

# Advanced Software Engineering

## Projektarbeitsdokumentation

im Rahmen der Prüfung zum

Bachelor of Science (B.Sc.)

des Studienganges

Informatik

an der

Dualen Hochschule Baden-Württemberg Karlsruhe

von

**Dominik Ochs**

Abgabedatum:

XX. Mai 2023

Bearbeitungszeitraum:

01.10.2022 - XX.05.2023

Matrikelnummer, Kurs:

2847475, TINF20B2

Gutachter der Dualen Hochschule:

Dr. Lars Briem

# Inhaltsverzeichnis

<b>Abbildungsverzeichnis</b>	<b>III</b>
<b>Quellcodeverzeichnis</b>	<b>V</b>
<b>1 Einführung</b>	<b>1</b>
1.1 Übersicht über die Applikation . . . . .	1
1.2 Wie startet man die Applikation? . . . . .	2
1.3 Wie testet man die Applikation? . . . . .	3
<b>2 Clean Architecture</b>	<b>4</b>
2.1 Was ist Clean Architecture? . . . . .	4
2.2 Analyse der Dependency Rule . . . . .	4
2.3 Analyse der Schichten . . . . .	6
<b>3 SOLID</b>	<b>10</b>
3.1 Analyse Single-Responsibility-Principle (SRP) . . . . .	10
3.2 Analyse Open-Closed-Principle (OCP) . . . . .	12
3.3 Analyse Interface-Segregation-Principle (ISP) . . . . .	15
<b>4 Weitere Prinzipien</b>	<b>19</b>
4.1 Analyse GRASP: Geringe Kopplung . . . . .	19
4.2 Analyse GRASP: Hohe Kohäsion . . . . .	23
4.3 Don't Repeat Yourself (DRY) . . . . .	24
<b>5 Unit Tests</b>	<b>28</b>
5.1 10 Unit Tests . . . . .	28
5.2 ATRIP: Automatic . . . . .	30
5.3 ATRIP: Thorough . . . . .	30
5.4 ATRIP: Professional . . . . .	33
5.5 Code Coverage . . . . .	38
5.6 Fakes und Mocks . . . . .	38
<b>6 Domain Driven Design</b>	<b>41</b>
6.1 Ubiquitous Language (UL) . . . . .	41
6.2 Entities . . . . .	42
6.3 Value Objects (VO) . . . . .	44
6.4 Repositories . . . . .	45
6.5 Aggregates . . . . .	48

<b>7</b>	<b>Refactoring</b>	<b>50</b>
7.1	Code-Smells . . . . .	50
7.2	2 Refactorings . . . . .	58
<b>8</b>	<b>Entwurfsmuster</b>	<b>66</b>
8.1	Entwurfsmuster: Observer-Pattern . . . . .	66
8.2	Entwurfsmuster: Strategie . . . . .	67

# Abbildungsverzeichnis

1.1	Spielphasen. . . . .	2
2.1	UML-Diagramm 1 zum Einhalten der Dependency Rule von <i>Camp</i> . . . .	5
2.2	UML-Diagramm 2 zum Einhalten der Dependency Rule von <i>Game</i> . . . .	6
2.3	Beispielklasse der Plugin-Schicht: <i>ErrorBuilder</i> . . . . .	7
2.4	Beispielklasse der Domain-Schicht: <i>ResourceStash</i> . . . . .	9
3.1	UML-Diagramm von <i>ase.plugin.cli.parsers.DiceParser</i> . . . . .	10
3.2	UML-Diagramm von <i>application.RollHandler</i> . . . . .	11
3.3	Abbildung 3.2 aufgeteilt in <i>EndeavorHandler</i> und <i>EncounterHandler</i> . . .	12
3.4	UML-Klassendiagramm vom <i>RollDxCommand</i> und weiteren Commands, wovon die meisten allerdings ausgelassen wurden der Übersichtlichkeit wegen. . . . .	13
3.5	UML-Klassendiagramm von <i>CardInvalidator</i> und <i>Card</i> . . . . .	14
3.6	Abbildung 3.5 mit Card-Interface und Implementationen statt Card-Enum und CardCategory. Aus Übersichtlichkeit wurden nicht alle Implementationen eingezeichnet. . . . .	15
3.7	UML-Diagramm von Buildable, Endeavor, Tool und Building und deren Implementationen. . . . .	16
3.8	UML-Diagramm von <i>Deck</i> und dessen Implementationen. . . . .	16
3.9	Abbildung 3.8 mit abgespaltem <i>Serializable</i> -Interface. . . . .	18
4.1	UML-Diagramm von <i>ase.application.PersistenceWriter</i> . . . . .	20
4.2	UML-Diagramm von <i>ase.domain.crafting.Workbench</i> . . . . .	22
4.3	Abbildung 4.2 mit eingezogenem <i>ResourceConsumer</i> -Interface zur Auflösung der Kopplung. . . . .	23
4.4	UML-Diagramm von <i>ase.domain.crafting.ResourceRequirement</i> . . . . .	24
5.1	UML-Diagramm des <i>CampMock</i> , der zur Testung der <i>encounter</i> -Methode des <i>RollHandler</i> benötigt wird. . . . .	39
5.2	UML-Diagramm des <i>RescueMock</i> , der zur Testung der <i>endeavor</i> -Methode des <i>RollHandler</i> benötigt wird. . . . .	40
6.1	UML-Diagramm von <i>GameState</i> . . . . .	43
6.2	UML-Diagramm von dem Value Object <i>Roll</i> . . . . .	45
6.3	UML-Diagramm von dem Repository <i>PersistenceReader</i> . . . . .	47
6.4	UML-Diagramm vom <i>crafting</i> -Aggregat mit <i>Camp</i> als Root-Entity. . . .	49
7.1	UML-Diagramm von <i>CardDeck</i> <u>vor</u> der Method-Extraction. . . . .	60

7.2	UML-Diagramm von <i>CardDeck</i> <u>nach</u> der Method-Extraction durch Commit cb55ad97. Beachte die neue private Methode <i>countCardOccurrences</i> in <i>CardDeck</i> . . . . .	61
7.3	UML-Diagramm von <i>StartCommand</i> <u>vor</u> dem RTQ-Refactoring. . . . .	63
7.4	UML-Diagramm von <i>StartCommand</i> <u>nach</u> dem RTQ-Refactoring durch Extraktion des Erstellens eines neuen Decks in Commit 827cd843. Beachte die neue private Methode <i>countCardOccurrences</i> in <i>CardDeck</i> . . . .	65
8.1	UML-Diagramm des Observer-Patterns zwischen <i>GameState</i> und <i>GameEndReporter</i> . . . . .	67
8.2	UML-Diagramm (gekürzt) des Strategie-Pattern der <i>Rescue-Endeavor</i> -Logik. Zur besseren Übersichtlichkeit wurden überflüssige Klassen ausgelassen. . . . .	68

# Quellcodeverzeichnis

4.1	DRY-Code der <i>Game</i> -Klasse zum Commit b4399813. . . . .	25
4.2	DRY-Code der <i>Game</i> -Klasse zum Commit e246d77b. . . . .	26
5.1	<u>Thorough</u> Test-Code zur <i>add</i> -Methode der <i>NonNegativeInteger</i> -Klasse. .	30
5.2	<u>Nicht thorough</u> Test-Code zum Konstruktor der <i>NonNegativeInteger</i> -Klasse.	33
5.3	<u>Professioneller</u> Test-Code zur <i>raiseRollBy</i> -Methode der <i>Roll</i> -Klasse. . . .	33
5.4	<u>Unprofessioneller</u> Test-Code zur statischen <i>fromNumberCapped</i> -Funktion der <i>RollInteger</i> -Klasse. . . . .	36
7.1	Long-Class-Code-Smell der <i>ArgumentParser</i> -Klasse zum Commit 174283ab.	50
7.2	Long-Class-Code-Smell behoben durch Aufteilung der Klassen zum Commit b0ce98af. . . . .	52
7.3	Switch-Statement-Code-Smell der <i>CardDeck</i> -Klasse zum Commit dd2a39d2.	54
7.4	Switch-Statement-Code-Smell behoben durch neue <i>DeckConfiguration</i> - Klasse und Map zum Zeitpunkt des Commit a0e99692. . . . .	56
7.5	<i>isValid</i> -Methode <u>vor</u> Method-Extraction (aus u.a. Commit a0e99692). Vgl. Code-Ausschnitt 7.4. . . . .	59
7.6	<u>Nach</u> Method-Extraction der Methode <i>countCardOccurrences</i> aus der <i>isValid</i> -Methode in Commit cb55ad97. . . . .	60
7.7	<i>execute</i> -Methode <u>vor</u> RTQ-Refactoring. . . . .	62
7.8	<i>execute</i> -Methode und <i>deckFrom</i> -Methode <u>nach</u> RTQ-Refactoring in Com- mit 827cd843. . . . .	63

# 1 Einführung

## 1.1 Übersicht über die Applikation

Bei dieser Applikation handelt es sich um ein singleplayer Kartenspiel. Hier versucht ein Spieler auf einer einsamen Insel zu überleben und von ihr zu flüchten, indem er Karten zieht und Ressourcen (*wood, plastic, metal*) sammelt, um Gegenstände zu bauen (*Scavage*), die ihn vor seinen Feinden (*Encounter*) schützen oder ihm eine Rettung (*Endeavor*) von der Insel ermöglichen. Während des Spiels trifft der Spieler auf Tiere (*spider, tiger, snake*), die ihm gesammelte Ressourcen kosten können, oder Katastrophen, die Ressourcen zerstören. Das Spiel endet, wenn der Spieler sich retten konnte oder keine Aktionen mehr möglich sind.

Das Kartenspiel hat verschiedene Spielphasen (s. Abbildung 1.1), in denen der Spieler unter bestimmten Bedingungen Karten ziehen und Gegenstände bauen kann (*Scavage*). Wenn ein Gegenstand aus der Kategorie Rettungen (*Endeavor*) gebaut wurde, muss der Spieler würfeln, um zu entscheiden, ob die Rettung erfolgreich ist. Ein Segelboot und ein Hanggleiter stellen eine Rettung dar, wenn beim Würfeln eine bestimmte Augenzahl erzielt wird. Wenn der Rettungsversuch fehlschlägt, kann der Spieler weiter Karten ziehen und Gegenstände bauen, solange Karten im Stapel vorhanden sind. Das Bauen eines Dampfschiffs und eines Heißluftballons ist nur mit einer Feuerstelle möglich und garantiert eine erfolgreiche Rettung. Wenn ein Tier gezogen wird, muss der Spieler gegen es kämpfen (*Encounter*) und auch hier bestimmt Würfeln über Sieg oder Niederlage. Für unterschiedliche Aktionen werden unterschiedliche Würfel (*vier-, sechs-, acht-seitig*) verwendet. Das Spiel endet, wenn keine weiteren Aktionen mehr möglich sind oder wenn eine garantierte Rettung erfolgt ist. Wenn das Spiel endet (*End*), bleibt es in diesem Zustand, bis ein neues Spiel gestartet wird, bis das Spiel neu initialisiert wird oder die Anwendung beendet wird.

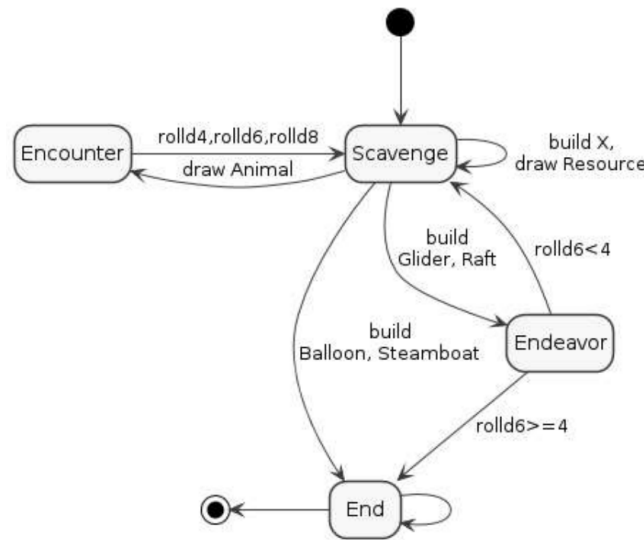


Abbildung 1.1: Spielphasen.

## 1.2 Wie startet man die Applikation?

Vorraussetzung ist **Java 17**<sup>1</sup>.

Wahlweise wird eine gängige Java-IDE<sup>2</sup> oder eine *maven*-Installation benötigt.

### Über Maven:

Im Wurzelverzeichnis folgende Maven-Befehle ausführen:

```
mvn compile
mvn package
java -jar target/ase-1.0-SNAPSHOT.jar
```

<sup>1</sup>Check mit `java -version`.

<sup>2</sup>Hier wurde *Jetbrains' IntelliJ* verwendet.



**Über IDE:**

Projekt mit IDE im Wurzelverzeichnis öffnen. Zu folgendem Pfad navigieren

```
./src/main/java/de/dhbw/karlsruhe/ase/plugin/cli
```

und dort die *main*-Methode der Klasse *Main* ausführen lassen.

**Erste Schritte:**

Nun läuft das Command Line Interface der Applikation und es kann mit

```
help
```

eine Übersicht über die möglichen Befehle (Name, Regex, Beschreibung) aufgerufen werden, oder mit

```
start?
```

ein Spiel mit zufälligem Kartenstapel gestartet werden. Daraufhin kann mit

```
draw
```

begonnen werden Karten zu ziehen.

## 1.3 Wie testet man die Applikation?

**Über Maven:**

Im Wurzelverzeichnis:

```
mvn test
```

**Über IDE:**

Die Tests befinden sich unter

```
./src/test/java
```

und können dort mit einer IDE-Funktion ausgeführt werden.

Es existieren Blackbox-Integration-Tests und Unit-Tests, die aber keine besonderen Anforderungen außer eine **JUnit5**-Abhängigkeit haben.

## 2 Clean Architecture

### 2.1 Was ist Clean Architecture?

Clean Architecture ist ein allgemeines Konzept zum Design von Softwarearchitekturen, das darauf abzielt, die Struktur einer Anwendung so zu gestalten, dass sie unabhängig von ihren Benutzerschnittstellen, Datenquellen und anderen äußeren Faktoren bleibt. Dies bedeutet, dass die verschiedenen Komponenten und Funktionen einer Anwendung in abgekapselten Schichten angeordnet werden (*Onion-Architektur*), um sicherzustellen, dass Veränderungen in einer Schicht die tieferliegenden Schichten nicht beeinflussen. Äußere Schichten haben dabei Abhängigkeiten von tieferen Schichten aber niemals umgekehrt. Aufrufe von tieferen Schichten an äußere Schichten werden über Abstraktionen (*Dependency Inversion* und *Injection*) realisiert. Tiefere Schichten sind dabei langlebiger als Schichten weiter außen.

Dies führt zu einer Anwendung, die flexibler und leichter zu entwickeln und zu warten ist, da Änderungen an einem Teil der Anwendung die tieferliegenden Schichten nicht berühren.

### 2.2 Analyse der Dependency Rule

Schichten im Sinne der Dependency Rule sind hier `plugin`, `application`, `domain` und `abstraction` aufsteigend geordnet nach zunehmender Tiefe in der Onion-Architektur<sup>1</sup>. Sie werden durch gleichnamige Packages, die parallel unter `de.dhbw.karlsruhe.ase` zu finden sind, erschöpfend repräsentiert. Es sind Abhängigkeiten von äußerden Schichten in tiefere Schichten erlaubt, aber nicht umgekehrt. Diese Regel wird für die oben genannten Schichten immer eingehalten, was die folgenden beiden Beispiele illustrieren.

---

<sup>1</sup>Siehe Abschnitt 2.3 für mehr Informationen.

### Positiv-Beispiel 1:

Abbildung 2.1 zeigt das verlangte UML-Diagramm der Beispielklasse 1 *Camp* der Schicht *domain*, welches die Dependency Rule einhält. Wie im Diagramm zu sehen ist hängt *Camp* ausschließlich von Klassen in der selben Schicht (*domain*) ab. *Crafting* ist dabei keine Schicht sondern ein *Aggregat*<sup>2</sup>. Schichten sind ausschließlich die oben genannten. Weiter zu sehen ist, dass ausschließlich Klassen in *application* von *Camp* abhängen, aber keine aus höheren Schichten (was in diesem konkreten Fall nur *abstraction* sein könnte).

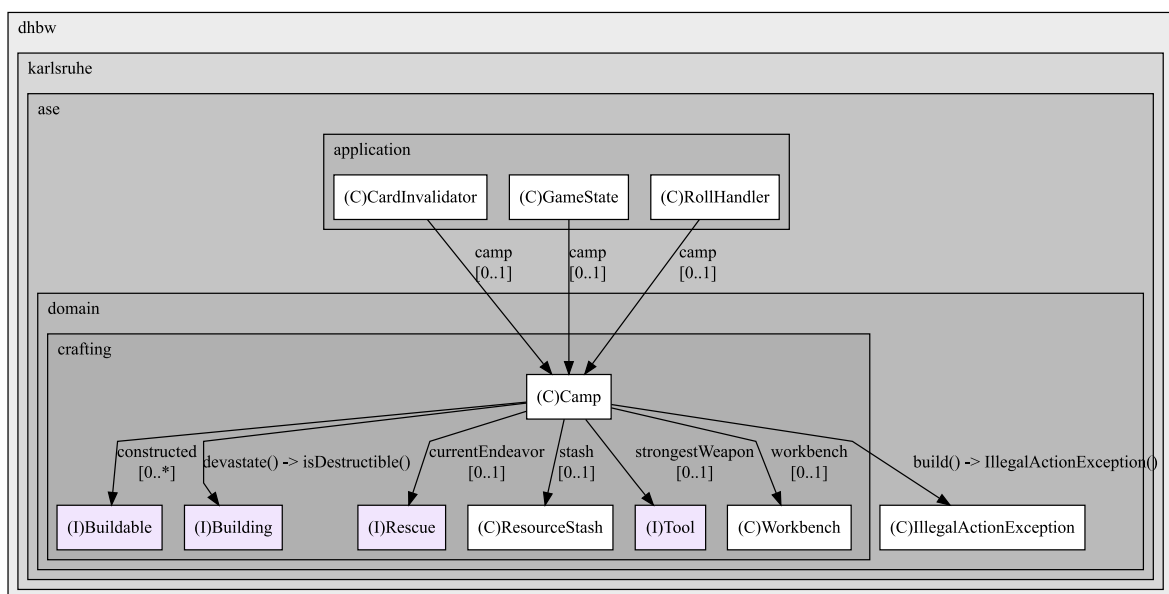


Abbildung 2.1: UML-Diagramm 1 zum Einhalten der Dependency Rule von *Camp*.

### Positiv-Beispiel 2:

Abbildung 2.2 zeigt ein weiteres positives Beispiel für die Dependency Rule, da diese in dieser Softwarearchitektur nicht verletzt wird. Abgebildet ist die Klasse *Game*, welche lediglich von Klassen aus der eigenen Schicht (*application*) und Klassen aus der tieferen Schicht *domain* abhängt. Abhängig von *Game* sind nur Klassen aus der Schicht *plugin*, bzw. dem konkreten *cli*-Plugin.

<sup>2</sup>Siehe Kapitel 6.

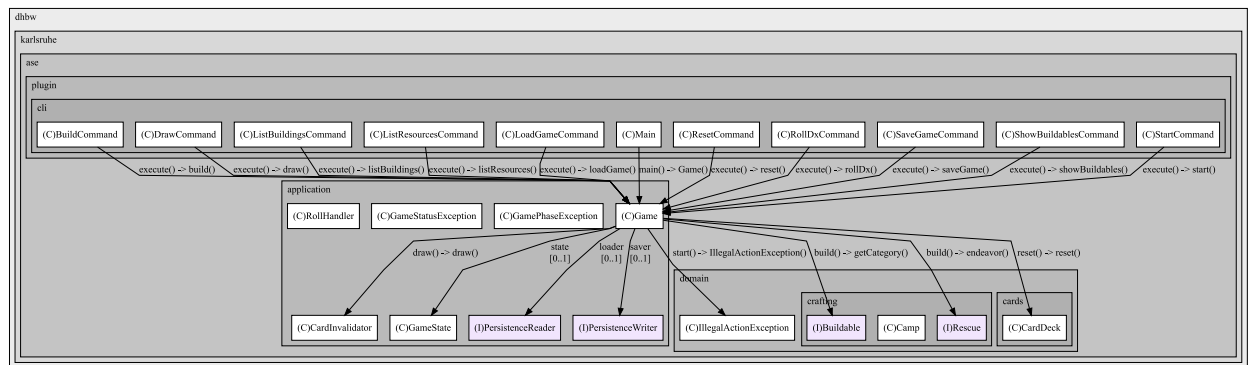


Abbildung 2.2: UML-Diagramm 2 zum Einhalten der Dependency Rule von *Game*.

## 2.3 Analyse der Schichten

In dieser Softwarearchitektur gibt es folgende Schichten aufsteigend sortiert nach Tiefe: *plugin*, *application*, *domain* und *abstraction*. Hierbei stellt *plugin* die äußerste und *abstraction* die tiefste Schicht dar.

### Schicht Plugin:

Bei einem Refactoring der Plugin-Schicht ist aufgefallen, dass viele der (Fehler-)Hinweise, die aufgrund von falschem Befehlssyntax oder einem falschen Befehl an den User über das CLI ausgegeben werden, keine gemeinsame einheitliche Form aufwiesen, da diese direkt als String ausgegeben wurden. Um eine einheitliche Fehlermeldung zu gewährleisten wurde der *ErrorBuilder* eingefügt, dessen Aufgabe es ist, die Fehler einheitlich zu formatieren und auszugeben. Abbildung 2.3 zeigt das UML-Klassendiagramm dieser Klasse. Sie wird im gesamten CLI-Plugincode verwendet um (Fehler-)Hinweise auszugeben.

Über verschiedene Konstruktoren kann ein Grund für den (Fehler-)Hinweis und eine mögliche Abhilfeempfehlung eingegeben werden. Mit der *print*-Methode kann der erzeugte (Fehler-)Hinweis dann formatiert an den User ausgegeben werden. Hierzu wird aus Gründen der Testbarkeit an die Funktion *printError* des Proxys *Terminal* delegiert.

Diese Klasse befindet sich im Plugin-Layer (insb. im CLI-Plugin), da die konkrete Ausgabe an den User in Form von Text von dem CLI abhängt. Die high-level Fehlermeldungen die von den tieferen Schichten über Exceptions realisiert sind können von jedem Plugin

anders verarbeitet und an den User weitergegeben werden. In der CLI funktioniert dies mit dem `ErrorBuilder`, während es mit einem denkbaren GUI-Plugin über bspw. ein Popup funktionieren könnte. Der `ErrorBuilder` ist lediglich für die Ausgabe an den Nutzer zuständig und enthält keine Fehler-Logik und gehört somit in die Plugin-Schicht.

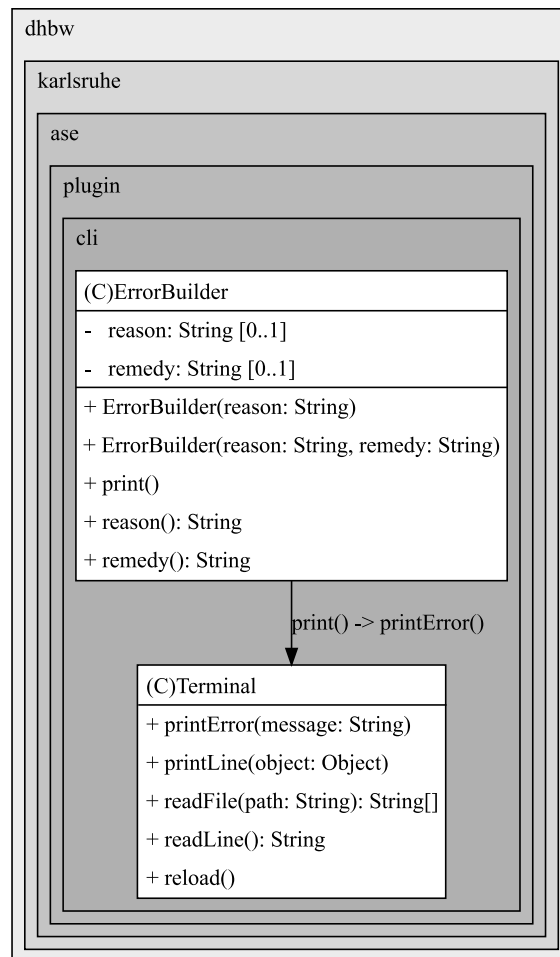


Abbildung 2.3: Beispielklasse der Plugin-Schicht: `ErrorBuilder`.

### Schicht Domain:

Der *ResourceStash* ist fester Teil der Domain-Schicht und stellt hier ein Wrapper mit Methodennamen in Domänensprache für eine Resource-Deque dar. Abbildung 2.4 zeigt das zugehörte UML-Klassendiagramm. Der *ResourceStash* beinhaltet und verwaltet alle Ressourcen, die der Spieler im Verlauf des Spiels ansammelt. Durch eine Katastrophe

oder das Verlieren gegen ein Tier wird der ResourceStash zerstört (*devastate*) und alle ungeschützten Ressourcen gelöscht. Das Erbauen eines *Shacks*<sup>3</sup> erlaubt es die obersten  $n \in [0; \text{inf})$  Ressourcen zu schützen (*protectTopMostNResources*), sodass diese nach einem *devastate* übrig bleiben. Zusätzlich können Ressourcen hinzugefügt, konsumiert oder deren Vorhandensein überprüft werden. *Camp* und *Workbench* teilen sich eine Referenz auf denselben Stash.

Die Klasse ist als Teil des Kartenspiels (der Domäne) im Code natürlich in der Domain-Schicht angesiedelt und wird von anderen Domänenklassen genutzt. Es gibt keine Möglichkeit die Klasse in einer der anderen Schichten sinnvoll anzusiedeln.

---

<sup>3</sup>nicht in der Abb.

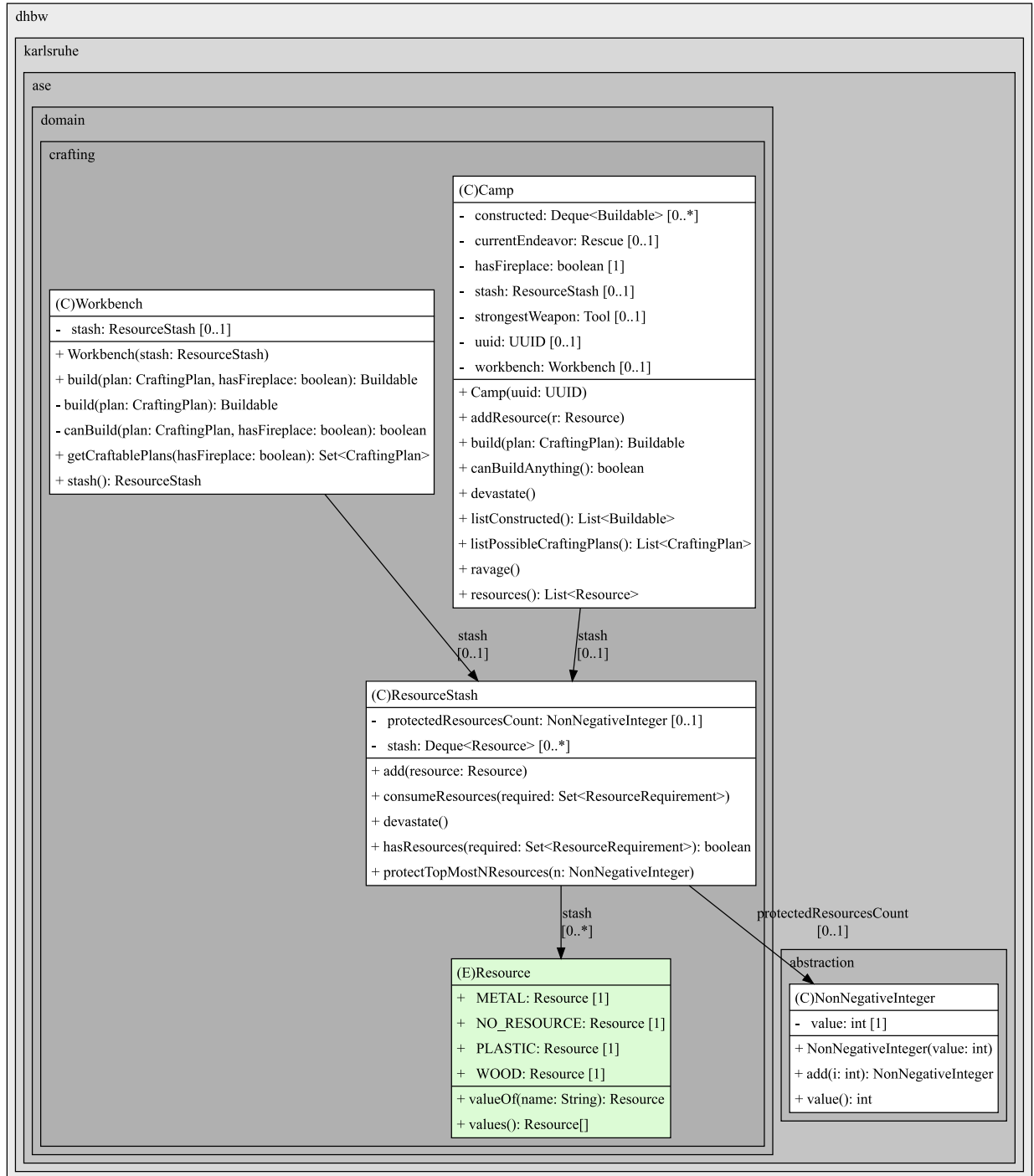


Abbildung 2.4: Beispielklasse der Domain-Schicht: ResourceStash.

## 3 SOLID

### 3.1 Analyse Single-Responsibility-Principle (SRP)

#### Positiv-Beispiel:

Abbildung 3.1 zeigt das UML-Klassendiagramm des *DiceParser*, welcher in der Plugin-Schicht unter `plugin.cli.parsers` zu finden ist und das Positiv-Beispiel darstellt. Dieser implementiert das *Parser*-Interface aus `plugin.cli`. Die einzige Aufgabe des *DiceParsers* ist es die Usereingabe zum Erstellen eines Würfelwurfs aus Text zu parsen und das entsprechende Objekt zu kreieren. Hierzu wird der *Regex*-Matcher, der bereits den Syntax überprüft hat an die *parse*-Methode übergeben und daraus die Argumente extrahiert, um den *Roll* zu erstellen.

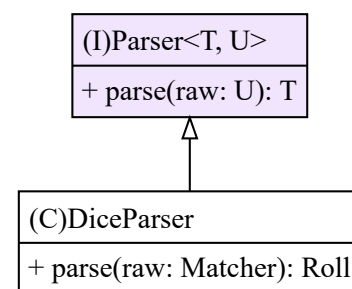


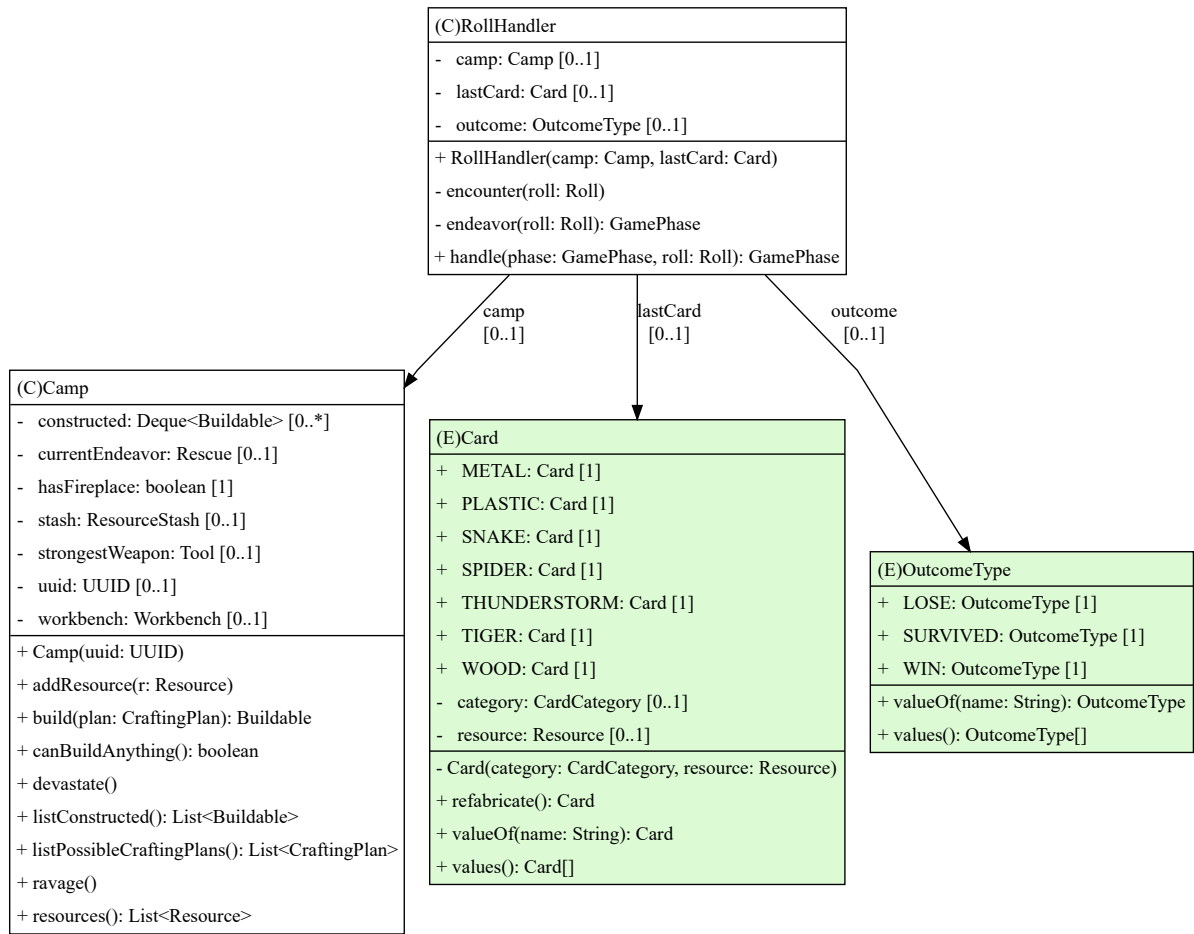
Abbildung 3.1:  
UML-Diagramm von  
*ase.plugin.cli.parsers.DiceParser*.

#### Negativ-Beispiel:

Abbildung 3.2 zeigt das UML-Klassendiagramm des *RollHandler*, welcher in der Applikations-Schicht die Würfelwürfe für die Rettungen (*Endeavor*) und Kämpfe (*Encounter*) bearbeitet. Hierbei entscheidet die öffentliche *handle*-Methode, ob es sich um einen *Encounter* oder ein *Endeavor* handelt (dies ist allerdings eigentlich schon bei Aufruf dieser bekannt) und führt die entsprechende private Methode aus. Die Klasse hat also zwei Aufgaben (Responsibilities).

Um dies zu lösen kann einfach die Klasse *RollHandler* in zwei Klassen aufgeteilt werden - den *EncounterHandler* und *EndeavorHandler* - die nun jeweils nur noch genau eine Aufgabe haben und somit das SRP einhalten, wie Abbildung 3.3 zeigt.



Abbildung 3.2: UML-Diagramm von *application.RollHandler*.

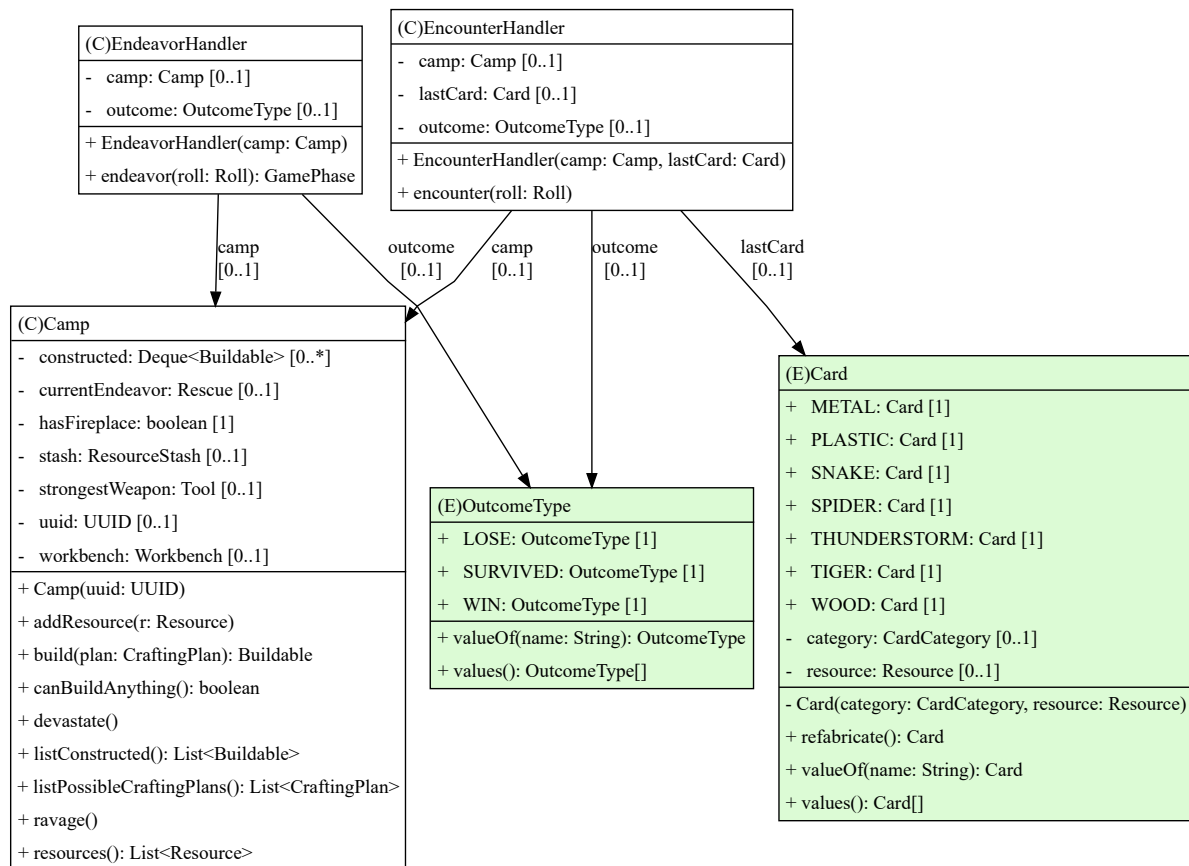


Abbildung 3.3: Abbildung 3.2 aufgeteilt in *EndeavorHandler* und *EncounterHandler*.

## 3.2 Analyse Open-Closed-Principle (OCP)

### Positiv-Beispiel:

Das Positiv-Beispiel zum OCP wird konstituiert durch das *Command*-Interface aus `ase.plugin.cli` und dessen Implementationen (z.B. *RollDxCommand*) aus `ase.plugin.cli.com` wie gezeigt in Abbildung 3.4. Die Main-Klasse kennt nur das *Command*-Interface und ruft darauf die *execute*-Methode auf, die dann in den unterschiedlichen *Command*-Implementationen verschiedene Wirkungen auf das übergebene *Game* haben. Dies ist auch die Begründung, wieso das OCP hier erfüllt wird: Um einen neuen Befehl zu implementieren muss lediglich eine weitere Klasse hinzugefügt werden, die das *Command*-Interface

implementiert. Es muss dazu keine der bestehenden Command-Klassen angepasst oder geändert werden.

Das OCP ist hier sehr sinnvoll, da für neue Features sehr wahrscheinlich regelmäßig neue Commands hinzugefügt werden müssen und dies soll somit möglichst einfach umsetzbar sein und keinen bestehenden Code breaken. Zuvor wurden die unterschiedlichen Befehle über zahlreiche `switch`-Blöcke realisiert (siehe z.B. Commit 034a5c28), was Änderungen des Programmcodes an vielen Stellen nötig machte, um einen neuen Befehl hinzuzufügen. Außerdem wurden Runtime-Exceptions geworfen, wenn vergessen wurde den Code an einer Stelle anzupassen. Durch das neue System müssen keine Änderungen (geschlossen für Änderungen) an vielen Stellen mehr durchgeführt werden, sondern nur noch Additions durchgeführt werden (offen für Erweiterung).

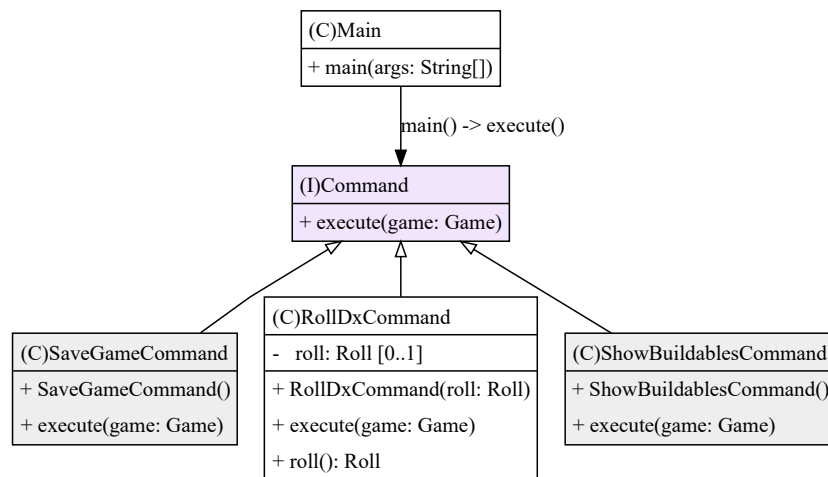


Abbildung 3.4: UML-Klassendiagramm vom *RollDxCommand* und weiteren Commands, wovon die meisten allerdings ausgelassen wurden der Übersichtlichkeit wegen.

### Negativ-Beispiel:

Abbildung 3.5 zeigt das Negativ-Beispiel für das OCP. Die hier relevanten Klassen sind der *CardInvalidator* und das *Card*-Enum, welches eine *CardCategory* und eine *Resource* enthält. Der *CardInvalidator* zieht bisher eine Karte vom *CardDeck* mit der *draw*-Methode und invalidiert die Karte, indem er deren Effekt einlöst. Dies wird mithilfe eines `switch`-Statements über die *CardCategory* der Karte entschieden. Wenn nun also eine neue Karte mit einem neuen Karteneffekt hinzugefügt werden soll, muss also die *draw*-Methode

verändert werden. Das OCP ist also nicht gewahrt.

Abbildung 3.6 zeigt eine mögliche Lösung, um das OCP einzuhalten: Das Card-Enum wurde mit einem Card-Interface mit der *invalidate*-Methode ersetzt und die Karten (Metal, Wood, Tiger, etc.) sind nun Implementationen des Card-Interface und *CardCategory* wurde entfernt. Somit kann der CardInvalidator in *draw* einfach die invalidate-Methode der gezogenen Karte aufrufen und die Karte führt selbst ihren Effekt aus. Somit können in Zukunft beliebig neue Karten eingeführt werden, für die einfach nur eine Implementation des Card-Interface geschrieben werden muss. Es ist keine Veränderung bestehenden Codes mehr nötig und somit wird das OCP mit dieser Lösung eingehalten.

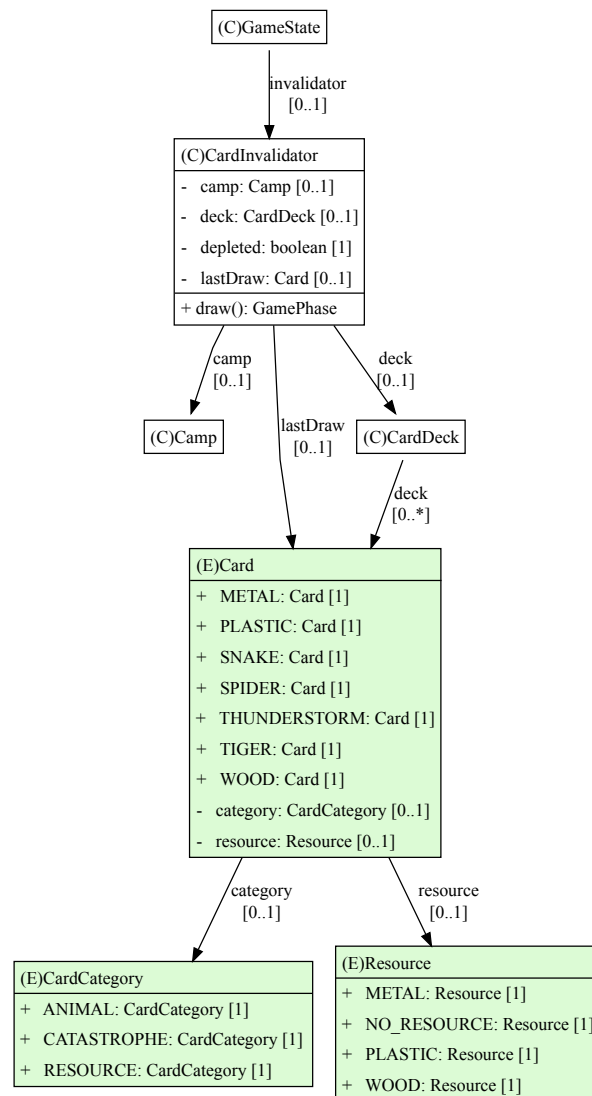


Abbildung 3.5: UML-Klassendiagramm von *CardInvalidator* und *Card*.

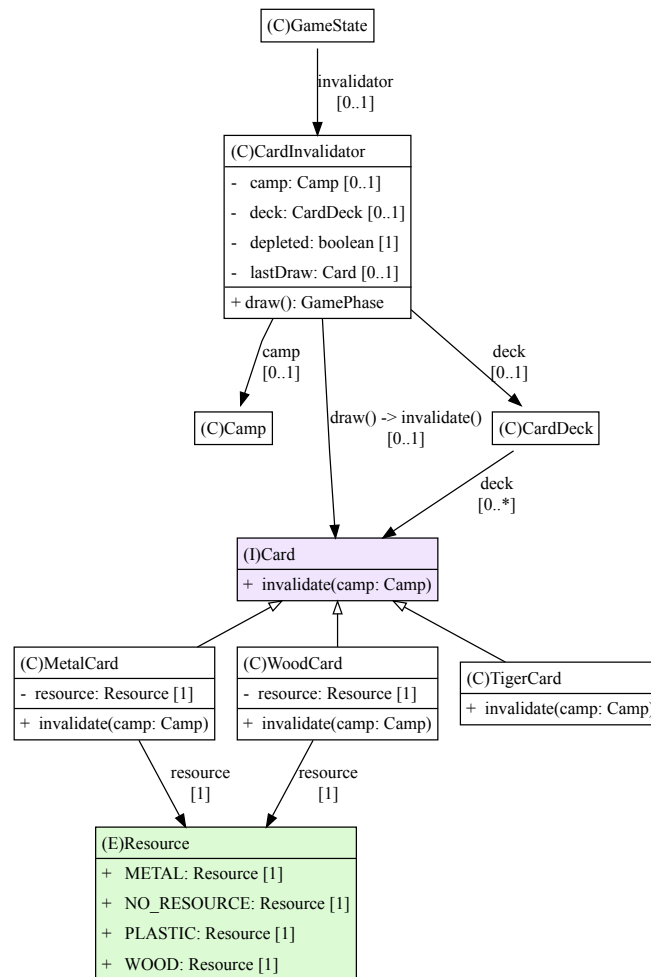


Abbildung 3.6: Abbildung 3.5 mit Card-Interface und Implementationen statt Card-Enum und CardCategory. Aus Übersichtlichkeit wurden nicht alle Implementationen eingezeichnet.

### 3.3 Analyse Interface-Segregation-Principle (ISP)

#### Positiv-Beispiel:

Abbildung 3.7 zeigt wie die Interfaces *Buildable*, *Tool*, *Building* und *Rescue* das ISP einhalten. Die Abhängigkeiten von anderen Klassen auf *Tool*, *Building* und *Rescue* sind aus Übersichtlichkeitsgründen ausgelassen. Alle Interfaces enthalten nur eine Methode und alle *Tool*-, *Building*- und *Rescue*-Implementationen sind auch *Buildables* aber nicht

anders herum. So können die Interfaces sehr flexibel angewandt werden und das ISP ist gewahrt.

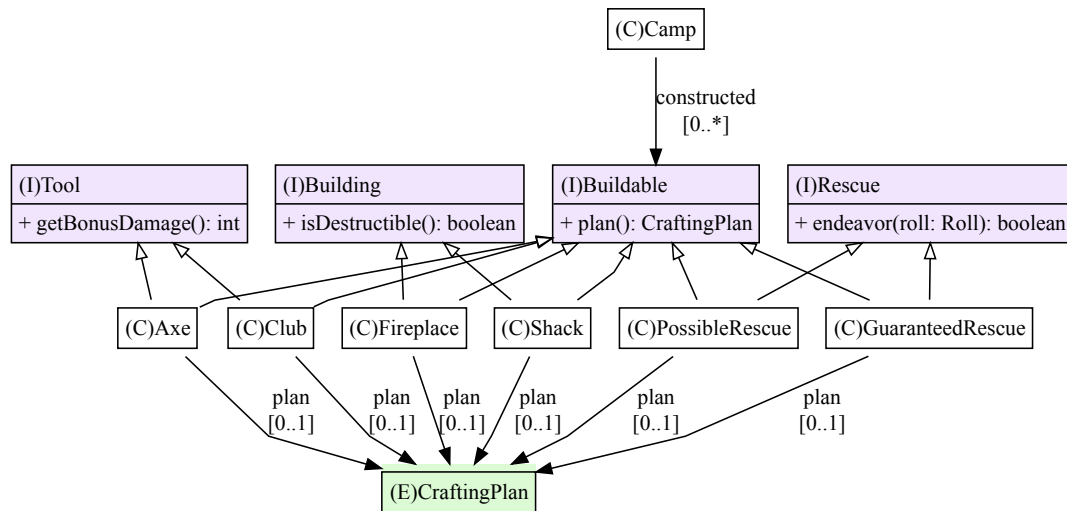


Abbildung 3.7: UML-Diagramm von Buildable, Endeavor, Tool und Building und deren Implementierungen.

### Negativ-Beispiel:

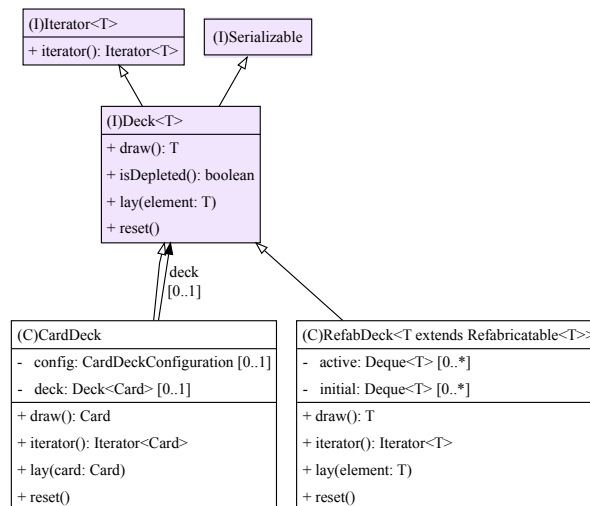


Abbildung 3.8: UML-Diagramm von Deck und dessen Implementierungen.

Abbildung 3.8 zeigt das *Deck*-Interface - das einzige Interface in der Applikation mit mehr als einer Methode. Zunächst ist es allerdings diskutabel, ob es sich hierbei wirklich um ein Negativ-Beispiel handelt, da es sich bei dem Deck-Interface um eine Datenstruktur handelt, die besondere Eigenschaften aufweisen soll, wie z.B. dass der ursprüngliche Zustand der Datenstruktur bei Aufruf der *reset*-Methode wieder hergestellt werden soll. Alle Methoden des Deck-Interfaces werden also gesammelt und gemeinsam benötigt, damit eine Klasse von der Abstraktion des Deck-Interfaces abhängen kann und den nötigen Funktionsumfang verwenden kann. Hier kann also erstmal das Deck-Interface nicht aufgeteilt werden. Außerdem hängt das Deck-Interface von dem *Iterator*-Interface ab, sodass alle Implementationen von Deck auch automatisch Iterator implementieren müssen. Diese Funktion ist aber ebenfalls für ein Deck gewünscht - es soll möglich sein, mit einer Schleife über die Elemente des Deck zu iterieren. Damit ist also auch nicht das Iterator-Interface abspaltbar.

Allerdings wird es unmittelbar klar, dass das ISP hier eindeutig und ohne Diskussion gebrochen wird, wenn betrachtet wird, dass Deck ebenfalls das *Serializable*-Interface beinhaltet, wodurch automatisch alle Implementationen das Serializable-Interface implementieren, was natürlich absolut *keinen* Sinn ergibt. Es ist sogar gefährlich: Beim Implementieren des Deck-Interface wird dem Programmierer nicht klar, dass er die Implementation implizit und ohne Kenntnis als Serializable markiert, was ein großes Problem darstellen kann.

Abbildung 3.9 zeigt nun, dass dieses Problem einfach behoben werden kann, indem die Implementationen explizit und direkt Serializable implementieren und so bewusst die jeweilige Implementation als Serializable markieren. Damit ist das ISP wieder gewahrt, da das Deck so weit wie sinnvoll möglich (Erläuterung s.o.) verkleinert wurde und die Implementationen nun von mehreren kleineren Interfaces abhängen, anstelle von einem größeren abhängen.

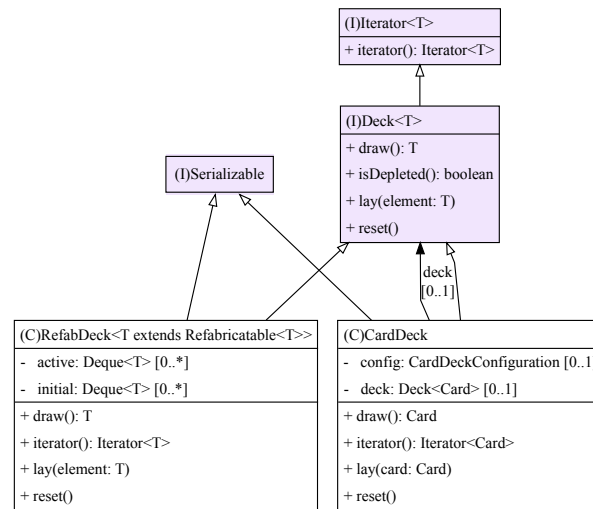


Abbildung 3.9: Abbildung 3.8 mit abgespaltem *Serializable*-Interface.



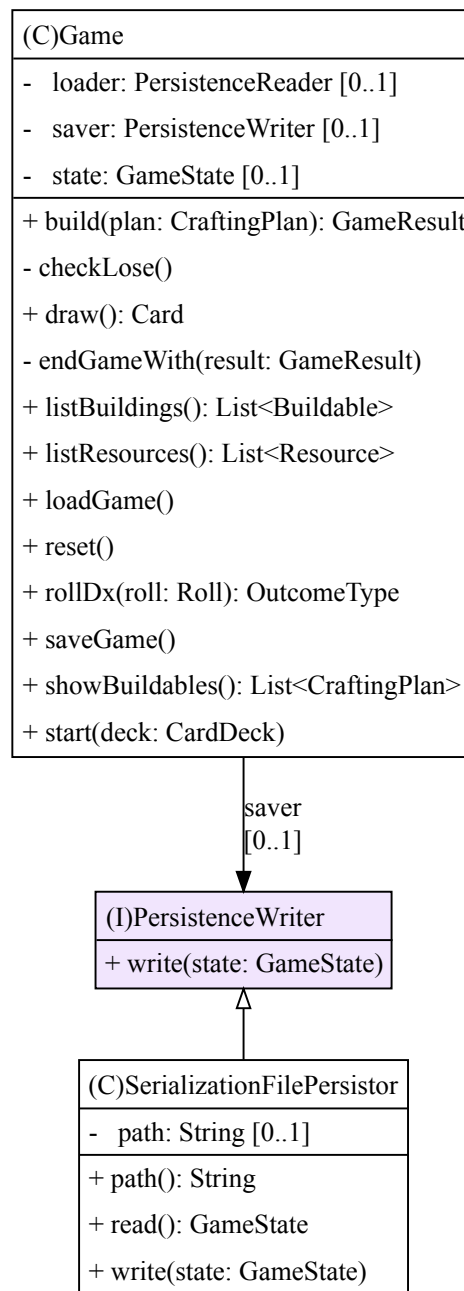
## 4 Weitere Prinzipien

### 4.1 Analyse GRASP: Geringe Kopplung

#### Positiv-Beispiel:

Abbildung 4.1 zeigt das Positiv-Beispiel zur geringen Kopplung.

Das Interface *ase.application.PersistenceWriter* entkoppelt *ase.application.Game* von *ase.plugin.localpersistence.SerializationFilePersistor*. In *Game* soll es die Möglichkeit geben den aktuellen Spielstand zu speichern, damit dieser in Zukunft wieder geladen werden kann und das Spiel fortgesetzt werden kann. *SerializationFilePersistor* bietet diese Möglichkeit, in dem es den aktuellen Spielstand als Java-Objekt serialisiert und als Datei im lokalen Dateisystem speichert. Es darf aber auf keinen Fall eine direkte Abhängigkeit von *Game* zu *SerializationFilePersistor* bestehen, da dies die Dependency Rule brechen würde und außerdem zu einer starken Kopplung führen würde. Mit der Abstraktion durch das Interface ist es möglich in Zukunft andere Speichermethoden einzuführen, ohne dass sich etwas für *Game* ändert, wodurch sich die positiven Effekte der geringen Kopplung entfalten.

Abbildung 4.1: UML-Diagramm von *ase.application.PersistenceWriter*.

**Negativ-Beispiel:**

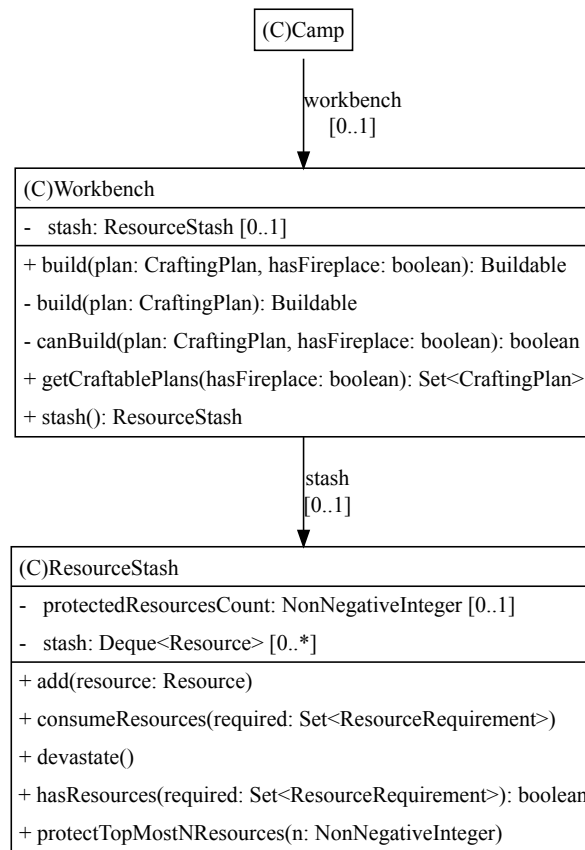
Abbildung 4.2 zeigt das Negativ-Beispiel, da eine sehr starke Kopplung zwischen *Workbench* und *ResourceStash* als direkte Abhängigkeit vorliegt<sup>1</sup>.

Aufgabe der *Workbench* ist es zu überprüfen, welche *CraftingPlans* mit den vorhandenen Ressourcen im *ResourceStash* herstellbar sind und Gegenstände mit der *build*-Methode herzustellen, indem dafür die nötigen Ressourcen vom *ResourceStash* verwendet werden. Aufgabe des *ResourceStash* ist es in erster Linie eine Datenstruktur zur Lagerung von Ressourcen bereitzustellen. Die *Workbench* nutzt dabei nur die Methoden *consumeResources* und *hasResources* des *ResourceStash*.

Wie Abbildung 4.3 zeigt, kann die hohe Kopplung einfach aufgelöst werden, indem zwischen *Workbench* und *ResourceStash* ein Interface eingezogen wird, welches die Methoden bereitstellt, die *Workbench* zum verrichten seiner Aufgaben benötigt. Dadurch könnte in Zukunft *ResourceStash* mit einer anderen Datenstruktur ausgetauscht werden, ohne dass die *Workbench* davon etwas mitbekommen würde, aufgrund der nun geringen Kopplung.

---

<sup>1</sup>Das gleiche Problem besteht auch zwischen *Camp* und *Workbench* aber hier wird nur ein Negativ-Beispiel gewünscht und behoben.

Abbildung 4.2: UML-Diagramm von *ase.domain.crafting.Workbench*.

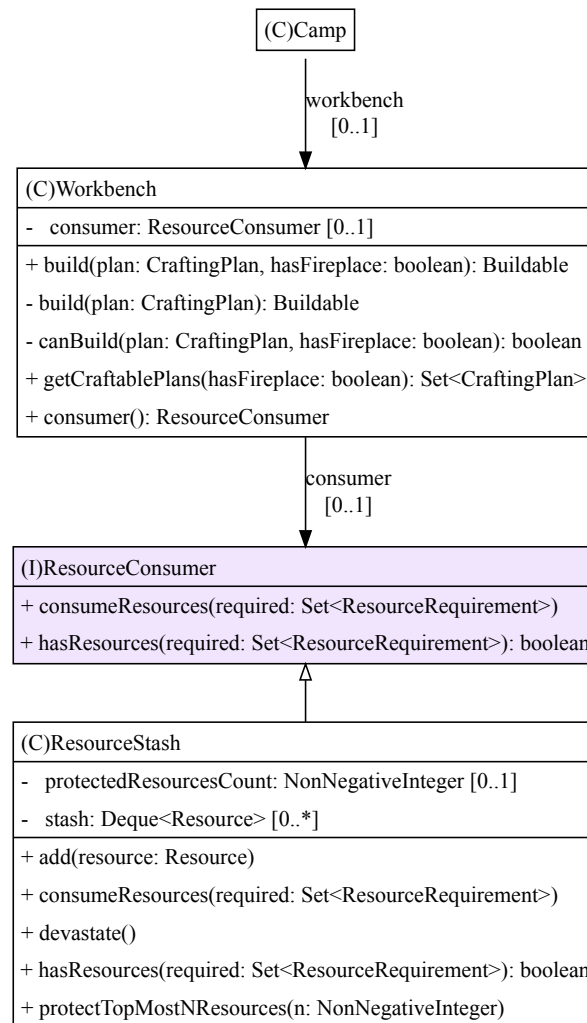


Abbildung 4.3: Abbildung 4.2 mit eingezogenem *ResourceConsumer*-Interface zur Auflösung der Kopplung.

## 4.2 Analyse GRASP: Hohe Kohäsion

Abbildung 4.4 zeigt das UML-Diagramm der Klasse *ResourceRequirement*, welches eine sehr hohe Kohäsion hat. Es handelt sich um ein Datentupel aus (Ressource, Menge), welches im *CraftingPlan*-Enum verwendet wird, um die benötigten Ressourcen für das jeweilige Buildable anzugeben. Die Kohäsion ist sehr hoch, da zu einer Bedarfsangabe immer gehört, welcher Gegenstand (*resource*) benötigt wird und wie viel davon (*amount*).

Die beiden Angaben sind im Rahmen der Bedarfsangabe semantisch maximal zusammenhängend und können inhaltlich nicht voneinander getrennt werden.

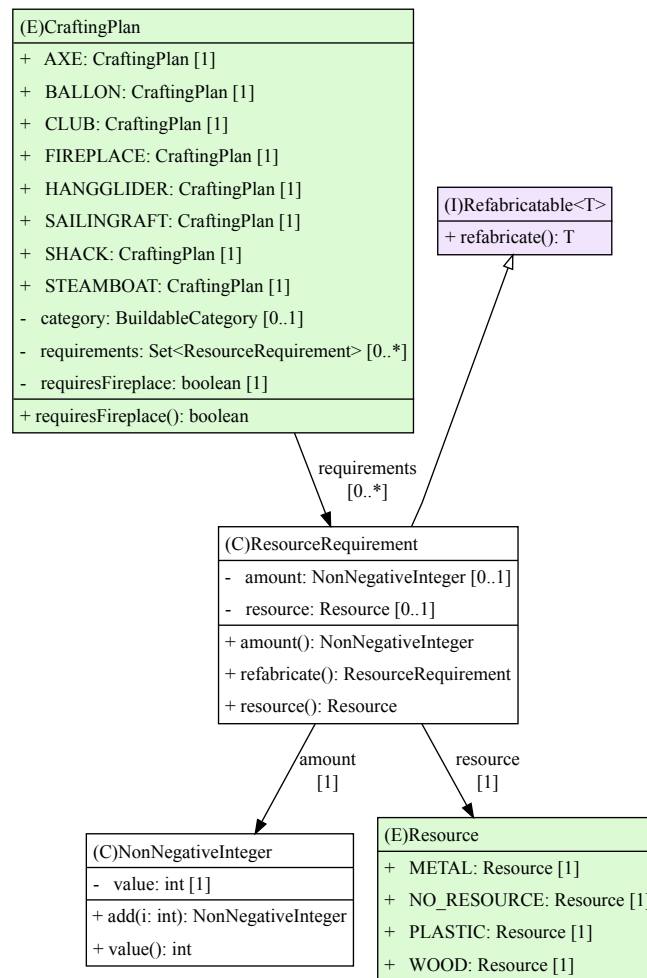


Abbildung 4.4: UML-Diagramm von *ase.domain.crafting.ResourceRequirement*.

### 4.3 Don't Repeat Yourself (DRY)

In Commit e246d77b wurde der folgende duplizierte Code aus der *Game*-Klasse gebündelt:

```
state.setPhase(GamePhase.END);
state.setStatus(GameStatus.ENDED);
state.setResult(X);
```

Code-Ausschnitt 4.1 zeigt den Code ein Commit vorher (ID: b4399813). Code-Ausschnitt 4.2 zeigt den Code mit der neuen Methode *endGameWith*, die die das Code-Duplikat an allen drei Stellen entfernt.

Es ist wichtig den Code an der Stelle anzupassen, um bei Änderungen der Game-Schlusslogik, keine Stelle ausversehen zu vergessen, was zu Bugs führen könnte. Ein Beispiel wäre, dass ein zusätzlicher Zustand gesetzt werden muss, Observern ein Statusupdate mitgeteilt werden soll, oder ein anderer Zustand gesetzt werden soll, weil sich das Modell ändert.

Durch das Entfernen der Duplikate ist der Code kürzer, einfacher, besser lesbar (sprechender Name der Methode anstatt willkürlich erscheinende Statusänderungen) und besser wartbar aus den zuvor genannten Gründen.

```
1  // ...
2  // Stelle 1 (Zeile 81-90):
3
4      if (!success) {
5          checkLose();
6          return GameResult.LOSE;
7      }
8
9      state.setPhase(GamePhase.END);
10     state.setStatus(GameStatus.ENDED);
11     state.setResult(GameResult.WIN);
12     return GameResult.WIN;
13 }
14
15 // ...
16 // Stelle 2 (Zeile 106-113):
17
18     if (state.getInvalidator().isDepleted()
19         && !state.getCamp().canBuildAnything()
20         && state.getPhase() != GamePhase.ENCOUNTER) {
21         state.setPhase(GamePhase.END);
22         state.setStatus(GameStatus.ENDED);
23         state.setResult(GameResult.LOSE);
24         return state.getInvalidator().getLastDraw();
```

```
25         }
26
27 // ...
28 // Stelle 3 (Zeile 185-193):
29
30     public void checkLose() {
31         if (state.getInvalidator().isDepleted()
32             && !state.getCamp().canBuildAnything()
33             && state.getPhase() == GamePhase.SCAVENGE) {
34             state.setPhase(GamePhase.END);
35             state.setStatus(GameStatus.ENDED);
36             state.setResult(GameResult.LOSE);
37         }
38     }
39 // ...
```

Code-Ausschnitt 4.1: DRY-Code der *Game*-Klasse zum Commit b4399813.

```
1 // ...
2 // Stelle 1 (Zeile 81-88):
3
4     if (!success) {
5         checkLose();
6         return GameResult.LOSE;
7     }
8
9     endGameWith(GameResult.WIN);
10    return GameResult.WIN;
11 }
12
13 // ...
14 // Stelle 2 (Zeile 104-109):
15
16    if (state.getInvalidator().isDepleted()
17        && !state.getCamp().canBuildAnything()
18        && state.getPhase() != GamePhase.ENCOUNTER) {
19        endGameWith(GameResult.LOSE);
```



```
20         return state.getInvalidator().getLastDraw();
21     }
22
23     // ...
24     // Stelle 3 (Zeile 181-187):
25
26     private void checkLose() {
27         if (state.getInvalidator().isDepleted()
28             && !state.getCamp().canBuildAnything()
29             && state.getPhase() == GamePhase.SCAVENGE) {
30             endGameWith(GameResult.LOSE);
31         }
32     }
33     // ...
34     // Neue Methode (189-193):
35
36     private void endGameWith(GameResult result) {
37         state.setPhase(GamePhase.END);
38         state.setStatus(GameStatus.ENDED);
39         state.setResult(result);
40     }
41     // ...
```

Code-Ausschnitt 4.2: DRY-Code der *Game*-Klasse zum Commit e246d77b.

# 5 Unit Tests

## 5.1 10 Unit Tests

Tabelle 5.1 zeigt die geforderten Tests.

Unit Test	Beschreibung
1 RollIntegerTest #from-NumberCappedTest	Testet statische <i>fromNumberCapped</i> -Funktion der <i>RollInteger</i> -Klasse, ob das richtige, ggf. beschränkte, Ergebnis erzeugt wird, bzw. illegale Eingaben abgewiesen werden.
2 Roll#raiseRollByTest	Testet <i>raiseRollBy</i> -Methode der <i>Roll</i> -Klasse, ob die Erhöhung den Domänenregeln entsprechend durchgeführt wird.
3 RollHandlerTest #encounter	Testet <i>encounter</i> -Methode des RollHandler auf verschiedenen legalen Input von gemocktem Bonusschaden, Tieren und Würfelwürfen.
4 RollHandlerTest #endeavor	Testet <i>endeavor</i> -Methode des RollHandler mit RescueMock mit verschiedenen Ergebnissen.
5 CollectionStringerTest #collectionToString-Test	Testet <i>collectionToString</i> -Methode der <i>CollectionStringer</i> -Klasse, ob die String-Konvertierung insb. mit den Zeilenumbrüchen ordentlich funktioniert.
6 DiceParserTest #parse-Test	Testet <i>parse</i> -Methode der <i>DiceParser</i> -Klasse, ob die verschiedenen Würfeltypen und Augenzahlen, sowie die Random-Roll-Funktion für die jeweiligen Würfel korrekt parsen.
7 NonNegativeIntegerTest #addTest	Testet <i>add</i> -Methode von <i>NonNegativeInteger</i> gründlich, ob richtiges Ergebnis erzeugt wird bzw. illegale Eingaben abgewiesen werden
8 NonNegativeIntegerTest #newNonNegativeIntegerTest	Testet Konstruktor von <i>NonNegativeInteger</i> ungründlich, ob Eingabe zugelassen wird.
9 ResourceStashTest #devastateTest	Testet <i>devastate</i> -Methode der <i>ResourceStash</i> -Klasse, ob die Ressourcen bis auf genau die obersten <i>n</i> gelöscht werden.
10 ResourceStashTest #hasResourcesTest	Testet <i>hasResources</i> -Methode der <i>ResourceStash</i> -Klasse, ob verschiedene <i>ResourceRequirements</i> immer korrekt behandelt werden.

Tabelle 5.1: Zehn Unit Tests mit Namen (ausgehend von *de.dhbw.karlsruhe.ase*) und Beschreibung. (Packages ausgelassen aus Platzgründen).

## 5.2 ATRIP: Automatic

*Automatic* verlangt, dass mit minimalem Aufwand die Ausführung der Unit Tests angestoßen werden kann. Hierfür wird das Unit-Testing-Framework *JUnit* verwendet, wonach Methoden erstellt werden können, die mit *@Test* annotiert sind, wodurch sie durch JUnit ausgeführt werden, wenn über die IDE oder Maven ein entsprechender Befehl erfolgt. Zur Unterstützung der automatisierten Ausführung sind die Unit Tests so geschrieben, dass sie *Repeatable* und *Independent* sind, damit es keine Probleme bei der (wiederholten) automatischen Ausführung der Tests in beliebiger Reihenfolge gibt. Außerdem haben die Tests keine externen Abhängigkeiten - auch nicht zu diversen Dateien mit Testinhalten oder Ähnliches. Nach Durchführen der Tests gibt JUnit eine Übersicht über die erfolgreichen und ggf. fehlgeschlagenen Tests. Es sind keine Nutzereingaben notwendig. Es muss nichts weiter konfiguriert werden, um die Tests durchzuführen. Der Konsolenbefehl `mvn test` genügt direkt nach dem Klonen des Repositorys.

## 5.3 ATRIP: Thorough

### Positiv-Beispiel

Code-Ausschnitt 5.1 zeigt das Positiv-Beispiel zu *thorough* Test-Code. Hier wird die *add*-Methode der *NonNegativeInteger*-Klasse getestet, die auf einen bestehenden *NonNegativeInteger* einen Java-Integer-Primitive addiert, der sowohl negativ als auch positiv sein kann (solange die Summe/Differenz nicht-negativ ist) und das Ergebnis als neuen *NonNegativeInteger* zurückgibt. Ist das Resultat negativ, soll eine *IllegalArgumentExcep-tion* geworfen werden.

Der Test ist *thorough*, weil durch den *ParameterizedTest* und die dazugehörige *Method-Source add* Stichproben getestet werden, die die Verteilung der möglichen Eingaben *gründlich* abdecken sollten, da positive Standardfälle, Edge-Cases und pathologische Eingaben getestet werden.

```
1      @ParameterizedTest
2      @MethodSource("add")
```

```

3      void addTest (NonNegativeInteger base, int addValue, ↵
        ↳ Class<? extends Throwable> expectedException, ↵
        ↳ NonNegativeInteger expectedReturn) {
4          if (expectedException != null) {
5              Assertions.assertThrowsExactly(expectedException, ↵
                ↳ () -> base.add(addValue));
6              return;
7          }
8
9          NonNegativeInteger result = base.add(addValue);
10
11         Assertions.assertEquals(expectedReturn, result);
12         Assertions.assertEquals(expectedReturn.value(), ↵
                ↳ result.value());
13     }
14
15     private static List<Arguments> add() {
16         return List.of(
17             Arguments.of(new NonNegativeInteger(0), 1, ↵
                ↳ null, new NonNegativeInteger(1)),
18             Arguments.of(new NonNegativeInteger(0), 1337, ↵
                ↳ null, new NonNegativeInteger(1337)),
19             Arguments.of(new NonNegativeInteger(42), 9, ↵
                ↳ null, new NonNegativeInteger(51)),
20             Arguments.of(new NonNegativeInteger(42), -2, ↵
                ↳ null, new NonNegativeInteger(40)),
21             Arguments.of(new NonNegativeInteger(12), -12, ↵
                ↳ null, new NonNegativeInteger(0)),
22             Arguments.of(new NonNegativeInteger(0), -1, ↵
                ↳ IllegalArgumentException.class, null),
23             Arguments.of(new NonNegativeInteger(7), -1337, ↵
                ↳ IllegalArgumentException.class, null),
24             Arguments.of(new ↵
                ↳ NonNegativeInteger(Integer.MAX_VALUE), ↵
                ↳ 0, null, new ↵
                ↳ NonNegativeInteger(Integer.MAX_VALUE)),

```

```

25      Arguments.of(new ↵
           ↳ NonNegativeInteger(Integer.MAX_VALUE), ↵
           ↳ -1, null, new ↵
           ↳ NonNegativeInteger(Integer.MAX_VALUE - 1)),
26      Arguments.of(new ↵
           ↳ NonNegativeInteger(Integer.MAX_VALUE), ↵
           ↳ 1, IllegalArgumentException.class, ↵
           ↳ null), // overflow -> = Integer.MIN_VALUE
27      Arguments.of(new ↵
           ↳ NonNegativeInteger(Integer.MAX_VALUE), ↵
           ↳ Integer.MAX_VALUE, ↵
           ↳ IllegalArgumentException.class, null), ↵
           ↳ // overflow results in -2
28      Arguments.of(new ↵
           ↳ NonNegativeInteger(Integer.MAX_VALUE), ↵
           ↳ Integer.MIN_VALUE, ↵
           ↳ IllegalArgumentException.class, null), ↵
           ↳ // results in -1
29      Arguments.of(new ↵
           ↳ NonNegativeInteger(Integer.MAX_VALUE), ↵
           ↳ Integer.MIN_VALUE + 1, null, new ↵
           ↳ NonNegativeInteger(0))
30      );
31  }

```

Code-Ausschnitt 5.1: *Thorough* Test-Code zur *add*-Methode der *NonNegativeInteger*-Klasse.

## Negativ-Beispiel

Code-Ausschnitt 5.2 zeigt das Negativ-Beispiel zu *thorough* Test-Code. Hier wird der Konstruktor der *NonNegativeInteger*-Klasse getestet, der über eine *IllegalArgumentException* sicherstellen muss, dass der übergebene Java-Integer-Primitive nicht negativ, also  $\geq 0$  ist.

Der Test ist *nicht* thorough, weil durch den Test nur ein einziger positiver Fall ( $5 \geq 0$ ) getestet wird, keine Edge-Cases (Eingabe = *Integer.MAX\_INT*, Eingabe = 0) oder

pathologischen Eingaben (z.B. Eingabe = -3) getestet werden. Die Hauptfunktionalität - also das Ablehnen von negativen Eingaben zum Einhalten des Vertrags - wird so nicht ein mal getestet und damit eigentlich nur triviales Konstruktorenverhalten.

```
1      @Test
2      void newNonNegativeIntegerTest() {
3          NonNegativeInteger actual = new NonNegativeInteger(5);
4
5          Assertions.assertEquals(5, actual.value());
6      }
```

Code-Ausschnitt 5.2: Nicht thorough Test-Code zum Konstruktor der *NonNegativeInteger*-Klasse.

## 5.4 ATRIP: Professional

### Positiv-Beispiel

Code-Ausschnitt 5.3 zeigt das Positiv-Beispiel zu *professional* Test-Code. Hier wird die *raiseRollBy*-Methode der *Roll*-Klasse getestet, die einen bestehenden *Roll* um einen *NonNegativeInteger* erhöhen soll, aber nur bis zu dem maximalen legalen Wert des bestehenden Rolls, der abhängig von dessen *DiceType* ist, und das Resultat als neuen Roll zurückgeben soll.

Der Test ist professionell da keinerlei Code-Duplication vorliegt und das SRP eingehalten ist, da der Test als *ParameterizedTest* all die Test-Cases, die durch die *MethodSource* gegeben sind, ausführt. Durch den *ParameterizedTest* ist auch leicht zu erkennen, welche Test-Cases erfolgreich waren und welche nicht. Die Variablen- und Methodennamen sind leicht verständlich und der Code ist gut lesbar. Es bestehen auch keine sonstigen Code-Smells. Zudem testet der Test eine relevante und nicht-triviale Methode, die wichtig für das Einhalten der Domänenregeln ist und deswegen korrekt sein muss, ansonsten würden Bugs verteilt in der Codebase auftreten.

```
1      @ParameterizedTest
2      @MethodSource("raiseRollBy")
3      void raiseRollByTest(Roll base, NonNegativeInteger raise, ↵
4                          ↵ Roll expected) {
```

```
4      Roll actual = base.raiseRollBy(raise);
5
6      Assertions.assertEquals(expected.roll(), actual.roll());
7      Assertions.assertEquals(expected.type(), actual.type());
8      Assertions.assertEquals(expected, actual);
9  }
10
11  private static List<Arguments> raiseRollBy() {
12      return List.of(
13          Arguments.of(
14              new Roll(DiceType.FOUR_SIDED, new ↵
15                  ↵ RollInteger(1)),
16              new NonNegativeInteger(1),
17              new Roll(DiceType.FOUR_SIDED, new ↵
18                  ↵ RollInteger(2))
19          ),
20          Arguments.of(
21              new Roll(DiceType.FOUR_SIDED, new ↵
22                  ↵ RollInteger(1)),
23              new NonNegativeInteger(10),
24              new Roll(DiceType.FOUR_SIDED, ↵
25                  ↵ DiceType.FOUR_SIDED.integerRepresentation)
26          ),
27          Arguments.of(
28              new Roll(DiceType.FOUR_SIDED, ↵
29                  ↵ DiceType.FOUR_SIDED.integerRepresentation),
30              new NonNegativeInteger(1337),
31              new Roll(DiceType.FOUR_SIDED, ↵
32                  ↵ DiceType.FOUR_SIDED.integerRepresentation)
33          ),
34          Arguments.of(
35              new Roll(DiceType.FOUR_SIDED, new ↵
36                  ↵ RollInteger(1)),
37              new NonNegativeInteger(0),
38              new Roll(DiceType.FOUR_SIDED, new ↵
39                  ↵ RollInteger(1))
40          )
41      );
42  }
```



```

32         ),
33         Arguments.of(
34             new Roll(DiceType.SIX_SIDED, new ↵
35                 ↵ RollInteger(3)),
36             new NonNegativeInteger(2),
37             new Roll(DiceType.SIX_SIDED, new ↵
38                 ↵ RollInteger(5))
39         ),
40         Arguments.of(
41             new Roll(DiceType.SIX_SIDED, new ↵
42                 ↵ RollInteger(3)),
43             new NonNegativeInteger(10),
44             new Roll(DiceType.SIX_SIDED, ↵
45                 ↵ DiceType.SIX_SIDED.integerRepresentation)
46         ),
47         Arguments.of(
48             new Roll(DiceType.EIGHT_SIDED, new ↵
49                 ↵ RollInteger(7)),
50             new NonNegativeInteger(10),
51             new Roll(DiceType.EIGHT_SIDED, ↵
52                 ↵ DiceType.EIGHT_SIDED.integerRepresentation)
53         )
54     );
55 }

```

Code-Ausschnitt 5.3: Professioneller Test-Code zur *raiseRollBy*-Methode der *Roll*-Klasse.

## Negativ-Beispiel

Code-Ausschnitt 5.4 zeigt das Negativ-Beispiel zu *professional* Test-Code. Hier wird die statische *fromNumberCapped*-Funktion der *RollInteger*-Klasse getestet, die aus einem Java-Integer-Primitive und einem *DiceType* einen neuen *RollInteger* erstellen soll, dessen Wert dem Primitive entspricht, aber maximal dem Maximalwert des übergebenen *DiceType*. Zahlen  $< 1$  sollen durch eine *IllegalArgumentException* abgefangen werden.

Der Test ist unprofessionell aus folgenden Gründen:

1. Es besteht extrem viel Code-Duplication, da der Test wohl durch Copy-Paste erzeugt wurde. Dies sollte durch eine Method-Extraction behoben werden.
2. Alle Test-Cases stehen in der einen Test-Methode (→ SRP verletzt), wodurch es schwieriger wird bei teilweise fehlschlagenden Assertions zu erkennen, wo der Fehler liegt. Die Test-Cases sollten in eigene Tests ausgelagert werden, sodass das SRP gewahrt wird. Zusammen mit dem vorherigen Punkt sollte also ein *ParameterizedTest* mit *MethodSource* für die Test-Cases erstellt werden.
3. Es werden die temporären Variablen *inputInteger* und *inputDiceType* nicht konsequent angewandt, sondern trotzdem noch magische Konstanten (s. z.B. Zeile 15, wo eine 2 steht, statt *inputInteger*).
4. Bei den Tests, bei denen der *inputInteger* das Maximum des Würfeltyps überschreitet (z.B. Block Zeile 17 ff.), sollte keine magische Konstante (im Beispiel 6 für *DiceType.SIX\_SIDED*) als erwarteter Wert verwendet werden, sondern *inputDiceType.integerRepresentation*, welche das Maximum darstellt.
5. In Zeile 35 steht ein Multiline-Kommentar mitten im Code, um eine magische Konstante zu erklären. Um diesen Code-Smell aufzuheben, sollte die Konstante *Integer.MIN\_VALUE* eingefügt werden.

```
1      @Test
2      void fromNumberCappedTest() {
3          int inputInteger = 2;
4          DiceType inputDiceType = DiceType.FOUR_SIDED;
5          RollInteger expected = new RollInteger(2, ↵
              ↵ DiceType.FOUR_SIDED);
6          RollInteger actual = ↵
              ↵ RollInteger.fromNumberCapped(inputInteger, ↵
              ↵ inputDiceType);
7          Assertions.assertEquals(expected, actual);
8          Assertions.assertEquals(2, actual.value());
9
10         inputInteger = 2;
11         inputDiceType = DiceType.SIX_SIDED;
12         expected = new RollInteger(2, DiceType.SIX_SIDED);
```

```
13      actual = RollInteger.fromNumberCapped(inputInteger, ↵
        ↵ inputDiceType);
14      Assertions.assertEquals(expected, actual);
15      Assertions.assertEquals(2, actual.value());
16
17      inputInteger = 100;
18      inputDiceType = DiceType.SIX_SIDED;
19      expected = new RollInteger(6, DiceType.SIX_SIDED);
20      actual = RollInteger.fromNumberCapped(inputInteger, ↵
        ↵ inputDiceType);
21      Assertions.assertEquals(expected, actual);
22      Assertions.assertEquals(6, actual.value());
23
24      inputInteger = 100;
25      inputDiceType = DiceType.EIGHT_SIDED;
26      expected = new RollInteger(8, DiceType.EIGHT_SIDED);
27      actual = RollInteger.fromNumberCapped(inputInteger, ↵
        ↵ inputDiceType);
28      Assertions.assertEquals(expected, actual);
29      Assertions.assertEquals(8, actual.value());
30
31      Assertions.assertThrows(IllegalArgumentException.class,
32          () -> RollInteger.fromNumberCapped(0, ↵
            ↵ DiceType.FOUR_SIDED));
33
34      Assertions.assertThrows(IllegalArgumentException.class,
35          () -> RollInteger.fromNumberCapped(0x80000000 ↵
            ↵ /* min integer */, DiceType.FOUR_SIDED));
36
37      Assertions.assertThrows(IllegalArgumentException.class,
38          () -> RollInteger.fromNumberCapped(-1, ↵
            ↵ DiceType.FOUR_SIDED));
39    }
```

Code-Ausschnitt 5.4: **Unprofessioneller** Test-Code zur statischen *fromNumberCapped*-Funktion der *RollInteger*-Klasse.

## 5.5 Code Coverage

Line	Method	Class
91% (611/671)	95% (237/249)	100% (67/67)

Tabelle 5.2: Code-Coverage der Tests relativ und absolut nach Statement-, Method- und Class-Coverage.

Tabelle 5.2 zeigt die Code-Coverage aller Tests im Projekt nach Statement-, Method- und Class-Coverage sowohl als relativer (prozentualer) Wert, als auch als absoluter Wert. Der Wert von 91% Statement-Coverage wird als gut für das konkrete Projekt angesehen. Insbesondere, wenn beachtet wird, welche Zeilen nicht abgedeckt sind: zumeist das Werfen von Runtime-Exceptions oder das Behandeln von Exceptions im Plugin-Code in Form von Fehlerausgaben an den Nutzer. Es gibt zwar kein allgemein anerkannten Wert für eine „gute“ Code-Coverage, aber für ein brettspielartiges Kartenspiel ist dies ein ausreichender Wert, da keine größeren Bugs vorhanden sein sollten und das Spiel im Großen und Ganzen gut funktionieren sollte, abgesehen von möglicherweise ein paar wenigen kleineren Bugs. Insbesondere die Blackbox-Integration-Tests sichern einen normalen Spielverlauf zu. Da es sich außerdem um kein Online-Spiel handelt und es keine Leaderboards gibt, muss das Spiel auch nicht zu sehr auf diverse kleinere Exploits oder Glitches getestet werden. Für andere, sicherheitskritische Software wäre eine Code-Coverage von 91% sicher unzureichend, aber für dieses Anwendung ist es ausreichend.

## 5.6 Fakes und Mocks

### CampMock

Abbildung 5.1 zeigt das UML-Diagramm eines Mocks zum Testen der *encounter*-Methode des *RollHandler*. Der *CampMock* ist realisiert als Subklasse der zu mockenden Oberklasse *Camp*, wobei *CampMock* die relevanten zu mockenden Methoden überschreibt. Konkret ist das hier lediglich die *getBonusDamage*-Methode, die im Mock das private Attribut *bonusDamage* zurückgibt, welches durch den Konstruktor gesetzt wird.

Grund für den Mock ist, dass in der *encounter*-Methode auf die *getBonusDamage*-Methode

von Camp zurückgegriffen wird, um zu bestimmen, ob ein Spieler gegen ein Tier gewinnt oder nicht. Das Setup, um ein Camp zu erzeugen, wobei die `getBonusDamage`-Methode tatsächlich einen gewünschten Wert zurückgibt wäre aber ohne Mock unverhältnismäßig schwierig, da zunächst genügend Ressourcen in das Camp eingefüllt werden müssten, dass es reicht ein *Tool* zu bauen, wovon der `bonusDamage` genommen wird. Außerdem könnten dann nur bereits über Tools implementierte Werte für `getBonusDamage` getestet werden und nicht beliebige. Daher wird hier ein Mock benötigt.

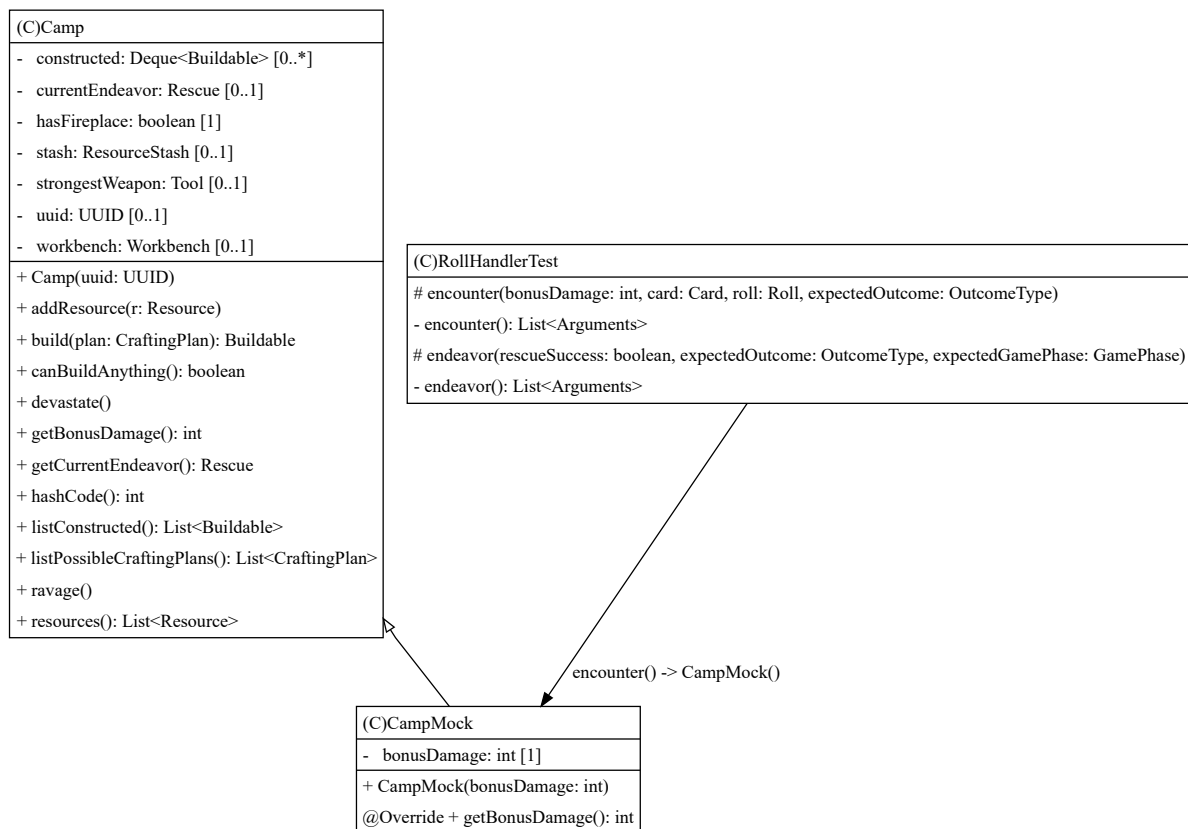


Abbildung 5.1: UML-Diagramm des *CampMock*, der zur Testung der *encounter*-Methode des *RollHandler* benötigt wird.

## RescueMock

Abbildung 5.2 zeigt das UML-Diagramm eines Mocks zum Testen der *endeavor*-Methode des *RollHandler*. Der *RescueMock* ist realisiert als Implementation des zu mockenden Interfaces *Rescue*, wobei *RescueMock* die *endeavor*-Methode so überschreibt, dass ein

Boolean-Wert zurückgegeben wird, der zuvor über den Konstruktor von `RescueMock` injiziert wurde.

Grund für den Mock ist, dass in der `endeavor`-Methode auf die `getCurrentEndeavor`-Methode von `Camp` zurückgegriffen wird, um auf dem zurückgegebenen Interface die `endeavor`-Methode aufzurufen, um zu überprüfen, ob der Spieler von der Insel entfliehen kann. Hierfür wird also in `Camp` der `RescueMock` injiziert, sodass `getCurrentEndeavor` den Mock zurückgibt. Das Setup, um ein `Camp` zu erzeugen, welches eine `Rescue` hält, die einen gewünschten Wert zurückgibt wäre aber ohne Mock unverhältnismäßig schwierig, da zunächst genügend Ressourcen in das `Camp` eingefüllt werden müssten, dass es reicht eine `Rescue` zu bauen, deren `endeavor`-Methode den gewünschten Wert zurück gibt. Außerdem könnten dann nur bereits über `Rescues` implementierte Verhalten für `Rescue#endeavor` getestet werden und nicht beliebige. Zusätzlich würde der Test dann auch die `Rescue#endeavor`-Methode der konkreten `Rescues` testen und nicht nur isoliert die `RollHandler#endeavor`-Methode. Daher wird hier ein Mock benötigt.

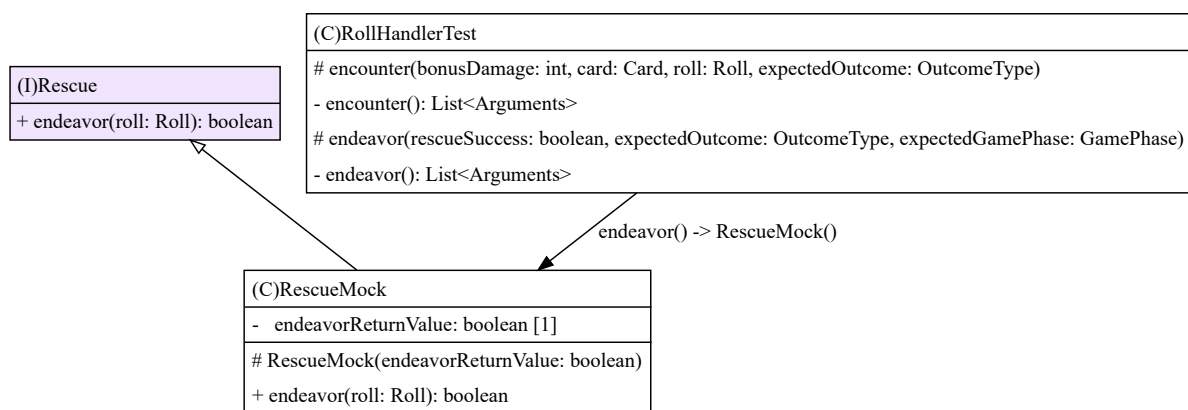


Abbildung 5.2: UML-Diagramm des *RescueMock*, der zur Testung der *endeavor*-Methode des *RollHandler* benötigt wird.

## 6 Domain Driven Design

### 6.1 Ubiquitous Language (UL)

#### Card

**Bedeutung:** Ein Objekt, welches von einer zuvor zufällig zusammengestellten Menge an Karten (Kartenstapel = CardDeck) erhalten (gezogen = draw) werden kann und dann durch einen Effekt den Spielverlauf beeinflusst.

**Begründung:** Card gehört zur UL, da es wichtig ist, dass Domänenexperten (die Spiel-Designer) und die Softwareentwicklern über Kartentypen und deren Effekte sprechen können, wenn neue solcher Karten hinzugefügt werden sollen, oder bestehende verstanden werden sollen.

#### AnimalEncounter

**Bedeutung:** Eine Spielphase (= GamePhase) die durch den Karteneffekt einer Karte des Typs Tier (= Animal) ausgelöst wird und erst verlassen werden kann, in dem der RollDx-Befehl erfolgreich ausgeführt wird (Kampf gegen das Tier).

**Begründung:** AnimalEncounter ist Teil des UL, da die Spielphasen (vgl. Abbildung 1.1) im Software-Modell direkt und unverändert übernommen werden, um keine Übersetzung im Spielverlauf zwischen Software- und Kartenspiel haben zu müssen, was die Kommunikation sehr erschweren würde und Missverständnisse provozieren könnte.

#### Axe

**Bedeutung:** Ein Objekt, welches das Tool- und Buildable-Interface implementiert, um so unter bestimmten Voraussetzungen erzeugt (crafted = hergestellt) werden zu können und im Spiel bessere Chancen (bonus damage = zusätzlicher Schaden) in der Encounter-Spielphase zu ermöglichen.

**Begründung:** Axe ist (wie das andere Tool *Club* auch) Teil der UL, um die Klasse in der gemeinsamen Kommunikation mit den Domain-Experten identifizieren zu können. Dieses Objekt benötigt in jedem Fall einen Namen und statt z.B. *StrongProbabilityIncreaser* ist Axe griffiger und leichter zu merken - auch für die Entwickler.

## Roll

**Bedeutung:** Ein Verbundsobjekt aus einer Zufallsvariable eines diskreten natürlich-zahligen Intervalls (Würfel-Art = Roll.type) und einer assoziierten Realisierung dieser Zufallsvariable (Augenzahl = Roll.roll).

**Begründung:** Roll als Würfelwurf ist in der UL, da es einem Würfeltyp eine (Augen-)Zahl zuordnet, wodurch eine technischere Bezeichnung, wie *BoundedRandomInteger* keine Assoziation zu den verschiedenen Würfeln (vier-, sechs- und achtseitig) zulässt, die aber in der weiteren Entwicklung und Kommunikation mit den Domänenexperten wichtig sind. Außerdem beschreibt der Name den Zweck und den Ursprung, anders als der generische *BoundedRandomInteger*, der für alles stehen könnte und womit die Domänenexperten (und auch Entwickler) deutlich weniger sich vorstellen und anfangen könnten.

## 6.2 Entities

Abbildung 6.1 zeigt die *GameState*-Klasse. Sie beinhaltet den ganzen Zustand des Spiels und wird genutzt um das Spiel zu speichern (und dann wieder zu laden), da alle relevanten Objekte hier verwurzelt sind. Dabei manipuliert die Game-Klasse den GameState und nicht die Klasse sich selbst. Somit ist kein kaum Verhalten in der Klasse realisiert, was ein Punkt einer Entity ist. Diese Klasse befindet sich zwar nicht in der Domain-Schicht, aber hat sonst alle Merkmale einer Entity und aus gutem Grund. Die Identität eines GameStates soll nur von einer ID (hier der Surrogatschlüssel *uuid* vom Typ *UUID*) abhängen, da der Hauptzweck der Klasse die einfache Speicherbarkeit des Spielzustandes ist und ein genau gleicher Spielzustand von den Attributen her nicht als identisch angesehen werden soll, wenn die ID nicht stimmt, um unterschiedliche Speicherungen (selbst bei zufällig identischen Attributen) unterscheiden zu können. Folglich wurden in GameState *hashCode* und *equals* im Sinne der Identität einer Entität überschrieben, sodass diese nur vom



Surrogatschlüssel *uuid* abhängen. Außerdem hat der GameState einen Lebenszyklus der insbesondere über die GamePhase und den GameStatus, aber auch sonstige Attribute, ausgedrückt wird.

Dies ist der einzige wirklich sinnvolle Einsatz von Entitäten im Programm, auch wenn es sich in der Application-Schicht befindet. Die Klasse *Camp* wurde ebenfalls als Entität implementiert, wobei der Schlüssel allerdings nicht annähernd so eine wichtige und sinnvolle Bedeutung hat wie beim GameState. Darüber hinaus sollte es pro GameState auch nur ein und nicht mehrere Camps geben, weswegen das zum GameState gehörige Camp die gleiche uuid erhält, wie der GameState. Da es aber faktisch keine anderen Camps in einer Instanz der Applikation gibt, werden diese auch nie verglichen, weswegen diese Entität bisher wenig sinnvoll ist.

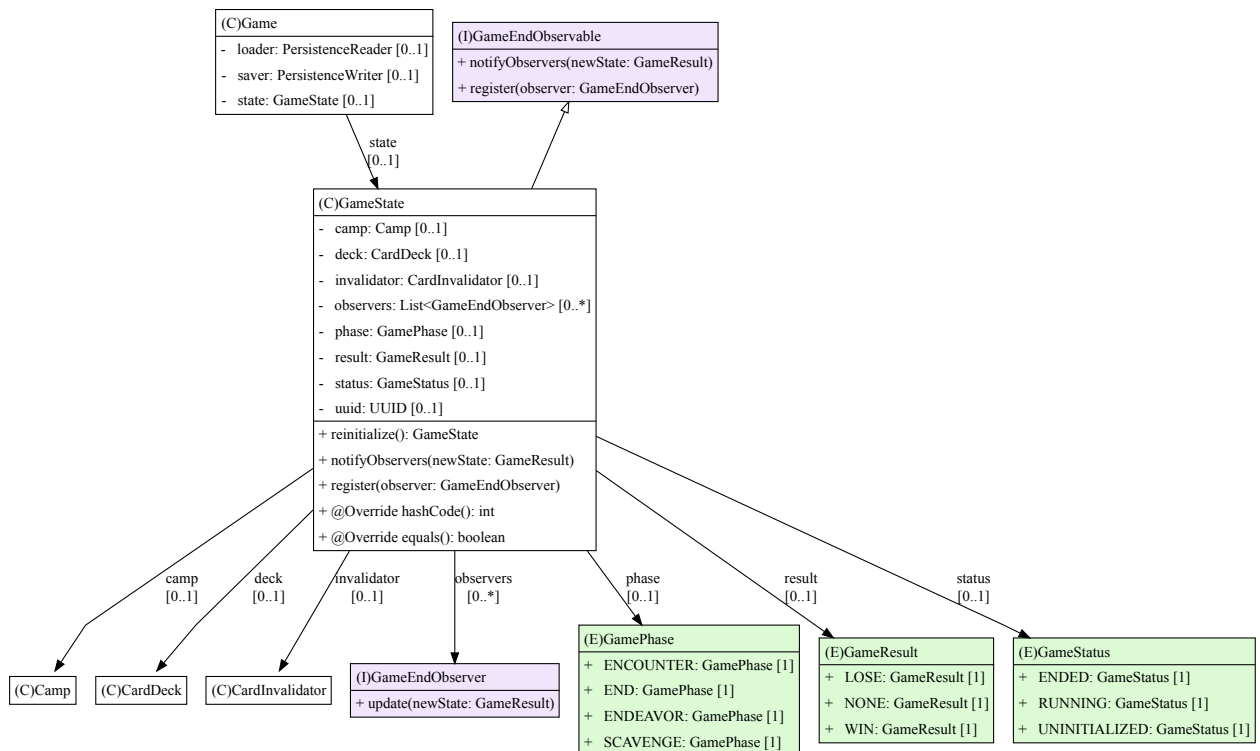
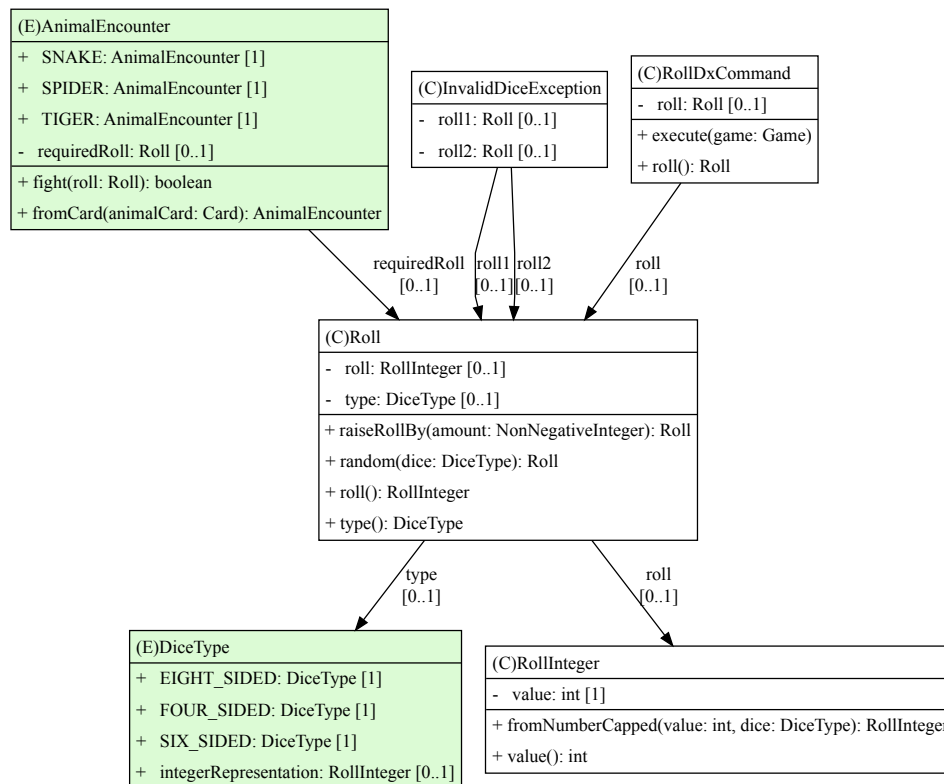


Abbildung 6.1: UML-Diagramm von *GameState*.

## 6.3 Value Objects (VO)

Abbildung 6.2 zeigt das VO *Roll*, welches in Java als *Record* implementiert ist. Demnach ist *hashCode* und *equals* bereits implizit entsprechend überschrieben, weswegen es nicht extra im Diagramm auftaucht. Das immutable VO *Roll* definiert eine Augenzahl (*Roll.roll*) für einen bestimmten Würfeltyp (*Roll.type*) und stellt die Domänen-Constraint sicher, dass Augenzahlen  $a$  stets im Intervall von eins bis Seitenanzahl des Würfels  $s$  liegen muss:  $a \in [1, s]$ ;  $s \geq 1$ . Dieser Constraint insbesondere wird dabei in das VO *RollInteger* ausgelagert. Weiter bietet *Roll* verschiedene Methoden bereit um Verhalten bezüglich Rolls durchzusetzen. Beispielsweise kann die Augenzahl mit der Methode *raisRollBy* erhöht werden, wobei ein neues *Roll*-Objekt mit der erhöhten Augenzahl aber maximal  $Roll.roll = s$  zurückgegeben wird.

*Roll* wurde also als VO realisiert, da es so das Domänenkonzept kapselt und verdeutlicht, die Domänenregeln eingehalten werden, Domänenverhalten umgesetzt wird, keine Seiteneffekte auftreten und der *Roll* einfach mit anderen Rolls (z.B. dem benötigten *Roll*, um gegen ein bestimmtes Tier zu gewinnen (s. Klasse *AnimalEncounter*)) verglichen werden kann.

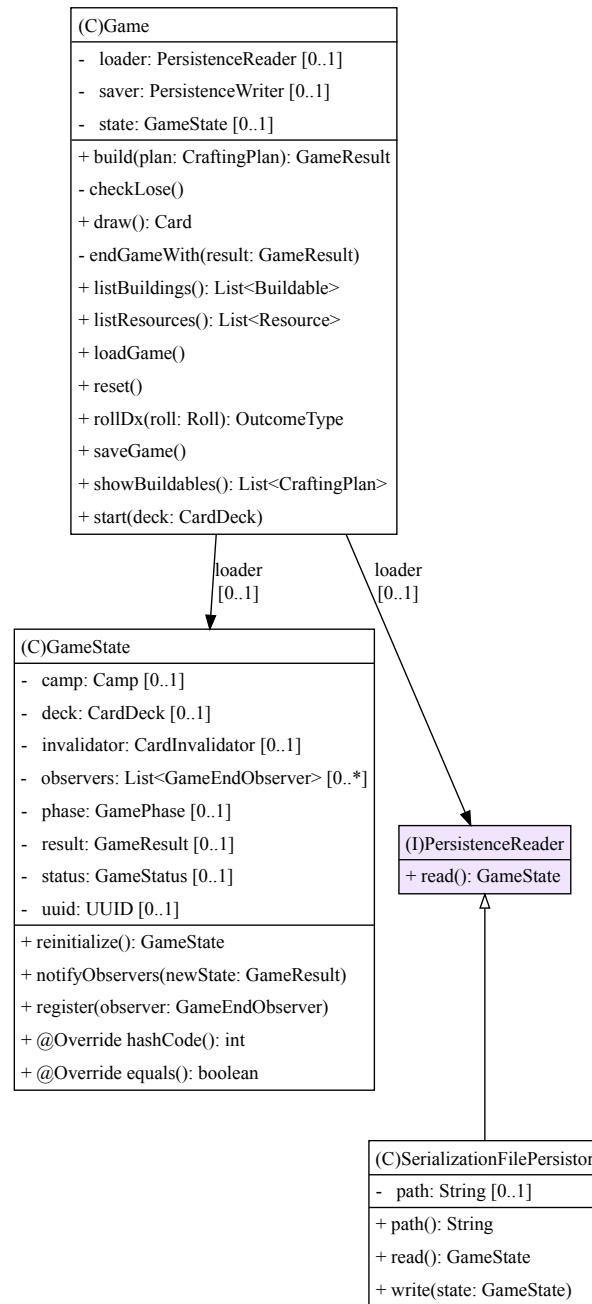
Abbildung 6.2: UML-Diagramm von dem Value Object *Roll*.

## 6.4 Repositories

Abbildung 6.3 zeigt das *PersistenceReader*-Interface, was eine Art „Repository“ auf Application-Layer-Ebene darstellt. Wie schon in Abschnitt 6.2 angedeutet, spielt die Speicherung des GameStates die zentrale Rolle, um ein Spiel wiederherzustellen und später fortzusetzen. Es ergibt allerdings wenig Sinn, die einzelnen Aggregate der Domain-Schicht einzeln zu persistieren, den GameStatus, die GamePhase der Applikationsschicht einzeln zu persistieren und dann später beim Laden zusammenzupflücken und einzeln wieder zusammenzubauen. Um das Spiel zu Speichern muss lediglich der GameState (der alle vorher genannten Objekte enthält) gespeichert werden. Dieser ist aber - und aus gutem Grund - Teil der Applikationsschicht, weswegen PersistenceReader per Definition erst mal kein Repository sein kann (Repository-Interfaces müssen im Domain-Code definiert sein). Hier ist es aber sinnvoll nicht den puristischen Ansatz zu wählen, sondern die Konzepte etwas aufzuweichen, sodass PersistenceReader ein Repository darstellt, um den

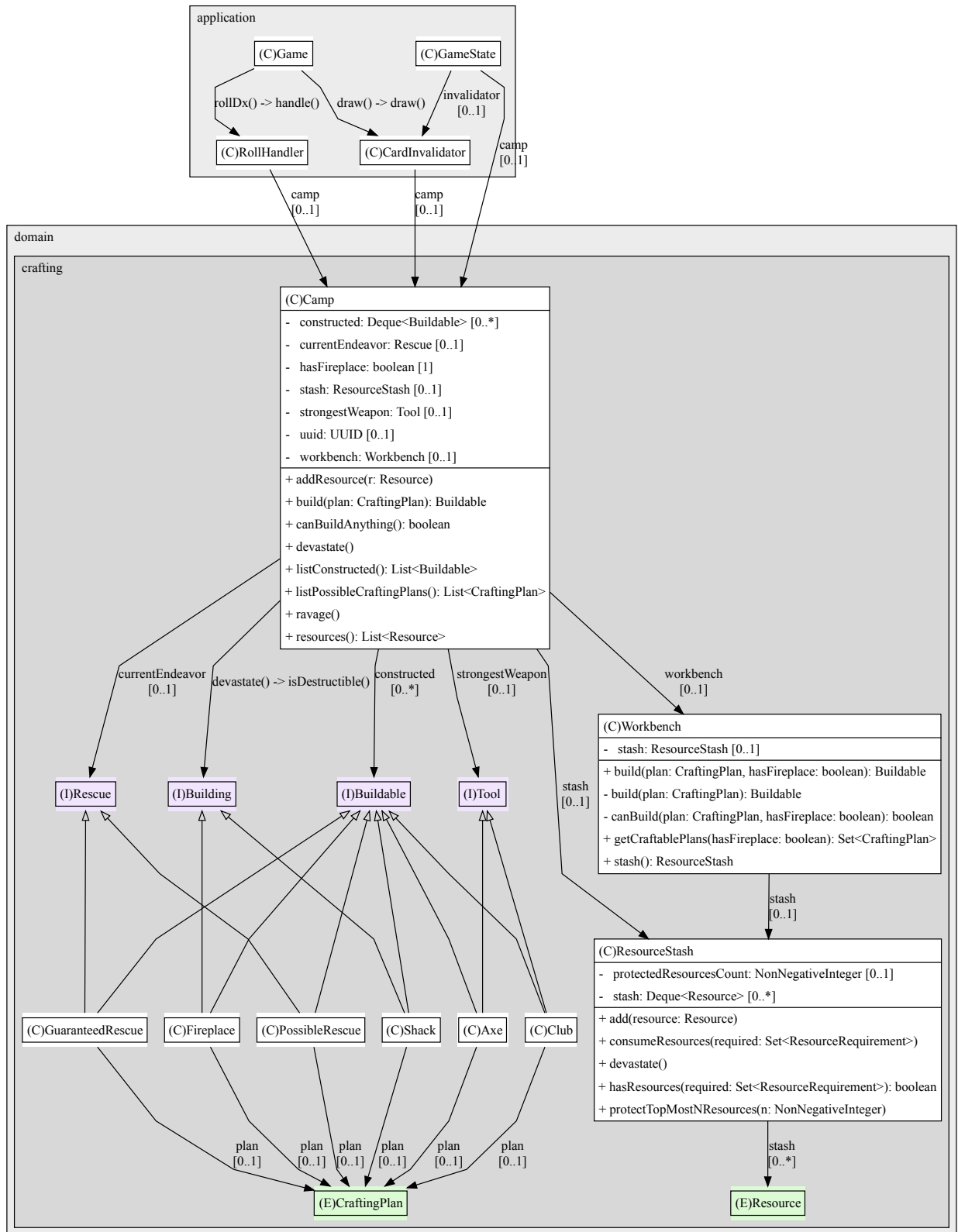
GameState zu lesen.

Das Repository ist nötig, um geringe Kopplung (vgl. *PersistenceWriter* aus Abschnitt 4.1) zu gewährleisten, das OCP und die Dependency Rule einzuhalten. Somit kann die Implementation des PersistenceReader in der Plugin-Schicht beliebig getauscht werden, ohne dass die höheren Schichten (Applikation und Domain) etwas davon wissen müssen oder gar von den Details abhängen müssen und stattdessen von der Abstraktion abhängen können.

Abbildung 6.3: UML-Diagramm von dem Repository *PersistenceReader*.

## 6.5 Aggregates

Abbildung 6.4 zeigt das *crafting*-Aggregat mit der Root-Entity *Camp*. Es ist deutlich zu sehen, dass Zugriff auf das Aggregat durch außenstehende Klassen nur über die Root-Entity *Camp* möglich ist. Hierfür bietet *Camp* öffentliche Methoden an, die lediglich an die anderen Klassen des Aggregats delegieren. Dadurch können die Domain-Regeln, die sich auf das Aggregat beziehen zentral in den öffentlichen Methoden von *Camp* überprüft werden. Außerdem werden die Zugriffe auf das Crafting-Aggregat und damit auch die Crafting-Funktionalität vereinfacht und die Objektbeziehungen werden entkoppelt, dadurch, dass für außenstehende Klassen nur das *Camp* existiert als Kopplungspartner und die Komplexität im Inneren des Aggregats versteckt wird.

Abbildung 6.4: UML-Diagramm vom *crafting*-Aggregat mit *Camp* als Root-Entity.

# 7 Refactoring

## 7.1 Code-Smells

### Long-Class

Code-Ausschnitt 7.1 und Code-Ausschnitt 7.2 zeigen den Code eines Long-Class-Code-Smells vor (Commit 174283ab) und nach (Commit b0ce98af) der Behebung mittels Aufteilen der Klasse in drei neue Klassen, die aber jeweils ein gemeinsames Interface (*Parser*) implementieren.

Vorher gab es eine *ArgumentParser*-Klasse, die als Methoden-Sammlung für das Parsen von *CraftingPlans* (*parseConstructible*), *Dice* (*parseDice*) und *Cards* (*parseCards*) zuständig war. Hierdurch war die Klasse nicht SRP- oder OCP-konform und trug den Long-Class-Code-Smell.

Dies wurde in Code-Ausschnitt 7.2 gelöst, indem die *ArgumentParser*-Klasse aufgeteilt wurde in *CardParser*, *CraftingPlanParser* und *DiceParser*, die jeweils das *Parser*-Interface implementieren und somit nur die *parse*-Methode haben. (Außerdem wurden die *switch*-Statements gleich mit aufgelöst, aber dieser Code-Smell wird hier nicht betrachtet). Dadurch wahrt die neue Lösung das SRP und OCP und es ist keine long-class mehr.

```
1 public class ArgumentParser {
2
3     public CraftingPlan parseConstructible(final String type) {
4         switch (type) {
5             case "axe":
6                 return CraftingPlan.AXE;
7             case "club":
8                 return CraftingPlan.CLUB;
9             case "shack":
10                return CraftingPlan.SHACK;
11             case "fireplace":
12                return CraftingPlan.FIREPLACE;
13             case "sailingraft":
```



```
14         return CraftingPlan.SAILINGRAFT;
15     case "hangglider":
16         return CraftingPlan.HANGGLIDER;
17     case "steamboat":
18         return CraftingPlan.STEAMBOAT;
19     case "ballon":
20         return CraftingPlan.BALLON;
21     default:
22         throw new UnsupportedOperationException(type + ↵
                ↵ " has not been implemented");
23     }
24 }
25
26 public Dice parseDice(final Matcher mchr) {
27     final String roll;
28     final DiceType diceType;
29
30     if (mchr.group(1) != null) {
31         diceType = DiceType.FOUR_SIDED;
32         roll = mchr.group(2);
33     } else if (mchr.group(3) != null) {
34         diceType = DiceType.SIX_SIDED;
35         roll = mchr.group(4);
36     } else {
37         diceType = DiceType.EIGHT_SIDED;
38         roll = mchr.group(6);
39     }
40
41     return new Dice(diceType, Integer.parseInt(roll));
42 }
43
44 public Card parseCard(final String type) {
45     switch (type) {
46         case "wood":
47             return Card.WOOD;
48         case "metal":
```

```
49         return Card.METAL;
50     case "plastic":
51         return Card.PLASTIC;
52     case "spider":
53         return Card.SPIDER;
54     case "snake":
55         return Card.SNAKE;
56     case "tiger":
57         return Card.TIGER;
58     case "thunderstorm":
59         return Card.THUNDERSTORM;
60     default:
61         throw new UnsupportedOperationException(type + ↵
        ↵ " has not been implemented");
62     }
63 }
64 }
```

Code-Ausschnitt 7.1: Long-Class-Code-Smell der *ArgumentParser*-Klasse zum Commit 174283ab.

```
1 public interface Parser<T, U> {
2     T parse(U raw);
3 }
4
5 public final class CardParser implements Parser<Card, String> {
6
7     @Override
8     public Card parse(String raw) {
9         for (var card : Card.values()) {
10             if (card.toString().equals(raw)) {
11                 return card;
12             }
13         }
14
15         throw new UnsupportedOperationException(raw + " has ↵
        ↵ not been implemented");
```

```
16     }
17 }
18
19 public final class CraftingPlanParser implements ↵
    ↳ Parser<CraftingPlan, String> {
20
21     @Override
22     public CraftingPlan parse(final String raw) {
23         for (var plan : CraftingPlan.values()) {
24             if (plan.toString().equals(raw)) {
25                 return plan;
26             }
27         }
28
29         throw new UnsupportedOperationException(raw + " has ↵
            ↳ not been implemented");
30     }
31 }
32
33 public final class DiceParser implements Parser<Dice, Matcher> {
34
35     @Override
36     public Dice parse(final Matcher raw) {
37         final String roll;
38         final DiceType diceType;
39
40         if (raw.group(1) != null) {
41             diceType = DiceType.FOUR_SIDED;
42             roll = raw.group(2);
43         } else if (raw.group(3) != null) {
44             diceType = DiceType.SIX_SIDED;
45             roll = raw.group(4);
46         } else {
47             diceType = DiceType.EIGHT_SIDED;
48             roll = raw.group(6);
49         }
```

```
50
51         return new Dice(diceType, Integer.parseInt(roll));
52     }
53
54 }
```

Code-Ausschnitt 7.2: Long-Class-Code-Smell behoben durch Aufteilung der Klassen zum Commit b0ce98af.

## Switch-Statement

Code-Ausschnitt 7.3 und Code-Ausschnitt 7.4 zeigen den Code eines Switch-Statement-Code-Smells vor (Commit dd2a39d2) und nach (Commit a0e99692) der Behebung mittels Einführung einer Map, die das Zählen dynamisch übernimmt.

Vorher gab es nur die *CardDeck*-Klasse, die mit der *isValid*-Methode überprüft, ob die Karten-Konfiguration (also die Anzahl der verschiedenen Karten im Deck) zulässig ist. Das Zählen der Karten funktionierte über ein Switch-Statement, welches einen explizit für diesen Kartentyp vorgesehenen Zähler inkrementierte, wenn die Karte bei Iteration über die Sammlung von eben diesem Typ war. Der Vergleich auf Validität erfolgte über eine Kaskade von verundeten Gleichheitstests mit den erwünschten Werten. Diese Methode war furchtbar unerweiterbar.

Die Lösung in Code-Ausschnitt 7.4 funktioniert so: Es wurde das neue VO *CardDeckConfiguration* eingeführt, welches eine immutable Map von Card -> Integer mappt und ermöglicht zu überprüfen, ob zwei CardDeckConfigurations wertetechnisch gleich sind (*record's* implizites *equals*). Zur besseren Vergleichbarkeit hat CardDeckConfiguration die *withoutZeroOccurrences*-Methode, um Karten, die null Mal vorkommen, zu entfernen. CardDeck hat nun das Attribut *config*, welches die erwünschte CardDeckConfiguration beinhaltet. Der Switch in *isValid* wurde nun ersetzt, indem durch einen Loop über das CardDeck die Anzahl an Karten eines Typs dynamisch über eine Map-Struktur gezählt werden (Kartentyp ist der Schlüssel) und dann schließlich die Ist- und Soll-Konfiguration über ein *equals*-Aufruf verglichen werden. Hierdurch wurde der Switch-Code-Smell entfernt und CardDeck ist nun OCP-konform, da nichts an der Klasse (also der frühere Switch und die Zähler pro Kartentyp und deren Vergleich) geändert werden muss, wenn ein neuer Kartentyp hinzugefügt wird.

```
1 public class CardDeck implements Deck<Card> {
2     // the numbers of occurrences of each card in the deck
3     private static final int NUMBER_OF_WOOD = 16;
4     private static final int NUMBER_OF_METAL = 16;
5     private static final int NUMBER_OF_PLASTIC = 16;
6     private static final int NUMBER_OF_SPIDER = 5;
7     private static final int NUMBER_OF_SNAKE = 5;
8     private static final int NUMBER_OF_TIGER = 5;
9     private static final int NUMBER_OF_THUNDERSTORM = 1;
10
11     private final Deck<Card> deck = new RefabDeck<>();
12
13     // ...
14
15     public boolean isValid() {
16         int wood = 0;
17         int metal = 0;
18         int plastic = 0;
19         int spider = 0;
20         int snake = 0;
21         int tiger = 0;
22         int thunderstorm = 0;
23
24         for (final Card card : this) {
25             switch (card) {
26                 case WOOD:
27                     wood++;
28                     break;
29                 case METAL:
30                     metal++;
31                     break;
32                 case PLASTIC:
33                     plastic++;
34                     break;
35                 case SPIDER:
36                     spider++;
```

```

37             break;
38         case SNAKE:
39             snake++;
40             break;
41         case TIGER:
42             tiger++;
43             break;
44         case THUNDERSTORM:
45             thunderstorm++;
46             break;
47         default:
48             throw new ↵
                    ↳ UnsupportedOperationException(card + ↵
                    ↳ " has not been implemented yet");
49     }
50 }
51
52     return wood == NUMBER_OF_WOOD && metal == ↵
        ↳ NUMBER_OF_METAL && plastic == NUMBER_OF_PLASTIC
53         && spider == NUMBER_OF_SPIDER && snake == ↵
        ↳ NUMBER_OF_SNAKE && tiger == NUMBER_OF_TIGER
54         && thunderstorm == NUMBER_OF_THUNDERSTORM;
55 }
56 }

```

Code-Ausschnitt 7.3: Switch-Statement-Code-Smell der *CardDeck*-Klasse zum Commit dd2a39d2.

```

1 public record CardDeckConfiguration(Map<Card, Integer> ↵
    ↳ cardOccurrences) {
2
3     public CardDeckConfiguration {
4         cardOccurrences = Map.copyOf(cardOccurrences); // ↵
5         ↳ Ensures deep immutability
6         // ...
7     }

```

```
8      public CardDeckConfiguration ↵
          ↳ withoutZeroOccurrenceEntries() {
9          final Map<Card, Integer> newOccurrences = new ↵
              ↳ HashMap<>();
10
11      cardOccurrences.forEach((card, amount) -> {
12          if (amount == 0) return;
13          newOccurrences.put(card, amount);
14      });
15
16      return new CardDeckConfiguration(newOccurrences);
17  }
18 }
19
20 public class CardDeck implements Deck<Card> {
21
22     private final Deck<Card> deck = new RefabDeck<>();
23     private final CardDeckConfiguration config;
24
25     public CardDeck() {
26         this(DEFAULT_CONFIGURATION);
27     }
28
29     public CardDeck(final CardDeckConfiguration config) {
30         this.config = config;
31     }
32
33     // ...
34
35     public boolean isValid() {
36         final Map<Card, Integer> cardOccurrences = new ↵
            ↳ HashMap<>();
37
38         for (Card card : Card.values()) {
39             cardOccurrences.put(card, 0);
40         }
```

```

41
42         for (final Card card : this) {
43             Integer amount = cardOccurrences.get(card);
44             cardOccurrences.put(card, amount + 1);
45         }
46
47         return config.withoutZeroOccurrenceEntries().equals(
48             new ↵
49                 ↵ CardDeckConfiguration(cardOccurrences).withoutZeroOccu
50
51     public static final CardDeckConfiguration ↵
52         ↵ DEFAULT_CONFIGURATION = new CardDeckConfiguration(
53             Map.of(Card.WOOD, 16,
54                 Card.METAL, 16,
55                 Card.PLASTIC, 16,
56                 Card.SPIDER, 5,
57                 Card.SNAKE, 5,
58                 Card.TIGER, 5,
59                 Card.THUNDERSTORM, 1)
60     );

```

Code-Ausschnitt 7.4: Switch-Statement-

Code-Smell behoben durch neue *DeckConfiguration*-Klasse und Map zum Zeitpunkt des Commit a0e99692.

## 7.2 2 Refactorings

### Extract-Method

Abbildung 7.1 und Code-Ausschnitt 7.5 zeigen die *isValid*-Methode der *CardDeck*-Klasse<sup>1</sup> vor dem *Extract-Method*-Refactoring und Abbildung 7.2 und Code-Ausschnitt 7.6 nach dem Refactoring durch Commit cb55ad97.

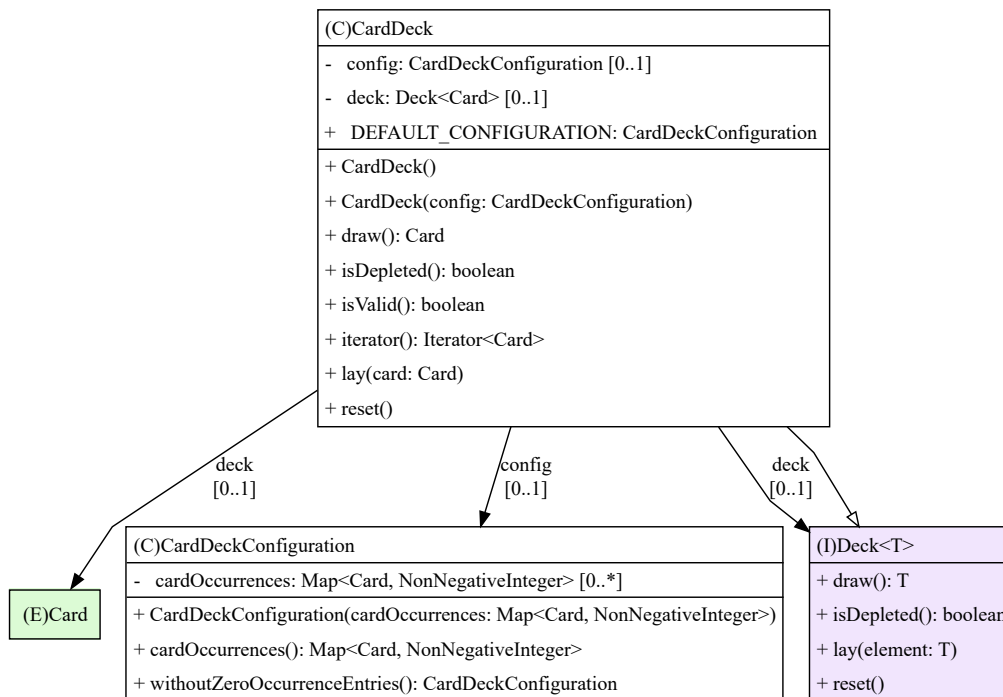
<sup>1</sup>Wie in Abschnitt 7.1.



Extract-Method wurde hier angewandt, um die Lesbarkeit der *isValid*-Methode deutlich zu erhöhen, da für den ersten Teil nicht direkt klar war, was passiert, wodurch fast ein Code-Comment-Smell benötigt worden wäre, ohne Method-Extraction. Nun ist in der *isValid*-Methode klar, was im ersten Zeil (zählen der eigenen Karten) und im zweiten Teil (Vergleich mit der Soll-Konfiguration) passiert. Außerdem ist die neue extrahierte Methode *countCardOccurrences* eine recht allgemeine Methode, die ggf. in Zukunft noch an anderer Stelle verwendet werden kann, um Code-Duplication zu verhindern.

```
1      public boolean isValid() {
2          final Map<Card, Integer> cardOccurrences = new ↵
              ↳ HashMap<>();
3
4          for (Card card : Card.values()) {
5              cardOccurrences.put(card, 0);
6          }
7
8          for (final Card card : this) {
9              Integer amount = cardOccurrences.get(card);
10             cardOccurrences.put(card, amount + 1);
11         }
12
13         return config.withoutZeroOccurrenceEntries().equals(
14             new ↵
                    ↳ CardDeckConfiguration(cardOccurrences).withoutZeroOccu
15     }
```

Code-Ausschnitt 7.5: *isValid*-Methode vor Method-Extraction (aus u.a. Commit a0e99692). Vgl. Code-Ausschnitt 7.4.

Abbildung 7.1: UML-Diagramm von *CardDeck* vor der Method-Extraction.

```

1  public boolean isValid() {
2      CardDeckConfiguration cardOccurrences = ↵
3          ↵ countCardOccurrences();
4
5      return config.withoutZeroOccurrenceEntries().equals(
6          cardOccurrences.withoutZeroOccurrenceEntries());
7  }
8
9  private CardDeckConfiguration countCardOccurrences() {
10     final Map<Card, NonNegativeInteger> cardOccurrences = ↵
11         ↵ new HashMap<>();
12
13     for (Card card : Card.values()) {
14         cardOccurrences.put(card, new ↵
15             ↵ NonNegativeInteger(0));
16     }
17
18     for (final Card card : this) {

```

```

16         NonNegativeInteger amount = ↗
           ↘ cardOccurrences.get(card);
17         cardOccurrences.put(card, amount.add(1));
18     }
19
20     return new CardDeckConfiguration(cardOccurrences);
21 }

```

Code-Ausschnitt 7.6: Nach Method-Extraction der Methode *countCardOccurrences* aus der *isValid*-Methode in Commit cb55ad97.

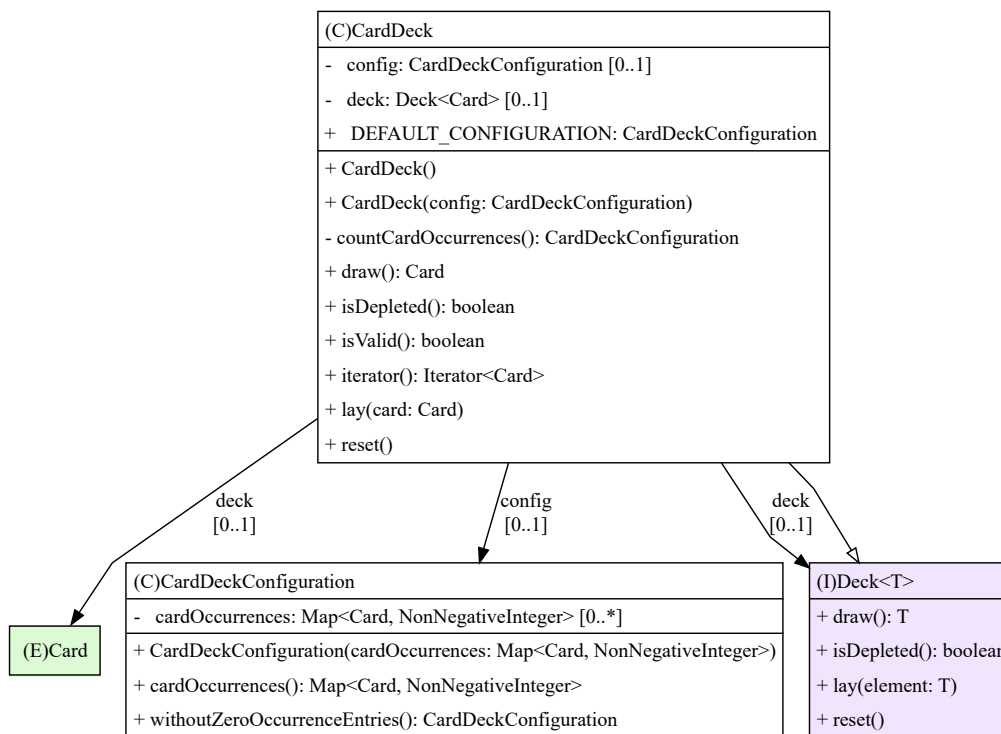


Abbildung 7.2: UML-Diagramm von *CardDeck* nach der Method-Extraction durch Commit cb55ad97. Beachte die neue private Methode *countCardOccurrences* in *CardDeck*.

## Replace-Temp-with-Query (RTQ)

Abbildung 7.3 und Code-Ausschnitt 7.7 zeigen die *execute*-Methode der *StartCommand*-Klasse vor dem *Replace-Temp-with-Query*-(RTQ)-Refactoring und Abbildung 7.4 und

Code-Ausschnitt 7.8 nach dem Refactoring durch Commit 827cd843.

Das Extrahieren der Berechnung des *CardDeck* von der *execute*-Methode in die *deckFrom*-Methode hilft dabei, dass die *execute*-Methode das SRP einhält und nur noch zuständig ist, den Output der *start*-Methode der *Game*-Klasse an den User auszugeben. Außerdem wird dadurch der *Long-Method-Code-Smell* der *execute*-Methode beseitigt.

```
1  public record StartCommand(List<Card> cards) implements ↵
    ↳ Command {
2
3      @Override
4      public void execute(final Game game) {
5          final CardDeck deck;
6
7          if (cards.isEmpty()) {
8              deck = new RandomCardDeckGenerator().generate();
9          }
10         else {
11             deck = new CardDeck();
12             for (int i = cards.size() - 1; i >= 0; i--) {
13                 deck.lay(cards.get(i));
14             }
15         }
16
17         try {
18             game.start(deck);
19         } catch (final IllegalArgumentException e) {
20             new ErrorBuilder("starting game due to " + ↵
                ↳ e.getMessage(),
21                 "check whether your command is ↵
                ↳ semantically correct").print();
22             return;
23         } catch (final GameStatusException e) {
24             new ErrorBuilder("game is running, can not use ↵
                ↳ start now", "perhaps you want to ↵
                ↳ reset").print();
25             return;
26         }
```

```

27         Terminal.println(CommonOutput.OK);
28     }
29 }

```

Code-Ausschnitt 7.7: *execute*-Methode vor RTQ-Refactoring.

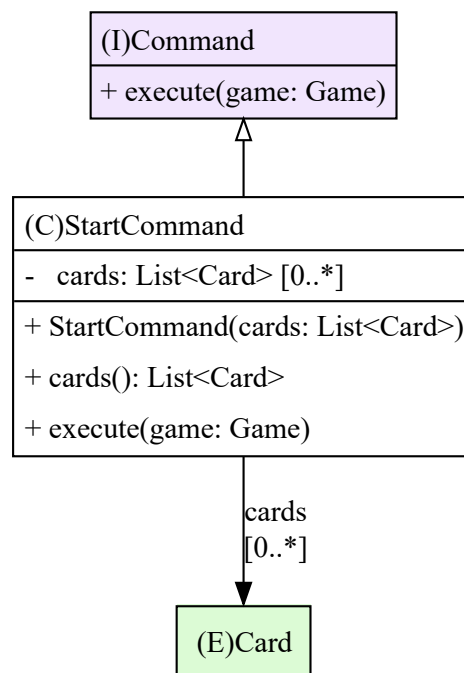


Abbildung 7.3: UML-Diagramm von *StartCommand* vor dem RTQ-Refactoring.

```

1  public record StartCommand(List<Card> cards) implements Command {
2      ↪ Command {
3
4      @Override
5      public void execute(final Game game) {
6          try {
7              game.start(deckFrom(cards));
8          } catch (final IllegalArgumentException e) {
9              new ErrorBuilder("starting game due to " +
10                 ↪ e.getMessage(),
11                 ↪ "check whether your command is
12                 ↪ semantically correct").print();
13          }
14          return;
15      }
16  }

```

```
11         } catch (final GameStateException e) {
12             new ErrorBuilder("game is running, can not use ↵
                ↵ start now", "perhaps you want to ↵
                ↵ reset").print();
13             return;
14         }
15         Terminal.println(CommonOutput.OK);
16     }
17
18     /**
19      * Returns a new CardDeck created from cards or a random ↵
20      * ↵ deck if cards is empty
21      */
22     private CardDeck deckFrom(List<Card> cards) {
23         final CardDeck deck;
24
25         if (cards.isEmpty()) {
26             deck = new RandomCardDeckGenerator().generate();
27         }
28         else {
29             deck = new CardDeck();
30             for (int i = cards.size() - 1; i >= 0; i--) {
31                 deck.lay(cards.get(i));
32             }
33
34             return deck;
35         }
36     }
```

Code-Ausschnitt 7.8: *execute*-Methode und *deckFrom*-Methode nach RTQ-Refactoring in Commit 827cd843.

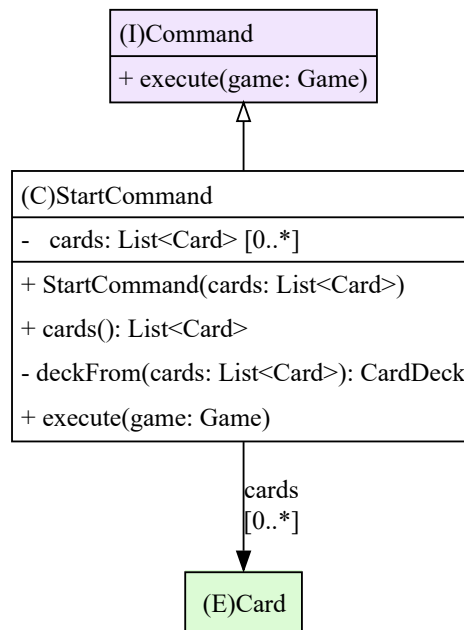


Abbildung 7.4: UML-Diagramm von *StartCommand* nach dem RTQ-Refactoring durch Extraktion des Erstellens eines neuen Decks in Commit 827cd843. Beachte die neue private Methode *countCardOccurrences* in *CardDeck*.

## 8 Entwurfsmuster

### 8.1 Entwurfsmuster: Observer-Pattern

Abbildung 8.1 zeigt das Observer-Pattern, wobei ein *GameEndObserver* von einem *GameEndObservable* benachrichtigt wird, wenn das Spielende erreicht ist und mit einem *GameResult* abgeschlossen wird. Das UML zeigt das konkrete *GameEndObservable* *GameState*, welches seine *GameEndObservers* updated, wenn das *GameResult* über einen Setter geändert wird. Der konkrete *GameEndObserver* *GameEndReporter* aus der Plugin-Schicht gibt dann das Spielergebnis an den User aus. Hierfür wird der *GameEndReporter* in der *main*-Methode erzeugt und über DI an die höhere Schicht weitergegeben.

Das Observer-Pattern ist hier sehr sinnvoll, da das Spielende ein Zustand ist, der über viele verschiedene Wege und User-Befehle erreicht werden kann, wie zum Beispiel das Ziehen der letzten Karte, ohne dass noch eine Aktion möglich ist, oder der Spieler mit dem letzten möglichen Endeavor nicht erfolgreich ist usw. Daher ist es sinnvoll dies über ein Event abzuwickeln, damit nicht viele verschiedene Commands separat auf Spielende checken müssen, was auch gegen das SRP gehen würde. Außerdem erlaubt es eine sehr flexible Erweiterbarkeit, wenn zum Beispiel ein Highscore-Mechanismus eingebaut werden soll, der am Spielende die benötigten Züge abspeichert.



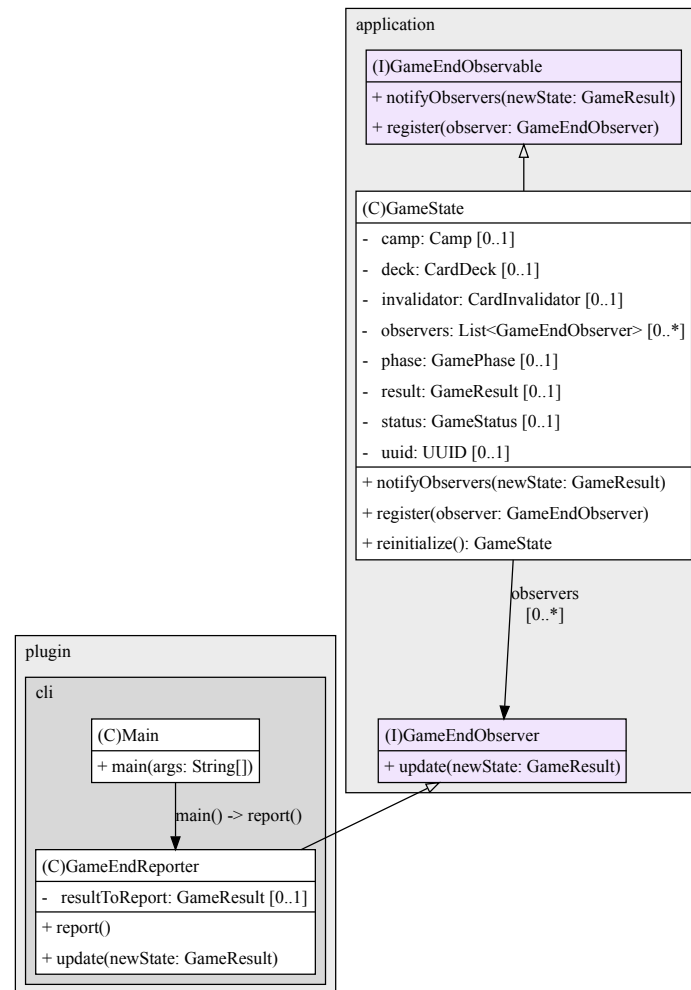


Abbildung 8.1: UML-Diagramm des Observer-Patterns zwischen *GameState* und *GameEndReporter*.

## 8.2 Entwurfsmuster: Strategie

Abbildung 8.2 zeigt das Strategie-Entwurfsmuster für die Logik der unterschiedlichen Endeavor-Möglichkeiten. Hierzu hält der Anwender *RollHandler* eine Referenz auf die Strategie *Rescue*, für die die Methode *endeavor* implementiert werden muss. Die konkreten Strategien *GuaranteedRescue* und *PossibleRescue* implementieren das Interface und können so flexibel von dem Anwender *RollHandler* verwendet werden.

Das Pattern ergibt hier sehr viel Sinn, da der *RollHandler* die konkrete Berechnung, ob

das Endeavor geglückt ist oder nicht, nicht selbst durchführen muss, bzw. davon nichts wissen muss. Das unterstützt sehr stark das OCP, SRP und geringe Kopplung. Sollte eine weitere Form der Rescue eingeführt werden und möglich werden, ist das Programm so sehr flexibel und einfach erweiterbar.

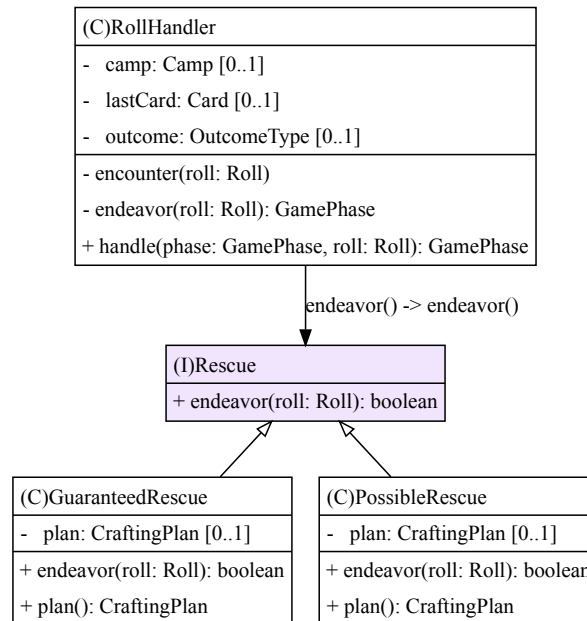


Abbildung 8.2: UML-Diagramm (gekürzt) des Strategie-Pattern der *Rescue-Endeavor*-Logik. Zur besseren Übersichtlichkeit wurden überflüssige Klassen ausgelassen.