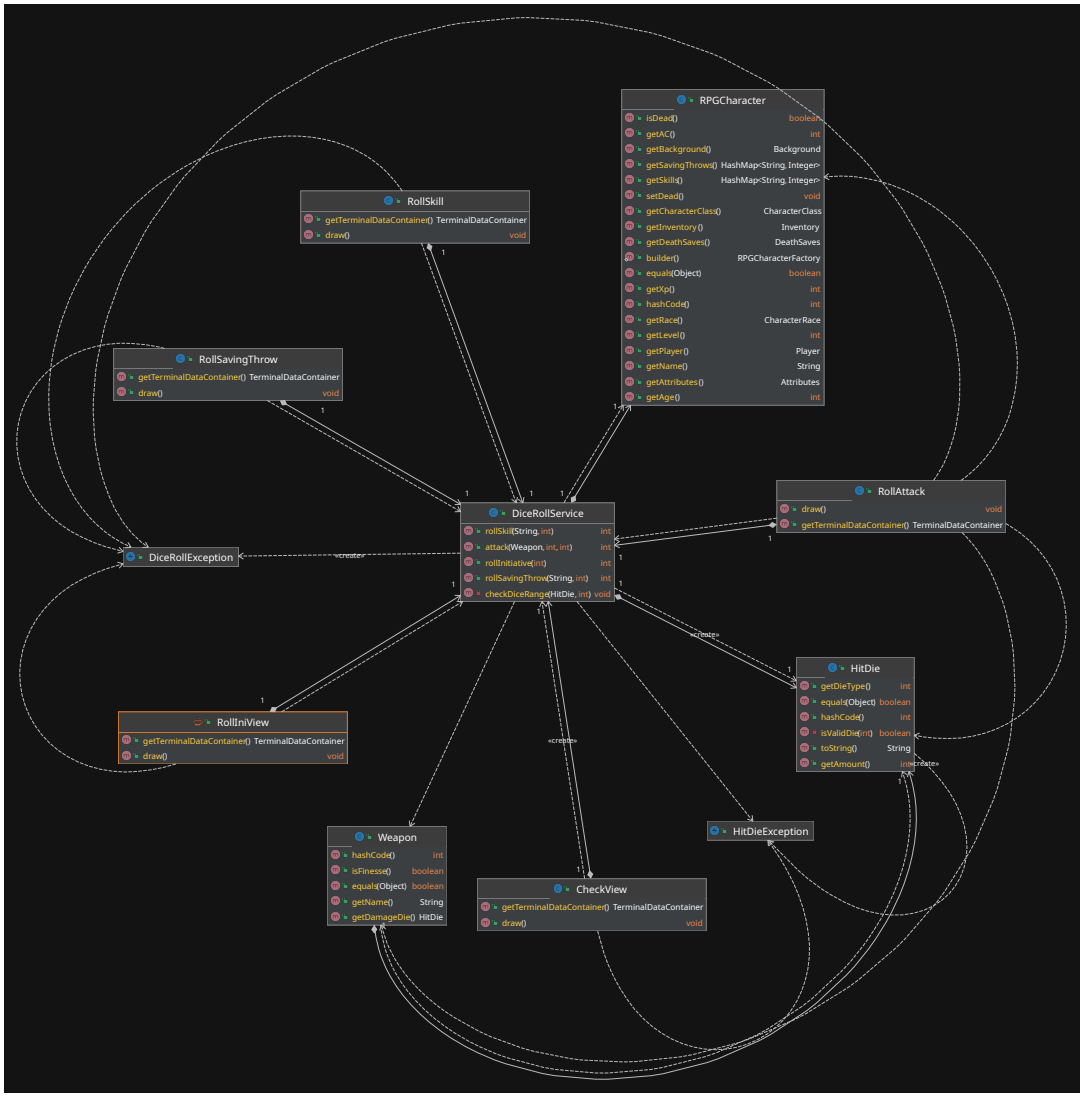


Abbildung 2.1: UML RollDiceService

Abbildung 2.1 zeigt das UML Diagramm der Klasse `DiceRollService`. Diese Klasse ist Teil des Application Layers der Clean Architecture und darf somit nur von unteren Layern abhängen. Wie der Abbildung zu entnehmen ist, ist dies der Fall. Die Klasse hängt ausschließlich von den Klassen `HitDie`, `HitDieException`, `Weapon`, `DiceRollException` und `RPGCharacter` ab. All diese Klassen liegen in der Domain Schicht. Abhängig von der Klasse `DiceRollService` sind die Klassen `RollSkill`, `RollSavingThrow`, `RollIniView`, `CheckView` und `RollAttack`, die in der Plugin Schicht liegen und Teil des User-Interfaces sind. Somit wird die Dependency Rule bei dieser Klasse eingehalten.

2.2.2 Negativ-Beispiel: Dependency Rule

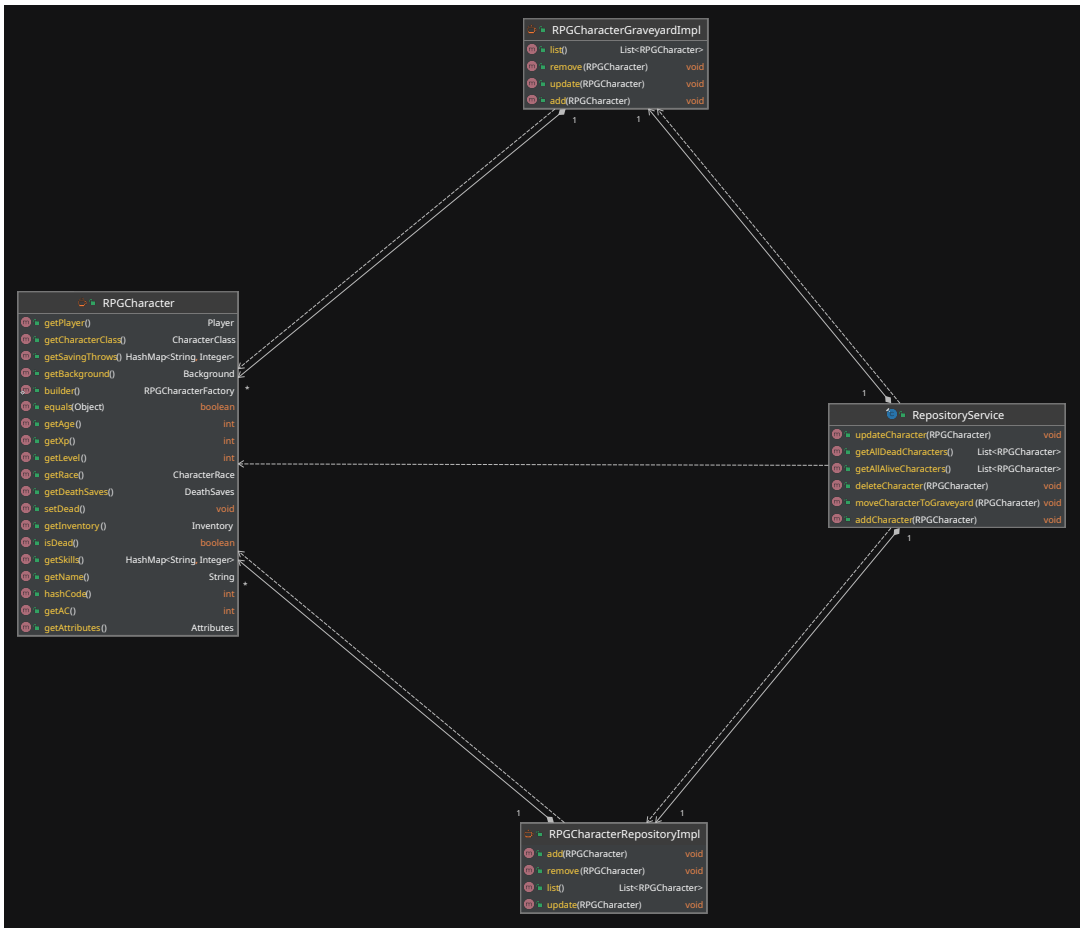


Abbildung 2.2: UML RepositoryService

Abbildung 2.2 zeigt das UML Klassendiagramm der Klasse `RepositoryService`. Diese Klasse befindet sich in der Application Schicht der Clean Architecture und stellt verschiedene Methoden / Use Cases für höhere Schichten bereit. Wie der Abbildung zu entnehmen ist, wurde hier die Dependency Rule nicht eingehalten, denn die Klasse `RepositoryService` hängt von den Klassen `RPGCharacterGraveyardImpl` und `RPGCharacterRepositoryImpl` ab. Beide Klassen sind Teil der Plugin Schicht und stellen die Implementierung der Repositorys zur Persistierung der Daten In-Memory bereit. Somit, wird hier die Dependency Rule verletzt, da eine Klasse der Application Schicht von Klassen in der Plugin Schicht abhängt. Neben den beiden Klassen aus der Plugin Schicht, ist die Klasse `RepositoryService` noch von der Klasse `RPGCharacter` aus der Domain Schicht abhängig. Um die Dependency Rule hier einzuhalten, wäre es korrekt anstatt die Implementation, die entsprechenden Interfaces in der Domain Schicht zu referenzieren. Ich musste dieses Beispiel aus meinem Code in einem späteren Commit entfernen, da maven aufgrund der entstehenden Circular Dependency keinen Workflow mehr zugelassen hat. Der Code ist in Commmit https://github.com/lkno0705/DnD-CharacterManager/tree/05b0b375a9279b1d63f8072d4d1756f30306457b einsehbar.

2.3 Analyse der Schichten

2.3.1 Plugin: MainMenu

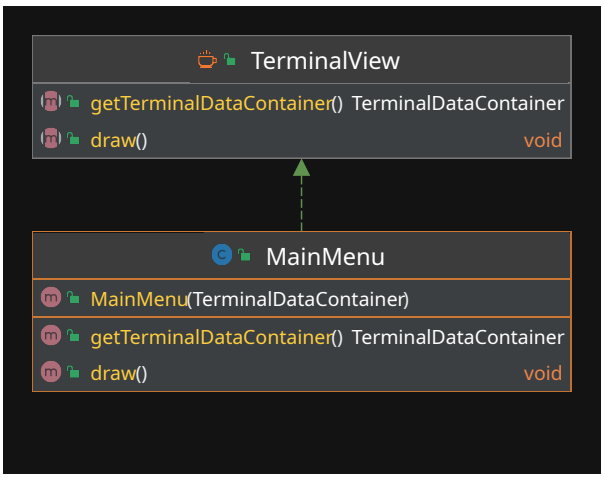


Abbildung 2.3: UML MainMenu

Die Klasse `MainMenu` liegt in der Plugin Schicht, da sie Teil des User Interfaces ist. Sie stellt den Einstiegspunkt der Nutzer Interaktion dar und stellt alle Möglichkeiten und Navigationspunkte dem User dar. Da die Implementierung des User Interfaces Framework spezifisch, bzw. Gerät spezifisch ist, gehört diese Klasse klar in die Plugin Schicht der Clean Architecture. Abbildung 2.3 zeigt das UML der Klasse.

2.3.2 Domain: Weapon

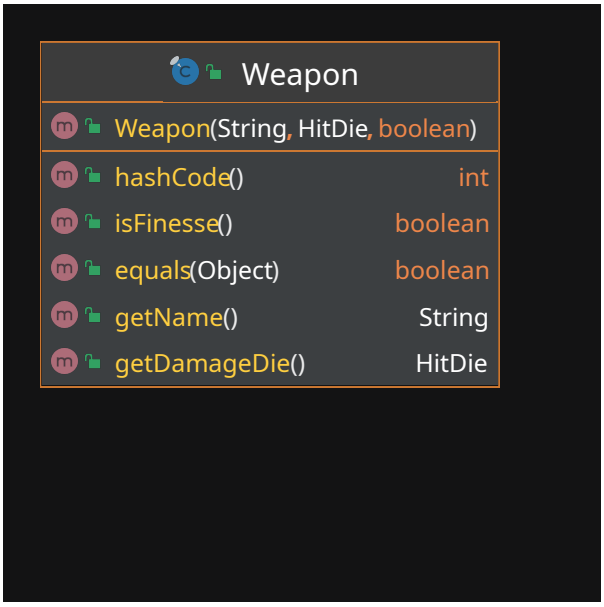


Abbildung 2.4: UML Weapon

Die Klasse Weapon Modelliert eine Waffe aus D&D 5e. Sie enthält ausschließlich Daten und modelliert einen wichtigen Teil der Anwendungsdomäne. Somit ist sie klar in die Domain Schicht der Clean Architecture einzuordnen.

3 SOLID

3.1 Analyse Single-Responsibility-Principle (SRP)

[jeweils eine Klasse als positives und negatives Beispiel für SRP; jeweils UML der Klasse und Beschreibung der Aufgabe bzw. der Aufgaben und möglicher Lösungsweg des Negativ-Beispiels (inkl. UML)]

3.1.1 Positiv-Beispiel

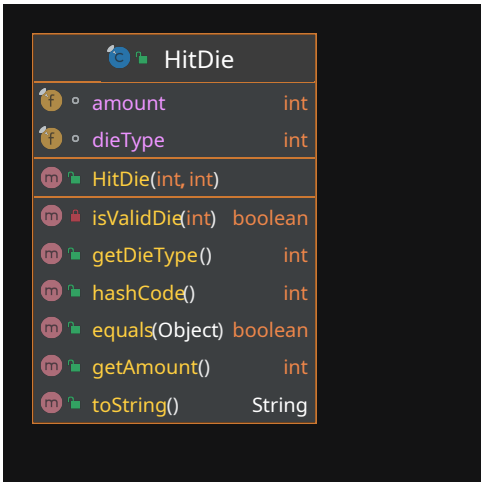


Abbildung 3.1: UML der HitDie Klasse

Abbildung 3.1 zeigt das UML-Diagramm der `HitDie` Klasse. Sie hält das Single Responsibility Principle ein, da sie ausschließlich für die Repräsentation eines Würfels zuständig ist und keinerlei andere Funktionalität enthält.

3.1.2 Negativ-Beispiel

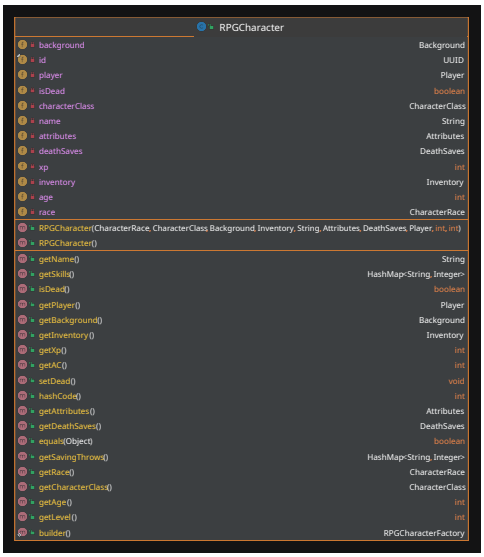


Abbildung 3.2: UML der RPGCharacter Klasse

Die in Abbildung 3.2 gezeigte `RPG-Character` Klasse hält das SRP nicht ein. Sie stellt nicht nur die verschiedenen Datenobjekte und Entitäten die ein Charakter hat zur Verfügung, sondern berechnet auch Status Werte wie z.B.: die ArmorClass (AC). Daher wäre es sinnvoll diese Berechnung in eine eigene ArmorClass Klasse auszulagern, wie in Abbildung 3.3.

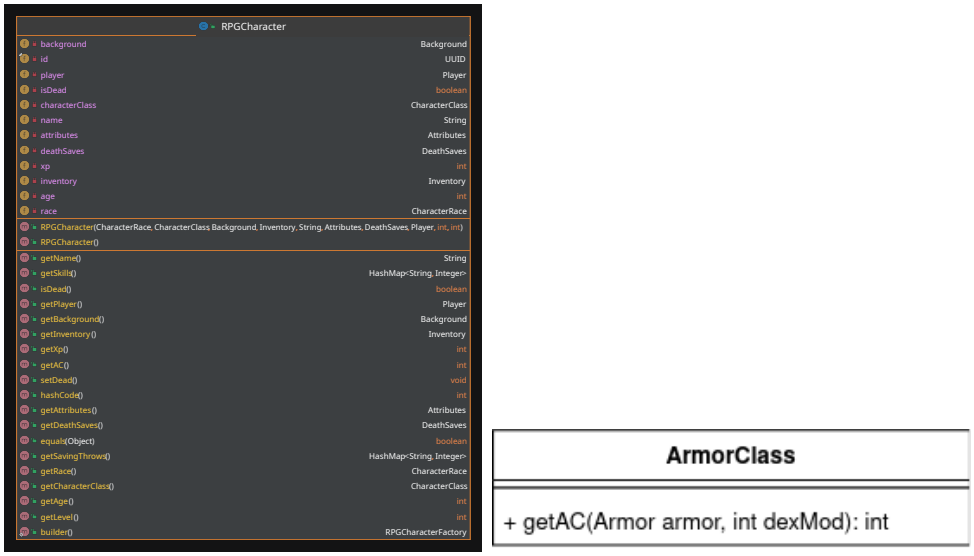


Abbildung 3.3: UML der RPGCharacter Klasse und der theoretischen ArmorClass

3.2 Analyse Open-Closed-Principle (OCP)

3.2.1 Positiv-Beispiel

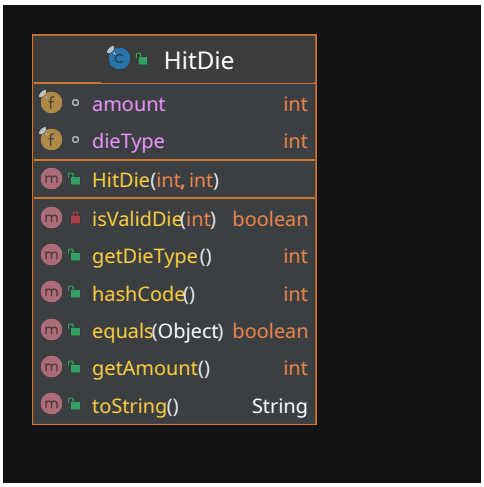


Abbildung 3.4: UML der HitDie Klasse

Die in Abbildung 3.4 gezeigte Klasse `HitDie` hält das OCP ein, da sie alle Funktionen bereitstellt, die ein Würfel benötigt. Möchte man nun einen weiteren gültigen Würfel

hinzufügen, muss nur diese Klasse angepasst werden und keine weiteren Änderungen im Projekt vorgenommen werden.

3.2.2 Negativ-Beispiel

Ich lasse an dieser Stelle das UML weg, da es keine Sinnvolle Aussagekraft hat. Die Klasse `SkillProficiencies` hält das OCP nicht ein, auch wenn sie eine Liste aller verfügbaren Skills enthält, muss man, wenn man einen neuen Skill hinzufügen möchte auch Änderungen in der `RPGCharacterClass` vornehmen. Da dort eine Hashmap zusammengebaut wird, die die tatsächlichen Werte enthalten. Somit liegt die Funktionalität der SkillProficiencies Klasse auf 2 Klassen aufgeteilt im Projekt und Änderungen und Modifikationen erfordern einen höheren Aufwand. Lösen könnte man dies, in dem man entweder in der Klasse SkillProficiencies eine neue Methode für die oben gennante Funktionalität anlegt, oder eine Klasse für die Skillwerte anlegt, die sich eine statische Liste aller validen Skills mit der SkillProficiencie Klasse teilt.

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

3.3.1 Positiv-Beispiel DIP

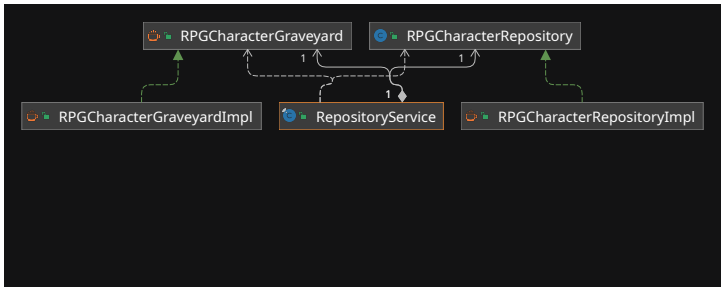


Abbildung 3.5: UML der RepositoryService Klasse

Der RepositoryService hält das DIP ein, da er ausschließlich von einer Abstraktion der Repositorys in Form der jeweiligen Interfaces Abhängig ist, nicht aber von der Klasse und der Implementation selbst. Somit kann die Implementation jederzeit geändert werden, ohne das der RepositoryService verändert werden muss. - Repositorys

3.3.2 Negativ-Beispiel

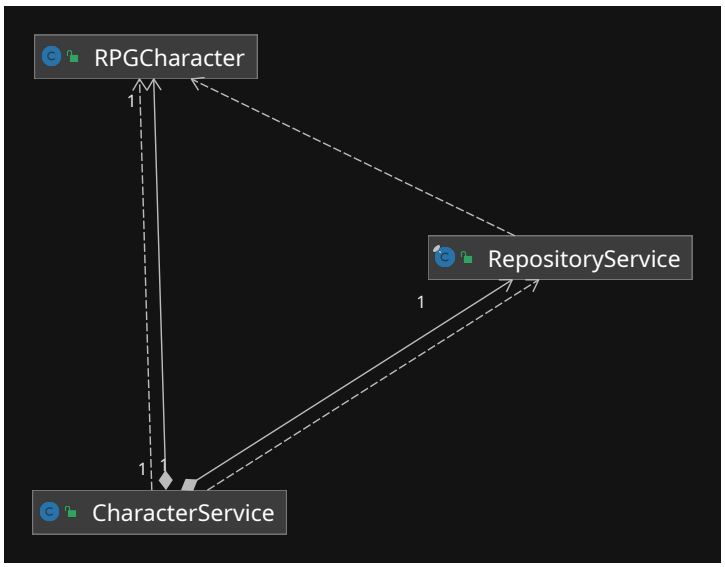


Abbildung 3.6: UML der CharacterService Klasse

Die Klasse CharacterService hält DIP wiederum nicht ein. Sie speichert direkt Verweise auf Objekte und ruft in allen Methoden, Methoden der gespeicherten Objekte sofort auf. Somit ist sie direkt Abhängig von den Objekten und nutzt keinerlei Abstraktionen.

4 Weitere Prinzipien

4.1 Analyse GRASP: Geringe Kopplung

[jeweils eine bis jetzt noch nicht behandelte Klasse als positives und negatives Beispiel geringer Kopplung; jeweils UML Diagramm mit zusammenspielenden Klassen, Aufgabenbeschreibung und Begründung für die Umsetzung der geringen Kopplung bzw. Beschreibung, wie die Kopplung aufgelöst werden kann]

4.1.1 Positiv-Beispiel

Gibt keines, da sich ein Listenerpattern im Projekt nicht angeboten hat.

4.1.2 Negativ-Beispiel

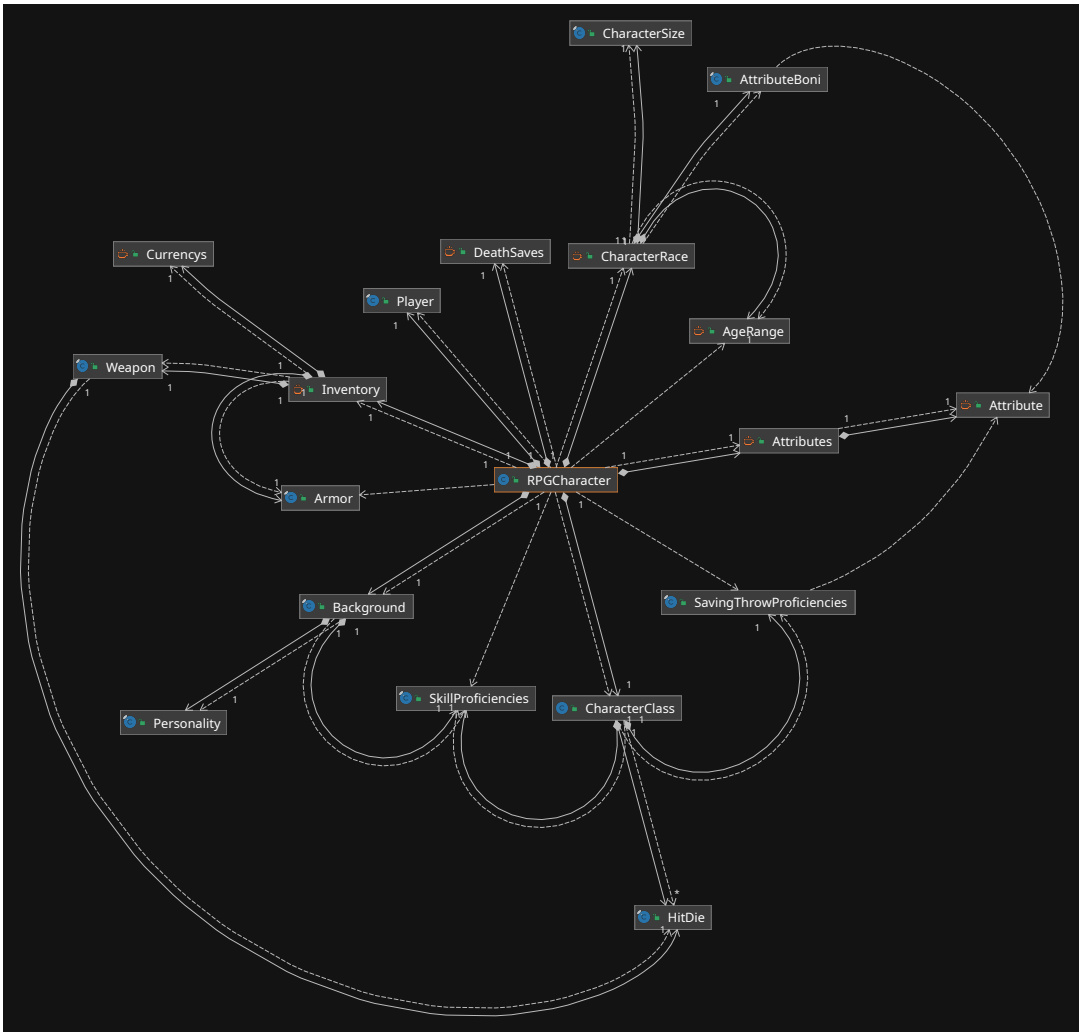


Abbildung 4.1: UML der RPGCharacter KLasse

Die RPGCharacter Klasse enthält viele Referenzen auf andere Instanzen von Klassen, die wiederum für die Klasse wichtige Daten halten. Somit ist sie sehr stark an diese gekoppelt. Änderungen in diesen Klassen haben somit einen hohen Einfluss auf die Resultate der RPGCharacterclass.

4.2 Analyse GRASP: Hohe Kohäsion

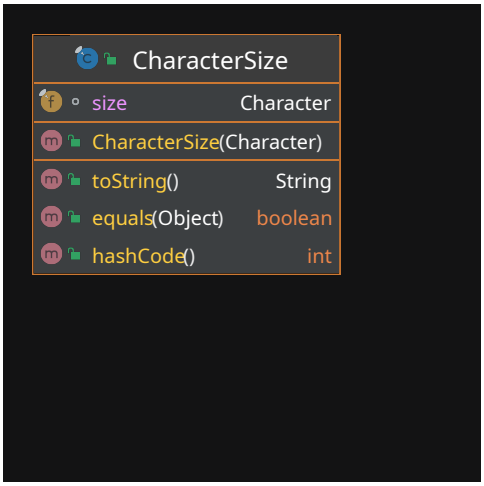


Abbildung 4.2: UML der CharacterSize Klasse

In der CharacterSize Klasse ist die Kohäsion hoch, da sie alle Funktionalitäten abdeckt, die die Größe des Charakters betreffen. Somit muss eine Änderung hier ausschließlich in dieser Klasse vorgenommen werden und nirgendwo anders.

4.3 Don't Repeat Yourself (DRY)

Gibt keinen. Ich habe von vornherein versucht duplizierten Code zu vermeiden.

5 Unit Tests

5.1 10 Unit Tests

Tabelle irgendwo in diesem Kapitel. Latex packt die dahin wo es das für richtig hält

5.2 ATRIP: Automatic

Automatic wurde auf zwei verschiedene Arten und weisen realisiert, einmal wurde die pom.xml des Maven Projektes im Hauptmodul der Applikation so angepasst, das das automatische Ausführen aller Tests Teil des Maven Workflows ist. Somit werden Tests automatisch mit jedem maven package, test und install ausgeführt. Sollte ein Test fehlschlagen, wird der jeweilige maven Workflow nicht erfolgreich abgeschlossen. Desweiteren wurde ein Github workflow zur automatischen Validierung aller Pullrequests und Commits angelegt. Dieser Workflow führt nach jedem Commit in einer isolierten Umgebung den maven package workflow aus. Kommt es während dieses zu einem Test Failure, schlägt der Workflow fehl und die Pullrequest kann nicht gemerged werden, oder es wird explizit am Commit ausgewiesen, dass dieser Commit fehlerhaft ist.

Tabelle 5.1: Evaluation der Datensätze

Unit Test	Beschreibung
RPGCharacterTest#getAC()	Es wird geprüft ob die Armor Class (AC) des Charackters korrekt berechnet wird.
RPGCharacterTest#getSavingThrows()	Es wird geprüft ob die SavingThrows Boni des Charackters korrekt berechnet werden.
RPGCharacterTest#getSkills()	Es wird geprüft ob die Skill Boni des Charackters korrekt berechnet werden.
DeathSavesTest#getFailures()	Prüft ob die Anzahl an Fehlschlägen korrekt berechnet wird.
DeathSavesTest#getSuccesses()	Prüft ob die Anzahl an Erfolgreichen Death Saves korrekt berechnet wird
DiceRollServiceTest#rollInitiative()	Prüft ob das Ergebnis des Initiative Wurfs korrekt berechnet wird
DiceRollServiceTest#attack()	Prüft ob der Damage beim Ausführen einer Attacke korrekt berechnet wird
DiceRollServiceTest#rollSkill()	Prüft ob das Ergebnis eines Skill Rolls korrekt berechnet wird
DiceRollServicetest#rollSavingThrow()	Prüft ob das Ergebnis eines SavingThrows korrekt berechnet wird
CharacterServiceTest#displayCharacter()	Prüft ob der String eines Charackters korrekt zusammengebaut wird.

5.3 ATRIP: Thorough

```
1       @Test
2       void attack() throws DiceRollException {
3           Weapon weaponWithFinesse = mockWeapon(true);
4           Weapon normalWeapon = mockWeapon(false);
5           assertEquals(5, this.diceRollService.attack(normalWeapon, 5, 11));
6           assertEquals(10, this.diceRollService.attack(normalWeapon, 5, 20));
7           assertEquals(7, this.diceRollService.attack(weaponWithFinesse, 5, 11));
8           assertEquals(0, this.diceRollService.attack(weaponWithFinesse, 5, 1));
9           assertEquals(0, this.diceRollService.attack(weaponWithFinesse, 5, 5));
10          assertThrows(DiceRollException.class, () ->{
11              this.diceRollService.attack(normalWeapon, 6, 22);
12          });
13          assertThrows(DiceRollException.class, () ->{
14              this.diceRollService.attack(normalWeapon, 15, 15);
15          });
16      }
```

Listing 5.1: Test der Attack Methode des DiceRollService

In Listing 5.1, ist ein Beispiel zu sehen, bei dem alle notwendigen Funktionalitäten der `attack()` Methode getestet werden. So testet der Unit Test das korrekte berechnen von Werten unter Einbezug aller möglicher Eigenschaften einer Waffe, sowie das auftreten von Exception, in dem absichtlich falsche Eingaben an die Funktion gereicht werden. Somit dekt dieser Test alle Funktionalitäten der Methode vollständig ab und hält damit das Thorough Prinzip ein. Im Vergleich dazu,hält der in Listing 5.2 gezeigte Test dieses Prinzip nicht ein. Er überprüft nur eine mögliche valide Eingabe und prüft keine Randfälle und falsch Eingaben. Somit wird nicht kontrolliert, ob nach veränderungen Exceptions noch korrekt geworfen werden oder ob ungewollte Seiteneffekte auftreten.

```
1       @Test
2       void rollSkill() throws DiceRollException {
3           assertEquals(13, this.diceRollService.rollSkill("Acrobatics", 10));
4       }
```

Listing 5.2: Test der rollSkill Methode des DiceRollService

5.4 ATRIP: Professional

```
1       private Attributes mockAttributes(){
2           Attributes mockedAttributes = mock(Attributes.class);
3           when(mockedAttributes.getDexMod()).thenReturn(2);
4           when(mockedAttributes.getStrengthMod()).thenReturn(2);
5           return mockedAttributes;
6       }
7
8       private HitDie mockDieD6(){
9           HitDie mockedDieD6 = mock(HitDie.class);
10          when(mockedDieD6.getDieType()).thenReturn(6);
11          when(mockedDieD6.getAmount()).thenReturn(2);
12          return mockedDieD6;
13      }
14
```

```
15      private Weapon mockWeapon(boolean isFinesse){
16          HitDie mockedDieD6 = mockDieD6();
17          Weapon mockedWeapon = mock(Weapon.class);
18          when(mockedWeapon.getDamageDie()).thenReturn(mockedDieD6);
19          when(mockedWeapon.isFinesse()).thenReturn(isFinesse);
20          return mockedWeapon;
21      }
22
23      private void mockChracter(){
24          Attributes mockedAttributes = mockAttributes();
25          this.mockedCharacter = mock(RPGCharacter.class);
26
27          when(this.mockedCharacter.getAttributes()).thenReturn(mockedAttributes);
28          when(this.mockedCharacter.getSkills()).thenReturn(new HashMap<>(){{
29              put("Acrobatics", 3);
30          }});
31          when(this.mockedCharacter.getSavingThrows()).thenReturn(new HashMap<>(){{
32              put("Strength", 3);
33          }});
34      }
35  }
```

Listing 5.3: Auszug aus dem Test des DiceRollService, ganzes File: https://github.com/lkno0705/DnD-CharacterManager/blob/main/2-dnd-charactermanager-application/src/test/java/rolls/DiceRollServiceTest.java

Listing 5.3 zeigt ein positives Beispiel des Professional Prinzips. In diesem Beispiel wurde Test Code wie Produktivcode behandelt und es wurde darauf geachtet, den Prozess des Mockings anstatt in einer riesigen Methode in mehrere kleine Untermethoden zu unterteilen. Somit ist der Code gut wartbar und falls eine Änderung gemacht werden muss, kann man sofort zu der jeweiligen Methode springen und muss nicht in einer Wall of Text die entsprechende Stelle raussuchen. Des weiteren kümmert sich jede Methode in diesem Beispiel genau um eine einzige Funktionalität. Somit wird in jeder Methode nur genau ein Mockobjekt generiert.

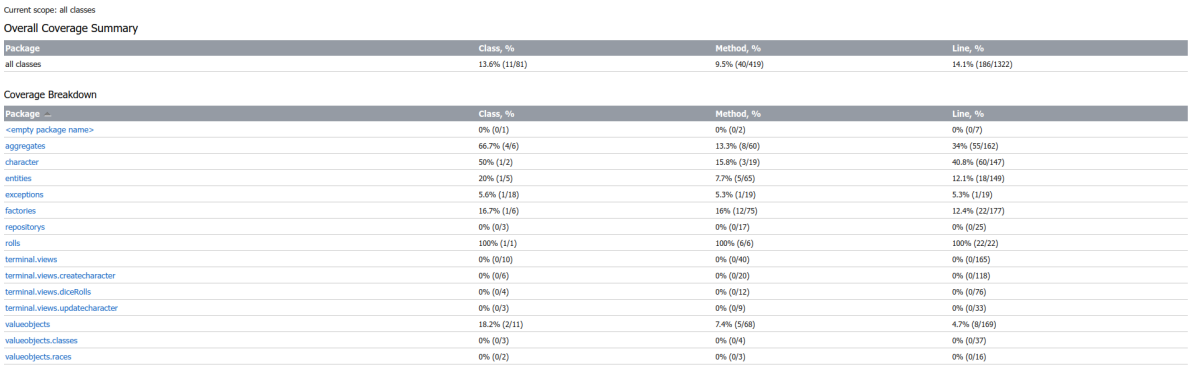
```
1       private HitDie mockDie() {
2           HitDie mockedDie = mock(HitDie.class);
3           when(mockedDie.toString()).thenReturn("HitDie{" +
4                   "dieType=" + 10 +
5                   ", amount=" + 1 +
6                   '}');
7           when(mockedDie.getDieType()).thenReturn(10);
8           when(mockedDie.getAmount()).thenReturn(1);
9           return mockedDie;
10      }
11
12      private void mockWeapon() {
13          HitDie mockedDie = mockDie();
14
15          this.mockedWeapon = mock(Weapon.class);
16          when(mockedWeapon.getDamageDie()).thenReturn(mockedDie);
17          when(mockedWeapon.getName()).thenReturn("Harte Hantel Hartholz");
18          when(mockedWeapon.isFinesse()).thenReturn(false);
```

```
19      }
```

Listing 5.4: Auszug aus dem Test des DiceRollService, ganzes File: https://github.com/lkno0705/DnD-CharacterManager/blob/main/2-dnd-charactermanager-application/src/test/java/character/CharacterServiceTest.java

Wie in Listing 5.4 kommt das Mocking von `HitDice`, `Weapons` etc. in anderen Tests auch zum Einsatz. Trotz dessen das auch in diesen Tests darauf geachtet wurde, den Mocking Prozess in kleine Methoden aufzuspalten und somit die Wartbarkeit und lesbarkeit zu erhöhen, stellt dies doch auch gleichzeitig ein negativ Beispiel dar. Da nun in jedem Test der ein Entsprechendes Objekt mockt, eine Änderung gemacht werden muss, wenn etwas an den jeweiligen Domain Objekten geändert wurde. Somit währe es hier sinnvoll gewesen, den Mocking Prozess in eine Utility Class auszulagern. Dies würde nicht nur die Wartbarkeit und lesbarkeit des Tests verbessern, sondern auch gleichzeitig die komplexität der Tests verringern.

5.5 Code Coverage



Current scope: all classes

Overall Coverage Summary

Package	Class, %	Method, %	Line, %
all classes	13.6% (11/81)	9.5% (40/419)	14.1% (186/1322)

Coverage Breakdown

Package	Class, %	Method, %	Line, %
<empty package name>	0% (0/1)	0% (0/2)	0% (0/7)
aggregates	66.7% (4/6)	13.3% (8/60)	34% (55/162)
character	50% (1/2)	15.8% (3/19)	40.8% (60/147)
entities	20% (1/5)	7.7% (5/65)	12.1% (18/149)
exceptions	5.6% (1/18)	5.3% (1/19)	5.3% (1/19)
factories	16.7% (1/6)	16% (12/75)	12.4% (22/177)
repositorys	0% (0/3)	0% (0/17)	0% (0/25)
rolls	100% (1/1)	100% (6/6)	100% (22/22)
terminal.views	0% (0/10)	0% (0/40)	0% (0/165)
terminal.views.createcharacter	0% (0/6)	0% (0/20)	0% (0/118)
terminal.views.diceRolls	0% (0/4)	0% (0/12)	0% (0/76)
terminal.views.updatecharacter	0% (0/3)	0% (0/9)	0% (0/33)
valueobjects	18.2% (2/11)	7.4% (5/68)	4.7% (8/169)
valueobjects.classes	0% (0/3)	0% (0/4)	0% (0/37)
valueobjects.races	0% (0/2)	0% (0/3)	0% (0/16)

Abbildung 5.1: Code Coverage Report generiert von IntelliJ

Abbildung 5.1 zeigt den Code Coverage report der Tests über das gesamte Projekt. Die generelle Coverage ist mit einer Line Coverage von 14.1%, einer Class Coverage von 13.6% und einer Method Coverage von 9.5% recht gering. Dies liegt daran das der größte Teil des Projektes ausschließlich aus der Modellierung von Domänenobjekten besteht, die ausschließlich Setter und Getter enthalten und nicht sehr viel Logik mit sich bringen. Diese Objekte werden in den jeweiligen Unit Tests durch mocks ersetzt

und somit nicht durchlaufen. Daher entstehen sehr geringe Coverage Prozent zahlen. In Packages wie dem `rolls` Package ist das anders. Dieses Package enthält die UseCases des `DiceRollService`, die jeweils für den Nutzer wichtige Ergebnisse berechnen. Diese lassen sich sinnvoll und gut testen. Somit erreicht die Code Coverage in diesem Package auch einen Wert von 100% in allen 3 Coverage Kategorien. Um die Code Coverage zu verbessern wären in diesem Projekt Integration Tests sinnvoll. Da sie neben der reinen Funktionalität einzelner Methoden, auch das zusammenspiel der einzelnen Klassen und Objekte prüfen und somit auch prüfen können ob Dinge richtig gesetzt und gelesen werden und ob die Semantik der Prozesse korrekt ist.

5.6 Fakes und Mocks

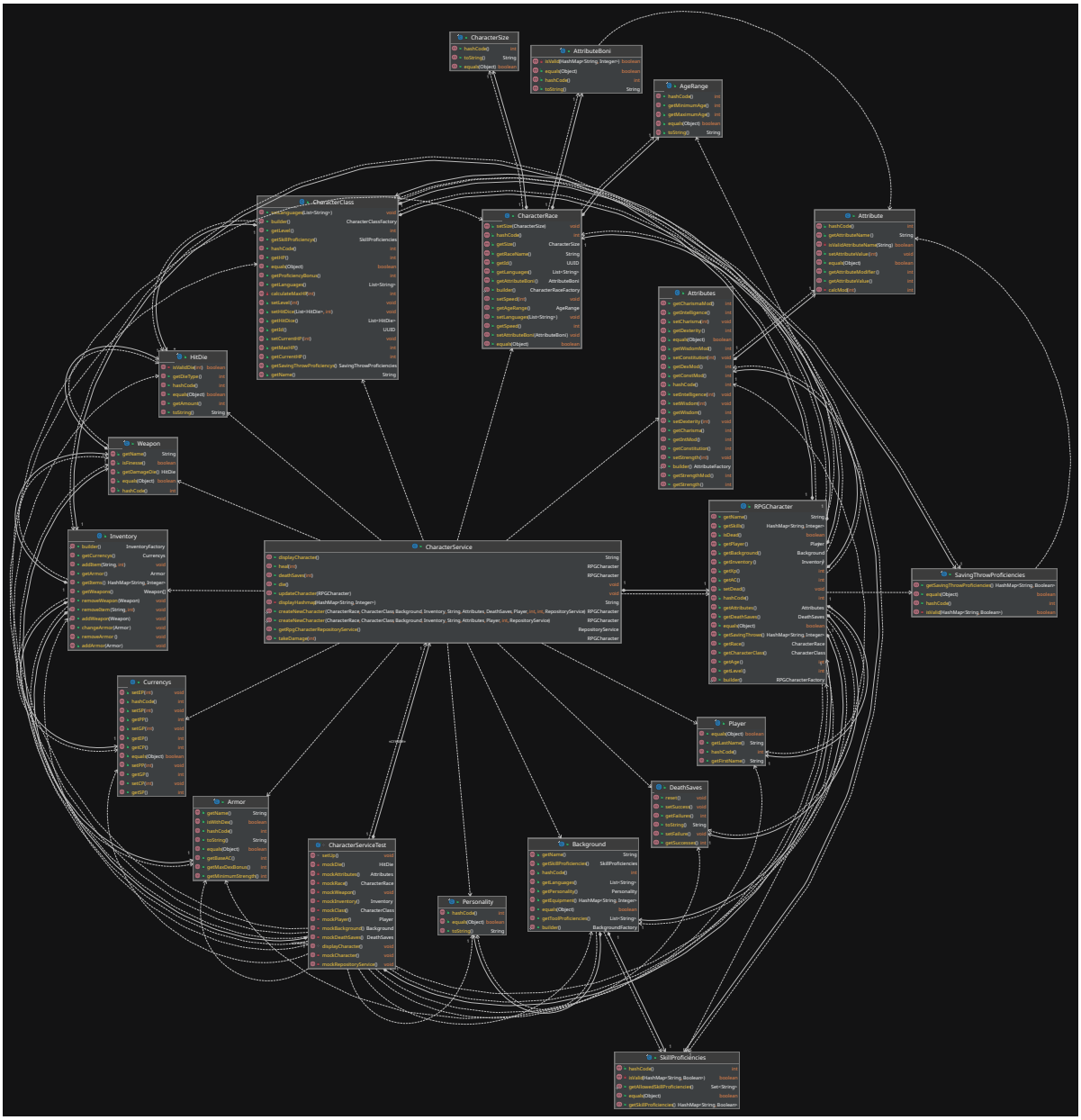


Abbildung 5.2: Abhängigkeiten der CharacterService Klasse

Wie Abbildung 5.2 zu entnehmen ist, hängt die Klasse `CharacterService` von allen Klassen der Domain Schicht ab und hat somit sehr viele Abhängigkeiten. Ein Unit Test soll jedoch möglichst wenige bis gar keine Abhängigkeiten besitzen, um den Test und

die auftretenden Effekte auf einen Ort zu isolieren und zu beschränken. Um dies zu ermöglichen und die Abhängigkeiten der Klassen zu reduzieren, wurde jedes einzelne Objekt, von dem der `CharacterService` abhängig ist durch einen Mock / einen Fake ersetzt. Dementsprechend wurden folgende Klassen ersetzt:

- `HitDie`
- `Weapon`
- `RepositoryService`
- `Player`
- `CharacterClass`
- `Personality`
- `Background`
- `CharacterRace`
- `DeathSaves`
- `Attributes`
- `Currencys`
- `Armor`
- `Inventory`
- `RPGCharacter`

Wären diese Objekte nicht durch Fakes ersetzt worden, hätte dieser Test praktisch das gesamte Projekt getestet. Ein ähnlicher Fall stellt auch der `DiceRollServiceTest` dar, auch wenn dieser weniger Abhängigkeiten besitzt.

6 Domain Driven Design

6.1 Ubiquitous Language

Klasse Die Klasse des Charakters. Sie bestimmt Lebenspunkte, Sprachen, Heilungsmöglichkeiten uvm. Sie hat einen direkten Einfluss auf die Spielweise des Charackters. Sie gehört zur Ubiquitous Language, dar sie einen spezielle Eigenschaft des Charakters beschreibt. Andere könnten unter diesem Begriff vor allem im Programmier Umfeld unterschiedliche Dinge verstehen. Im Projekt heißt sie `CharacterClass`.

Rasse Die Rasse des Charakters. Sie bestimmt Attribut Boni, die Geschwindigkeit, Sprachen, Alter und Größe. Somit hat sie einen großen Einfluss auf die äußerliche Erscheinung des Charakters. Sie ist Teil der Ubiquitous Language, da sie im Domänen Umfeld Einfluss auf viele Dinge hat. Außenstehende würden diesen Begriff sicherlich meist mit negativen Ausdrücken assoziieren oder ihn allein auf die Hautfarbe eines Menschen beschränken. Im Projekt heißt sie `CharacterRace`.

Equipment Die Ausrüstung die ein Charakter mitführt. Dieser Begriff ist Teil der Ubiquitous Language, da es für alle Projektteilnehmer wichtig ist, was ein Charackter mitführen kann. Dies hat unweigerliche Auswirkungen auf Statuswerte, Attackiermöglichkeiten uvm.

HitDice Die Würfel die ein Spieler würfeln kann, um die Lebenspunkte seines Charakters wieder aufzufüllen. Sie sind Teil der Ubiquitous Language, da Außenstehende diesen Begriff warscheinlich nicht kennen.

6.2 Entities

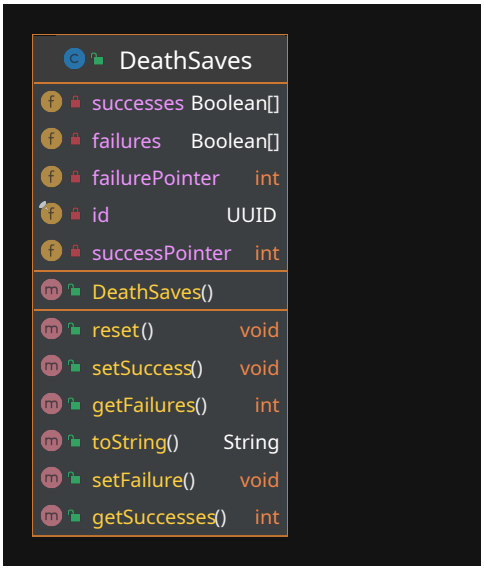


Abbildung 6.1: UML der DeathSaves Klasse

Abbildung 6.1 zeigt eine Entity der Domain Modellierung des DnDCharacterManagers. Die Klasse `DeathSaves` modelliert die Death Saves eines Spielers. Diese kommen zum Einsatz, falls ein Charackter 0 Lebenspunkte hat. Ist dieser erneut am Zug, wird ein Death Save mit einem D20 geworfen. Dieser kann entweder erfolgreich sein oder fehlschlagen. Hat man 3 erfolgreiche Death Saves kommt man mit 1 Lebenspunkt zurück und kann weiter spielen. Hat man 3 Fehlschläge stirbt der Charackter und kommt auf den Friedhof. Diese Klasse stellt eine Entity dar, da sich ihr Inhalt mit dem Laufe der Zeit verändert. So können Successes oder Failures hinzugefügt werden. Nichts desto trotz muss die Instanz der Klasse eindeutig identifizierbar sein und ist Teil der Charackters, unabhängig ihres Inhalts / der Werte ihrer Attribute. Somit stellt sie eindeutig eine Entity dar und wird mittels einer ID eindeutig identifiziert.

6.3 Value Objects

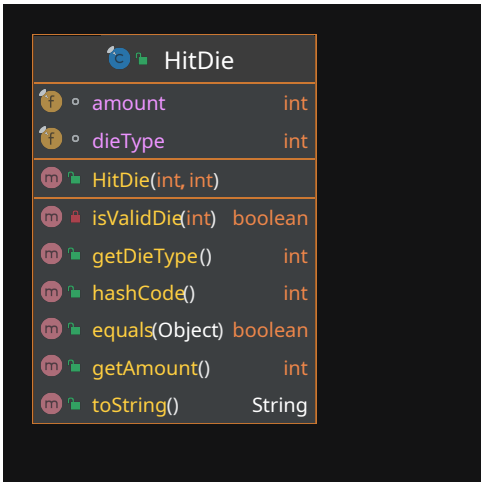


Abbildung 6.2: UML der HitDie Klasse

Abbildung 6.2 zeigt das UML Diagramm der Klasse `HitDie`. Diese Klasse modelliert einen Würfel für den Angriff, zum heilen oder für verschiedene Checks. Sie stellt also sicher, das ausschließlich die in DnD 5e erlaubten Würfel verwendet werden. Da ein Würfel ausschließlich über seine Merkmale wie die Seitenanzahl, oder wie viele Würfel man zur Verfügung hat identifieziert wird, stellt diese Klasse ein Value Object dar. Somit müssen die Attribute `amount` und `dieType` auf den selben Hashwert abbilden, damit eine Instanz der Klasse als äquivalent gilt.

6.4 Repositories

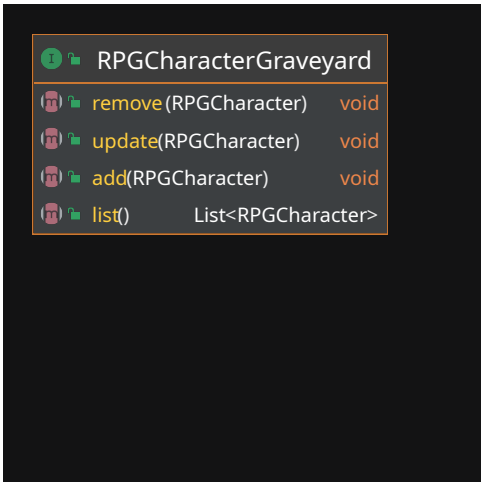


Abbildung 6.3: UML des Repository Interface

Abbildung 6.3 zeigt das UML Klassendiagramm des `RPGCharacterGraveyard` Interfaces. Dieses Interface stellt die Schnittstelle / Implementierungsvorlage für die persistierung des Charakterfriedhofs dar. Da hier Daten persistiert und verwaltet werden bildet dies ein Repository.

6.5 Aggregates

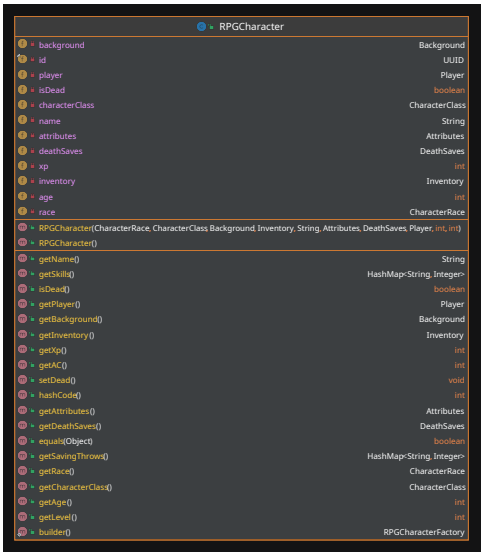


Abbildung 6.4: UML der RPGChracter Klasse

Abbildung 6.4 zeigt das UML Diagramm der Klasse `RPGCharacter`. Sie sammelt alle Eigenschaften eines Charakters und stellt sie in einem Zentralen Objekt zur Verfügung. Da auch diese Klasse unabhängig von ihrem Attributinhalt identifiziert werden muss, um z.b: Repositorys zu updaten, enthält sie eine ID. Da diese Klasse alle Eigenschaften eines Charakters sammelt und zentral zur Verfügung stellt, ist sie ein Aggragate.

7 Refactoring

7.1 Code Smells

```
1     public RPGCharacter(CharacterRace race, CharacterClass ↙
          ↳ characterClass, Background background, Inventory ↙
          ↳ inventory, String name, Attributes attributes, ↙
          ↳ DeathSaves deathSaves, Player player, int xp, int ↙
          ↳ age) throws RPGCharacterException
```

Listing 7.1: Beschreibung eines in einem Bild enthaltenen Objektes mittels des Yolo Datei Formats

Listing 7.1 zeigt den Konstruktor der Klasse `RPGCharakter`. Dieser stellt den CodeSmell Long Parameter List dar. Um diesen zu lösen, müsste man mehrere der bereits zusammengefassten Objekte in weitere Objekte zusammenfassen. So wäre es z.B. möglich die Parameter `race`, `characterClass`, `Background`, `Attributes` in ein neues Objekt namens `CharacterProperties` zusammen zu fassen.

Die gesamte Klasse `InventoryService` besteht aus dead code. Sie wurde im Vorfeld angelegt um Änderungen im Inventory vornehmen zu können. Später wurde der Scope des Projektes jedoch verändert und der Code wird nicht mehr benötigt, wurde aber nicht gelöscht. Dementsprächend wäre es sinnvoll diesen Code nun zu löschen.

7.2 2 Refactorings

7.2.1 Rename Method

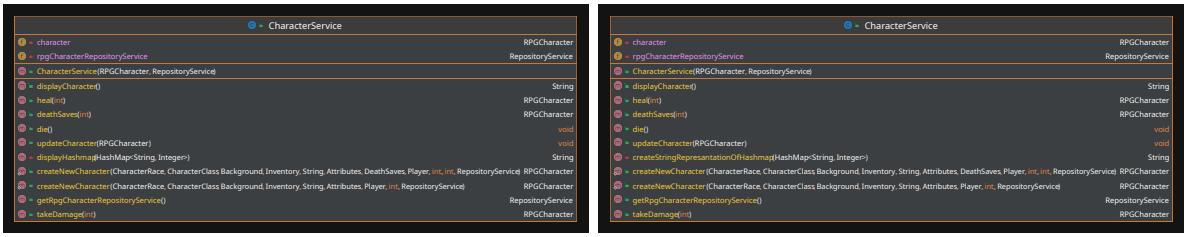


Abbildung 7.1: UML Diagramm des Character Service vor und nach dem Refactoring

Die Methode `displayHashmap` wurde in `createStringRepresantationOfHashmap` umbenannt, da die Methode keine Hashmap darstellt, sondern eine Stringrepräsentation einer übergebenen Hashmap baut und zurück gibt. Die UML Diagramme sind in Abbildung 7.1 dargestellt. Alter commit: https://github.com/lkno0705/DnD-CharacterManager/blob/497c202ebeea dnd-charactermanager-application/src/main/java/character/CharacterService.java. Neuer commit: https://github.com/lkno0705/DnD-CharacterManager/blob/main/2-dnd-charactermanager-application/src/main/java/character/CharacterService.java

7.2.2 Extract Method

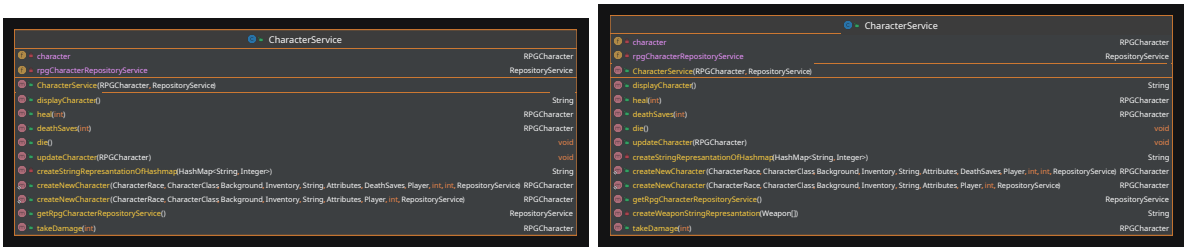


Abbildung 7.2: UML Diagramm des Character Service vor und nach dem Refactoring

In der Methode `displayCharacter`, wurde eine `for-Schleife` extrahiert, die eine String Repräsentation des Waffen arrays des Inventorys aufgebaut hat. Dies dient dazu die recht lange Methode `displayCharacter` weiter zu verkürzen und so ihre lesbarkeit zu

erhöhen. Alter commit: https://github.com/lkno0705/DnD-CharacterManager/blob/main/2-dnd-charactermanager-application/src/main/java/character/CharacterService.java. Neuer commit: https://github.com/lkno0705/DnD-CharacterManager/blob/1469ce1aef5b10f2685ddd448b70dnd-charactermanager-application/src/main/java/character/CharacterService.java

8 Entwurfsmuster

8.1 Entwurfsmuster: Erbauer

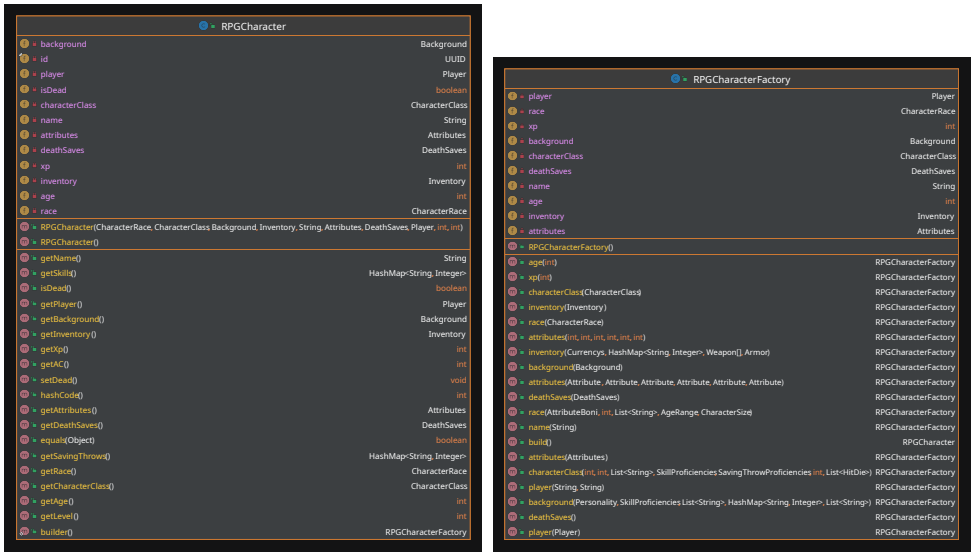


Abbildung 8.1: UML der RPGChracter und der RPGCharacterFactory Klasse

Um komplexe Objekte, wie z.B.: den `RPGCharacter` zu erzeugen, wurde das Erbauer Entwurfsmuster eingesetzt. Dieses ermöglicht es die Objekte Schritt für Schritt auf verschiedene Art und weisen zu Erzeugen und stellt gleichzeitig noch die Korrektheit der Objekte sicher. Abbildung 8.1 zeigt die UML Diagramme der `RPGCharacter` Klasse und des dazugehörigen Erbauers. Dies verhindert das sehr lange Konstruktoren und Parameter auf einmal zur Objekterzeugung verwendet werden müssen und ermöglicht eine einfachere Handhabung der Objekterstellung in höheren Schichten.

8.2 Entwurfsmuster: Stellvertreterle

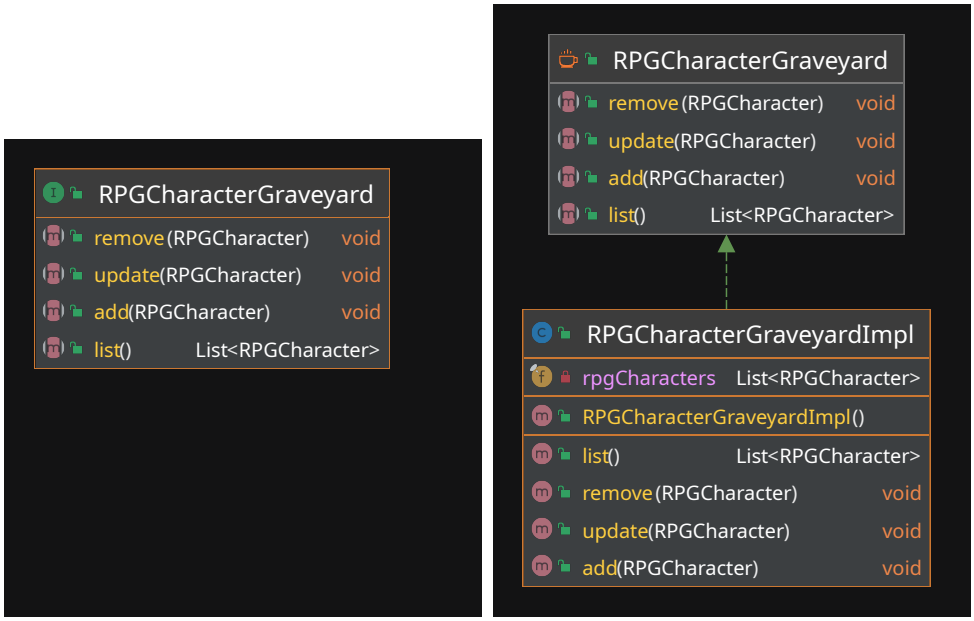


Abbildung 8.2: UML der RPGChracter und der RPGCharacterFactory Klasse

Zur Realisierung der Repositorys wurde das Entwurfsmuster des Stellvertreters verwendet. Abbildung 8.2 zeigt die UML Diagramme des `RPGCCharacterGraveyards` und deren Implementation. Da die Implementation der Repositorys Gerät bzw. Implementationsabhängig ist, wird in den unteren Schichten ausschließlich mit den Interfaces als Stellvertreter für die tatsächliche Implementation gearbeitet. In den äußeren Schichten kann so die tatsächliche Persistierung mittels des Interfaces implementiert werden, die unteren Schichten kennen allerdings nur das Interface und somit ausschließlich den Stellvertreter. Wie die Funktionalität genau realisiert wird ist in den unteren Schichten nicht von relevanz.