



Advanced Software Engineering Textbasierter Dungeon Crawler

Programmentwurf

im Rahmen der Prüfung zum

Bachelor of Science (B.Sc.)

des Studienganges Informatik

an der Dualen Hochschule Baden-Württemberg Karlsruhe

von

Sascha Vollmer

Abgabedatum: 30. Mai

Matrikelnummer, Kurs: 5700656, TINF19B5

Gutachter der Dualen Hochschule: Maurice Müller

Eidesstattliche Erklärung

Wir Versichern hiermit, dass wir diese Programmentwurf mit dem Thema:

Advanced Software Engineering Textbasierter Dungeon Crawler

gemäß § 5 der „Studien- und Prüfungsordnung DHBW Technik“ vom 29. September 2017 selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Wir versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Oberkirch, den 30. Mai 2022

Gez. Sascha Vollmer

Vollmer, Sascha

Inhaltsverzeichnis

1	Einführung	1
1.1	Übersicht über die Applikation	1
1.2	Wie startet man die Applikation	1
1.3	Technischer Überblick	1
2	Clean Architecture	3
2.1	Was ist Clean Architecture	3
2.2	Analyse der Dependency Rule	3
2.3	Analyse der Schichten	4
3	SOLID	8
3.1	Analyse SRP	8
3.2	Analyse OCP	10
3.3	Analyse LSP	11
4	Weitere Prinzipien	13
4.1	Analyse GRASP: Geringe Koppelung	13
4.2	Analyse GRASP: Hohe Kohäsion	15
4.3	DRY	15
5	Unit Tests	17
5.1	10 Unit Tests	17
5.2	ATRIP: Automatic	17
5.3	ATRIP: Thorough	17
5.4	ATRIP: Professional	17
5.5	Fakes und Mocks	18
6	Domain Driven Design	19
6.1	Ubiquitous Language	19
6.2	Repositories	19
6.3	Aggregates	20
6.4	Entities	20
6.5	Value Objects	21
7	Refactoring	22
7.1	Code Smells	22
7.2	2 Refactorings	23

8 Entwurfsmuster	26
8.1 Entwurfsmuster: Konstruktor/Erbauer	26

1 Einführung

1.1 Übersicht über die Applikation

Bei der Applikation handelt es sich um einen Textabsierten Dungeon Crawler, bzw. ein Framework mit dem ein solcher einfach umgesetzt werden kann. In der Applikation ist Test Text, sowie ein Test Dungeon enthalten durch das Nutzer sehen können was mit diesem Framework möglich ist. Diese Applikation hat als Ziel Dungeon Crawler ohne konkret Coden zu müssen anhand von speziel Formartierten Daten erstellen zu können.

1.2 Wie startet man die Applikation

1.3 Technischer Überblick

Für die Implementierung wurden folgende Techniken eingesetzt.

1.3.1 Java

aus de zwei erlaubten sprachen ist Java dem Autor am vertauteren. Ist die haupt Sprache in der diese Applikation geschrieben ist.

1.3.2 Swing

Da für den Dungeon Crawler sehr viele Informationen auf einmal ersichtlich sein sollen und die Ausgabe des Textes in einer Kommandozeile hierdurch eingeschränkt ist, wurde sich entschieden die Textfelder in eine GUI einzubauen. Da Vorwissen in Swing vorhanden ist wurde sich für Swing entschieden.

write exe-
cution
instruc-
tions

1.3.3 Maven

für die Dependency Kontrolle wird Maven eingesetzt. Es werden nur 1 nicht Java interne Librarys verwendet,

1.3.4 org.json & JSON

Es wurde als Datenspeicherformat JSON gewählt da es von den grundlegenden eigenschaften her vergleichbar mit xml für dieses Projekt ist und eine präferenz für json vorhanden ist. Als Datenspeicher war eines der beiden notwendig um schnell und übersichtlich die informationen in die richtigen Klassen zu konvertieren. Für diese Konvertierung wurde zusätzlich die org.json Library verwendet.

1.3.5 JUnit Testing

Als Testing Framework wurde sich für JUnit entschieden. Durch dieses werden die Unit-Tests für die Applikation durchgeführt.

2 Clean Architecture

2.1 Was ist Clean Architecture

Clean Architecture ist ein ansatz die Software Architektur so aufzubauen, das die Applikation Langlebig ohne viel Maintenance am Kern Programm funktionieren kann. Dies wird erreicht durch ein 'schicht' aufbau der Dependencys. Dadurch wird versucht Externe Librarys und Technologien getrennt vom Kern des Programms zu halten. wodurch anpassungen an diesen Technologien durch ein ersätzen von Schnittstellen funktionieren kann.

2.2 Analyse der Dependency Rule

2.2.1 Negativ-Beispiel: Dependency Rule

Im hier zusehenden Fall wird durch die Main Klasse der Initilizer erzeugt und gestartet. Die MyJsonParser Klasse dient 'Plugin'. Der Grund warum es sich um ein negativ beispiel handelt liegt daran das Initilizer selbst auch org.JSON nutzt. Hierdurch hängen beide Klassen von org.JSON ab. Dies wäre für MYJsonParser in Ordnung. Die Abhängigkeit zu Initilizer ist jedoch ein Sprung über die Schichten der Clean Architecture und eine falsche Dependency. Dies ist in 2.1 zu sehen.

2.2.2 Positiv-Beispiel: Dependency Rule

Die GUI Klasse welche für die SWING gui verantwortlich ist wird durch ein IO interface genau definiert. Hierdurch ist die GUI über Adapter Code abstrahiert. Die Aktuell weiß die GUI Klasse nichts über die Main / BattleInterface Klassen. Dies ist in 2.2 zu sehen.

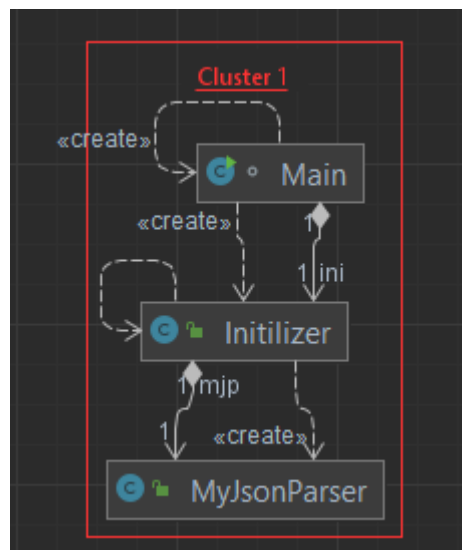


Abbildung 2.1: Die Dependencies JSon Parser

2.3 Analyse der Schichten

2.3.1 Schicht: Plugins

Die GUI ist die Schnittstelle über die ein SWING Interface erstellt und gesteuert wird. Hierbei werden in der GUI dependencies zu den SWING Klassen ausgelegt. Die GUI wird als Plugin eingeordnet, da sie für die Applikation so angelegt ist, das sie einfach tauschbar bleibt solange das Ersatz Plugin auch das IO Interface implementiert. Die GUI hat nichts mit der konkreten Domain Logik zu tun weswegen diese einordnung getroffen werden konnte. Dies wird in 2.3 auch nochmal gezeigt

2.3.2 Schicht: Domain Schicht

In die Domain Schicht werden die in ?? zu sehenden Klassen in die Domainen Schicht zugeordnet. Wie auch zusehen ist handelt es sich bei einer Vielzahl dieser Klassen um Enums. Diese Enums und die Klassen die auf ihnen aufbauen stellen die Grundlagen der Game Entities dar. Sie selbst erfüllen ohne den Applikation Code keinen weiteren Sinn, außer die Daten der Domain in ein nutzbares Format zu bringen, und mit diesen

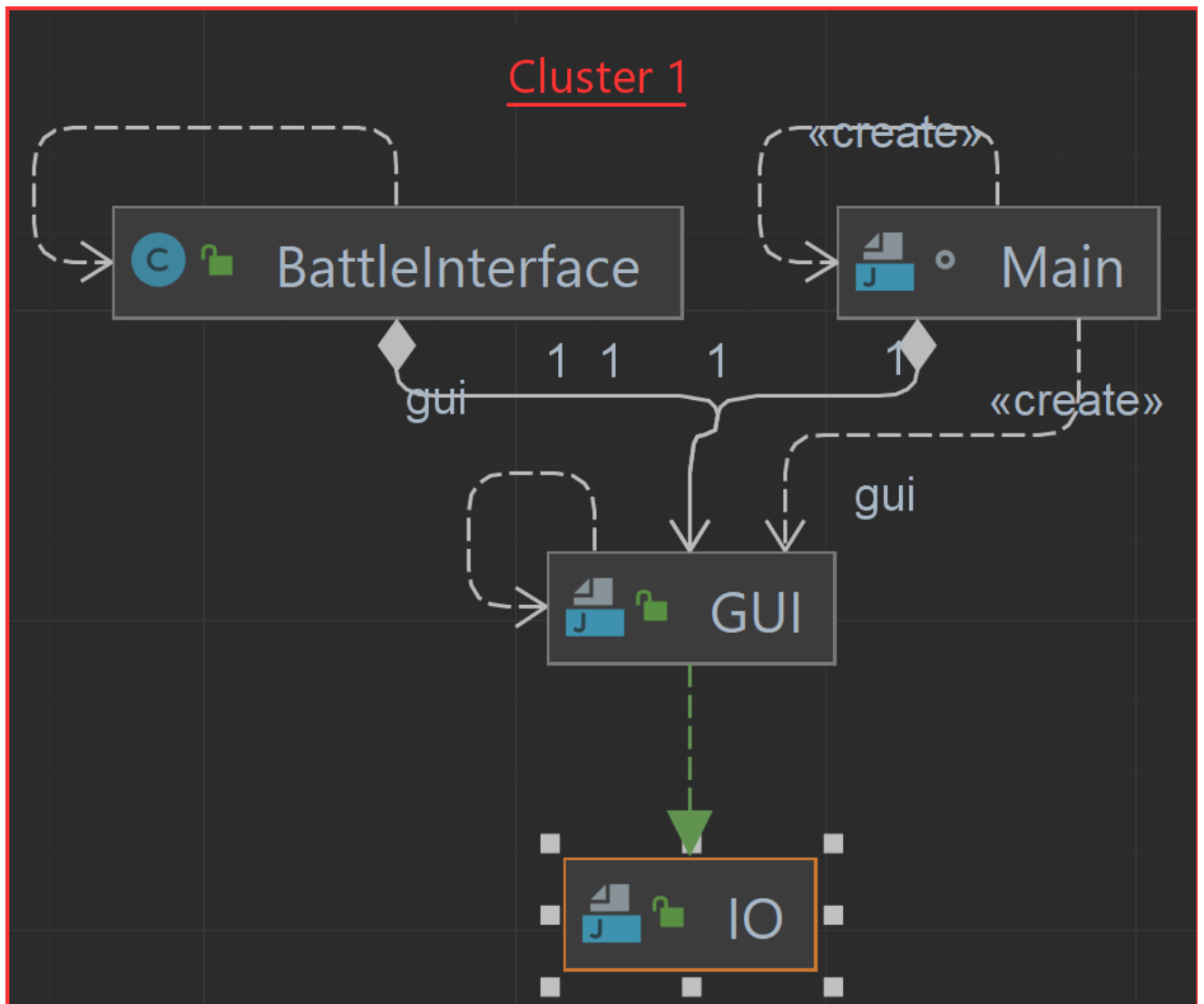


Abbildung 2.2: Alle Dependencys der GUI Klasse

Umgehen zu können. Diese Klassen hängen wie in dem UML-Diagramm ersichtlich ist alle durch die Zugrundeliegenden Enums wie DamageType zusammen.

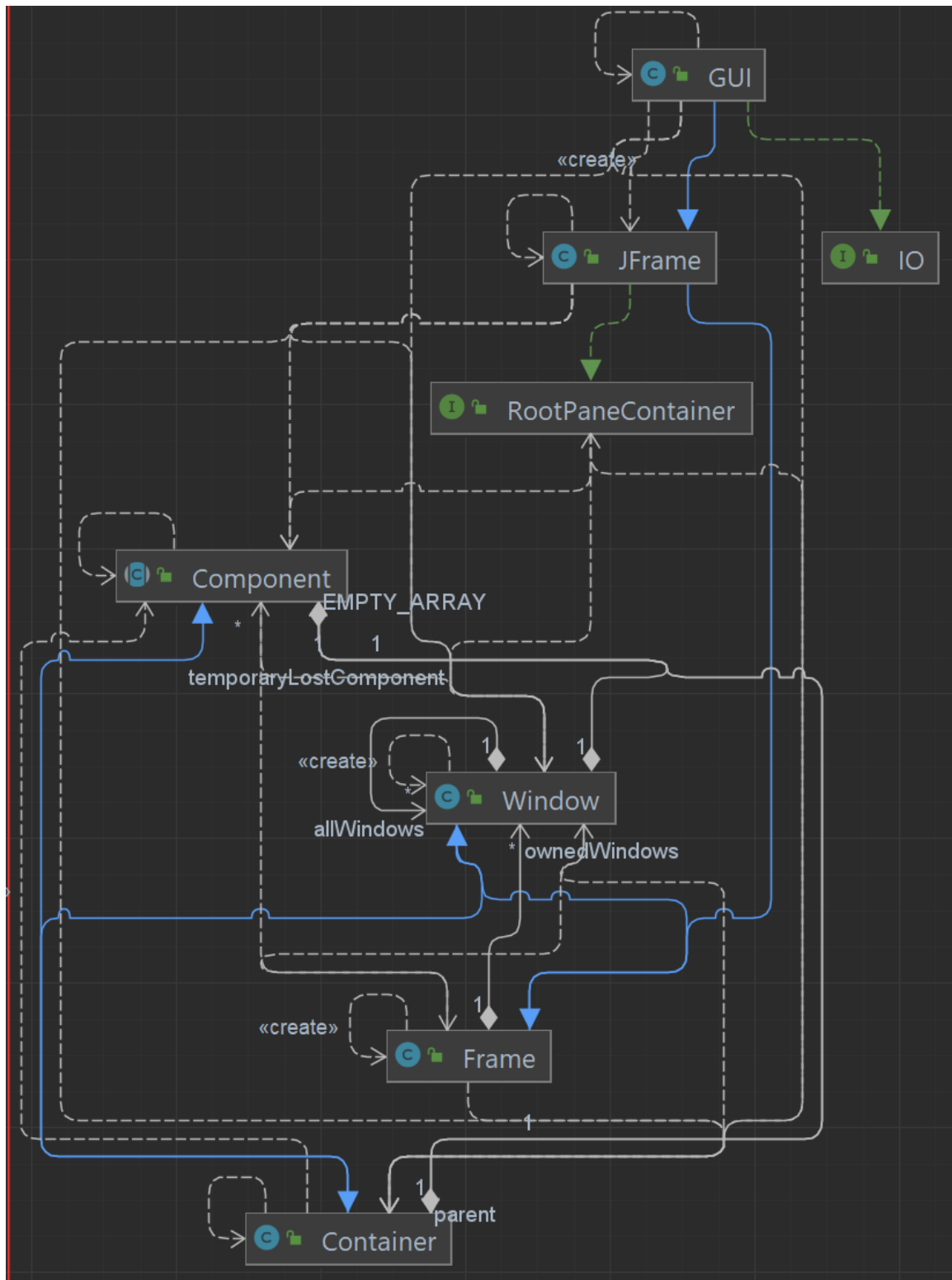


Abbildung 2.3: UML der GUI in Hinsicht auf die SWING Abhängigkeiten

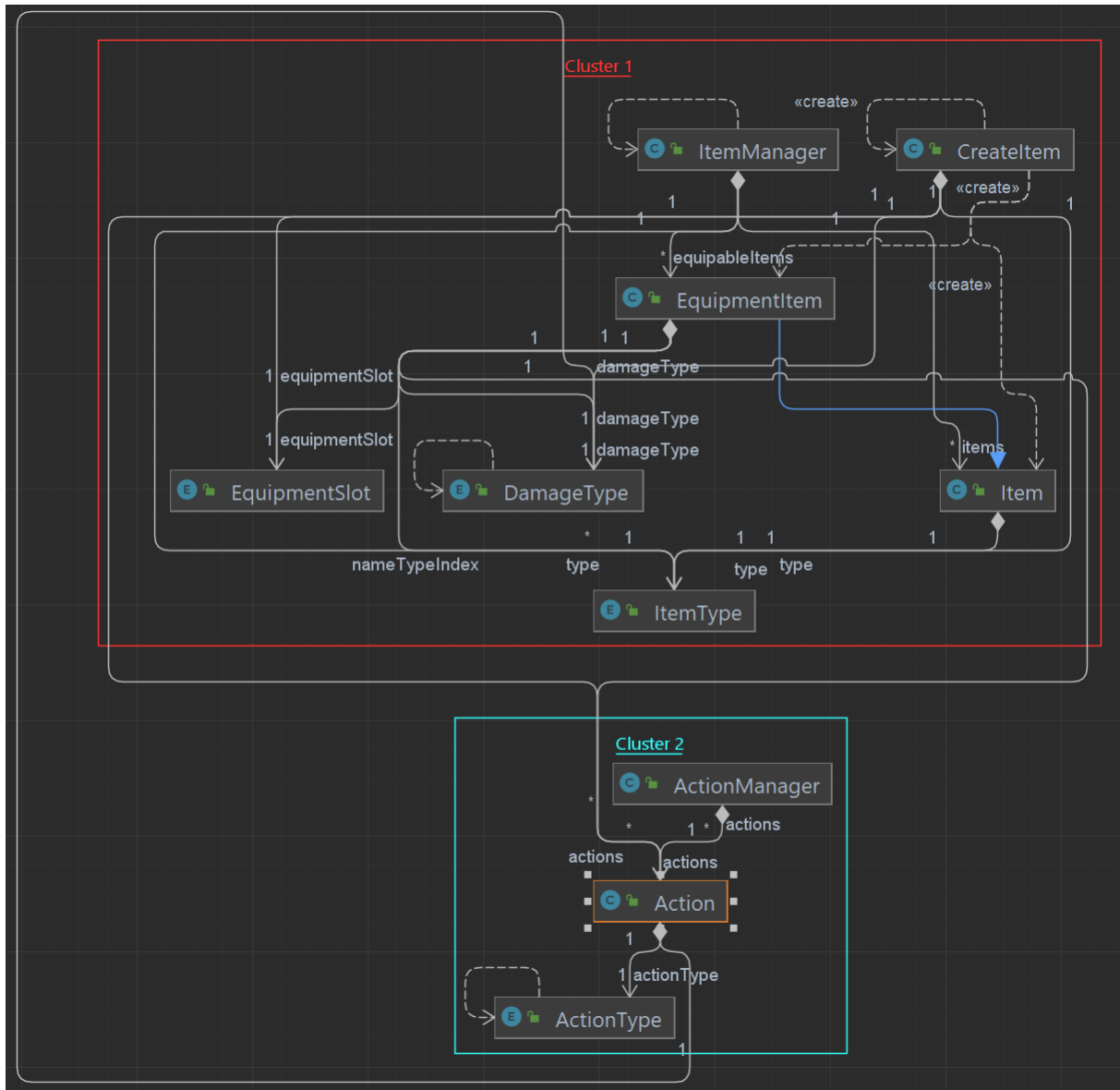


Abbildung 2.4: UML Einblick in die Domainenschicht

3 SOLID

3.1 Analyse SRP

3.1.1 Positiv-Beispiel: CreateNPC

Bei der CreateNPC Klasse handelt es sich um einen Konstruktor. Diese Klassen hat den Zweck die Unterschiedlichen arten und füllen von NPCs erstellen zu können. Dabei handelt es sich um die einzige Aufgabe die CreateNPC umsetzt. Sie wird dabei durch eine interne Klasse CreateableNPC unterstützt, da nicht jeder Parameter des NPCs verpflichtend für die Erstellung ist. Die unterschiedlichen Methoden die in CreateNPC(und Unterklassen) vorhanden sind erlauben es alle Pflicht und Optionalen Felder bei der Erstellung eines NPCs abzudecken. Dies ist auch in 3.1 zu sehen.

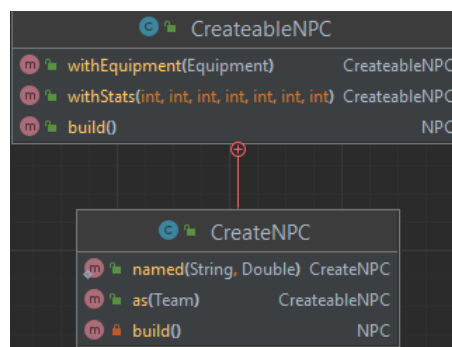


Abbildung 3.1: UML-Diagramm der CreateNPC Klasse

3.1.2 Negativ-Beispiel: Initilizer

Die Initilizer Klasse hat die Aufgabe die JSON objekte welche Domainrelevante Daten beinhalten zu den jeweiligen Objekten zu konvertieren. Hierzu werden die Jeweiligen Manager der Objekte sowie die Konstruktoren verwendet. Folgend werden alle Aufgaben konkret aufgezählt:

- Initialisieren der Items
- Initialisieren der Karte & Räume
- Initialisieren der NPCs
- Initialisieren der Spieler presets
- erzeugen von Equipment objekten für Spieler und NPCs
- erzeugen von Stats objekten für Spieler
- erzeugen von Health objekten für Spieler

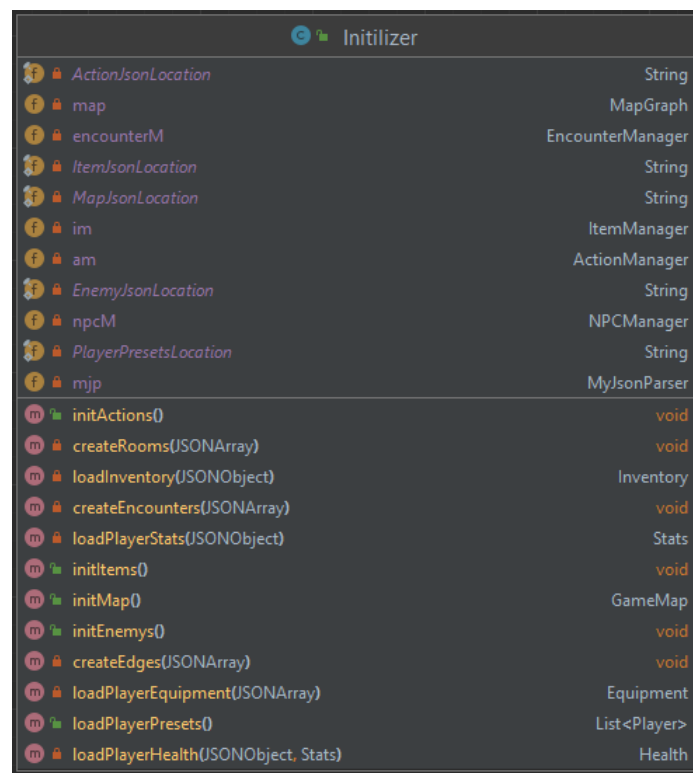


Abbildung 3.2: UML-Diagramm der CreateNPC Klasse

Dies wird auch in dem UML Diagramm deutlich 3.2. Eine mögliche Lösung dieses Verstoßes gegen SRP ist die Aufteilung in einzelne unter Initilizer Klassen welche durch Initilizer aufgerufen werden. Hierdurch kann z. B. das initialisieren von Items in eine eigene Klasse IniItems umgelagert werden. Da zwischen den unterschiedlichen aufgaben nur begrenzt Informationen ausgetauscht werden ist ein solcher Umbau die einfachste Lösung.

3.2 Analyse OCP

3.2.1 Positiv-Beispiel

Der Item Klassen Komplex wie er in 3.3 zu sehen ist, erfüllt OCP aus folgenden Gründen. Items sind zwar nicht Abstrakt beinhalten aber nur die notwendigen Attribute um in der Domäne funktionieren zu können. Durch die ENUMs die auch zu sehen sind lassen sich ohne Änderung an der Nutzung von Items neue Arten von Items hinzufügen. Wenn eine neue Art Items mit Attributen wie die EquipmentItems hinzugefügt werden muss Erben diese von Items. Hierdurch funktioniert die Logik weiterhin und es lassen sich große Änderungen an anderen Stellen vermeiden wenn Neues hierzu kommt. Der Grund warum Items nicht abstrakt ist hängt damit zusammen dass mit den grundlegenden Attributen die immer in Items vorhanden sind Objekte Domänen Spezifisch erzeugt werden können. Wenn an Items etwas geändert wird würde dies OCP verletzen, da damit viele weitere

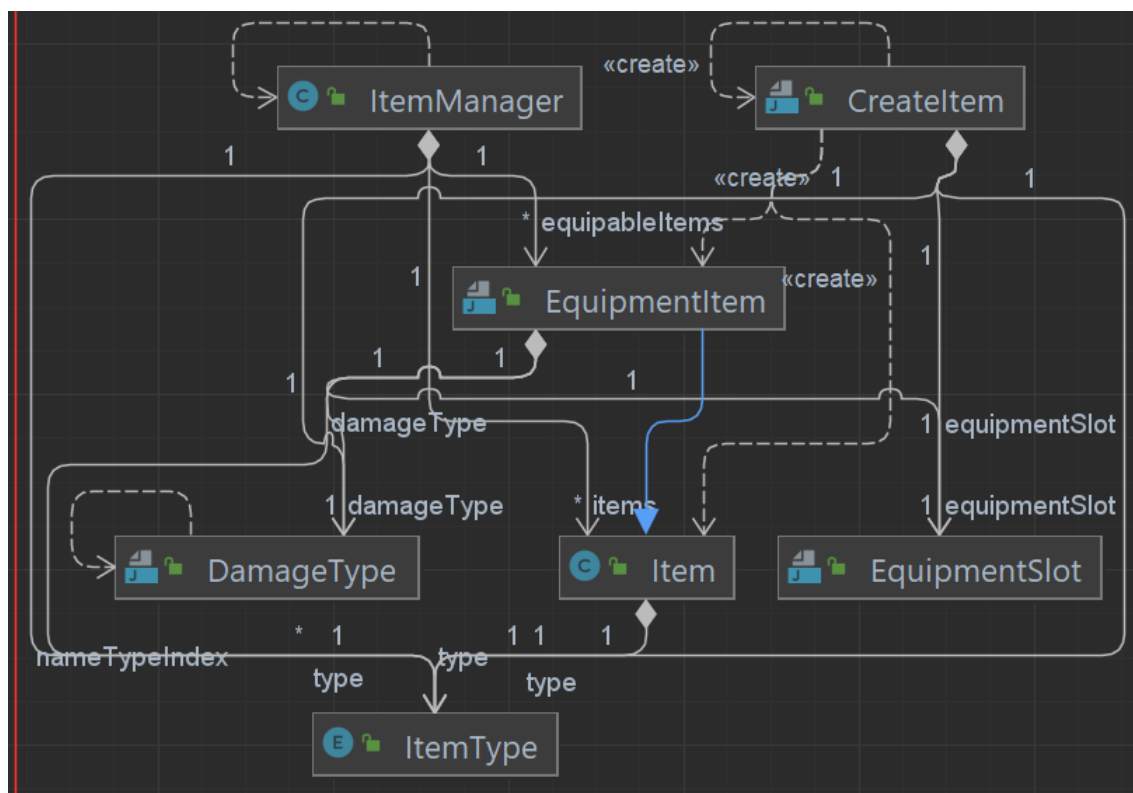


Abbildung 3.3: UML-Diagramm des Item Klassen Komplexes

Dinge zusammenhängen. Da es sich hierbei aber um eine Basis Klasse handelt welche

alle notwendigen Informationen halten kann auch ohne Änderungen, wird dies in dieser Analyse jedoch ausgeschlossen.

3.3 Analyse LSP

3.3.1 Positiv-Beispiel

Beim ersten Beispiel handelt es sich um Item und die davon abgeleitete Klasse EquipmentItem. Bei beiden dieser Klassen handelt es sich um Datenklassen, welche keine Funktionen außer den Konstruktoren haben. EquipmentItem Objekte können dadurch das die selbe felder vorhanden sind, und diese identisch genutzt werden an den selben Stellen verwendet wie Item Objekte. Dies ist in ?? zu sehen. Das bedeutet das es sich hierbei um eine Kovarianz handelt. Denn Items können nicht EquipmentItems ersetzen was für eine Invarianz Voraussetzung wäre.

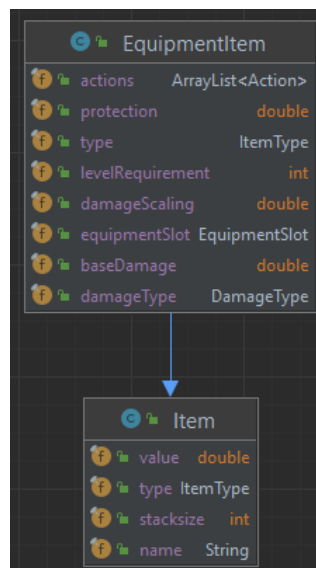


Abbildung 3.4: UML-Diagramm der Item & EquipmentItem Vererbung

3.3.2 Negativ-Beispiel

Die einzige andere Stelle an der Vererbung in diesem Projekt eine Rolle spielt ist bei den Entities. Hierbei wird von einer Abstrakten super Klasse auf zwei Unterklassen abgeleitet wie in Grafik 3.5 zu sehen ist. Da in diesem Fall LSP nicht anwendbar ist kann ich nur darauf eingehen wie der gezeigte Fall umgeformt werden kann damit es LSP erfüllen würde. LSP ist nur anwendbar wenn von beiden betroffenen Klassen Objekte ableitbar sind. Damit diese Vererbung LSP erfüllen kann muss Player und Entity verbunden werden

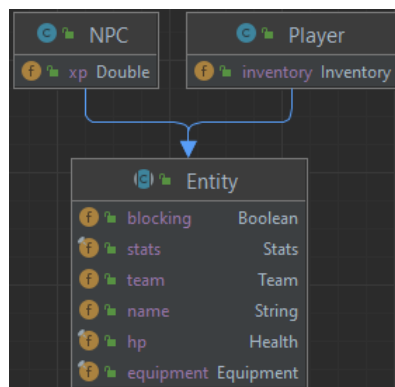


Abbildung 3.5: UML-Diagramm der Entity Vererbung

zu einer neuen Super Klasse. Eine von dieser Klasse abgeleiteten Klasse NPC erfüllt ohne Änderung derer Methoden die Kovarianz. Dies ist der Fall da Inventory in dem Fall leer erzeugt würde, und dies dennoch Funktionsfähig ist. anderstherum wäre es nicht möglich, da NPC folgende extra funktionen besitzt:

- clone() : NPC
- chooseTarget(List<Entity>):Entity
- actionChoice(): Action

Diese Methoden besitzt die Player Klasse nicht. ChooseTarget würde in einem solchen Konstrukt statt Entity Player als übergabe- und rückgabe-Typen haben.

4 Weitere Prinzipien

4.1 Analyse GRASP: Geringe Koppelung

4.1.1 Positiv-Beispiel

Im ItemManager werden alle Existierenden Items gessammelt, indexiert und suchbar Gemacht. Dies hat den Zweck den zugriff auf Items insgesamt zu vereinfachen und die erzeugung der Items zum start der Applikation zu schieben. Diese Klasse besitzt eine Geringe Koppelung durch eingeschränkten zugriff auf die jeweiligen Items. Die Items werden von außerhalb in die Klasse hineingekippt. In der Klasse selbst werden nur auf den namen und den typ des jeweiligen Items reagiert. Dies erlaubt zum einen die nutzung von abgeleiteten Klassen wie EquipmentItem im ItemManager ohne änderung. Hierbei wirkt die Item Klasse wie ein Interface an das sich gehalten wird, da es sich dabei um Domainen Logik handelt die nicht oder nur selten verändert werden muss. Die Kohäsion in der ItemManager Klasse ist auch hoch da das Wissen in der Klasse direkt von Funktionen in dem Objekt verwendet wird, und nicht an andere Objekte durch Hilfsmethoden weitergereicht werden muss.

4.1.2 Negativ-Beispiel

Durch die Verbindung und nutzungsweise der Elemente in einem Player Objekt besitzt die Player Klasse eine geringe Kohäsion. Es gibt im Player keine Methode welche auf die einzelnen unter-Objekte zugreift. Grundlegend sollten diese Elemente protected statt public sein. Dies würde bewirken das nur über Methoden im Player auf die einzelnen Variablen zugegriffen wird. Dies würde wiederum bedeuten das die Kohäsion erhöht wird aber die Koppelung weiterhin gering bleibt. Die Koppelung ist so gering in dieser Klasse, weil Healt, Inventory, Equipment, Team und Stats von ihrer Implementierung her irrelevant für den Player sind. Diese Objekte werden außerhalb des Players erzeugt und ihm übergeben. Letztlich zeigt sich diese geringe Koppelung dadurch das diese Unterelemente

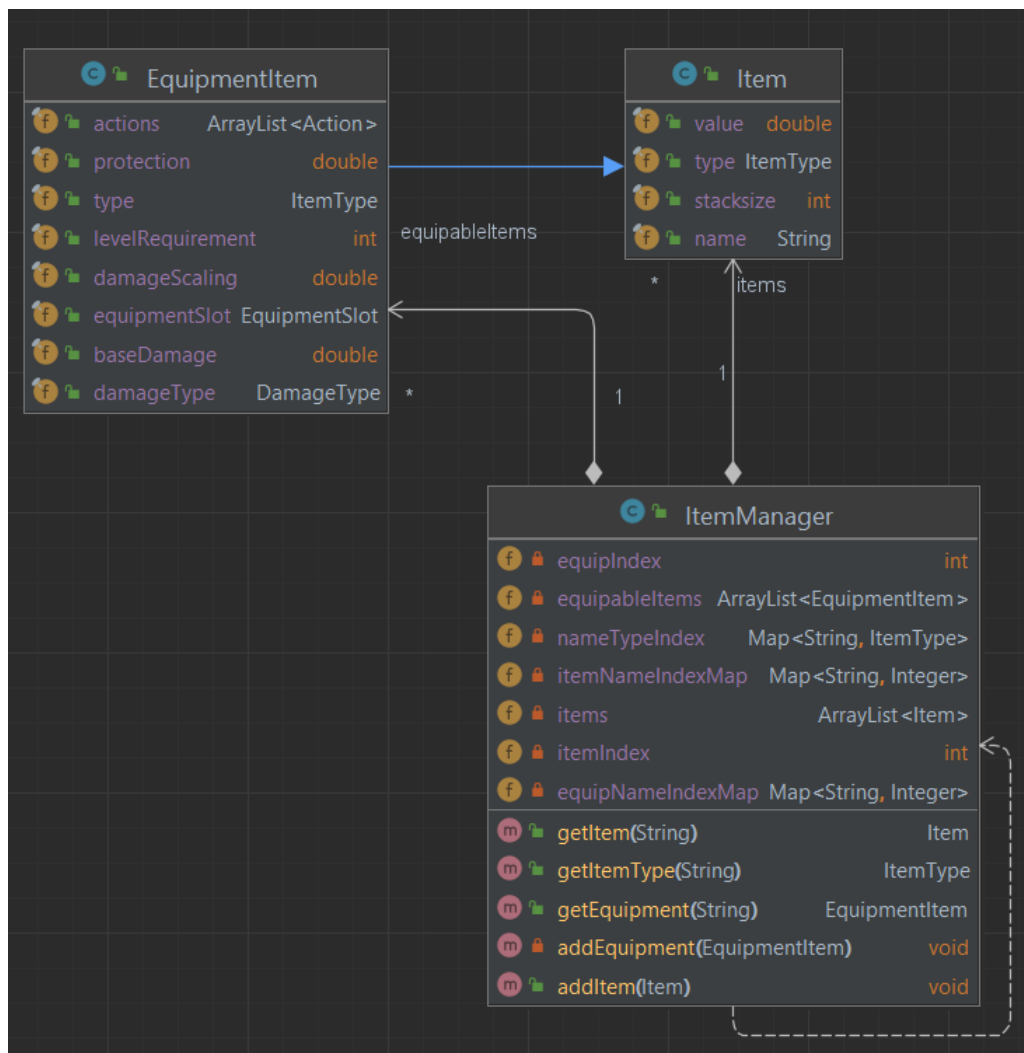


Abbildung 4.1: UML-Diagramm CreateItem

auch bei NPC verwenden lassen. Die Logik liegt in den einzelnen Elementen dem Player ist die implementierung somit 'egal'.

Der Grund warum ich dies als negativ Beispiel bringe, liegt mitunter daran das die Kapselung nicht erfüllt ist wodurch die Kohäsion der Player Klasse stark reduziert wird. Ein Ansatz die Vorteile bei dieser Architektur zu behalten und die Kapselung dennoch zu erreichen ist möglich durch die Verwendung an Interface für die einzelnen Unter Aspekte. Da ansonsten die Koppelung erhöht wird, was hier vermieden werden wollte. Es Sollten gleichzeitig Funktionen in Player/ Entity dazukommen welche die Daten nutzen die ihnen vorliegen und nicht die Logik in andere Klassen verschieben. Das Bedeutet z. B. das

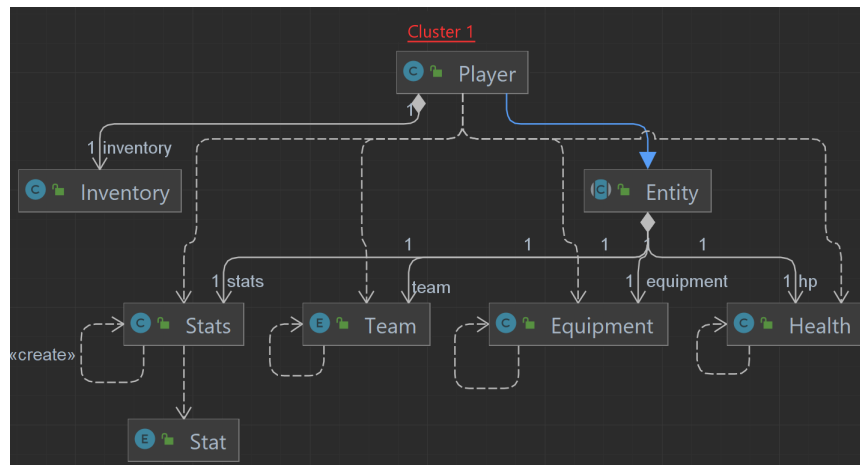


Abbildung 4.2: UML-Diagramm Player

Änderungen am Inventar oder dem Equipment über den Player passieren sollten und nicht am Player vorbei mit direkt zugriff auf diese beiden Objekte.

4.2 Analyse GRASP: Hohe Kohäsion

Als Beispiel für eine positive hohe Kohäsion existiert im Projekt die MapGraph Klasse. Diese Klasse baut einen Graphen aus einzelnen Räumen und Kanten zwischen diesen Räumen auf. Dieser Graph bedeutet eine hohe Bindung zwischen den Variablen der Klasse und den Funktionen in ihr. Der Graph ist von den einzelnen Räumen abhängig da durch diese die eigentliche Datenstruktur aufgebaut wird.

Durch Diese Klasse lässt sich eine Spielkarte als Datenstruktur aufbauen in der die einzelnen Räume direkt teil der Datenstruktur sind. Dies zeigt sich auch sehr gut im UML Diagramm 4.3 diese Komplexes.

4.3 DRY

In dem Commit "d35f0a7743581a13df1df5f5e04fd06dd1cfe9bf" wurden die add- und removeItem Methoden des Inventars vereinfacht. Zuvor gab es jeweils eine Methode in die nur das Item gegeben wurde, wodurch 1 hinzugefügt oder aus dem Inventar entfernt wurde.

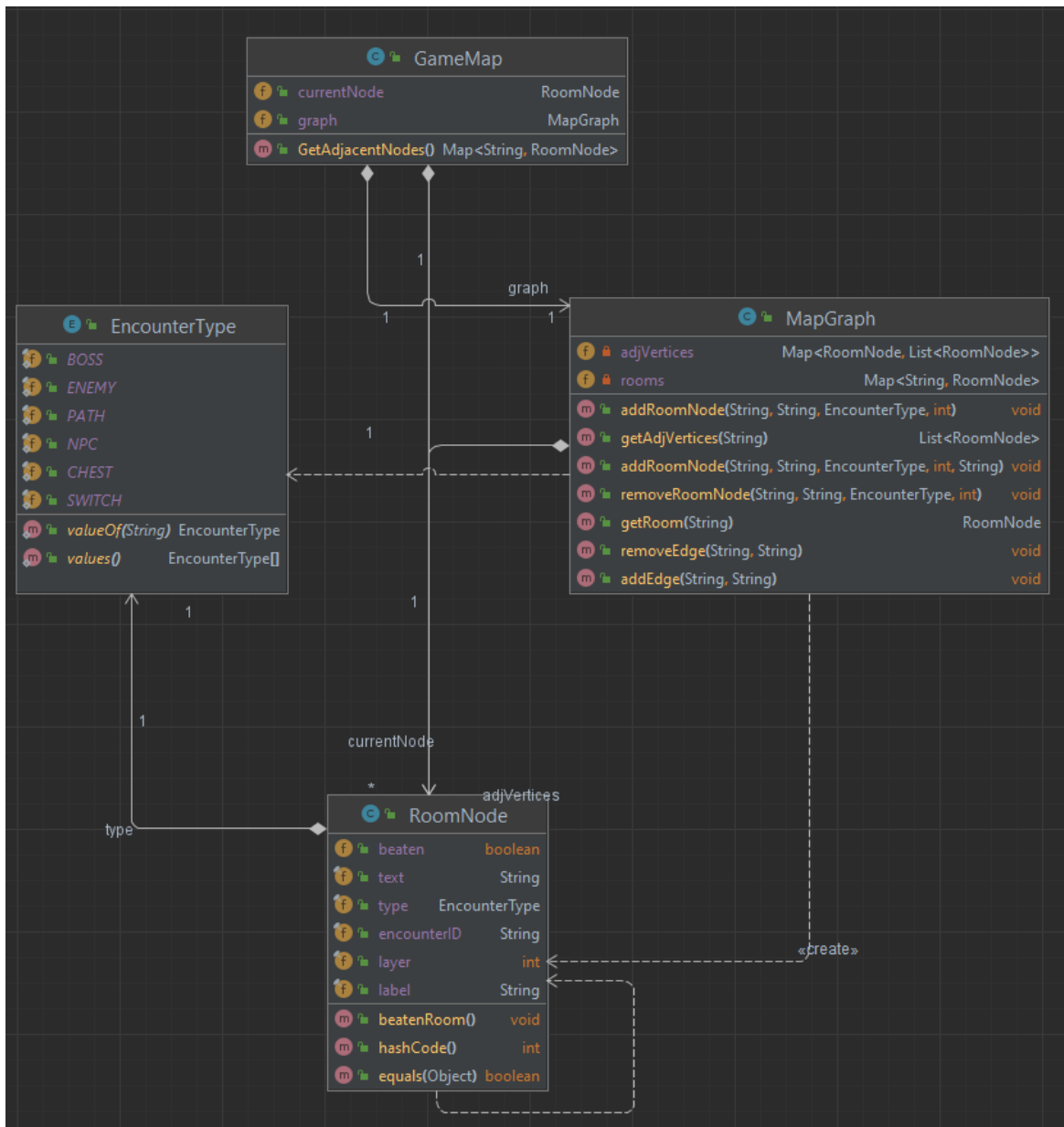


Abbildung 4.3: UML-Diagramm MapGraph

Jetzt nutzen diese Methoden die andere Overloadete Version in dem sie zusätzlich den amount 1 mitgeben.

Diese Reduktion von Code Logik verändert nichts an der Funktionsweise erhöht jedoch die Wartbarkeit da nur noch an jeweils einer stelle Code angepasst werden muss.

5 Unit Tests

5.1 10 Unit Tests

5.2 ATRIP: Automatic

Der ATRIP punkt Automatic hat zum Ziel das das Ausführen der Tests problemfrei einfach funktioniert und direkt ein konkretes Ergebnis liefert(bestanden, fehlgeschlagen). Durch diesen Punkt soll somit bewirkt werden das zum ausführen der Tests sehr wenige Anforderungen benötigt werden.

Dadurch das in dem Projekt Maven in kombination mit Junit verwendet wird lassen sich Test durch einen Knopfdruck bmit Maven ausführen. Hierbei liefern Junit Tests wenn man sich an die konventionen hält auch nur bestanden oder fehlgeschlagen zurück. Auf diese weiße ist Automatic im Projekt umgesetzt.

5.3 ATRIP: Thorough

Die Code Coverage in diesem Projekt ist viel zu gering. Nicht alle funktionsrelevanten aspekte des Codes werden durch Unit-Test abgedeckt. Dies führt dazu das die Funktionsweiße der Applikation nicht garantiert werden kann. Es werden zudem nicht die Systemkritischen Methoden getestet.

Der einzige Grund warum dies der Fall ist ist durch ein Zeitmangel der durch das vergessen eines Projektes entsteht.

5.4 ATRIP: Professional

Ein Negatives Beispiel im punkt Professional liegt im Fehlen der UNitTests, welche ein essentieller Bestandteil der Dokumentation und garnatie der funktion stellen. Als Code

Würde sich als negativ aber auch ein Test wie dieser zeigen. `@Test public void getDamageTypePerName() assert(DamageType.getByName("basic")).equals(DamageType.BASIC); Assert.assertNotEquals(DamageType.getByName("test"),DamageType.BLEEDING); @Test public void getDamageTypeByValueOf() assert(DamageType.BLEEDING).equals(DamageType.valueOf("BLEEDING"));` Dieser Code ist nicht Professional da hier ein Enum auf seine standard funktion getestet wird. Solche Test sind unprofessionel und bewirken nicht wirklich etwas.

5.5 Fakes und Mocks

...

6 Domain Driven Design

6.1 Ubiquitous Language

Bezeichnung	Bedeutung	Begründung
Ein Player hat veränderbare stats	Ein Spieler besitzt eine Sammlung an veränderbaren Statuswerte welche das aktuelle Level repräsentieren	Es Beschreibt ein Aspekt des Players mit den Begriffen wie sie in der Domain verwendet werden
Die Gamemap wird aus Roomnodes zu einem Graphen anhand einer JSON Repo gebaut	Der MapGraph wird aufgebaut indem informationen aus einer JSON Datei ausgelesen und in Räume und Kanten umgeformt wird	Der Prozess welcher in der Domain notwendig ist um die Map vorzubereiten
Items and Equipment can both be put in the Inventory, so both need the same fields	Die beiden Klassen müssen im selben inventar gespeichert werden, was bedeutet das sie beide die selbe schnittstelle benötigen	
.	.	.

6.2 Repositories

Repositorys stellen die Schnittstelle zu Datenmodellen und somit zum Speicher her. Sie Abstrahieren die vorhandene Komplexität für den rest der Domäne.

Nach dieser Definition handelt es sich bei der Initialiser und der JsonParser Klasse um Repositories. Diese stellen über einfache Methoden Möglichkeiten bereit aus dem Datenmodell (JSON) die value Objects und Entities zu generieren. Es werden hier Repositorien eingesetzt da dadurch eine bessere Trennung zwischen der Gamelogik und dem Speicher möglich ist. Dadurch lässt sich theoretisch immer noch der Speicherprozess 'einfacher' austauschen.

6.3 Aggregates

Aggregate sind eine Sammlung an Value Object und Entities. Diese werden durch jeweils eine zentrale Instanz verwaltet. Das bedeutet jeder Zugriff läuft über die Aggregate.

Aufgrund der erhöhten Komplexität und dem begrenzten Nutzen der Aggregates für die Menge an Value Objects und Entities welche in der Domain verfügbar sind wurde dieser Aspekt des DDD nicht in seiner Reinform implementiert.

Es lässt sich maximal argumentieren dass die Manager Klassen Aggregates sein können, da diese jedoch nicht zwischen jeder Interaktion von z. B. Items und dem Controller als Zwischenschicht dienen ist dies grundlegend eine inkorrekte Annahme.

6.4 Entities

Eine Entity ist nach der DDD Definition ein Object mit einem Lifecycle, welches eindeutig identifizierbar in der gesamten Domain ist. Dabei können Werte der Entity verändert werden.

Im Fall dieses Projektes gibt es folgende Entity Arten.

- NPC
- Player

Diese beiden Arten von Objekten bieten sich an als Entity strukturiert zu sein, dadurch dass sie im Gamecycle sterben/gelöscht werden können. Im GameCycle dazu auftauchen können und während Kämpfen verändert wird.

Bei den beiden handelt es sich jedoch nicht um 'perfekte' Entities, da sie zwar in Teilen der Domain eindeutig identifizierbar sind, aber es für diese Aussage Ausnahmen gibt. NPCs werden für Kämpfe Deepcopied wodurch keine komplett neuen Gegner angelegt werden müssen. Hierdurch haben theoretisch mehrere Entities den selben Identifier. Dies ist vermeidbar indem beim Kopieren eine neue UUID für die Entity generieren lassen.

6.5 Value Objects

Eine Sammlung an unveränderlichen Daten, welche keine Lebenszyklus haben und somit in Form eines Objektes gespeichert werden. Hierdurch sind die Daten in einer nutzbaren Semantik.

Im Fall dieses Projektes gibt es unterschiedliche arten von Value Objects. Folgend ist eine Liste der relevantesten.

- Item
- EquipmentItem
- Action (as seen in 6.1)
- Encounter

RoomNodes sind fast Valueobjects, der Grund warum sie nicht als reine value Objekts angelegt sind hängt mit dem einen veränderbaren Wert eines Raumes zusammen, 'beaten'. Wenn ein Raum abgeschlossen wird wird das Objekt geändert. Großteils bietet es sich an

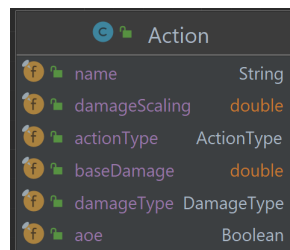


Abbildung 6.1: UML-Diagramm Action Value Object

diese Daten in form eines Value Objects zu haben da diese Daten durch andere Prozesse in der Domain auslesbar sein müssen und gleichzeitig egal wo sie verwendet werden nicht neuerzeugt werden müssen. Hierdurch ist es möglich jeweils eine globale liste der Objekte zu führen die einmal erzeugt wurde.

7 Refactoring

7.1 Code Smells

7.1.1 Long Method

`Gui.createAndShowGUI()` //ca. 120 Zeilen Code //creating the text output area //creating input area //Battle log //status area Player //Display the window Um eine solche lange Methode zu refactorn ist der erste schritt die aufteilung des Codes in mehrere einzelne Methoden. Diese können anschließend entweder in eine separate Klasse ausgelagert werden oder in der ursprüngliche Klasse gelassen werden.

Wenn nach diesem ersten Schritt auffällt das die neuen Methode thematisch wieder aufteilbar sind wird der schritt wiederholt.

7.1.2 Duplicated Code

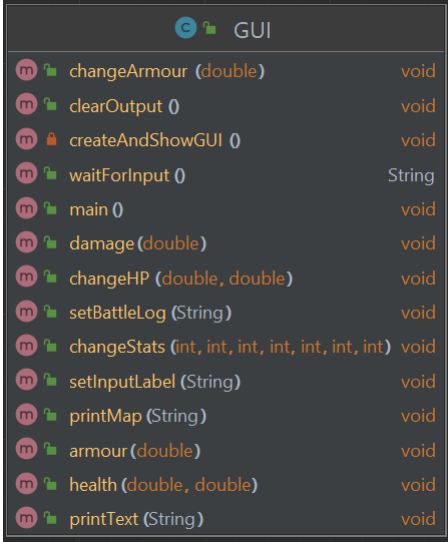
Mehrfach vorkommener Code in einem Programm ist ein risiko auf die Dauer, da bei veränderungen an einer stelle die gleichen stellen auch verändert werden sollten. Anderenfalls gehen die Code stellen immer weiter auseinander, bis die funktionsweiße nicht mehr garantierbar ist. Um dies zu verhindern sollen die betroffenen Stellen in eine separate Methode/Klasse ausgelagert werden. Dies kann nach einem ähnlichen Prinzip wie bei dem Long Method Smell gemacht werden. Heißt es wird zuerst geschaut welche funktion der Code erfüllt und welche teile man direkt in eine andere Methode machen kann. Nachdem das erledigt ist muss garantiert werden dass an allen stellen wo nun die neu entstanden Methoden verwendet werden die Ursprüngliche Funktionsweiße immer noch identisch ist. Ein Beispiel für ein solchen Refactor kann so aussehen: zuvor: `public void doSmth(String smth) if(smth.equals(this.smthelse)))` complexe abfragen... `this.smth=smth;` `public void doSmth(String smth,String smthelse) if(smth.equals(smthelse)))` selbe complexe abfragen... `this.smth=smth;` Wie in diesem Beispiel zusehen ist werden hier zweimal die selbe abfrage gemacht nur mit unterschiedlichen eingabe parameteren. Dies kann z. B.

konsolidiert werden wie folgt. `public void doSmth(String smth) doSmth(smth,this.smt-helse) public void doSmth(String smth,String smthelse) if(smth.equals(smthelse))) selbe complexe abfragen... this.smth=smth;`

7.2 2 Refactorings

7.2.1 Method Extraction

Bei Code Smells wie Long Methods können aus der lange methode sehr viele kleine gemacht werden. In 7.1 sieht man den Vergleich zu 7.2. Hier kann man sehen wie viele



Method Name	Return Type
changeArmour (double)	void
clearOutput ()	void
createAndShowGUI ()	void
waitForInput ()	String
main ()	void
damage(double)	void
changeHP (double, double)	void
setBattleLog (String)	void
changeStats (int, int, int, int, int, int, int)	void
setInputLabel (String)	void
printMap (String)	void
armour (double)	void
health (double, double)	void
printText (String)	void

Abbildung 7.1: UML-Diagramm bevor long method refactor

Methoden aus einer solchen lange Methode gemacht werden kann. Der Code hierzu ist in dem Commit "668420970c7c32ef2e12725800d78ce416e7eb53" in der GUI Klasse sehen.

7.2.2 Rename Method

Wie in der Grafik 7.2 zusehen ist gibt es hier auch drei Methoden welche anhand ihres namens nicht deutlich machen was sie tun, somit können diese wie folgend angepasst werden ohne etwas an der Funktionsweise zu ändern. Wie in Grafik 7.3 zu sehen ist sind

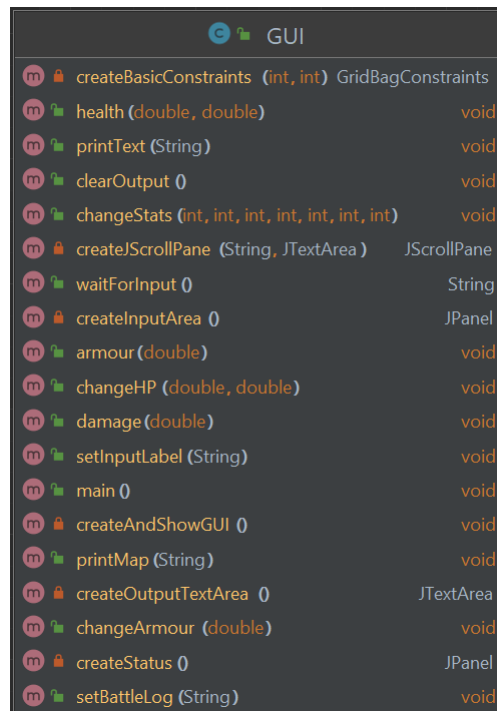


Abbildung 7.2: UML-Diagramm nach long method refactor

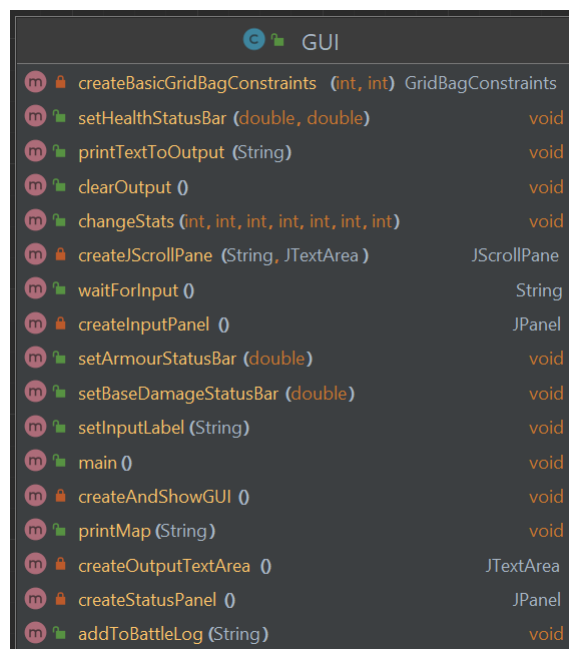


Abbildung 7.3: UML-Diagramm nach Method Renaming Refactor

die namen mit einzelnen ausnahmen sehr viel besser lesbar. Dieser Change wurde in Commit "8e8be45dafa0e0b8f77e413e305518c490cee266" gemacht.

8 Entwurfsmuster

8.1 Entwurfsmuster: Konstruktor/Erbauer

Aufgrund der großen Menge an NPC die dieses Programm nutzen können soll und die weiß wie diese in der Datenstruktur abgelegt sind, ist es möglich sehr unterschiedliche Objekte zu generieren. Um dieser Aufgabe gerecht zu werden ohne 6+ Konstruktoren mit eigener Logik in die NPC Klasse zu programmieren bietet sich ein Konstruktor Pattern an. Dieses erlaubt es anhand von 'Optionalen' und notwendigen Methodenaufrufen das passende NPC Objekt zu erstellen. Wie in Grafik ?? zu sehen ist lassen sich sehr komplexe Objekte durch dieses Pattern bauen. Jeder der Privaten Attribute im CreateNPC ist ein durch eine Methode setzbare variable.



Abbildung 8.1: UML-Diagramm Zu dem NPC Constructor Pattern