Advanced SWE am Beispiel einer Spielhilfe für das Spiel "Die Werwölfe vom Düsterwald"

Projektarbeit

Duale Hochschule Baden-Württemberg Karlsruhe

von

Luna Zacharias Zetsche / Simon Meisetschläger

Abgabedatum 13.05.2022

Matrikelnummer	5000688 / 7781484
Kurs	TINF19B1
Gutachter der Studienakademie	Daniel Lindner

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

1.1 Nutzen des Codesubstrates

Entwickelt wurde eine Desktopanwendung, die das Spiel „Die Werwölfe von Düsterwald" simulieren kann. Die Software hilft dem Spielleiter bei der Planung und Durchführung einzelner Spielrunden. Weiterhin beinhaltet sie eine Funktion, Informationen zu einzelnen Spielrollen und Spielregeln anzuzeigen. Die Software kann über Konsolenbefehle oder eine graphische Oberfläche bedient werden, wobei der Fokus für die Projektabgabe auf der Konsolenanwendung liegt.

1.2 Entwicklung

Die Software wurde mit *Eclipse 2021-06* entwickelt. Als JDK wurde der *Java-17 Release der Amazon Coretto Distribution*[1] verwendet. Weiterhin wurden externe und interne Abhängigkeiten durch den Einsatz von *Maven* erleichtert.

1.3 Programminfos

Es wurde sich für ein Datenbankspeicher entschieden. Diese Designentscheidung wurde getroffen, da -unabhängig von dieser Prüfungsleistung- in Zukunft dieses Projekt weitergeführt werden soll. Für die aktuell umgesetzten Funktionen hätte

[1] https://docs.aws.amazon.com/corretto/latest/corretto-17-ug/downloads-list.html

eine einfachere Speicherlösung ausgereicht.

Nötige Informationen für den Verbindungsaufbau mit der erstellten Datenbank sind in der bereitgestellten readme-Datei vorhanden. Aus Sicherheitsgründen wurden diese Daten nicht auf GitHub hochgeladen.

Es ist zu beachten, dass die Umschaltung zwischen Konsolen- und GUI-Modus aktuell nur über das Auskommentieren der ungewollten Methode und dem Aktivieren der gewollten Funktion möglich ist . Näher Informationen hierzu finden sich ebenfalls in der readme-Datei.

Kapitel 2

Clean Architecture

Das Schichtenmodell der Clean Architecture wurde mithilfe von Maven gepflegt. Hierfür wurde ein übergeordnetes Maven-Projekt angelegt, das die verwendeten Schichten (Plugins, Adapters, Application und Domain) als jeweils eigenständige Module enthält (Siehe Abbildung 2.1). Weiterhin wurden die geschriebenen Tests im späteren Verlauf der Entwicklung in ein eigenes Modul ausgelagert. Anfänglich wurden sie jeweils im getesteten Modul gespeichert.

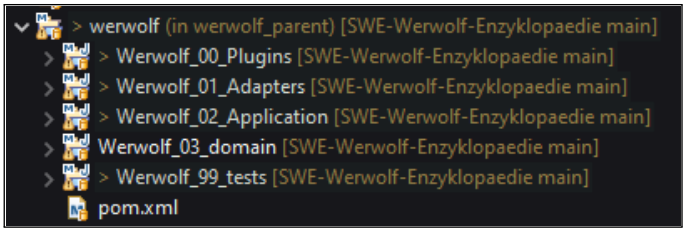

Abbildung 2.1: Übersicht der erstellten Maven Projekte

In den einzelnen Schichten wurde über die pom.xml die Dependencys so konfiguriert, dass äußere Schichten die jeweils inneren in ihrem Build-Path beinhalten. Innere Schichten haben jedoch keinen Zugriff auf äußere Schichten. So wird sichergestellt, dass bei Verletzung des Schichtenmodells bereits der Compiler eine Fehlermeldung zurückgibt.

2.1 Schichteninhalt

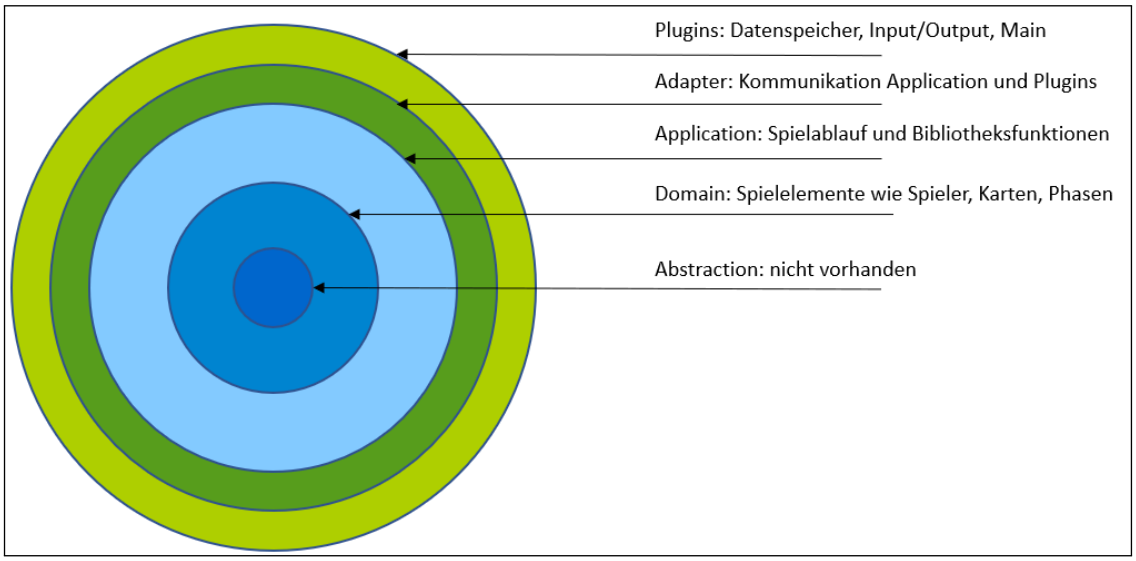

Abbildung 2.2: Übersicht der geplanten und umgesetzten Schichten

Abstraction

Die Abstractionschicht wird nicht verwendet, da sie nicht benötigt wird. In der entwickelten Applikation werden keine unveränderbaren Value Objects und keine grundlegenden mathematischen beziehungsweise naturwissenschaftlichen Prinzipien verwendet, die in dieser Schicht sonst Einzug finden würden.

Domain

In der Domain finden sich grundlegende Elemente des Spieles. Dort sind Karten, Rollen und Spieler gespeichert. Weiterhin enthalten sind zwei Interfaces, eines für die Erzeugung und Verwaltung von Rollenobjekten (*rollenRepository*), sowie ein weiteres mit ähnlichen Funktionen für Kartenobjekte (*kartenRepository*).

Application

Die Applicationschicht beinhaltet Klassen, welche die Logik des Spielverlaufs abbilden. Weiterhin wurde eine Klasse zur Verwaltung der in der Domain beschriebenen Repositorys-Interfaces erstellt. Über den *LibraryManager* können Karten und Rollen aus verschiedenen Implementationen der Repository-Interfaces geladen und darauf zugegriffen werden.

Adapter

Die Adapterschicht beinhaltet verschiedene Wege, mit der Applicationschicht zu kommunizieren. Dort existiert eine Implementation der angesprochenen Repositorys zum Laden der Spielelemente aus einer Datenbank (*SQLKartenRepository und SQLRollenRepository*). Es existieren ebenfalls Schnittstellen zum Ausgeben von Output-Strings sowie Funktionen zur Steuerung des Spielablaufs und Möglichkeiten auf die Bibliotheksfunktionen zuzugreifen.

Plugins

Die Main Klasse der Anwendung findet sich in der Pluginschicht. In dieser müssen verschiedene andere Plugins initialisiert werden in der die verwendeten Elemente initialisiert werden. Hier befindet sich ein Plugin für die Herstellung einer Datenbankverbindung und die Ausführung von einfachen mySQL-Befehlen. Es exisiteren verschiedene Plugins zur Steuerung des Programmes, eine Konsolenanwendung sowie ein graphisches Userinterface.

2.2 Dependency Inversion

Für die korrekte Umsetzung des Schichtenmodells wurde an einigen Stellen eine Dependency-Inversion vorgenommen. Genauere Informationen dafür finden sich im Kapitel 3.1.

Kapitel 3

Programming Principles

3.1 SOLID

Single Responsibility Principle

Das Single Responsibility Prinzip kann an einigen Klassen beispielhaft aufgezeigt werden. So gibt es Beispielsweise für Kartenobjekte und für Rollenobjekte eine eigene Repositoryklasse. Innerhalb der GUI wird das Prinzip nicht vollständig erfüllt, hier könnte eine bessere Trennung zwischen Logik und Darstellungscode stattfinden.

Die nachfolgende Tabelle soll die Umsetzung dieses Prinzipes übersichtlich darstellen. Die GUI Klassen wurden aus dem oben erwähnten Grund nicht mit einbezogen.

Schicht	**Klasse**	**Responsibility**
Domain	Karte	Bildet eine Spielkarte ab
Domain	Rolle	Bildet eine Spielrolle ab
Domain	Spieler	Bildet ein Spieler ab
Domain	Rollen-Repository	Interface zur Verwaltung (Erzeugung & Auslesen) von Rollen
Domain	Karten-Repository	Interface zur Verwaltung (Erzeugung & Auslesen) von Karten
Domain	GameExeption	Eigene Exception, die aus dem Gameloop geworfen wird
Application	GameLoop	Verwaltet Spielphasen, zugehörige Spieler und Spielelemente
Application	Spielphase	Beinhaltet geteilte Logik für Tag- und Nacht-Phasem
Application	Nacht	Beinhaltet Logik der Nachtphasen
Application	Nacht	Beinhaltet Logik der Nachtphasen
Application	Library-Manager	Verknüpft Karten mit Rollen. Bietet Zugriff auf Karten und Rollen.
Adapters	GameController	Ermöglicht die Steuerung des GameLoops durch Plugins
Adapters	OutputAdapter	Verwaltet Input/Output des Programmes
Adapters	SQLKarten-Repository	SQL-spezifische Implementierung des Karten-Repository-Interfaces
Adapters	SQLRollen-Repository	SQL-spezifische Implementierung des Rollen-Repository-Interfaces
Adapters	SQL-Verbindung	Interface, das Methoden zur Durchführung von SQL-Befehlen bereitstellt
Plugins	Main	Legt die verwendeten Plugins fest und startet das Programm
Plugins	KonsolenMain	Startet die Kommandozeilensteuerung
Plugins	Kommandos	Enum, enthält alle möglichen Kommandos und führt diese aus
Plugins	MySQL-Authentifizierung	Stellt eine Verbindung zu einer MySQL-Datenbank her
Plugins	MySQL-Verbindung	Implementierung des Interfaces SQLVerbindung für mySQL-Datenbanken

OPEN/CLOSED Principle

Ein Beispiel für das Open/Closed-Principle findet sich in der Klasse *GameController*. Dieser erlaubt Zugriff auf die Funktionen des GameLoops und beinhaltet die Logik für das Ausführen von Spielfunktionen und erzeugt Rückgabewerte. Da ein User aber nur mit dem *OutputAdapter* kommuniziert, wurde dieser um diese Funktionen erweitert. Über diesen werden die GameController-Funktionen aufgerufen und der Rückgabewert weitergereicht, es wird aber keine weitere Logik angewendet. Durch diese Schichten können Veränderungen innerhalb der Spielelogik stattfinden, ohne Änderungen an den Adaptern durchzuführen.

Liskov Substitution Principle

Das Liskov-Substitution-Priciple wird insofern erfüllt, dass in dem erstellten Programmcode zum aktuellen Zeitpunkt keine Vererbung vorhanden ist. Im Rahmen des Refactoring (siehe 6.2) wurde eine Vererbung in die Application Schicht eingebaut. Da es sich jedoch um eine abstrakte Oberklasse handelt die nicht instanziiert werden kann, findet das LSP hier ebenfalls keine Anwendung.

Interface Segregation Principle

Die Interfaces *Rollenrepository* und *Kartenrepository* definieren sehr ähnliche Funktionen, nur mit jeweils einer anderen verwalteten Entität. Hier wurde sich dazu entschieden zwei Interfaces zu verwenden, wenn ein Allgemeines auch möglich wäre. Dies würde allerdings auch gegen weitere Prinzipien verstoßen, vor allem gegen das Single Responsibility Principle.

Dependency Inversion Principle

Für manche Funktionen wurde eine Dependency Inversion umgesetzt. Zum einen wurden die Klassen zum Laden und Verwalten von Karten und Rollen (*RollenRepository und KartenRepository*) nur als Interfaces in der Domain-Schicht erstellt. Diese werden dann erst in der Adapter-Schicht konkret implementiert. Dies ermöglicht es, für verschiedene externe Datenspeicher das gleiche Interface zu verwenden. Die definierten Funktionen werden innerhalb der Application-Schicht benötigt. Um dem

Schichtenmodell zu entsprechen muss das generische Interface verwendet werden, da keine Compile-Time-Dependency auf eine äußere Schicht möglich ist. Somit entsteht eine Dependency Inversion über drei Schichten. Es wäre auch möglich, die *Rollen- und Kartenrepositorys* in der Apllication-Schicht umzusetzen. Dies würde das Schichtenmodell ebenfalls nicht verletzen.

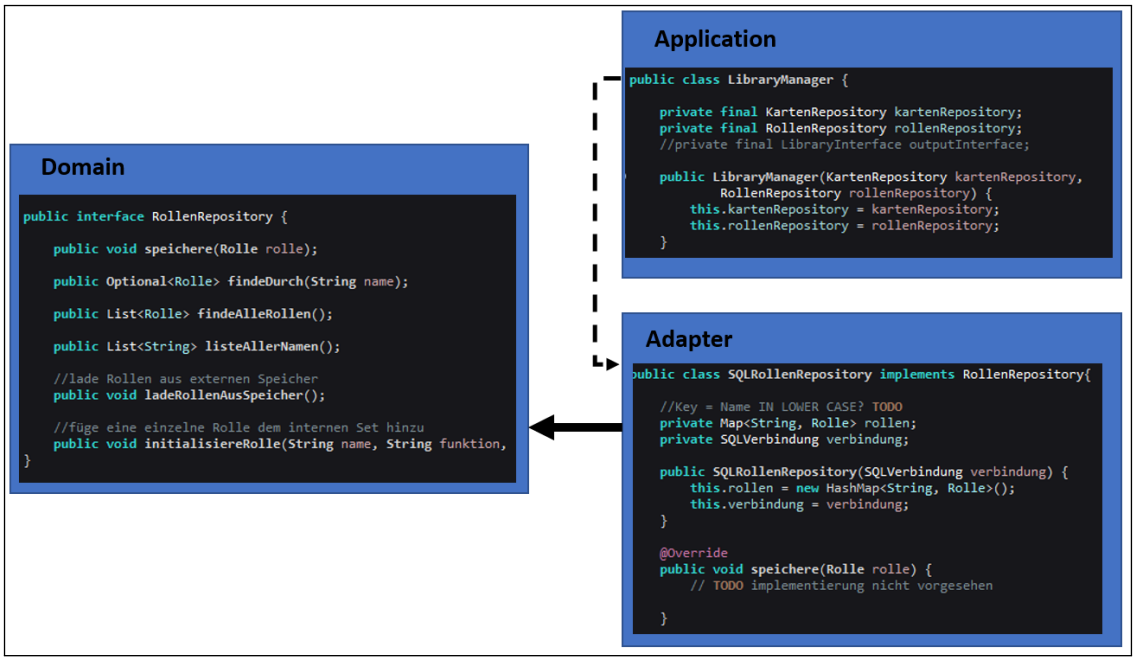

Abbildung 3.1: Dependency Inversion bei Implementierung eines RollenRepository

3.2 GRASP

Kopplung

Innerhalb der Domänen und Application Schicht gibt es eine relativ starke Kopplung der einzelnen Klassen. So exisiteren in der Klasse *Spieler* getter, die Attribute der Klasse *Rolle* zurückgeben. Da es gewollt ist, dass Objekte der Klasse *Rolle* nur mit einem Objekt der Klasse *Karte* oder einem Objekt der Klasse *Spieler* verwendet werden können, ist diese Kopplung als hinnehmbar zu betrachten. Die KartenRepository weisen ebenfalls eine höhere Kopplung auf, was mit der Verknüpfung zischen

Karten und Rollen zusammenhängt. In den restlichen Schichten des Projektes ist eine eher lockere Kopplung vorhanden, was unter anderen mit dem guten Einhalten des Single Responsibility Principles zusammenhängt.

Kohesion

Durch das Single Responsibility Principle ist neben der lockeren Kopplung auch eine relativ hohe Kohesion in dem Projekt gegeben. Besonders in der Plugin Schicht ist jedoch eine noch bessere Kohesion zu erreichen. Hier kann wie bereits erwähnt eine bessere Trennung zwischen Darstellungs und Logikfunktionen stattfinden. Die verwendeten Klassen zur Datenbankverbindung könnten auch verbessert werden, sodass eine bessere Kohesion entsteht.

3.3 DRY

Das „Don't repeat yourself" Prinzip wurde unter anderem im *OutputAdapter* verletzt, in dem mehrere Methoden sehr ähnlichen Code ausführten. Diese Verletzung wurde im Refactoring unter Punkt 6.4 beschrieben und behoben.

Kapitel 4

Domain Driven Design

4.1 Ubiquitous Language

Umsetzung der Business Objekte

Als Business Objekte werden jene Objekte verstanden, welche in der Business-Welt der Auftraggeber*innen benötigt werden und im Code umgesetzt werden sollen. In diesem Fall sind dies die Objekte des Spiels "Werwölfe vom Düsterwald". Für ein Werwolf Spiel werden *Spielkarten* an eine Gruppe von *Spielern* ausgegeben. Die Spielrunden bestehen aus einem Zyklus von *Tag* und *Nacht* und der Spielablauf wird durch festgelegte *Regeln* beschrieben.

Auch wenn es sich bei den Auftraggebern um uns selbst - und damit um die Entwickler - handelt ist es wichtig sich bei der Sprache an eben diesen Business Objekten zu orientieren. Dies macht ein späteres Code Verständnis und damit ggf. notwendiges Debuggen und Weiterentwickeln der Anwendung einfacher. Die Java Klassen werden dementsprechend (mit ein paar Ausnahmen in Begrifflichkeiten wie Controller etc.) auf deutsch nach dem jeweiligen Objekt benannt. Es ergeben sich folgende Klassen:

- Regeln
- Spieler
- Tag

- Nacht
- Rollen
- Karten

Es fällt auf, dass die Spielkarten in zwei unterschiedliche Klassen aufgeteilt wurden - *Karten* und *Rollen*. Bei dem klassischen Kartenspiel ist dies natürlich nicht der Fall, für die technische Umsetzung wurde sich jedoch auf Grund der zwei angestrebten Use Cases des Simulieren eines Spielablaufes und der Hilfe in Form einer Kartenenzyklopädie entschieden. Für den Spielablauf werden in der digitalen Anwendung keine Karten benötigt, es werden Spieler*innen lediglich Rollen (ohne zugehöriges Kartenbild) zugewiesen. In der Enzyklopädie soll jedoch auch nach dem Bild einer Karte gesucht werden. Bilder müssen daher als Blob in der Datenbank hinterlegt sein und innerhalb der GUI sichtbar werden. Zu Vereinfachung des Codes und der besseren Lesbarkeit wurde daher die Spielkarte in Karte und Rolle aufgeteilt.

In der bisherigen Umsetzung des Projektes ist die GUI nur für den Spielablauf fertiggestellt. Die Enzyklopädie läuft über die Kommandozeile und stellt daher noch keine Bilder dar. Dies ist eine Funktion welche später hinzugefügt wird.

Die restlichen Klassen sind dem Konzept des realen Spiels jedoch sehr ähnlich gehalten. Alle Eigenschaften die auch im Kartenspiel dem Objekt zugeordnet werden sind auch in der Java Klasse vertreten.

Methoden

Neben der Benennung der Klassen in eine gemeinsame Projektsprache sollten auch Methoden nach diesem Prinzip benannt werden. Bei der Erstellung wurde daher darauf geachtet die Namen möglichst auf deutsch und sprechend zu wählen, so das beim Lesen des Codes direkt klar wird welchen Zweck die Methode erfüllt. Im besten Fall soll dies auch jenen Leser*innen klar sein, welche wenig Ahnung von der technischen, jedoch viel Ahnung von der spielerischen Seite dieses Projektes haben.

Dies ist in den meisten Fällen sehr gut gelungen. Einzelne Methoden haben zwar noch die standardmäßigen Bezeichnungen wie *getKarten()* anstelle von *bekommekKarten*, dies wurde jedoch als nicht sonderlich unverständlich bewertet und

daher beibehalten. Die restlichen Methoden besitzen sehr sprechende Namen wie beispielsweise *findeAlleKarten()*, welche genutzt werden kann um alle Karten zu finden und in einer Liste zu speichern.

Doch nicht nur die Namen, auch der Inhalt der Methoden sollte so weit wie möglich verständlich und in der gemeinsamen Projektsprache geschrieben werden.

Durch sprechende Methodennamen ist dies zum größten Teil bereits umgesetzt. Rufen Methoden andere Methoden auf, so ist recht klar was im Code vor sich geht. Dies ist jedoch nicht überall der Fall. Die Bereiche in denen die gemeinsame Projektsprache nicht genutzt und auf technische Sprache des Programmierens zurückgegriffen werden muss können durch Inline Kommentare verständlicher gestaltet werden. Ein Beispiel befindet sich in Abbildung 4.1.

```
/**
 * Gibt eine bestimmte Art von Karten zurueck
 * @param filter: moegliche Werte: "gut", "boese", "spezial"
 */
@Override
public Map<String, Karte> getKartenMitFilter(String filter) {

    Map<String, Karte> filteredKarten = new HashMap<>();

    for (String name: karten.keySet()) {
        if(filter == "gut") {
            if(!karten.get(name).getRolle().istBoese())
                filteredKarten.put(name, karten.get(name));
        }
        if(filter == "boese") {
            if(karten.get(name).getRolle().istBoese())
                filteredKarten.put(name, karten.get(name));
        }
        if(filter == "spezial") {
            if(karten.get(name).getRolle().istSpezial())
                filteredKarten.put(name, karten.get(name));
        }
    }
    return filteredKarten;
}
```

Abbildung 4.1: Beispiel einer kommentierten Methode

Die Kommentare in dem für das Projekt genutzten Code sind noch nicht bei jeder Methode vorhanden. Die Lesbarkeit ist durch die gewählten Methodennamen dennoch gewährleistet.

Umsetzung innerhalb des CLI

Wie bereits mehrfach erwähnt ist die Anwendung in eine Command-Line- und eine GUI-Anwendung unterteilt. Hauptfokus war zunächst die CLI, da hier die wichtigsten Funktionen des Spielablaufs und der Enzyklopädie ohne den großen

Aufwand einer grafischen Oberfläche umgesetzt werden können.

```
starte-spiel anna,simon,reka,daniel werwolf,werwolf,hexe
Neues Spiel gestartet.
Ein neuer Tag hat begonnen
spielschritt
Phase wird fortgefuehrt
Ein neuer Buergermeister muss gewaehlt werden!
waehle anna
Spieler*in anna ist neue*r Buergermeister*in
```

Abbildung 4.2: Beispiel eines CLI Spielablaufs

Die CLI Kommandos wurden ebenfalls in der gemeinsamen Projketsprache verfasst und gleichen sich damit bis auf ein paar Ausnahmen einer normalen Kartenspielrunde. Während einer nicht digitalen Runde Werwolf werden natürlich keine Kommentare wie *Spielschritt* genutzt. Dieses Kommando war jedoch während der Entwicklung notwendig um den Tag / Nacht Zyklus im Programm umsetzen zu können.

Zum Eliminieren eines Spielers wird in der Konsolenanwendung noch der Befehl *kill <Spielername>* statt *eliminiere <Spielername>* genutzt. Eine Überlegung wäre diese begriffliche Unstimmigkeit noch zu beeitigen indem das Kommando abgeändert wird. Sobald die GUI fertig aufgebaut ist, wird hier darauf geachtet bei dem Begriff des Eliminierens zu bleiben.

Alle weiteren Kommandos sind für sich sprechend und sehr nah an der Spielrealität. Ein Beispiel kann in Abbildung 4.2 gesehen werden. Das Spiel teilt dem Nutzenden mit, dass ein Bürgermeister gewählt werden soll, welches durch das Kommando *waehle <Spielername>* erfolgen kann.

Umsetzung innerhalb der GUI

Eine besonders wichtige Einsatzstelle für die gemeinsame Projektsprache ist innerhalb der GUI, da dies die Oberfläche ist welche von den Auftrageber*innen und späteren Anwender*innen am ehesten betrachtet wird. Unstimmigkeiten bei den Begrifflichkeiten der GUI und des Auftrags sind daher zu vermeiden.

Dennoch weißt der aktuelle Entwurf der GUI eine solche Unstimmigkeit auf. Der Button, welcher zur Enzyklopädie führen wird ist noch als *Help* beschriftet. Begründet ist dies dadurch, dass alle Grafiken eigenständig erstellt wurden und auf dem ersten Buttonentwurf, welcher in der aktuellen GUI zu finden ist, kein Platz für das Wort *Enzyklopädie* ist.

Diese Unstimmigkeit zwischen Business Welt und Code muss im weiteren Entwicklungsverlauf noch beseitigt werden, wurde jedoch zunächst nicht weiter beachtet, da die Enzyklopädie noch nicht in der GUI implementiert ist.

In einem grafisch noch nicht aufgehübschten Teil der GUI beim Spielablauf sind die Buttons zum Eliminieren der Spielenden noch mit *Töte Name* beschriftet. Da diese Ansicht noch nicht fertig ist (die Button sollen am Ende natürlich auch nicht grün sein) wird dies ebenfalls aus der Bewertung ausgeschlossen.

Die restlichen Begrifflichkeiten sind einfach und unmissverständlich für die Anwender*innen gewählt.

4.2 Entities

Verwendete Entities in diesem Projekt sind *Spieler*, *Rollen* und *Karten*. Spieler besitzen anders eine eigene Identität, haben anders als die Value Objects veränderbare Eigenschaften und sind über den Lauf eines Spieles an einen Lebenszyklus gebunden. Bei Rollen war lange nicht klar ob sie als Entity oder als Value Object behandelt werden sollen. Da sich aber durch Spielabläufe die „Gesinnung“ einer Rolle ändern kann, wurden sie als Entity eingestuft. Gleiches galt auch bei *Karten*. Da Karten aber auch eine eindeutige Identität besitzen (ausgedrückt durch ihre Rolle, Bild und Beschreibung) wurden auch sie als Entity bewertet. Auch die *Tag*, und *Nacht* und deren Oberklasse *Spielphase* sind Entities im Sinne der Domäne. Sie befinden sich allerdings in der Application-Schicht, da sie volatiler als die restlichen Domänenobjekte sind und wesentliche Logik beinhalten.

4.3 Value Objects

In diesem Projekt gibt es keine Value Objects. Mit einer veränderten Implementation der Domänenobjekte könnten wie vorab beschrieben die *Karten* und *Rollen* Objekte als Value Objects gehandhabt werden.

4.4 Aggregate

Durch die Abhängigkeit von *Rollen* und *Spielern* sowie *Rollen* und *Karten* je nach Usecase ist eine Aggregatsbildung über diese Klassen nicht sinnvoll. Diese bilden demnach jeweils ein eigenes Aggregat.

Ein weiteres Aggregat finden wir in den Entities für die Spielelogik. Hier entsteht ein Aggregat aus *GameLoop* und *Spielphase* (und damit auch *Nacht* und *Tag*). Die Root Entity, die hierbei den Zugriff auf die restlichen Elemente erlaubt ist die Klasse *GameLoop*. Auch wenn diese keine Entity im Sinne der Domäne ist, ist sie trotzdem durch ihre Schnittstellenfunktio ein Teil des Aggregrats.

4.5 Repositories

Als Repositories werden schon seit Beginn die beiden so benannten Interfaces *KartenRepository* und *Rollenrepository* verwendet. Diese beiden Interfaces und ihre Implementierung wurden bereits mehrfach erläutert. Da keine weiteren Entities persistent gespeichert werden, sind für die restlichen Aggregate beziehungsweise die restlichen Entities keine Repositories vorhanden.

Kapitel 5

Unit Tests

Bevor mit Unit Tests begonnen wurde, wurde zunächst die Verbindung zur Datenbank erstellt. Zum einen war so ein Grundsubstrat vorhanden auf welches aufgebaut werden konnte und zum anderen wurde mit diesem Teil des Programms bereits weit vor der entsprechenden Vorlesungseinheit begonnen.

Bei der weiteren Entwicklung wurde versucht sich an die Prinzipien des Test Driven Designs (TDD) zu halten.[1] Dabei wurde wann immer möglich wie folgt vor gegangen:

1. Überlegen der benötigten Funktion
2. Überlegung über das gewünschte Ergebnis der Funktion
3. Schreiben des Unit Tests
4. Schreiben der benötigten Methode
5. Testen

Besonders durch die genaue Überlegung des erwünschten Ergebnisses erwies sich diese Vorgehensweise als sehr produktiv. Durch die begrenzte Zeit des Projektes

[1] unter anderem:
https://github.com/diemeise/SWE-Werwolf-Enzyklopaedie/commit/9a6b1f71f598701f74d10a002d95e04efebd9436
https://github.com/diemeise/SWE-Werwolf-Enzyklopaedie/commit/2d1c0d45f96e164ddfd7bf9ea32e4b5665f5afd1
https://github.com/diemeise/SWE-Werwolf-Enzyklopaedie/commit/d5d715ac996a7ea5f172a6496e8183a8d7bf39a6

wurde sich dennoch dazu entschieden nicht jede mögliche Methode durch einen Unit Test abzusichern.

Insgesamt besitzt dieses Projekt eine eher schlechte Testabdeckung. Innerhalb der Pluginschicht ist die schlechteste Coverage. Dies liegt darab, dass für GUI-Methoden Unit-Tests nicht sinnvoll sind. Des weiteren wurden nicht für alle CLI-Kommandos Tests erstellt, da die zugrundeliegenden Methoden oft schon getestet wurden.
Innerhalb der Adapterschicht sind viele Methoden ebenfalls nur "Durchreichen"von Informationen aus den tieferen Schichten, hier ist eine Coverage von fast 50% ausreichend.
Innerhalb der Domain- und der Application-Schicht ist die größte Coverage gegeben. Da hier die wichtigesten Elemente des Projektes liegen, ist dies sinnvoll. Leider wurden nicht für alle Teile ein Unittest erstellt, da dies zuviel Zeit in Anspruch genommen hätte.

Element	Coverage	ed Instructions	ed Instructions	tal Instructions
Werwolf_00_Plugins	26,9 %	602	1.635	2.237
Werwolf_01_Adapters	46,1 %	433	506	939
Werwolf_02_Application	63,2 %	431	251	682
Werwolf_99_tests	88,8 %	1.494	188	1.682
Werwolf_03_domain	60,2 %	112	74	186

Abbildung 5.1: Code-Coverage aufgeteilt nach Modul/Schicht

5.1 Ausgewählte Unit Tests

Die nachfolgenden Unterkapitel gehen jeweils kurz auf ausgewählte Testklassen und entsprechende Tests des Projektes ein. Der erste Test wird dabei ausführlich beschrieben, bei den späteren Tests wurde hierauf verzichtet und lediglich auf den Verwendungszweck eingegangen, da sie in ihrem Aufbau alle sehr ähnlich sind.

RollenGesinnungTest.java

Eine der ersten Neuerungen, welche umgesetzt werden sollte war die Umstrukturierung der Datenbank. In der alten Struktur wurde die Gesinnung und Spezialität

einer Rolle noch nicht abgebildet. Es musste also manuell im Code eingegeben werden ob eine Rolle gut oder böse ist und ob es sich bei ihr um eine Spezialrolle handelt (Hexe, Seher...).

Um nach den Prinzipien des TDD zu arbeiten, wurden zunächst vier Unit Tests geschrieben, welche vier Mögliche Fälle abdecken.

1. Rolle ist gut
2. Rolle ist böse
3. Rolle ist spezial
4. Rolle ist nicht spezial

Da die eigentliche Verbindung zur Datenbank innerhalb eines Unit Tests keine Rolle spielt, wird auf diese mit Hilfe eines Mocks simuliert. So wird die eigentlich vorhandene Abhängigkeit ersetzt und der Test kann die Einheit komplett abgekabselt vom restlichen Code überprüfen.

Bei dem Einsatz des Mocks wird auf die externe Bibliothek *EasyMock* zurückgegriffen. Damit er als Ersatz für die Datenbank genutzt werden kann muss er zunächst die drei Phasen **Einlernen, Abspielen, Überprüfen** durchlaufen. Die ersten beiden Phasen sind in Abbidlung 5.2 unter dem Begriff *Capture* zusammengefasst.

In einem ersten Schritt wird der Mock erstellt (*EasyMock.createMock()*). Anschließend werden die für den Test benötigten Bereiche der Datenbank innerhalb des Mocks simuliert (*EasyMock.expect()*) und der Mock abgespielt (*EasyMock.replay()*).

Nach dieser Vorbereitung kann mit dem eigentlichen Unit Test begonnen werden. Hierbei wird die AAA Normalform (**Arrange, Act, Assert**) für Unit Tests genutzt. Zum Testen der Funktion wird im Assert jeweils folgende zu erwartende Ergebnisse angegeben:

1. Dorfbewohner sind böse - *false*
2. Werwölfe sind böse - *true*

3. Werwölfe sind eine Spezialrolle - *false*

4. Der weiße Werwolf ist eine Spezialrolle - *true*

```java
@Test
public void fuerGuteRollen() throws SQLException {
    //Capture
    ResultSet rs = EasyMock.createMock(ResultSet.class);
    EasyMock.expect(rs.next()).andReturn(true);
    EasyMock.expect(rs.getString("Name")).andReturn("Dorfbewohner");
    EasyMock.expect(rs.getString("Funktion")).andReturn("Lebt");
    EasyMock.expect(rs.getBoolean("istBoese")).andReturn(false);
    EasyMock.expect(rs.getBoolean("istSpezial")).andReturn(false);
    EasyMock.expect(rs.getInt("prioritaet")).andReturn(0);

    EasyMock.expect(rs.next()).andReturn(false);
    EasyMock.replay(rs);

    //Arrange
    SQLRollenRepository repo = new SQLRollenRepository(null);

    //Act
    repo.initialisiereRollen(rs);

    //Assert
    Assertions.assertFalse(repo.findeDurch("Dorfbewohner").get().istBoese());

    //Verify
    EasyMock.verify(rs);
}
```

Abbildung 5.2: Beispiel eines Datenbank Mocks

Da die Rückgaben der entsprechenden Methoden *istboese()* bzw. *istSpezial()* standardmäßig den Wert *false* besitzen und manuell geändert werden mussten ist zu erwarten, dass zwei der vier Tests nicht erfolgreich sind (Failure, siehe Abbildung 5.3).
Nach Umstrukturierung der Datenbank werden die Tests erneut durchlaufen und liefern das erwartete positive Ergebnis. So kann schnell überprüft werden, ob die Umstrukturierung erfolgreich war.

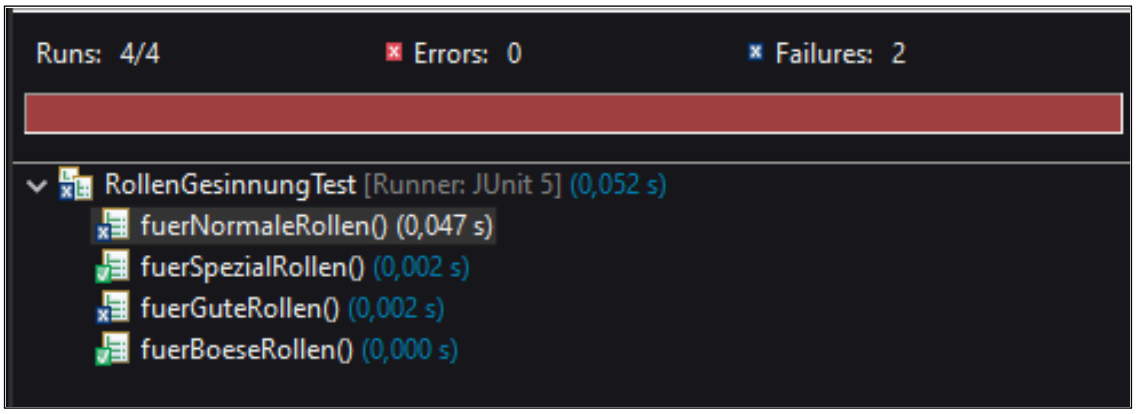

Abbildung 5.3: Testergebnisse vor der Änderung

KartenUndRollenTest.java

In dieser Klasse befinden sich die Tests, welche die Methoden zur Initialisierung der Karten und die Verknüpfung mit der entsprechenden Rolle überprüfen. Umgesetzt wird dies indem eine Testliste erstellt wird, welche das erwartete Ausgabeergebnis der jeweiligen Methode beinhaltet. Anschließend wird über eine Assertion geprüft ob die Testliste mit dem tatsächlichen Ergebnis übereinstimmt.

Das restliche Vorgehen ist analog zu den bereits beschriebenen Tests, daher wird hier nicht näher darauf eingegangen.

Auch hier wurden die oben genannten Schritte für das TDD eingehalten. Es fiel allerdings auf, dass die Tests auch nach Implementierung der Methoden kein positives Ergebnis anzeigten. Grund hierfür war, dass die als Ergebnis entstehende Liste ihre Ergebnisse nicht in einer festen Reihenfolge speichert und die beiden im Assert geprüften Listen daher nicht identisch waren. Gelöst wurde das Problem durch ein nachträgliches Einfügen einer Sortierung in der implementierten Methode.

Die Methode funktioniert zwar auch mit unsortierter Liste ohne Probleme, allerdings wird sie später auch für die Enzyklopädie genutzt. Hier sollen alle Elemente der Liste ausgegeben werden können und so eine Kartenliste zu sehen sein, welche Details über die Karten für Anwender*innen bereithält. Wäre die Liste unsortiert wäre die User Experience verringert. Der Test hat diesen Fehler bereits frühzeitig

aufgezeigt, so dass es in der späteren Entwicklung zu weniger Problemen kam.

KonsolenEingabenTest.java

Diese Klasse beinhaltet Tests, welche die Konsoleneingaben für die Enzyklopädie überprüfen. Diese Eingaben sind eine Möglichkeit zur Interaktion mit dem Programm und wurden als Grundlage gebaut falls eine GUI zu viel Zeit in Anspruch nehmen sollte (Die GUI ist Stand Ende des Projektes nicht komplett umgesetzt). Das Programm funktioniert also auch nur über die Konsole.

Da manuelle Eingaben bei Tests nicht erwünscht sind (vgl. Kapitel 5.2) müssen diese simuliert werden. Die Eingabe wird also durch eine String Variable in dem jeweiligen Test festgelegt. Die eigentliche Ein- und Ausgabe erfolgt nicht über die Konsole sondern wird durch *ByteArrayOutputStreams* und *PrintStreams* umgeleitet und ausgelesen. Diese müssen vor jedem Test initialisiert und nach jedem Test wieder zurückgesetzt werden, da es sonst zu Abhängigkeiten zwischen den einzelnen Tests kommen kann (vgl. Kapitel 5.2).

Im Arrange des jeweiligen Tests muss also neben der Initialisierung von Rollen und Karten und deren Verknüpfung also auch eine String Variable mit entsprechender Konsoleneingabe vorbereitet werden.

Es ist zu beachten, dass in dieser Klasse nur die Eingaben für die Enzyklopädie getestet werden und nicht die für die Spielmechanik. Das liegt daran, dass die Methoden welche für letzteres benötigt werden bereits durch separate Tests abgedeckt werden.

GameLoopTest.java

Der Spielablauf wurde anfangs auch mithilfe von Test Driven Development entwickelt. Es hat sich jedoch mit der Zeit herausgestellt, dass in der Planung des GameLoops ein paar Edgecases nicht asureichend bedacht waren. Daher konnte nicht für jede Funktion / jeden Edgecase ein eigener Test vorab entwickelt werden. Innerhalb der Spiellogik werden folgende Elemente getestet:

Getestet werden verschiedene Konfigurationen an Spieler*innen und verwendeten Rollen bei Spielstart. Bei keinen übergebenen Spielern oder im Fall dass keine

"bösen"Rollen vorhanden sind soll das Spiel nicht starten. Um unabhängige Tests zu gewährleisten wird hierfür vor jedem Test eine gültige Liste an Spielern erzeugt. Diese wird in den meisten Tests verwendet und wenn nötig in einem Test angepasst, zum Beispiel in dem "böse"Rollen entfernt werden.

Weiter getestet wird das Starten und das Beenden von einzelnen Spielphasen sowie das Eliminieren von Spielern. Bei diesen Tests wird ebenfalls Gebrauch von der immer gleichen Liste an Spielern gemacht.

5.2 Bewertung der Unit Tests

Bei der Erstellung der Unit Tests war es das Ziel sogenannte GUTs (Good Unit Tests) zu erstellen. Hierfür haben wir die Unit Tests immer zeitnah zu den jeweiligen Methoden geschrieben. In den meisten Fällen bereits vor der Implementierung der Methode, so dass sich an die Richtlinien für Test Driven Design gehalten werden konnte. Auch wurden die Tests bereits frühzeitig als hilfreich erkannt (vgl. KartenUndRollenTests.java) und dementsprechend versucht an sinnvollen Stellen einzusetzen.

Die weitere Bewertung der Unit Tests erfolgt anhand ATRIP-Regeln für gute Unit Tests:

- Automatic - Eigenständig
- Thorough - Gründlich
- Repeatable - Wiederholbar
- Independet - Unabhängig voneinander
- Professional - Mit Sorgfalt hergestellt

Diese fünf Regeln werden im folgenden Abschnitt kurz dargestellt und mit ihnen die erstellten Unit Tests überprüft.

Automatic

Tests sollten eigenständig ablaufen und ihre Ergebnisse selbstständig überprüfen. Ein Eingriff von außen ist nicht vorgesehen. Diese Eigenschaft wird von allen verwendeten

Unit Tests erfüllt. Um diese Regel nicht zu brechen wurde bspw. bei den Tests der Klasse *KonsolenEingabenTests.java* die Ein- und Ausgabe der Konsole über PrintStreams und ByteArrayOutputStreams simuliert.

Thorough

Die Testabdeckung sollte alles Notwendige beinhalten. Was als notwendig zählt ist hierbei anhand von den Rahmenbedingungen zu definieren. Im Bezug auf dieses Softwareprojekt sind folgende Funktionen unbedingt notwendig um die anfangs gestellten Anforderungen zu erfüllen:

- Initialisieren der Karten durch die Datenbank
- Abrufen von Karten und Informationen in der Konsole
- Starten und Durchführen des Gameloops

Die ersten beiden Punkte werden durch die Unit Tests ausreichend abgedeckt. Während das initialisieren durch die Tests der Klassen *KartenUndRollenTests.java* und *RollenGesinnungTest.java* abgedeckt wird, werden die Konsoleneingaben zur Suche nach bestimmten Karten und Informationen in den Tests der Klasse *KonsolenEingabenTest.java* behandelt. Der dritte Punkt dieser Liste ist jedoch noch nicht ganz abgedeckt. Zwar wird das Starten, Beenden, Eliminieren und auch einige Fehlerfälle volends getestet, doch bei den Spielfunktionen sind noch ein paar wenige Fälle offen. Dennoch kann alles in allem von einer relativ guten Testabdeckung der genannten Schwerpunkte des Programms gesprochen werden.

Außerdem sollte für jeden Bug Report ein Test existieren bzw. ggf. ein neuer Test geschrieben werden. Dadurch dass dieses Projekt noch recht jung ist, ist dies bisher nicht der Fall und kann daher in der Bewertung vernachlässigt werden.

Aufgrund der oben aufgezählten Punkte wird die Testabdeckung als ausreichend gut für die Hauptaufgaben der Anwendung bewertet.

Repeatable

Um die Zuverlässigkeit von Tests zu gewährleisten müssen diese jederzeit das gleiche Ergebnis liefern. Das ist bei den hier geschriebenen Tests der Fall, die besitzen keine

Abhängigkeiten äußeren Begebenheiten wie bspw. Datum, Zeit oder Systemvariablen. Der einzige Einfluss von außen (die Datenbank) wurde durch den Einsatz eines Mocks simuliert und die Abhängigkeit dadurch aufgelöst. Die Tests erzeugen daher zu jedem Zeitpunkt das gleiche Ergebnis.

Independent

Um Fehler zuverlässig und schnell erkennen zu können müssen Tests in jeder beliebigen Reihenfolge ablaufen können (also keine Abhängigkeiten untereinander besitzen) und möglichst jeden Aspekt eines Programms einzeln testen.

Auch dies wurde in den Unit Tests umgesetzt. Jeder Test testet einen spezifischen Anwendungsfall, beispielsweise die Überprüfung ob böse Rollen auch als solche gekennzeichnet werden. Läuft ein Test fehl, so kann man sehr genau differenzieren an welcher Stelle im Code nach einem Fehler gesucht werden muss.

Eventuell entstehende Abhängigkeiten durch das Aufsetzen eines Streams im *@BeforeEach* in der Klasse *KonsolenEingabenTests.java* werden durch das Zurücksetzen nach jedem Test (*@AfterEach*) verhindert. Auf diese Weise kommt es zu keinen Abhängigkeiten der Tests untereinander.

Professional

Um die Lesbarkeit der Tests zu erhöhen wurden diese in ein eigenes Projekt *Werwolf_99_tests* ausgelagert und für jede zu testende Schicht ein eigenes Package erstellt. Auf diese Weise lässt sich schnell erschließen, welcher Bereich durch die jeweiligen Tests abgedeckt wird.

Auch die Klassennamen wurden ausreichend sprechend gewählt um sofort einen Überblick darüber zu bekommen was getestet wird. So sind die Tests, welche Methoden zur Karten- und Rollenerstellung innerhalb des Adapters testen beispielsweise in dem Package *werwolf.adapter.test* in der Klasse *KartenUndRollenTest.java*.

Die Namen der jeweiligen Tests wurden so gewählt, dass direkt klar ist welche Methode getestet wird (*zeigeNamenUndFunktionTest()* testet bspw. die Methode *zeigeNamenUndFunktion()*). Durch die ebenfalls sprechenden Methodennamen, welche bereits beschrieben wurden, besteht auch hier ein einfaches Verständnis.

Die Lesbarkeits der Test hängt jedoch nicht nur von der Namensgebung ab. Auch

der eigentliche Code sollte lesbar gestaltet sein. Diese Lesbarkeit wird durch eine klare Codestruktur sowie Inline Kommentare ermöglicht. So ist der grundsätzliche Aufbau der Tests (Arrange, Act, Assert) immer gut voneinander abgetrennt. Auf diese Weise kann beim Lesen besser nachvollzogen werden was genau geschieht.

Die Variablennamen sind teilweise jedoch noch zu unspezifisch. So gibt *Testliste* beispielsweise zwar an, dass es sich um eine für den Test spezifische Liste handelt (welche anschließend mit der durch die Methode generierten Liste verglichen wird), jedoch nicht genau genug was Inhalt und Zweck dieser Liste ist.

Die Klasse *GameLoopTest.java* ist mit ihren ca. 250 Zeilen trotz der guten Codestruktur jedoch wesentlich zu umfangreich und sollte in mindestens zwei Klassen aufgeteilt werden.

Alles in allem kann aber dennoch (nicht zu letzt wegen der sauberen Code Struktur innerhalb der Tests) von einer guten Professionalität gesprochen werden.

Kapitel 6

Refactoring

6.1 Code Smell 1 - Kontrollfluss durch Exceptions

Innerhalb des GameLoops wurden bestimmte Abspekte eines Spielablaufes durch Exceptions gesteuert. Dies ist nicht Sinn einer Exception und sollte durch bessere Überprüfung von Edgecases und durch bessere Rückgabewerte ersetzt werden. Zu beachten ist jedoch, dass manche Exceptions eine Daseinsberechtigung besitzen. Dies wird an ein paar Beispielen verdeutlicht, es werden nicht alle refaktorierten Methoden schriftlich dargestellt. Eine vollständige Übersicht findet sich im Commit 7cf8819[1]

Ein Beispiel für sinnvolle Exceptions soll kurz aufgezeigt werden. In der Methode *ueberpruefeSpieler()* werden die durch den Benutzer eingetragenen Spieler und Rollen überprüft. Bei bestimmten Rollenkombinationen soll das Starten des Spiels nicht möglich sein. Zur Verdeutlichung der ungültigen Konfiguration wird eine Exception geworfen.

[1] https://github.com/diemeise/SWE-Werwolf-Enzyklopaedie/commit/7cf8819fcfb0aad2e46e69c76decb6acfa646ba3

```
private void ueberpruefeSpieler(List<Spieler> spieler2) throws GameException {
        if(spieler2.isEmpty()) {
                throw new GameException("Keine Spieler uebergeben!");
        }
        boolean boeseRolleVorhanden = false;
        for (Spieler s : spieler2) {
                if (s.istBoese()) {
                        boeseRolleVorhanden = true;
                }
        }
        if (!boeseRolleVorhanden) {
                throw new GameException("Es muss ein Wolf / eine boese Karte vorhanden sein!");
        }
}
```

Abbildung 6.1: Die Methode ueberpruefeSpieler

Refactoring

Methoden findeErstenSpieler und findeNachstenSpieler

Vor dem Refactoring warfen diese beide Methode eine Fehlermeldung, wenn keine (weiteren) Spieler in einer Nacht aktiv waren. Dies ist ein Punkt, der in jeder Phase kurz vor Ende immer eintritt und somit ein Bestandteil des Spielflusses. Diese Methoden wurden entsprechend geändert, dass keine Fehlermeldung mehr geworfen wird. Dafür wurde auch die Funktionsweise an sich verändert. Der neue aktive Spieler wird nun nicht mehr als Rückgabewert zurückgegeben, sondern direkt in der Methode als aktiv markiert. Stattdessen zeigt ein Wahrheitswert an, ob noch weitere Spieler in dieser Spielphase aktiv werden. Durch die Umstrukturierung mussten auch andere Methoden leicht angepasst werden.

Methoden neachsterSpielschritt

In dieser Methode wurde eine Exception geworfen, wenn die Phase bereits abgeschlossen war aber dennoch ein Spielschritt ausgeführt werden sollte. Hiermit wurde die Logik zum Starten der nächsten Spielphase in der *GameLoop-Klasse* gesteuert. Diese Exception wurde entfernt, zusammen mit den entsprechenden damit verknüpften Funktionen. Stattdessen wird nun ebenfalls über ein Wahrheitswert die Logik gesteuert.

```
private Spieler findeErstenSpieler() throws GameException {
        for (Spieler spieler : lebendeSpielerBeiStart) {
                if(spieler.getPrio() > 0) {
                        return spieler;
                }
        }

        throw new GameException("Keine aktiven Spieler diese Nacht!");
}


private Spieler findeNaechstenSpieler() throws GameException {
        int indexAktiverSpieler = lebendeSpielerBeiStart.indexOf(aktiverSpieler);

        //finde den nächsten nicht eliminerten Spieler
        while(indexAktiverSpieler < lebendeSpielerBeiStart.size()-1) {
                indexAktiverSpieler++;
                Spieler naechsterSpieler = lebendeSpielerBeiStart.get(indexAktiverSpieler);
                if (!eliminierteSpieler.contains(naechsterSpieler)) {
                        return naechsterSpieler;
                }
        }
        //keine weiteren Spieler vorhanden
        throw new GameException("Keine weiteren Spieler diese Nacht!");
}
```

Abbildung 6.2: Die Methode findeNaechstenSpieler vor dem Refactoring

```
private boolean setNaechstenSpieler() {
        int indexAktiverSpieler = lebendeSpielerBeiStart.indexOf(aktiverSpieler);

        //finde den nächsten nicht eliminerten Spieler
        while(indexAktiverSpieler < lebendeSpielerBeiStart.size()-1) {
                indexAktiverSpieler++;
                Spieler naechsterSpieler = lebendeSpielerBeiStart.get(indexAktiverSpieler);
                if (!eliminierteSpieler.contains(naechsterSpieler)) {
                        aktiverSpieler = naechsterSpieler;
                        return true;
                }
        }
        //keine weiteren Spieler vorhanden
        return false;
        //throw new GameException("Keine weiteren Spieler diese Nacht!");
}
```

Abbildung 6.3: Die Methode findeNaechstenSpieler nach dem Refactoring

```
public void naechsterSpielschritt() throws GameException {
        if(phaseAbgeschlossen) {
                throw new GameException("Phase ist bereits abgeschlossen!");
        }

        //setze ersten Spieler auf aktiv
        if (!phaseAngefangen) {
                aktiverSpieler = findeErstenSpieler();
                phaseAngefangen = true;
                aktiverSpieler.setAktiv(false);
                return;
        }

        //setze naechsten Spieler aktiv
        aktiverSpieler.setAktiv(false);
        try {
                aktiverSpieler = findeNaechstenSpieler();
        }catch (GameException e) {
                phaseAbgeschlossen = true;
        }
}
```

Abbildung 6.4: Die Methode neachsterSpielschritt vor dem Refactoring

```java
public boolean naechsterSpielschritt(){

    if(phaseAbgeschlossen) {
        throw new GameException("Phase ist bereits abgeschlosse
    }
    if (phaseAbgeschlossen) {
        return false;
    }

    if (!phaseAngefangen) {
        phaseAngefangen = true;

        if(setErstenSpieler()) {
            aktiverSpieler.setAktiv(true);
            return true;
        }
        phaseAbgeschlossen = true;
        return false;
    }


    aktiverSpieler.setAktiv(false);
    if(!setNaechstenSpieler()) {
        phaseAbgeschlossen = true;
        return false;
    }


    aktiverSpieler.setAktiv(true);
    return true;
}
```

Abbildung 6.5: Die Methode findeNaechstenSpieler nach dem Refactoring

6.2 Code Smell 2 - Large Class

Bisher wurden die gesamte Spielphasenlogik innerhalb der Klasse Nacht abgebildet. Dies hatte zur Folge, dass die Klasse durch viele Erweiterungen der Spielfunktionen zu überladen wurde. Es entstand eine unübersichtliche, zu große Klasse die mehrere gut abgrenzbare Aufgabenbereiche abdeckte. So war auch das Single Responsibility Prinzip ab einem bestimmten Punkt verletzt.

Refactoring

Innerhalb des Refactorings wurde eine weitere Klasse *Tag* erstellt, damit die unterschiedlichen Spielphasen richtig abgegrenzt werden können. Weiterhin wurde eine abstrakte Oberklasse *Spielphase* erstellt, die Code enthält, der von Tag und Nacht gleichermaßen benötigt wird. Innerhalb der Klassen Tag und Nacht werden nur noch die für diese Phase spezifischen Funktionen implementiert. Der GameLoop bzw. die GameAdapter wurden ebenfalls an benötigten Stellen angepasst damit diese Unterteilung funktioniert. Diese Änderungen sind im Commit „refactor Spielphasen“[2] erfolgt.

6.3 Code Smell 3 - Unübersichtlicher GUI Code

In der Vorlesung nicht explizit als Code Smell erwähnt und doch ein wichtiger Grund fürs Refactoring ist eine Bessere Lesbarkeit des Codes sowie klare Strukturen. So soll das Ziel der besseren Codequalität erreicht werden.

Betrachtet man die anfängliche Struktur des GUI Packages so kann keineswegs von einer guten Übersicht gesprochen werden. Durch die Umstellung auf Java FX und die zu Beginn des Projekts geringen Erfahrung damit sind Java Klassen, Stylesheets und fxml Dateien in einem einzigen Package zusammen. So verliert man beim Versuch den Code nachzuvollziehen sehr schnell den Überblick.

[2] https://github.com/diemeise/SWE-Werwolf-Enzyklopaedie/commit/b8bf7d3f933f52bb08cdcf735850a42906c894ad

Refactoring

In einem ersten Schritt des Refactorings wurden das neue Package fxml innerhalb des GUI Packages erstellt und die fxml Dateien hinzugefügt. Es wurde sich außerdem dazu entschieden auch das Stylsheet in dieses Package zu verlegen. Man könnte für css Datein ebenfalls ein eigenes Package erstellen, da das vorliegende Projekt jedoch nur eine css Datei beinhaltet wird die Übersicht des fxml Packages nicht gefährdet.

Durch das Verschieben der Dateien mussten natürlich auch einige Code Zeilen angepasst werden. Der Aufwand hielt sich jedoch in Grenzen, da lediglich der jeweilige Pfad angepasst werden musste. Hierbei war zu beachten, dass die Pfade nicht nur im Code sondern auch in dem zur Erstellung der fxml Dateien verwendeten Programm *Scenebuilder* angepasst werden mussten.

Um die Übersichtlichkeit und Lesbarkeit des Codes weiter zu erhöhen war es außerdem notwendig einige der fxml Dateien und die zugehörigen Controller Klassen umzubenennen. Durch das *einfach los programmieren* trotz mangelnder FX Erfahrung waren die Namen nicht sonderlich gut gewählt, so dass nicht gleich klar war welches fxml und welcher Controller für welche Ansicht zuständig war. Die Umbenennungen waren folgende:

$MainFXML.fxml \Rightarrow Startseite.fxml$
$PlayMain.fxml \Rightarrow SpielErstellen.fxml$

Die Controller wurden zum besseren Verständnis wie die fxml Dateien mit dem Zusatz *Controller* benannt. Die beiden Namen der Dateien *SpielScene.fxml* und *SpielController.java* konnten beibehalten werden.

6.4 Code Smell 4 - Duplication Code

Innerhalb des OutputAdapters gibt es einige Methoden zur Ausgabe von Karten mit bestimmten Eigenschaften. Innerhalb dieser Methoden werde immer sehr ähnliche Funktionen aufgerufen. Die Filterung der Karten sowie die Ausgabe des Rollennamens und der Rollenfunktion waren in allen Methoden vorhanden.

```java
public HashMap<String, String> getAlleGutenKarten(){

    HashMap<String, String> gut = new HashMap<>();

    karten = gamelib.getKartenRepository().getKarten(); //TODO außerhalb der Methoden initialisieren?

    for (String key: karten.keySet()) {

        if(!karten.get(key).getRolle().istBoese()) {
            name = karten.get(key).getRolle().getName();
            funk = karten.get(key).getRolle().getFunktion();

            gut.put(name, funk);
        }

    }

    return gut;
}
```

Abbildung 6.6: Methode für die guten Karten

```java
public HashMap<String, String> getAlleBoesenKarten(){

    HashMap<String, String> boese = new HashMap<>();

    karten = gamelib.getKartenRepository().getKarten(); //TODO außerhalb der Methoden initialisieren?

    for (String key: karten.keySet()) {

        if(karten.get(key).getRolle().istBoese()) {
            name = karten.get(key).getRolle().getName();
            funk = karten.get(key).getRolle().getFunktion();

            boese.put(name, funk);
        }

    }

    return boese;
}
```

Abbildung 6.7: Methode für die bösen Karten

```java
public Map<String, String> getAlleSpezialKarten(){

    Map<String, String> spezial = new HashMap<>();

    karten = gamelib.getKartenRepository().getKarten(); //TODO außerhalb der Methoden initialisieren?

    for (String key: karten.keySet()) {

        if(karten.get(key).getRolle().istSpezial()) {
            name = karten.get(key).getRolle().getName();
            funk = karten.get(key).getRolle().getFunktion();

            spezial.put(name, funk);
        }

    }

    return spezial;
}
```

Abbildung 6.8: Methode für die spezial Karten

Refactoring

Für das Refactoring wurden die beiden Methoden *Extract Method* sowie *Replace Temp with Query* angewendet. Innerhalb des *Kartenrepository* wurde eine neue Methode *getKartenMitFilter* implementiert. Diese erhält einen Filterparameter und gibt eine entsprechende Collection an Karten zurück. Innerhalb des *OutputAdapters* gibt es ebenfalls eine neue Methode *filterKarten*. Diese nimmt die Collection an Karten entgegen und durch eine weitere neue Methode werden aus den Kartenobjekten eine Ausgabemap bestehend aus Strings erstellt.

Als nächster Refactoring-Schritt könnten die alten Methoden *getAlleSpezialKarten, getAlleBoesenKarten, getAlleGutenKarten* entfernt werden und durch entsprechende Aufrufe der neuen Funktion *filterKarten* ersetzt werden.

```
public Map<String, String> getAlleSpezialKarten(){
    return filterKarten("spezial");
}

public Map<String, String> getAlleBoesenKarten(){
    return filterKarten("boese");
}

public Map<String, String> getAlleGutenKarten(){
    return filterKarten("gut");
}


private Map<String, String> konvertKartenMapZuStringMap(Map<String, Karte> karten) {
    Map<String, String> returnMap = new HashMap<>();
    for (String key: karten.keySet()) {
        name = karten.get(key).getRolle().getName();
        funk = karten.get(key).getRolle().getFunktion();
        returnMap.put(name, funk);
    }
    return returnMap;
}

public Map<String,String> filterKarten(String filter){
    Map<String, Karte> filteredKarten = gamelib.getKartenRepository().getKartenMitFilter(filter);

    return konvertKartenMapZuStringMap(filteredKarten);
```

Abbildung 6.9: Neue Methoden im *OutputAdapter*

```java
/**
 * Gibt eine bestimmte Art von Karten zurueck
 * @param filter: moegliche Werte: "gut", "boese", "spezial"
 */
@Override
public Map<String, Karte> getKartenMitFilter(String filter) {

    Map<String, Karte> filteredKarten = new HashMap<>();

    for (String name: karten.keySet()) {
        if(filter == "gut") {
            if(!karten.get(name).getRolle().istBoese())
                filteredKarten.put(name, karten.get(name));
        }
        if(filter == "boese") {
            if(karten.get(name).getRolle().istBoese())
                filteredKarten.put(name, karten.get(name));
        }
        if(filter == "spezial") {
            if(karten.get(name).getRolle().istSpezial())
                filteredKarten.put(name, karten.get(name));
        }
    }
    return filteredKarten;
}
```

Abbildung 6.10: Neue Methode im *KartenRepository*

6.5 Code Smell 5 - Inappropiate Information

Robert C. Martin beschreibt in seinem Buch *Clean Code* eine Reihe Code Smells welche sich auf Kommentare beziehen. Zwar sind Kommentare häufig nur ein Indiz für einen anderen Code Smell, jedoch gibt es auch einige Kommentare, welche selbst als solcher bezeichnet werden können.

Einer davon ist die Inappropiate Information. Metadaten sollten nicht im Code Substrat stehen sondern in einer externen Anwendung verwaltet werden. Zu Beginn des Projekts haben wir viel mit Todos gearbeitet.

Refactoring

Die erwähnten Todos werden nun in einer externen Anwendung verwaltet und die entsprechenden Kommentare aus dem Code entfernt. Ein Beispiel für einen solchen Kommentar kann in Abbildung 6.11 eingesehen werden.

In der Abbildung ist neben dem erwähnten Code Smell auch noch ein weiterer zu sehen. Der Kommentar unter dem Todo gibt an, dass die Methode nicht mehr

```
//TODO In eigene Klasse auslagern?
//wird eigentlich nicht mehr benÃ¶tigt!
@Override
public HashMap<String, String> zeigeNameUndFunktion() {
    String name;
    String funk;
    HashMap<String, String> kartenFunk = new HashMap<>();

    for (String key: karten.keySet()) {
        name = karten.get(key).getRolle().getName();
        funk = karten.get(key).getRolle().getFunktion();

        kartenFunk.put(name, funk);
    }

    return kartenFunk;
}
```

Abbildung 6.11: Inappropiate Information

benötigt wird. Statt diesen Kommentar zu verfassen hätte man direkt die Methode löschen können.

6.6 Code Smell 6 - Auskommentierter Code

Code welcher nicht mehr benötigt wird sollte nicht auskommentiert in der Code Basis stehen bleiben. Später folgende Entwickler*innen (bzw. auch man selbst nach einer bestimmten Zeit) wissen ggf. nicht warum dieser Code noch drin steht und löschen ihn ebenfalls nicht. Auf diese Weise wird das Substrat schnell unübersichtlich und voller uninteressanter Informationen.

Refactoring

Wir haben versucht die meisten auskommentierten Stellen beim nachträglichen Refactoring zu entfernen. Eine gewollte Ausnahme bildet die Code Zeile in der Main Klasse. Hier kann nach aktuellem Stand zwischen CLI und GUI nur durch das auskommentieren bzw aktivieren der entsprechenden Zeilen gewechselt werden. Dies wird im weiteren Verlauf der Entwicklung überarbeitet.

Kapitel 7

Entwurfsmuster

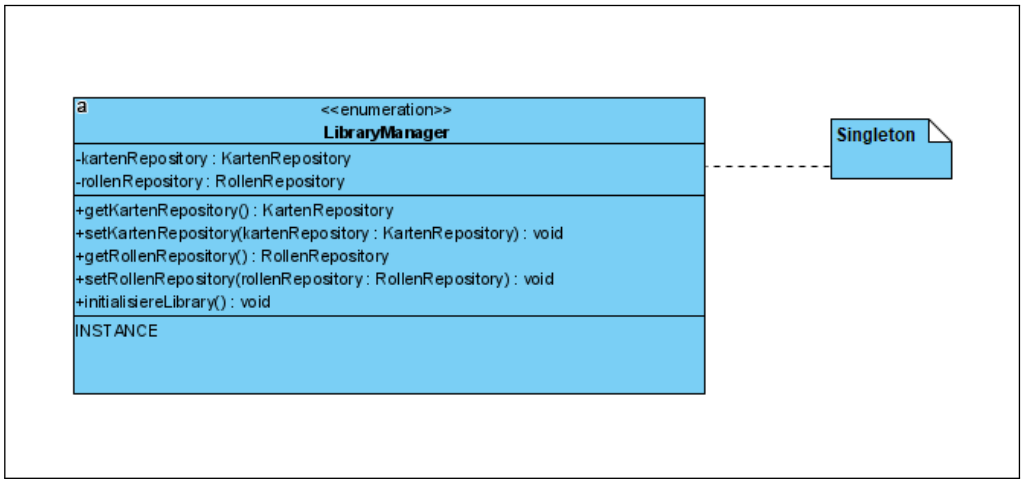

Abbildung 7.1: UML-Diagramm des Singleton

Innerhalb der Entwicklung der Anwendung hat sich kein eindeutiges Entwurfsmuster herauskristallisiert. Jedoch ist aufgefallen, dass die Klasse *LibraryManager* von mehreren unterschiedlichen Klassen referenziert wird. Da diese Klasse nur ein Wrapper für die Verwaltung Domänenelemente *Karte* und *Rolle* ist, eignet sie sich hervorragend als Singleton. Dadurch kann vermieden werden, die Klasse von der Initialisierung in der Main-Methode als Parameter in alle weiteren Klassen zu übergeben, die diese Klasse benötigen.

Zur Implementierung des Singletons wurde die Klasse in ein ENUM umgewandelt. Dieses wird weiterhin in der Main-Methode initialisiert und befüllt, aber kann nun

danach statisch aus der gesamten Codebasis (bis auf die darunterliegende Domain-Schicht) verwendet werden. Die größte Änderung befindet sich bei der Initialisierung. Da die Instanzvariablen nun nicht mehr im Konstrukter gesetzt werden können, geschieht dies über eigene Settermethoden. Dies hat ebenfalls zur Folge, dass die verwendeten Repositorys zum Speichern der *Rollen* und *Karten* nicht mehr als final deklariert werden können.

Weiter wurde die *LibraryManager*-Variable betroffener Klassen entfernt, und etwaige Methodenaufrufe durch *getInstance()*-Aufrufe ersetzt. Bei einigen Tests wurde eine neue *@BeforeAll*-Methode eingebaut, die das Singleton vor der Testausführung erzeugt.

```
LibraryManager gameLibrary= LibraryManager.INSTANCE;
gameLibrary.setKartenRepository(new SQLKartenRepository(mysql));
gameLibrary.setRollenRepository(new SQLRollenRepository(mysql));
gameLibrary.initialisiereLibrary();
```

Abbildung 7.2: Initialisierung des LibraryManagers als Klasse

```
LibraryManager gameLibrary= LibraryManager.INSTANCE;
gameLibrary.setKartenRepository(new SQLKartenRepository(mysql));
gameLibrary.setRollenRepository(new SQLRollenRepository(mysql));
gameLibrary.initialisiereLibrary();
```

Abbildung 7.3: Initialisierung des LibraryManagers als ENUM/Singleton

```java
//Singleton!
public enum LibraryManager {

    INSTANCE();

     private KartenRepository kartenRepository;
     private RollenRepository rollenRepository;

    private LibraryManager() {

    }

    public void initialisiereLibrary() {
        rollenRepository.ladeRollenAusSpeicher();
        kartenRepository.ladeKartenAusSpeicher();
        kartenRepository.verknuepfeKartenMit(rollenRepository);
    }

    public KartenRepository getKartenRepository() {
        return kartenRepository;
    }

    public RollenRepository getRollenRepository() {
        return rollenRepository;
    }
    public void setKartenRepository(KartenRepository kr) {
        this.kartenRepository = kr;
    }

    public void setRollenRepository(RollenRepository rr) {
        this.rollenRepository = rr;
    }
```

Abbildung 7.4: Der veränderte LibraryManager