Programmentwurf
DnD Charakter Verwaltung

Name: Kreuß, Tobias; Lüdtke Rafael
Martrikelnummer: 9093101; 5293675

Abgabedatum: 24.03.2023

Kapitel 1: Einführung

Übersicht über die Applikation

Die Anwendung dient zum Erstellen und Managen von DnD (5e) Charakteren. Zusätzlich verfügt sie über digitale Würfel und bietet Optionen, *Encounter* und Quests zu erstellen und Charaktere diesen zuzuordnen. Die Anwendung soll DnD-Spieler beim Spielen unterstützen. Hier eine Übersicht über alle verfügbaren Kommandos der Anwendung, bzw. deren Kurzform und was sie bewirken:

- exit - beendet das Programm
- roll - Würfel würfeln
- create_character, cc - Charakter erstellen
- create_encounter, ce -Encounter erstellen
- create_quest, cq - Quest erstellen
- list_characters, lc - alle Charaktere auflisten
- list_encounters, le - alle Encounter auflisten
- list_quests, lq - alle Quests auflisten
- select - Möglichkeit, ohne Name Objekte auszuwählen
- select_character, sc - einen Charakter auswählen
- select_encounter, se - einen Encounter auswählen
- select_quest, sq - eine Quest auswählen
- selected, current, curr - aktuell ausgewählte Objekte anzeigen
- selected_character, selected_char - aktuell ausgewählter Charakter anzeigen
- selected_encounter, selected_enc - aktuell ausgewählter Encounter anzeigen
- selected_quest, selected_que - aktuell ausgewählte Quest anzeigen
- deselect - gesamte Auswahl aufheben
- deselect_character - Characterauswahl aufheben
- deselect_encounter - Encounterauswahl aufheben
- deselect_quest - Questauswahl aufheben
- modify_character, modify_char - ausgewählten Charakter bearbeiten
- modify_encounter, modify_enc - ausgewählten Encounter bearbeiten
- modify_quest, modify_que - ausgewählte Quest bearbeiten
- get_Stat_Character, getC - Informationen zu ausgewähltem Charakter anzeigen
- get_Stat_Encounter, getE - Informationen zu ausgewähltem Encounter anzeigen
- get_Stat_Quest, getQ - Informationen zu ausgewählter Quest anzeigen
- start_encounter, start - einen Encounter starten
- delete_character - ausgewählten Charakter Löschen
- finish_quest - ausgewählte Quest beenden

Wie startet man die Applikation?

JAR-Datei: Java RE benötigt. Im Terminal mit java -jar <Name-der-Datei>

Code: Main-Methode in Main.java ausführen

Wie testet man die Applikation?

Terminal: zum Root-Verzeichnis der Source Dateien navigieren → mvn test

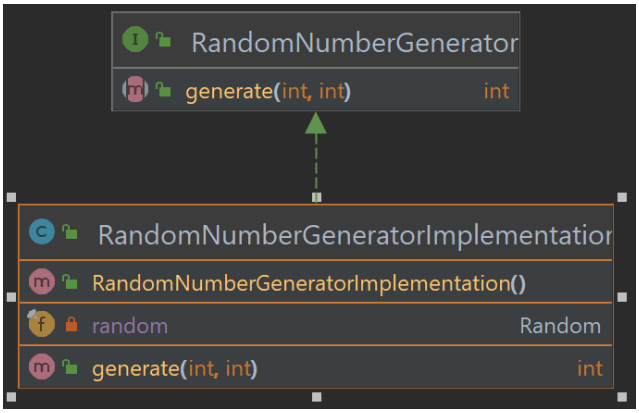
Kapitel 2: Clean Architecture

Was ist Clean Architecture?

Ein Prinzip der Softwareentwicklung, bei dem Bestandteile einer Softwareanwendung in Schichten (Domain, Application, Adapter, Plugin) aufgeteilt werden. Die inneren Schichten sollen nicht von äußeren abhängig sein (Dependency Inversion Principle).

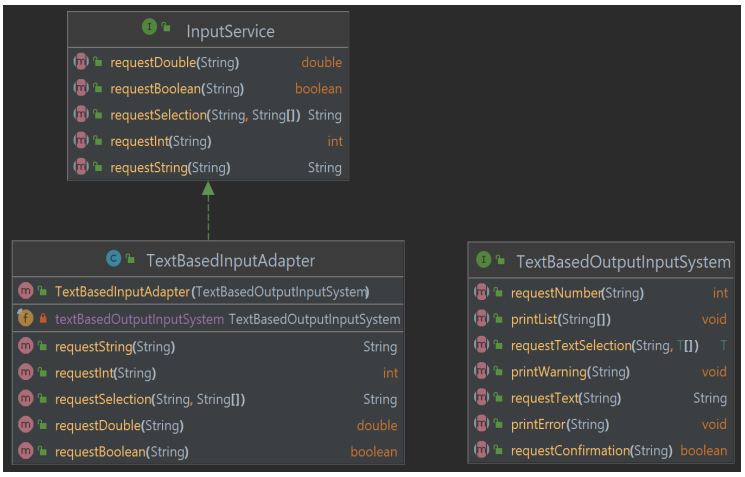
Dies soll ermöglichen, kleine Bestandteile des Codes in äußeren Schichten einfach auszutauschen. Währenddessen können die Kernelemente der Anwendung (Domain Layer) davon unberührt bleiben.

Analyse der Dependency Rule

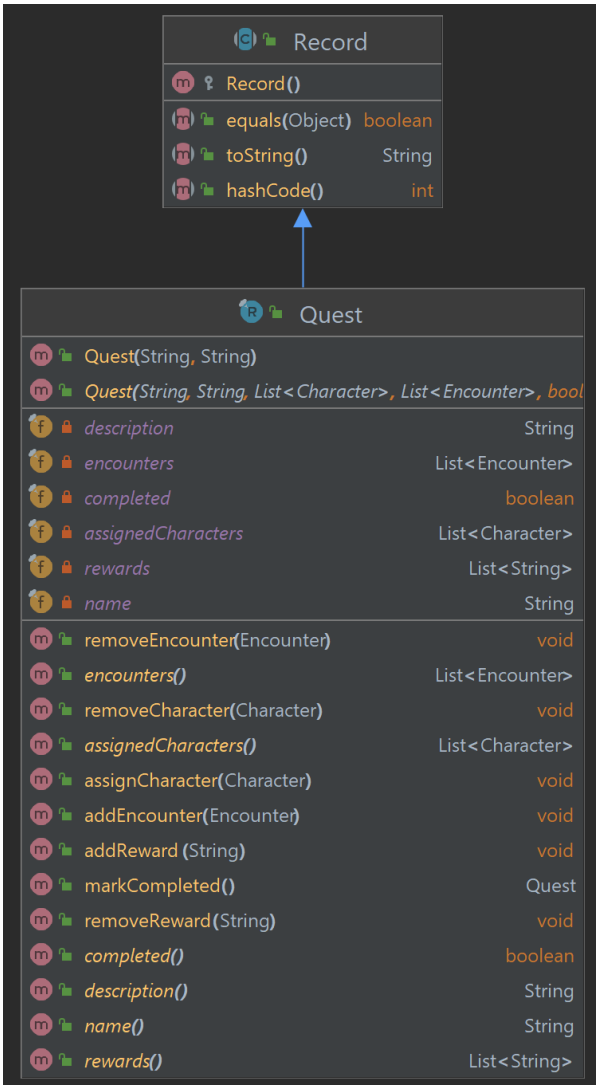


Positiv-Beispiel: Dependency Rule
Die *RandomNumber-GeneratorImplementation* Klasse ist in der Plugin-Schicht und implementiert ein Interface aus der Adapter-Schicht. Die Klasse hängt somit von der inneren Adapter-Schicht ab.

Negativ-Beispiel: Dependency Rule
kein negatives, daher ein zweites Positives:
Die Klasse *TextBasedInputAdapter* gehört zur Adapter-Schicht. Es implementiert das *InputService* Interface aus der Application Schicht. Die Klasse hat ein Attribut, das ein *TextBasedOutputInputSystem* als Datentyp hat. Das Interface fungiert als Schnittstelle zur weiter außen liegenden Plugin-Schicht. Es wird auf diese Weise die Dependency Rule sichergestellt.



Analyse der Schichten

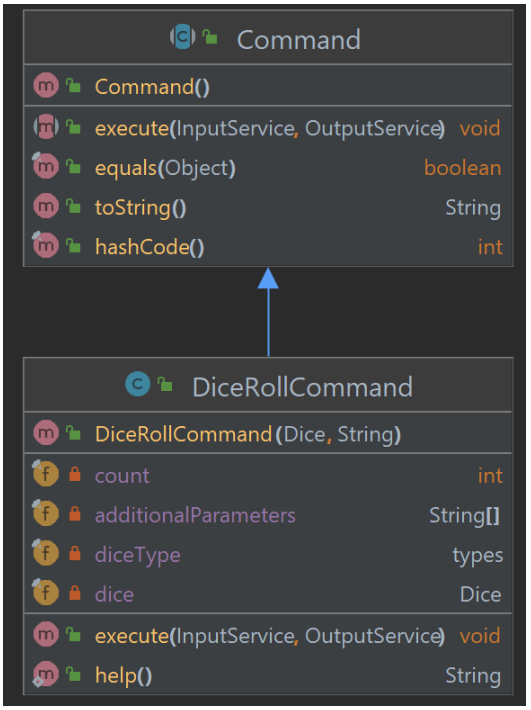


Schicht: Domain

Ein *Quest* Objekt dient dazu, eine Quest abzubilden. Sie besteht aus Charakteren, die sie bestreiten, verschiedenen Encountern, die im Laufe der Quest auftreten können, sowie einem Namen, einer Beschreibung, einer Belohnung und einem Marker, ob sie beendet ist.
Es gehört zur Domänenschicht, da die Klasse etwas aus den Grundregeln der Anwendung implementiert und bei Modernisierung der Anwendung nicht verändert werden muss, ist also sehr langlebig.

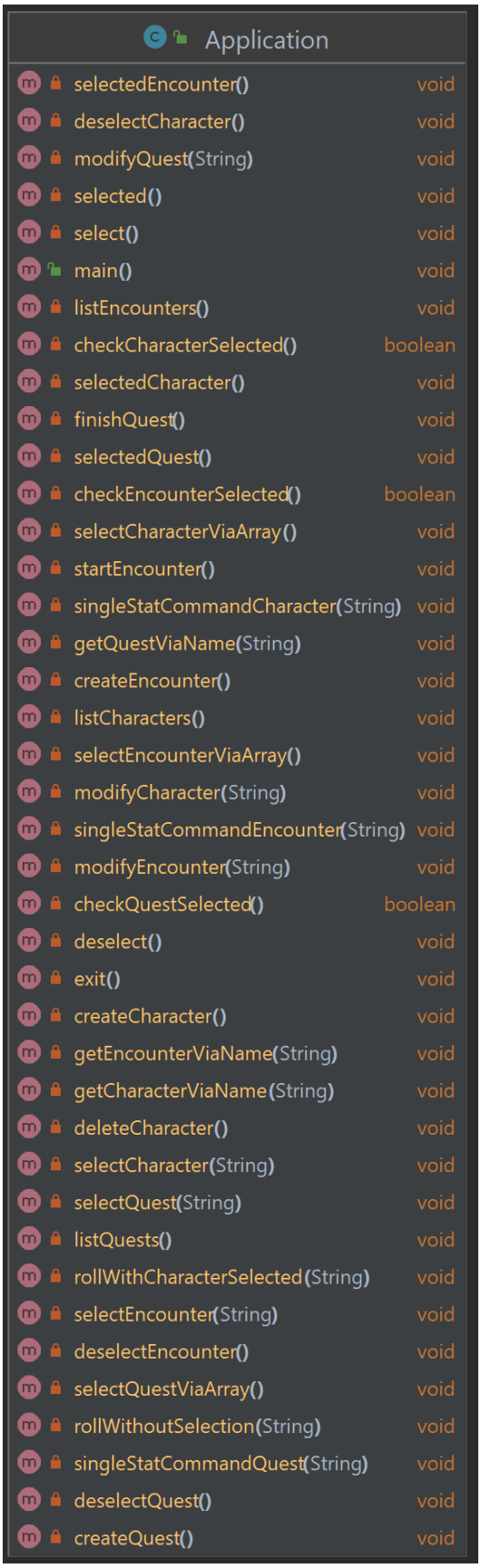
Schicht: Anwendung

Die *DiceRoll* Klasse gehört zur Anwendungsschicht, da sie das spezielle Verhalten der Anwendung implementiert, zum Beispiel, falls der Nutzer würfeln möchte.
Sie steuert weder die Ein-, noch Ausgabe des Nutzers.
Es handelt sich um einen Use Case, den der Nutzer mit der Anwendung erfüllen möchte, der nicht zu den Regeln gehört, die in der Domäne definiert werden.



Kapitel 3: SOLID

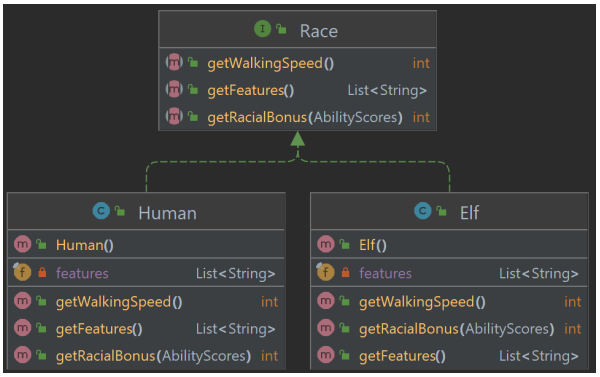
Analyse Single-Responsibility-Principle (SRP)



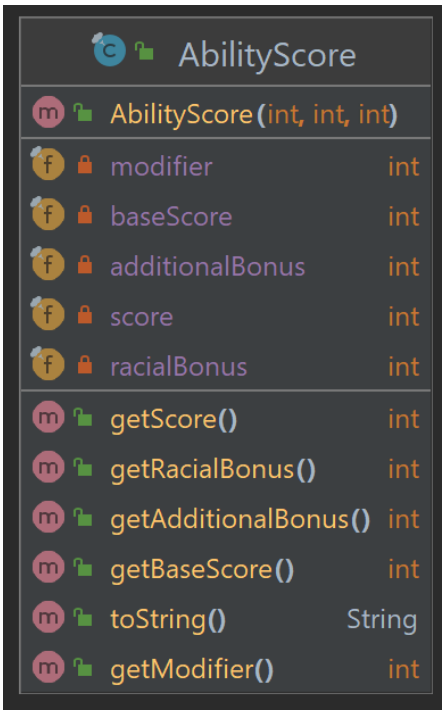
Positiv-Beispiel
Die *Dice* Klasse entspricht dem SRP insofern, als dass sie sich nur darum kümmert, mit Hilfe eines *RandomNumberGenerators* die für DnD benötigten Würfeltypen zu definieren und diese dann im Anwendungsfall zu “würfeln”.

Negativ-Beispiel
Die Klasse *Application* hat die Funktion, die Benutzereingabe dem richtigen *Command* zuzuordnen. Sie verstößt gegen das SRP, da es kleinere Befehle gibt, die direkt in einer Methode dieser Klasse definiert sind (z.B. *deleteCharacter*() oder *selectedCharacter*()).
Auch wenn es sich dabei derzeit nur um einfache Operationen auf inhärente Attribute der Klasse handelt, ist es denkbar, dass diese *Commands* bei einer Weiterentwicklung der Anwendung durchaus komplizierter werden.
Eine Lösung, um die Klasse SRP-konform zu machen, wäre es, diese Methoden ebenfalls in einer eigenen Unterklasse des *Commands* zu implementieren, so dass die Application-Klasse, wie ursprünglich geplant, nur die Zuordnung der Benutzereingaben zur entsprechenden *Command*-Implementierung vornehmen würde. Zusätzlich könnte die Verwaltung aller Charaktere, *Encounter* und *Quests* in Controller Klassen ausgelagert werden, um die Kopplung mit der Applikationsschicht zu entfernen.

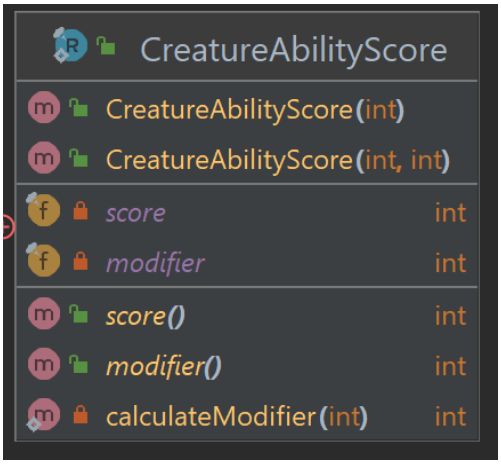
Analyse Open-Closed-Principle (OCP)



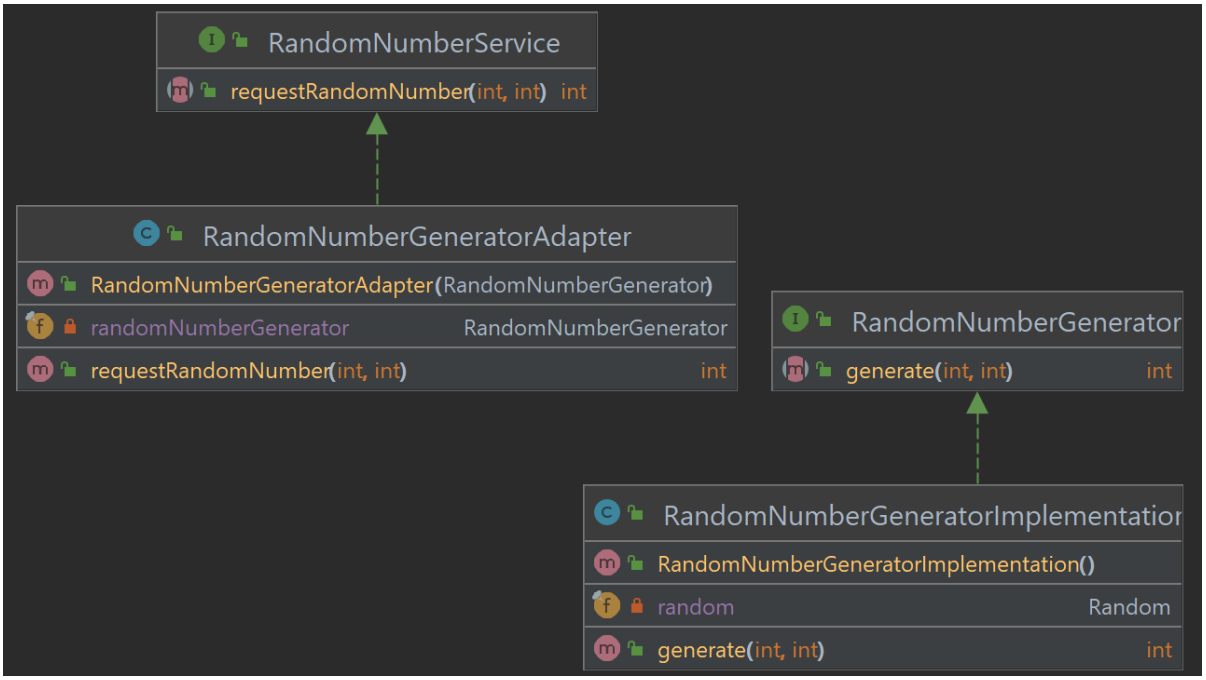
Positiv-Beispiel
Das OCP wird von der Art und Weise, wie die unterschiedlichen Rassen von DnD in der Anwendung implementiert sind, eingehalten. Durch das Interface *Race*, was jede konkrete Rasse implementiert. Dadurch ist es möglich, sehr einfach neue Rassen in die Anwendung hinzuzufügen.



Negativ-Beispiel
Ein Verstoß gegen das OCP ist die Implementation des *AbilityScore* und des *CreatureAbilityScore*. Sie haben beide nahezu die gleiche Funktion- und Arbeitsweise. Nur mit dem Unterschied, dass das eine für Charakter verwendet wird und es damit mehr Regeln hat, die es erfüllen muss. Ändert sich nun jedoch das Regelwerk von DnD, mit direkten Auswirkungen in Bezug auf die Berechnung dieser Werte, muss dies an beiden Stellen geändert werden. Eine Lösung dafür wäre, einen *Base Ability Score* zu implementieren und dann eine spezialisierte Variante für Kreaturen und Charaktere zu verwenden. So würde es dem OCP entsprechen und bei Regeländerung müssten diese nur an einer Stelle eingebracht werden. Zusätzlich könnte man hiermit einfach neue Varianten implementieren.



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



Positiv-Beispiel
Die Klasse *RandomNumberGeneratorAdapter* implementiert das Interface *RandomNumberService* der darüber liegenden Anwendungsschicht. Dies geschieht, um über das DIP sicherzustellen, dass die Art und Weise, wie Zufallszahlen generiert werden, in Zukunft einfach geändert werden kann, ohne dass die Anwendung geändert werden muss.

Negativ-Beispiel, kein negatives, daher ein zweites Positives
Das DIP wird auch in der Klasse *RandomNumberGeneratorImplementation* verwendet. Sie implementiert das *RandomNumberGenerator* Interface, um sicherzustellen, dass der Adapter nicht auf der konkreten Implementierung basiert und diese jederzeit mit geringem Aufwand ausgetauscht werden kann.

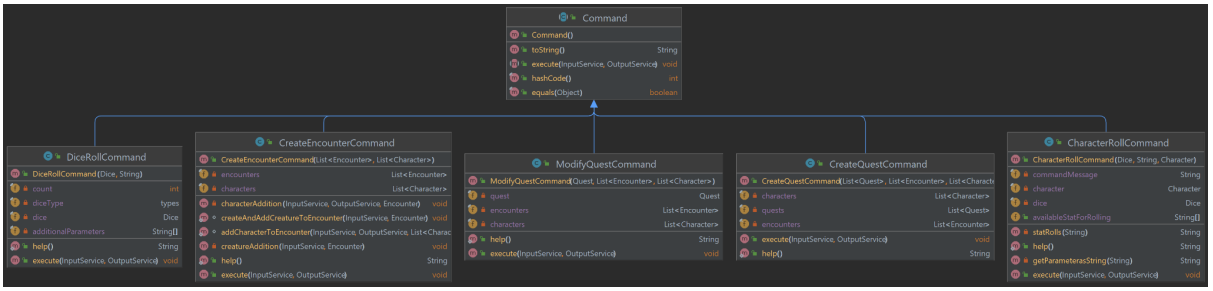
Kapitel 4: Weitere Prinzipien

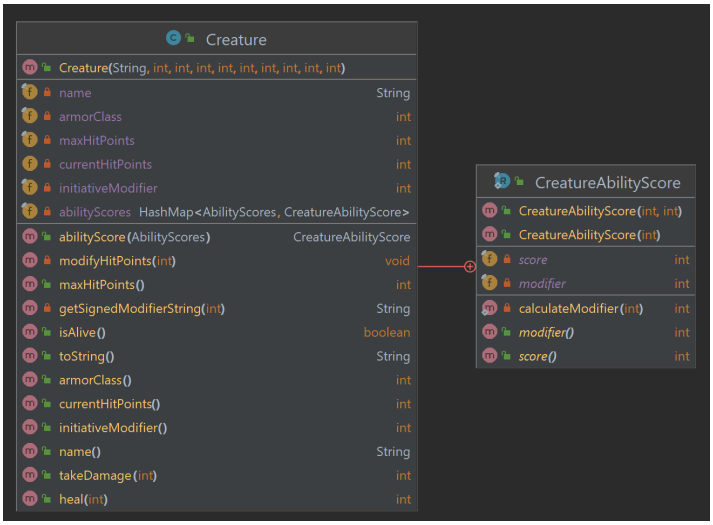
Analyse GRASP: Geringe Kopplung

Positiv-Beispiel

Ein positives Beispiel ist die Verwendung der abstrakten Klasse *Command*.

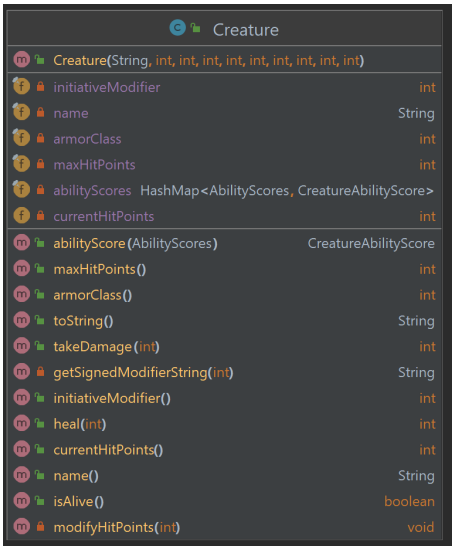
Dadurch, dass alle *Commands* auf sie aufbauen, ist sichergestellt, dass alle *Commands* von der Applikation (im *Application* Layer) mit Hilfe des *CommandControllers* ausgeführt werden können, obwohl sie zutiefst unterschiedliche Funktionen und Implementierungen aufweisen. Zudem kann jeder *Command* so individuell verändert werden, ohne dabei andere zu beeinflussen.





Negativ-Beispiel

Zwischen *Creature* und *CreatureAbilityScore* besteht eine hohe Kopplung. Wenn sich etwas in dem Ability Score verändert, wird das automatisch Auswirkungen auf die Kreaturen haben, da diese ein Kern der Kreaturen sind. Da dies aufgrund der DnD (Domänen) Regel allerdings so gegeben ist, kann das für die Anwendung nicht gelöst werden.



Analyse GRASP: Hohe Kohäsion

Die *Creature* Klasse hat eine hohe Kohäsion, da alle Funktionen, die sie übernimmt, zu einem gesamten Objekt, einer Kreatur, gehören. Sie verwaltet diese lediglich mit einer einzigen Ausnahme:

Die *CreatureAbilityScore* ist ausgelagert, um diese separat im Blick haben zu können. Ansonsten ist alles innerhalb der Klasse und dabei sind die Aufgaben der Methode klar definiert.

Don’t Repeat Yourself (DRY)

Im Commit wurde doppelter Code aufgelöst.

Vorher: doppelter Code sowohl in *CreateEncounterCommand* als auch in *ModifyEncounterCommand*:

```
String[] remainingCharacters = characters.stream()
                            .filter(character ->
!encounter.getPlayerCharacters().contains(character))
                            .map(Character::getName)
                            .toArray(String[]::new);
                    if (remainingCharacters.length == 0) {
                        output.displayMessage("There are no characters left to
add.");
                        return;
                    }
                    String name = input.requestSelection("Which character do you
want to add?", remainingCharacters);

encounter.addPlayerCharacter(characters.stream().filter(character ->
character.getName().equals(name)).findFirst().orElse(null));
```

Nachher: Code als statische Methode in *CreateEncounterCommand*:

```
static void addCharacterToEncounter(InputService input, OutputService
output, List<Character> characters, Encounter encounter) {
        String[] remainingCharacters = characters.stream()
                .filter(character ->
!encounter.getPlayerCharacters().contains(character))
                .map(Character::getName)
                .toArray(String[]::new);
        if (remainingCharacters.length == 0) {
            output.displayMessage("There are no characters left to
add.");
            return;
        }
        String name = input.requestSelection("Which character do you
want to add?", remainingCharacters);

encounter.addPlayerCharacter(characters.stream().filter(character ->
character.getName().equals(name)).findFirst().orElse(null));
    }
```

und in beiden Command Implementationen aufgerufen.

```
addCharacterToEncounter(input, output, characters, encounter);
```

Dies wurde durchgeführt, um sicherzustellen, dass beim Ändern der Art, wie ein *Charakter* einem *Encounter* hinzugefügt wird, nicht beide *Command* Implementierung geändert werden müssen.

Kapitel 5: Unit Tests

10 Unit Tests

Unit Test	Beschreibung
AbilityScoreTest# modifierTest()	Testet, ob die Modifier für die entsprechenden Fähigkeitswerte richtig berechnet werden
AbilityScoreTest# lowerBaseBoundaryTest()	Prüft, ob der Basiswert für Fähigkeiten bei Spielern, dass das Minimum von 3 nicht unterschreiten kann
AbilityScoreTest #totalBoundaryTest()	Stellt sicher, dass ein Fähigkeitswert eines Spielers nicht das Maximum von 20 überschreitet
CharacterTest# testHeal()	Testet, ob die heal(int) Methode des Charakters funktioniert und nicht über die maximalen Lebenspunkte hinaus Punkte wiederherstellt.
CharacterTest# testDamageTaken()	Testet, ob die damageTaken(int) Methode die Lebenspunkte abzieht und ein Charakter nicht weniger als 0 Lebenspunkte haben kann.
CharacterTest# testSavingThrowModifier()	Testet, ob Saving Throw Modifier richtig bestimmt werden, gerade in Bezug auf Proficiency durch die Klasse des Charakters
CharacterTest# testFlatAbilityModifier()	Prüft die Funktion einem Charakter einen festen AbilityScore zu geben, ohne dass sonstige Boni addiert werden. Dies kann z.B. durch Items geschehen.
CharacterTest# testEquipment()	Testet die Funktion, einem Charakter Items zu geben und zu nehmen. Gerade in Bezug auf die automatische Sortierung der Liste.

CreatureTest# testIsAlive()	Testet, ob die Methode *false* zurückgibt, nachdem eine Kreatur tödlichen Schaden bekommen hat.
ConsoleInputOutputTest# testRequestTextSelection()	Prüft auf richtige Funktionsweise der RequestTextSelection Funktion des TextBasedOutputInputSystem Interfaces, was durch ConsoleOutputInputSystem implementiert wird.

ATRIP: Automatic

Durch Maven (surefire-plugin) werden alle Tests bei jedem Maven Befehl automatisch ausgeführt.

ATRIP: Thorough

Positiv:

```
@Test
    public void testQuestAssignedCharacters() {
        assertEquals(0, testQuest.assignedCharacters().size());
        testQuest.assignCharacter(testCharacter);
        assertEquals(1, testQuest.assignedCharacters().size());
        assertEquals(testCharacter,
testQuest.assignedCharacters().get(0));
        testQuest.removeCharacter(testCharacter);
        assertEquals(0, testQuest.assignedCharacters().size());
    }
```

Auch wenn der Test aus der *QuestTest* Klasse nur einen Getter / Setter überprüft, war es dennoch wichtig für uns, diese hier zu überprüfen, da wir das erste Mal einen record in Java verwendet haben, und es daher für uns notwendig war, *Quests* auf richtige Funktionsweise zu überprüfen.

Negativ:

```
    void testEnqueueCommand() {
        Command command = new Command() {
            @Override
            public void execute(InputService input, OutputService
output) {
                output.displayMessage("Test");
            }
        };
        controller.enqueueCommand(command);
```

```
        Queue<Command> commandQueue = controller.getCommandQueue();
        assertEquals(1, commandQueue.size());
        assertEquals(command, commandQueue.peek());
    }
```

Der Test *testEnqueueCommand* der Klasse *CommandControllerTest* steht hier eher stellvertretend dafür, dass die Anwendungsschicht in unserer Anwendung fast keine Testabdeckung hat. Dies ist unprofessionell, da es sich um einen großen Teil der Anwendung und gleichzeitig um eine kritische Stelle im System handelt, die nicht durch Tests abgedeckt wird.

ATRIP: Professional

Positiv:

```
public class ConsoleInputOutputTest {
    private final ByteArrayOutputStream outputStream = new
ByteArrayOutputStream();
    private ConsoleOutputInputSystem console;

    @BeforeEach
    void setUp() {
        console = new ConsoleOutputInputSystem(outputStream, System.in);
    }
...
```

Dies ist ein Beispiel für professionelles Testen nach ATRIP, da der parametrisierte Konstruktor für das *ConsoleOutputInputSystem* nur zum Testen benötigt wird, da die Klasse in der normalen Anwendung direkt mit System.in und System.out initialisiert wird.

Negativ:

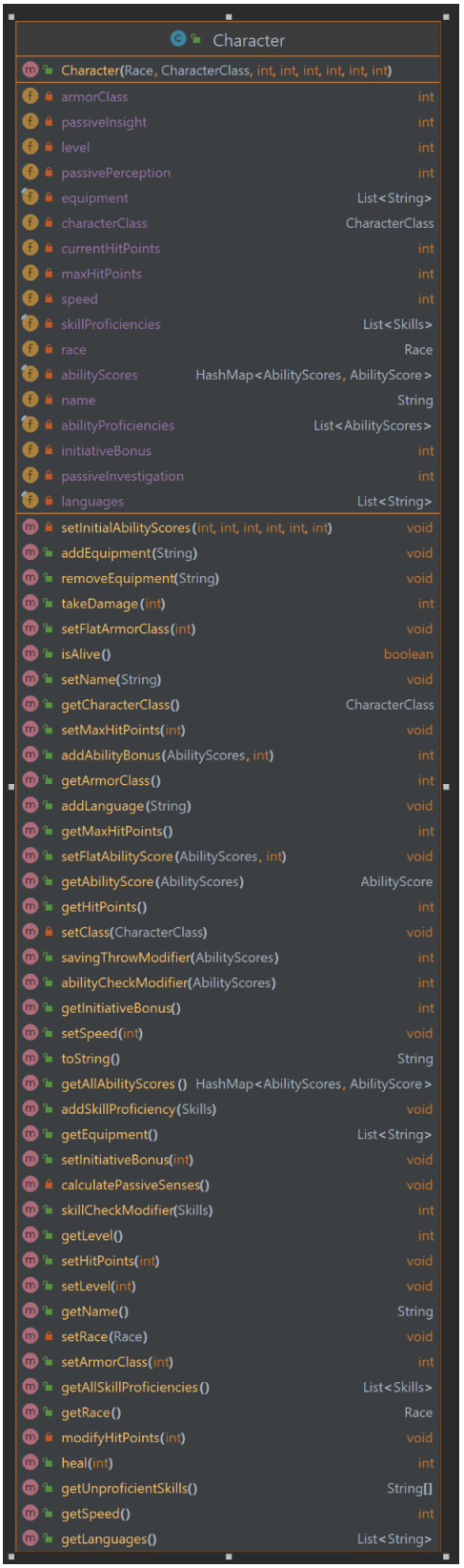
```
public class CharacterTest {
...
    @Test
    public void testSpeed() {
        assertEquals(30, character.getSpeed());
    }
...
```

Es handelt sich hier um einen unprofessionellen Test, da das gegebene Beispiel nur einen Getter testet, was in der Regel als unnötig angesehen werden kann.

Code Coverage

Die Codeabdeckung im Projekt beträgt 30%. Der Hauptfokus des Testens lag auf der Domäne, da dieser Code langlebig ist und die zugrundeliegende Regel für die gesamte Anwendung implementiert. Hier liegt die Abdeckung bei 66%. Dies scheint auf den ersten Blick nicht sehr umfassend zu sein, jedoch wurden nicht alle Getter und Setter mit Tests abgedeckt und aufgrund der Vielzahl an Charakterklassen und Rassen, die die Anwendung implementiert, ist es nicht notwendig, jede einzelne Methode und jede mögliche Variante mit Tests abzudecken.

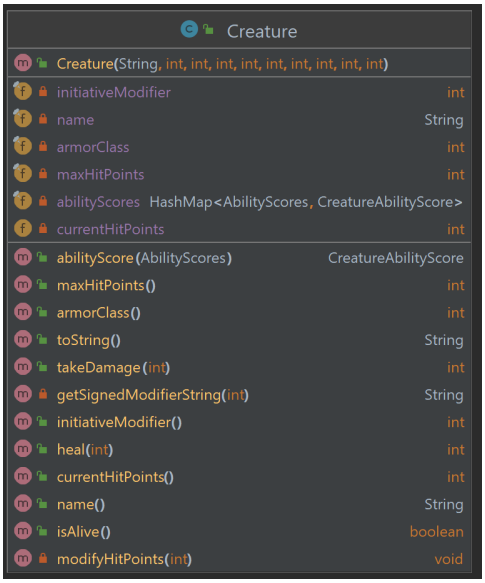
Eine große Anzahl von Codezeilen ist in der Anwendungsschicht enthalten. Da hier sehr viel Benutzerinteraktion stattfindet und es schwieriger ist, hier gut gestaltete Tests zu schreiben, wird hier der Ansatz verfolgt, Tests zu schreiben, wenn ein Fehler auftritt, um ihn nach der Behebung für alle folgenden Versionen zu schließen. Dieser Ansatz führt jedoch, wie hier in einer frühen Version, zu einer geringeren Abdeckung.



Fakes und Mocks

Um die Implementierung von Kreaturen oder Charakteren testen zu können, wird ein Objekt dieser Klassen benötigt. Da beim Testen ohnehin nicht auf reale Daten aus der Nutzung der Anwendung zugegriffen werden kann, werden dafür Mock-Objekte benötigt. Diese verhalten sich wie “echte” Instanzen, sind aber in ihrem Funktionsumfang auf ein für die Tests reduziertes Minimum beschränkt. Mock-Objekte sind daher im Produktivbetrieb in dieser Form nicht in der Anwendung zu finden.

Am Beispiel des Charakters Test kann man sehen, dass sich die dafür erstellte Charta bis auf wenige Ausnahmen auf das Minimum der benötigten Parameter beschränkt.

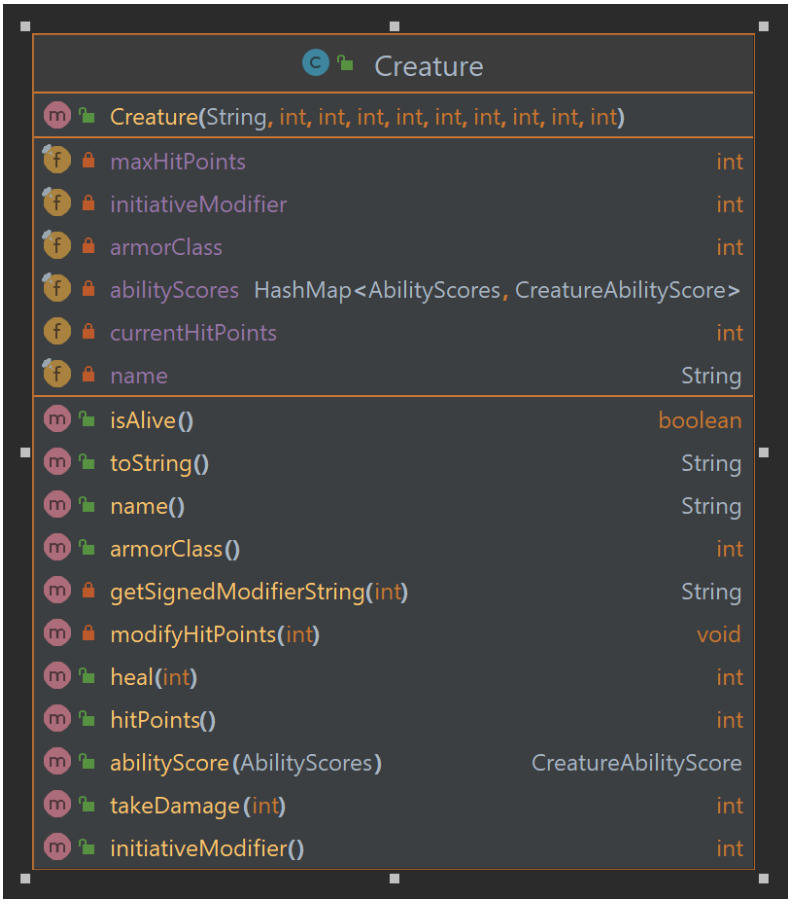


Kapitel 6: Domain Driven Design

Ubiquitous Language

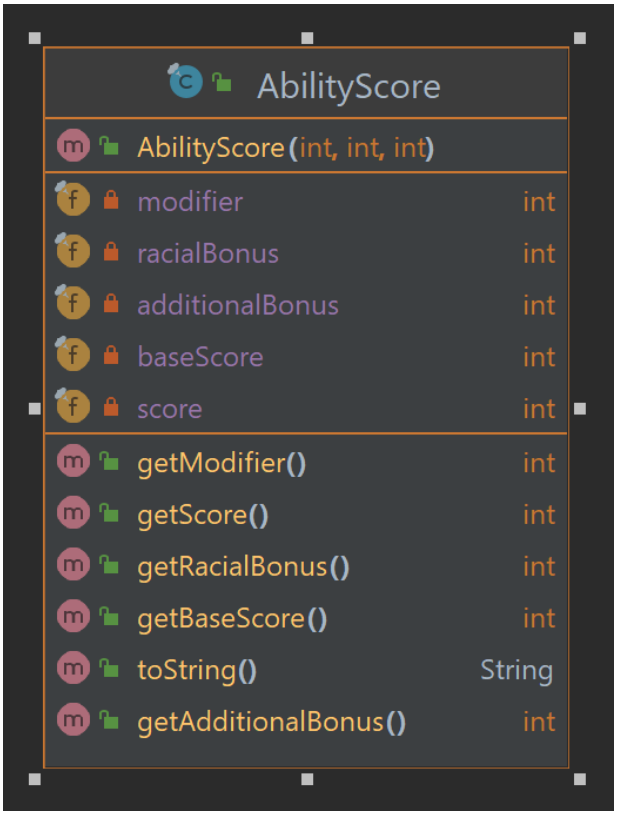
Bezeichung	Bedeutung	Begründung
Ability (-score) / Fähigkeit(-swert)	Einer der 6 Attribute eines DnD Charakters (Stärke, Geschicklichkeit, Konstitution, Intelligenz, Weisheit und Charisma). Dabei befinden sich die Werte für Spieler im Bereich von 3 bis 20 und bestimmen einen Modifikator, der auf Würfelwürfe hinzuaddiert wird.	Da es nahezu keinen Würfelwurf in DnD gibt, der von keinem der Fertigkeitswerte bzw. den sich daraus bestimmten Modifikatoren abhängt, ist es wichtig, diese Werte zusätzlich festzuhalten.
Hitdice / Trefferwürfel	Würfelart, die der Charakterklasse zugeordnet sind. Diese können unter bestimmten Voraussetzungen zum Heilen eingesetzt werden und werden bei einem Rangaufstieg gewürfelt, um festzustellen, wie viel sich die maximalen Trefferpunkte erhöhen.	Da sie zur Berechnung der Lebenspunkte essentiell sind, ist es wichtig, dass alle Beteiligten wissen, was die Trefferwürfel machen.
Saving throw / Rettungswurf	Ein W20 Wurf (also ein Wurf eines Würfels mit 20 Seiten), der für den Versuch steht, einem Zauber, einer Falle, einem Gift, einer Krankheit oder einer ähnlichen Bedrohung zu widerstehen.	Da Rettungswürfe nicht nur einfach Würfe auf eine gewisse Fähigkeit sind, sondern besondere Regeln dafür gelten, ist es wichtig, den Unterschied zwischen “normalen” Würfen und Rettungswürfen zu kennen.
Proficiency / Übung	Charaktere können in mehreren Fähigkeiten oder mit verschiedensten Ausrüstungsgegenständen Übung haben. Je nach Rang des Charakters wird dann bei einem W20 Wurf ein Übungsbonus zum Ergebnis addiert.	Der Begriff ist im Regelwerk weit verbreitet und die Kenntnis über ihn stellt daher sicher, dass alle Beteiligten das Konzept dahinter verstehen.

Entities



Ein *Creature* Objekt repräsentiert eine Kreatur von DnD. Es hat mehrere Attribute, die durch die Zugehörigkeit zu dieser Creature definiert sind. Ein Entity wird hier verwendet, da eine Creature ihre Lebenspunkte verwaltet und ihre Attribute speichert.

Value Objects

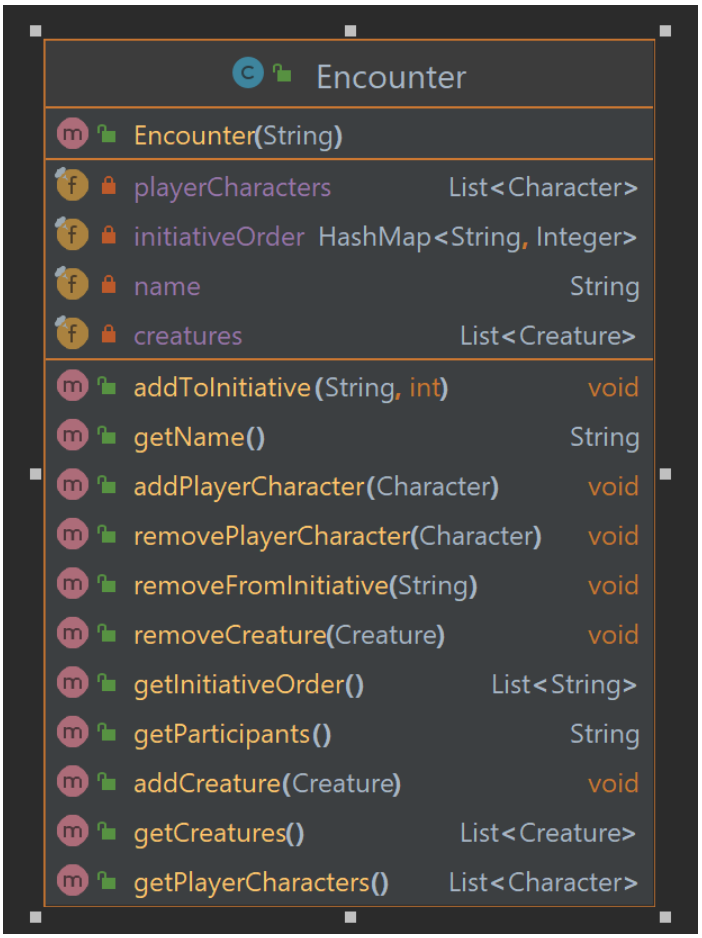


Um die Regeln, die an die einzelnen Fähigkeitswerte (*AbilityScore)* eines *Charakters* geknüpft sind, einzuhalten, eignet sich ein Value Object. So wird für jede Änderung des Wertes ein neues Objekt erzeugt, wobei die Gültigkeit der Regel überprüft wird. Für die verschiedenen Fertigkeiten gelten die gleichen Regeln, deshalb kann in diesem Fall ein Value Object verwendet werden.

Repositories

Die Verwendung eines Repositories ist derzeit nicht notwendig, da die Anwendung derzeit nur Daten im Hauptspeicher ablegt und somit keine Repräsentation der Objekte für z.B. eine Datenbank benötigt. Die Speicherung von Objekten außerhalb des Arbeitsspeichers ist eine sehr gut vorstellbare Erweiterung und Verbesserung der Anwendung, die die Verwendung eines Repositories mit sich bringen würde.

Aggregates



Ein *Encounter* ist eine in sich geschlossene Kampfhandlung in DnD. An ihr nehmen sowohl Charaktere als auch gegnerische Kreaturen teil. Um ein Chaos zu vermeiden, wird der Kampf rundenbasiert ausgetragen. Jede beteiligte Partei (Charakter, Kreatur oder manchmal auch der Ort) würfelt Initiative. Die Partei mit dem höchsten Wert beginnt und von dort aus geht es in absteigender Reihenfolge weiter. Bis zum Ende der Liste, dann geht es wieder von oben los.

Ein *Encounter* Objekt speichert alle gängigen Beteiligten in sich, um diese initiale Reihenfolge einfach erstellen zu können. Da sowohl Charaktere als auch Kreaturen eigenständige Entitäten sind, sorgt das Aggregat hier für eine kontextbasierte Verknüpfung dieser, um die Kohärenz innerhalb einer Kampfhandlung zu wahren.

Kapitel 7: Refactoring

Code Smells

Codesmell: Große Klasse: (Code im Anhang oder hier Link zum Code)

Die *Charakter*-Klasse ist sehr groß. Eine Möglichkeit dem entgegenzuwirken, wäre sie in kleinere Unterklassen zu zerteilen. So könnte man z.B. die Verwaltung der Lebenspunkte herausnehmen und bei der Kreatur wiederverwenden, oder alle Funktionen, die zur Verwaltung der Fähigkeiten gehören. Dies könnte z.B. so aussehen:

```
public class Character{
private abilityscoreController abilitys= ...
private hpController hpContoller= ....
private String name;
....
```

Codesmell: Lange Methode (Code im Anhang oder hier Link zum Code)

Die Methode *execute*() in *ModifyCharacterCommand* ist mit über 60 Zeilen sehr lang. Eine Möglichkeit, dieses Problem zu lösen, wäre, die einzelnen Fälle aus dem switch-Statement in jeweils eine eigene Methode zu extrahieren. Dadurch würde die Methode deutlich kürzer, was die Lesbarkeit verbessern würde. Pseudocode dafür:

```
switch (toModify) {
            case "Add equipment" -> addEquipment(output, character);
            case "Remove equipment" -> removeEquipment(output,
character);
            case "Add language" -> addLanguage(output, character);
            case "Add a skill proficiency" ->
addSkillProficiency(output, character);
            case "Set speed" -> setSpeed(output, character);
            case "Set armor class" -> setArmorClass(output, character);
            case "Set flat armor class" -> setFlatAC(output, character);
            case "Set initiative bonus" -> setInitiativBonus(output,
character);
            case "Set current hit points" -> setCurrentHP(output,
character);
                  case ""Set maximum hit points" ->  setMaxHP(output,
character);
            case "Set level" -> setLevel(output, character);
            case "Add ability bonus" -> setAblilityBonus(output,
character);
        }
    }
```

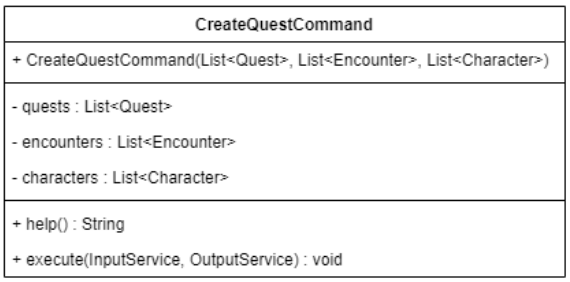
2 Refactorings

Vorher:

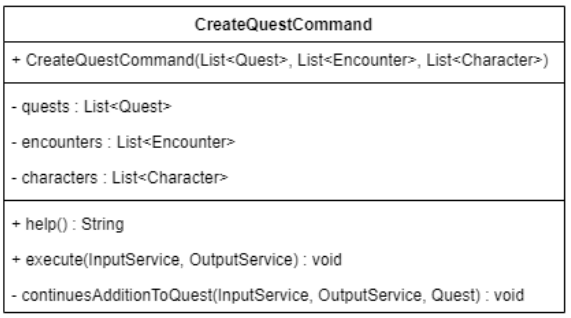
Nachher:

Die Methode *hitPoints*() in *currentHitPoints*() umbenannt, um widerzuspiegeln, was die Methode zurückgibt, da es current- und max-Hitpoints gibt. Commit

Vorher:



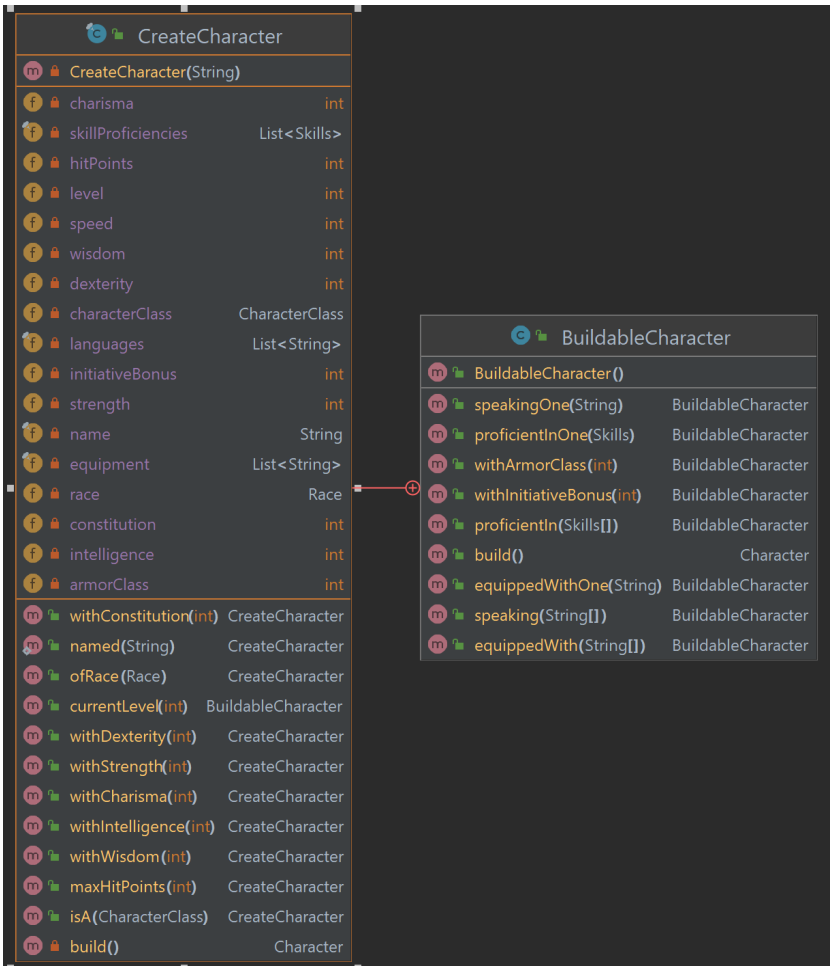
Nachher:



Die Methode *continuesAdditionToQuest*() aus der *Execute* Methode der *CreateQuestCommand* Klasse extrahiert, um die Klasse besser lesbar zu gestalten. Commit

Kapitel 8: Entwurfsmuster

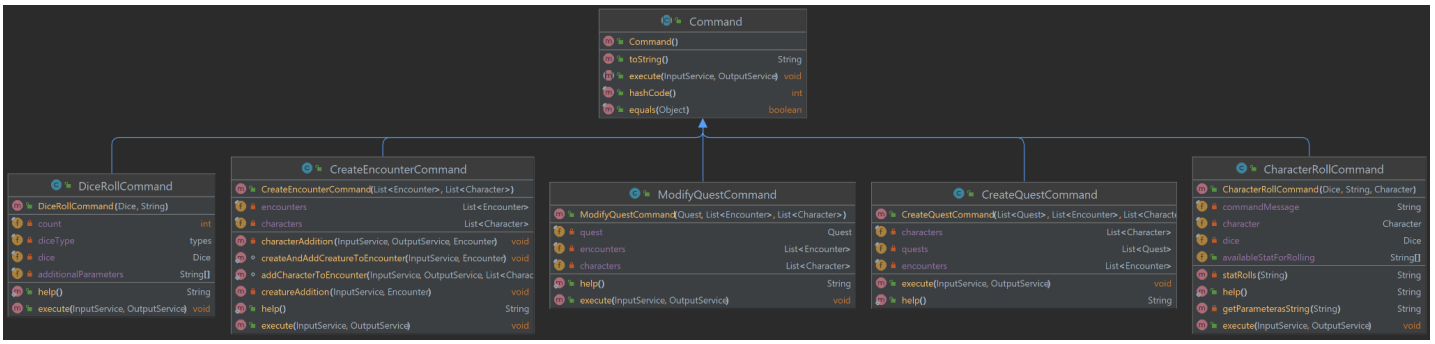
Entwurfsmuster: Builder



Da die Erstellung eines DnD-Charakters relativ komplex ist und das Regelwerk eher einen Rahmen vorgibt, ist es sinnvoll, für die Erstellung einen Erbauer zu verwenden. So ist es möglich, die komplexe Erstellung eines Charakters von der Darstellung zu trennen und eine bessere Lesbarkeit bei der Erstellung zu haben.

Entwurfsmuster: Strategie

Der *CommandController* arbeitet nur mit der abstrakten Klasse *Command*. Jeder *Command* führt etwas anderes aus, aber alle werden auf die gleiche Weise aufgerufen und ausgeführt. Durch die Verwendung dieser Strategie ist eine einfache Erweiterbarkeit um weitere *Commands* möglich und diese können sich nicht gegenseitig behindern.



Kapitel 9: Anhang

ModifyCharacterCommand execute Methode:

```
    @Override
    public void execute(InputService input, OutputService output) {
        String toModify = input.requestSelection("What do you want to
modify?", availableOptions);

        switch (toModify) {
            case "Add equipment" -> {
                String equipment = input.requestString("What equipment
do you want to add?");
                character.addEquipment(equipment);
                output.displayMessage("You added " + equipment + " to
your equipment.");
            }
            case "Remove equipment" -> {
                String equipment = input.requestString("What equipment
do you want to remove?");
                character.removeEquipment(equipment);
                output.displayMessage("You removed " + equipment + "
from your equipment.");
            }
            case "Add language" -> {
                String language = input.requestString("What language do
you want to add?");
                character.addLanguage(language);
                output.displayMessage("You added " + language + " to
your languages.");
            }
            case "Add a skill proficiency" -> {
                String[] unproficientSkills =
character.getUnproficientSkills();
                String skill = input.requestSelection("What skill do you
want to add?", unproficientSkills);
                character.addSkillProficiency(Skills.valueOf(skill));
                output.displayMessage("You added " + skill + " to your
skill proficiencies.");
            }
            case "Set speed" -> {
                int speed = input.requestInt("What speed do you want to
set?");
                character.setSpeed(speed);
                output.displayMessage("You set your speed to " + speed +
```

```
".");
            }
            case "Set armor class" -> {
                int armorClass = input.requestInt("What armor class has
the armor that the character wears? (Dex will be added)");
                output.displayMessage("The new armor class is " +
character.setArmorClass(armorClass) + ".");
            }
            case "Set flat armor class" -> {
                int flatArmorClass = input.requestInt("What flat armor
class do you want to set?");
                character.setFlatArmorClass(flatArmorClass);
                output.displayMessage("You set your flat armor class to
" + flatArmorClass + ".");
            }
            case "Set initiative bonus" -> {
                int initiativeBonus = input.requestInt("What initiative
bonus do you want to set?");
                character.setInitiativeBonus(initiativeBonus);
                output.displayMessage("You set your initiative bonus to
" + initiativeBonus + ".");
            }
            case "Set current hit points" -> {
                int hitPoints = input.requestInt("How many hit points
does the character have currently?");
                character.setCurrentHitPoints(hitPoints);
                output.displayMessage("You set your hit points to " +
hitPoints + ".");
            }
            case "Set maximum hit points" -> {
                int hitPoints = input.requestInt("How many hit points
does the character have maximum?");
                character.setMaxHitPoints(hitPoints);
                output.displayMessage("You set your hit points to " +
hitPoints + ".");
            }
            case "Set level" -> {
                int level = input.requestInt("What level do you want to
set?");
                character.setLevel(level);
                output.displayMessage("You set your level to " + level +
".");
            }
            case "Add ability bonus" -> {
                String[] abilities = new String[]{"Strength",
"Dexterity", "Constitution", "Intelligence", "Wisdom", "Charisma"};
```

```
                String ability = input.requestSelection("What ability do
you want to add?", abilities);
                int bonus = input.requestInt("What bonus do you want to
add?");
                output.displayMessage("You added " + bonus + " to your "
+ ability + " for a new total of: " +

character.addAbilityBonus(AbilityScores.valueOf(ability.toUpperCase()),
bonus) + ".");
            }
        }
    }
```

Character Klasse

```
public class Character {
    private String name;
    private Race race;
    private CharacterClass characterClass;

    private int level;
    private int armorClass;
    private int initiativeBonus;
    private int speed;
    private int maxHitPoints;
    private int currentHitPoints;

    private int passivePerception;
    private int passiveInvestigation;
    private int passiveInsight;

    private final List<AbilityScores> abilityProficiencies = new
ArrayList<>();
    private final List<Skills> skillProficiencies = new ArrayList<>();

    private final List<String> languages = new ArrayList<>();
    private final List<String> equipment = new ArrayList<>();
    private final HashMap<AbilityScores, AbilityScore> abilityScores =
new HashMap<>();

    public Character(Race race, CharacterClass characterClass, int
strength, int dexterity, int constitution, int intelligence, int wisdom,
int charisma) {
        setRace(race);
        setClass(characterClass);
```

```
        this.characterClass = characterClass;
        this.languages.add("Common");
        setInitialAbilityScores(strength, dexterity, constitution,
intelligence, wisdom, charisma);
        calculatePassiveSenses();
        setArmorClass(10);
    }

    private void setInitialAbilityScores(int baseStrength, int
baseDexterity, int baseConstitution, int baseIntelligence, int
baseWisdom, int baseCharisma) {
        this.abilityScores.put(STRENGTH, new AbilityScore(baseStrength,
this.race.getRacialBonus(STRENGTH), 0));
        this.abilityScores.put(DEXTERITY, new
AbilityScore(baseDexterity, this.race.getRacialBonus(DEXTERITY), 0));
        this.abilityScores.put(CONSTITUTION, new
AbilityScore(baseConstitution, this.race.getRacialBonus(CONSTITUTION),
0));
        this.abilityScores.put(INTELLIGENCE, new
AbilityScore(baseIntelligence, this.race.getRacialBonus(INTELLIGENCE),
0));
        this.abilityScores.put(WISDOM, new AbilityScore(baseWisdom,
this.race.getRacialBonus(WISDOM), 0));
        this.abilityScores.put(CHARISMA, new AbilityScore(baseCharisma,
this.race.getRacialBonus(CHARISMA), 0));
    }

    public int setArmorClass(int baseAmor) {
        this.armorClass = baseAmor +
this.abilityScores.get(DEXTERITY).getModifier();
        return this.armorClass;
    }

    public void setFlatArmorClass(int armorClass) {
        this.armorClass = armorClass;
    }

    private void calculatePassiveSenses() {
        if (this.skillProficiencies.contains(Skills.Perception))
            this.passivePerception = 10 +
this.abilityScores.get(WISDOM).getModifier() +
this.characterClass.getProficiencyBonus(this.level);
        else
            this.passivePerception = 10 +
this.abilityScores.get(WISDOM).getModifier();
```

```
        if (this.skillProficiencies.contains(Skills.Investigation))
            this.passiveInvestigation = 10 +
this.abilityScores.get(INTELLIGENCE).getModifier() +
this.characterClass.getProficiencyBonus(this.level);
        else
            this.passiveInvestigation = 10 +
this.abilityScores.get(INTELLIGENCE).getModifier();

        if (this.skillProficiencies.contains(Skills.Insight))
            this.passiveInsight = 10 +
this.abilityScores.get(WISDOM).getModifier() +
this.characterClass.getProficiencyBonus(this.level);
        else
            this.passiveInsight = 10 +
this.abilityScores.get(WISDOM).getModifier();
    }

    private void setRace(Race race) {
        this.race = race;
        this.speed = race.getWalkingSpeed();
    }

    private void setClass(CharacterClass characterClass) {
        this.characterClass = characterClass;

this.abilityProficiencies.addAll(characterClass.getAbilityProficiencies(
));
    }

    public void setName(String name) {
        this.name = name;
    }

    public void addEquipment(String item) {
        this.equipment.add(item);
        this.equipment.sort(String::compareTo);
    }

    public void removeEquipment(String item) {
        this.equipment.remove(item);
    }

    public List<String> getEquipment() {
        return this.equipment;
    }
```

```
    public void addLanguage(String language) {
        this.languages.add(language);
        this.languages.sort(String::compareTo);
    }

    public void addSkillProficiency(Skills skill) {
        this.skillProficiencies.add(skill);
        if (skill == Skills.Perception || skill == Skills.Investigation
|| skill == Skills.Insight)
            calculatePassiveSenses();
    }

    public void setFlatAbilityScore(AbilityScores abilityScore, int
value) {
        this.abilityScores.put(abilityScore, new AbilityScore(value, 0,
0));
        if (abilityScore == WISDOM || abilityScore == INTELLIGENCE)
            calculatePassiveSenses();
    }

    public int addAbilityBonus(AbilityScores abilityScore, int bonus) {
        AbilityScore ability = this.abilityScores.get(abilityScore);
        try {
            this.abilityScores.put(abilityScore, new
AbilityScore(ability.getBaseScore(),
                    this.race.getRacialBonus(abilityScore),
                    ability.getAdditionalBonus() + bonus));
        } catch (AbilityScoreLimitException e) {
            System.out.println("Ability Score must not exceed 20");
        }
        if (abilityScore == WISDOM || abilityScore == INTELLIGENCE)
            calculatePassiveSenses();
        return this.abilityScores.get(abilityScore).getScore();
    }

    public int skillCheckModifier(Skills skill) {
        if (skillProficiencies.contains(skill)) {
            return abilityScores.get(skill.ability).getModifier() +
this.characterClass.getProficiencyBonus(this.level);
        } else {
            return abilityScores.get(skill.ability).getModifier();
        }
    }

    public int savingThrowModifier(AbilityScores abilityScore) {
        if (abilityProficiencies.contains(abilityScore)) {
```

```
            return this.abilityScores.get(abilityScore).getModifier() +
this.characterClass.getProficiencyBonus(this.level);
        } else {
            return this.abilityScores.get(abilityScore).getModifier();
        }
    }

    public int abilityCheckModifier(AbilityScores abilityScore) {
        return this.abilityScores.get(abilityScore).getModifier();
    }

    public AbilityScore getAbilityScore(AbilityScores abilityScore) {
        return this.abilityScores.get(abilityScore);
    }

    public int getLevel() {
        return level;
    }

    public void setLevel(int level) {
        this.level = level;
    }

    public int getArmorClass() {
        return armorClass;
    }

    public int getInitiativeBonus() {
        return initiativeBonus;
    }

    public void setInitiativeBonus(int initiative) {
        this.initiativeBonus = initiative;
    }

    public int getSpeed() {
        return speed;
    }

    public void setSpeed(int speed) {
        this.speed = speed;
    }

    public int getCurrentHitPoints() {
        return currentHitPoints;
    }
```

```
public void setMaxHitPoints(int maxHitPoints) {
    this.maxHitPoints = maxHitPoints;
    this.currentHitPoints = maxHitPoints;
}

public void setCurrentHitPoints(int hitPoints) {
    if (hitPoints > this.maxHitPoints) {
        this.currentHitPoints = this.maxHitPoints;
    } else this.currentHitPoints = Math.max(hitPoints, 0);
}

private void modifyHitPoints(int hitPoints) {
    if (this.currentHitPoints + hitPoints > this.maxHitPoints) {
        this.currentHitPoints = this.maxHitPoints;
    } else if (this.currentHitPoints + hitPoints < 0) {
        this.currentHitPoints = 0;
    } else {
        this.currentHitPoints += hitPoints;
    }
}

public int takeDamage(int damage) {
    modifyHitPoints(-damage);
    return this.currentHitPoints;
}

public int heal(int healing) {
    modifyHitPoints(healing);
    return this.currentHitPoints;
}

public String getName() {
    return this.name;
}

public Race getRace() {
    return race;
}

public CharacterClass getCharacterClass() {
    return characterClass;
}

public List<String> getLanguages() {
    return languages;
}
```

```
    public String[] getUnproficientSkills() {
        List<String> unproficientSkills = new ArrayList<>();
        for (Skills skill : Skills.values()) {
            if (!this.skillProficiencies.contains(skill)) {
                unproficientSkills.add(skill.toString());
            }
        }
        return unproficientSkills.toArray(new String[0]);
    }

    public boolean isAlive() {
        return this.currentHitPoints > 0;
    }

    @Override
    public String toString() {
        String stillAlive = isAlive() ? "" : "[Dead]\n";
        return stillAlive +
                "Name: " + this.name + "\n" +
                "Race: " + this.race.getClass().toString() + "\n" +
                "Class: " + this.characterClass.getClass().toString() +
"\n" +
                "Level: " + this.level + "\n" +
                "Armor Class: " + this.armorClass + "\n" +
                "Initiative Bonus: " + this.initiativeBonus + "\n" +
                "Speed: " + this.speed + "\n" +
                "Hit Points: (" + this.currentHitPoints + "/" +
this.currentHitPoints + ")\n" +
                "Passive Perception: " + this.passivePerception + "\n" +
                "Passive Investigation: " + this.passiveInvestigation +
"\n" +
                "Passive Insight: " + this.passiveInsight + "\n" +
                "Ability Scores: " + "\n" +
                "Strength: " +
this.abilityScores.get(STRENGTH).getScore() + " (" +
this.abilityScores.get(STRENGTH).getModifier() + ")" + "\n" +
                "Dexterity: " +
this.abilityScores.get(DEXTERITY).getScore() + " (" +
this.abilityScores.get(DEXTERITY).getModifier() + ")" + "\n" +
                "Constitution: " +
this.abilityScores.get(CONSTITUTION).getScore() + " (" +
this.abilityScores.get(CONSTITUTION).getModifier() + ")" + "\n" +
                "Intelligence: " +
this.abilityScores.get(INTELLIGENCE).getScore() + " (" +
this.abilityScores.get(INTELLIGENCE).getModifier() + ")" + "\n" +
```

```
                "Wisdom: " + this.abilityScores.get(WISDOM).getScore() +
" (" + this.abilityScores.get(WISDOM).getModifier() + ")" + "\n" +
                "Charisma: " +
this.abilityScores.get(CHARISMA).getScore() + " (" +
this.abilityScores.get(CHARISMA).getModifier() + ")" + "\n" +
                "Ability Proficiencies: " + this.abilityProficiencies +
"\n" +
                "Skill Proficiencies: " + this.skillProficiencies + "\n"
+
                "Languages: " + this.languages + "\n" +
                "Equipment: " + this.equipment + "\n";
    }

    public int getMaxHitPoints() {
        return this.maxHitPoints;
    }

    public List<Skills> getAllSkillProficiencies() {
        return this.skillProficiencies;
    }

    public HashMap<AbilityScores, AbilityScore> getAllAbilityScores() {
        return this.abilityScores;
    }
}
```