Ludwig-Maximilians-Universität München
Fakultät für Mathematik, Informatik und Statistik
Department Institut für Informatik

Entwicklung eines neuartigen Tower Defense Spieles

vorgelegt von: Benjamin Knorr
MatrikelNr: 10967090

Ausarbeitung
Praktisches Programmieren

Betreuende Dozentin:
Paola Maneggia

Schopenhauerstraße 66, 80807 München
Knorr.Benjamin@campus.lmu.de
München, den 10. Juli 2018

Inhaltsverzeichnis

1 Projektidee

1.1 Bisherige Tower Defense Spiele

Das Tower Defense Genre (Abk. *TD*) bietet eine große Bandbreite an verschiedenen Umsetzungen. Das Grundkonzept ist dabei immer, dass der Spieler mehrere Wellen an KI-Gegnern, sogenannte „Creeps“, auf einer Karte davon abhalten muss, bestimmte Ziele zu erreichen. Dazu kann er Türme platzieren, die die Creeps töten. In den meisten Fällen handelt es sich um eher kleinere Spiele, wie Browser-, Flash- oder Mobile Games. Auch die ersten bekannteren TD-Spiele sind als Custom Maps im Warcraft oder StarCraft MapEditor entstanden und haben daher kein großes Entwicklerteam im Hintergrund. Der Umfang war damit für das eigene Projekt gut abschätzbar und der vorgesehenen Bearbeitungszeit angemessen. Vereinzelt wurde das Spielprinzip aber auch in größeren Produktionen, wie Sanctum, aufgegriffen, was die Erweiterbarkeit des Ansatzes zeigt. Es existieren bereits einige Spielvariationen, über die im Folgenden exemplarisch ein Überblick gegeben werden soll.

Klassisches Spiel

In der häufigsten (da einfachsten) Spielimplementierung bewegen sich die Creeps auf einem vordefinierten Pfad und die Türme werden entlang des Pfades frei oder auf bestimmten Knotenpunkten positioniert. Typischer Weise gibt es unterschiedliche Türme und verschiedene Creep-Arten. Zum Teil haben diese auch Spezialfähigkeiten, wie fliegende Creeps (folgen also nicht dem Pfad, sondern bewegen sich auf direkter Linie zum Ziel) bzw. Türme, die besonders gut gegen Bodeneinheiten oder gegen fliegende Einheiten sind. Häufig werden die Creeps auch in ihrer Geschwindigkeit variiert und es existieren Türme, die Creeps verlangsamen sowie Creeps die dagegen immun sind. Diese Spiele rücken besonders das taktisch geschickte Kombinieren der verschiedenen Türme und das Aufrüsten an entscheidenden Knotenpunkten in den Fokus des Spielerlebnisses.

Mazing Tower Defense

Eine weitere Variante sind die „Mazing Tower Defense Spiele“, in denen die Creeps sich nicht auf einem vordefinierten Pfad bewegen, sondern der Spieler mit den Türmen eine labyrinthartige Struktur baut, durch die die Creeps dann durchlaufen. Das erste Spiel dieser Art und daher prototypisch ist „Desktop Tower Defense“ aus dem Jahr 2007 von Paul Preece, das bereits in den ersten Monaten mehrere Millionen Mal gespielt wurde und inzwischen durch eine Erneuerung als Facebook Anwendung noch populärer

Abbildung 1: Karte eines klassischen Tower Defense Spiels ohne Änderungsmöglichkeit durch Spieler – Hier Bloons Tower Defense (Bildquelle: `http://bloons.wikia.com/wiki/File:Bloons_Tower_Defense_3_Track_1.png`)

geworden ist. Das Spielprinzip erweitert das normale Spiel um eine strategische Planung der Kreaturenführung durch das Labyrinth um die Türme möglichst effektiv aufrüsten zu können. Die Spielelemente sind dabei weiterhin die klassischen Kreaturenarten und Turmvarianten.

Abbildung 2: Karte eines typischen Aufbaus bei einem Mazing Tower Defense Spiels – hier Desktop Tower Defense. (Bildquelle: `https://www.ranker.com/list/best-video-games-to-secretly-play-at-work/collin-flatt`)

Andere Varianten

Es existieren noch viele weitere Varianten wie das sehr erfolgreiche Plants vs Zombies: Bei dem greifen die Creeps (=Zombies) die Türme an und auch wenn keine Wegfindung implementiert ist, bringt das Spiel dafür eine sehr große Vielfalt an verschiedenen Gegnern und Strategien ein. Eine weitere Variante sind Multiplayer Spiele, bei denen entweder in Kooperation verteidigt wird, oder auch ein Spieler die übliche Verteidigungsrolle übernimmt und ein anderer die Kreaturen sendet. Diese Spiele waren

jedoch keine Inspirationsquelle für das eigene Projekt und werden daher nicht genauer betrachtet.

1.2 Neuer Ansatz im eigenen Projekt

Die existierenden Mazing Tower Defense Spiele erweiterten das klassische Spiel um die Herausforderung, ein möglichst effektives Labyrinth zu bauen. Dabei ist jedoch die Wegsuche der Creeps optimal, wodurch zwar Labyrinthe (lange verschlungene Wege) aber keine Irrgärten (mit Sackgassen) entstehen. Denn Verzweigungen und Sackgassen nehmen nur unnötigen Platz weg und die Creeps werden nie in diese Bereiche hineinlaufen. Die Spielidee für dieses Projekt ist, dass die Anlage eines kompliziert gebauten Irrgartens dem Spieler einen Vorteil für das Spiel bringen soll. Um dies zu ermöglichen, wurde die Prämisse aus dem Spiel genommen, dass die Kreaturen einen allwissenden Überblick über das Labyrinth haben. Stattdessen sollten sie das Labyrinth erkunden müssen. Diese Variation hatte viele Designentscheidungen zur Folge, die im Folgenden kurz vorgestellt werden.

Abbildung 3: Beispiel für ein Irrgarten-Setup bei dem eigenen Projekt.

State der Kreaturen

In den bisherigen Spielvarianten benötigten die Kreaturen keine eigene Repräsentation des Spielfeldes. Dagegen müssen die Creeps sich nun individuell auf Grundlage ihres bisherigen Wissens über die Spielwelt bewegen. Ein wichtiges Spielelement ist daher die Klasse `VisitedMap`, mit der jede Kreatur in einem zweidimensionalen Enum-Array dieses Wissen speichert. Andererseits sollen die Kreaturen auch über eine gewisse Intelligenz verfügen und nicht alle in die selbe Sackgasse laufen. Das gelingt durch simulierte Besprechungen zwischen den Kreaturen. Die Karten müssen also effizient synchronisierbar sein (siehe 4.1.3).

Wegfindung

Die effiziente Wegfindung ist ebenfalls schwieriger: Während bei normalen TD-Spielen gar keine Berechnung notwendig ist und bei Mazing TD ein A*-Algorithmus effizient den kürzesten Weg von allen Kartenpunkten zum Ziel berechnen kann, muss der Algorithmus hier noch weiter abgewandelt werden, da die Creeps ihr Ziel nicht kennen. Es muss daher für jeden Creep ein eigenes Ziel ausgewählt werden, was zum Beispiel durch Breitensuchen umgesetzt werden kann.

Variationsmöglichkeiten

Dadurch, dass die Kreaturen ihre eigene Wegfindung haben und Absprachen treffen, ergeben sich neue Möglichkeiten der Variation der Creeps. Beispielsweise wurden Figuren implementiert, die sich nur zufällig bewegen, andere versuchen jedes Feld der Spielwelt zu besuchen. Weitere Möglichkeiten wären Kreaturen, die tatsächlich sehen können und damit z.B. größere Flächen direkt durchqueren, oder Kreaturen die andere dirigieren. Auch neue Turmarten, die auf diese Wegfindung Einfluss haben, sind möglich. Als Beispiel wurde dafür ein Amnesie-Turm programmiert, der bei abgeschossenen Creeps bewirkt, dass sie sich eine Zeit lang zufällig bewegen.

Labyrinthänderungen

Die für das Genre typischen Echtzeit-Änderungen des Labyrinths sind in der Variante schwieriger an Kreaturen zu kommunizieren. Da die Creeps den Zustand ihrer erkundeten Spielwelt speichern, sind Änderungen in bereits erkundeten Bereichen problematisch. Es gibt mehrere Möglichkeiten damit umzugehen, die in der Entwicklung abgewogen und mit User-Testings betrachtet wurden:

- Kreaturen erhalten kein Update über die Spielwelt. Bei dieser Lösung musste ein Verhalten implementiert werden, falls Creeps denken es gäbe keinen Ausweg. Typischerweise greifen eingesperrte Figuren die Türme an oder rennen darüber, was in der eigenen Variante so implementiert wurde. Das User Feedback zeigte, dass das Verhalten der Creeps insbesondere wegen der Kommunikation nicht nachvollziehbar ist.
- Kreaturen erhalten ein Update, falls sie den Bereich bereits erkundet hatten. Dies führte in User-Testings zu Strategien gegen das Spielprinzip, bei denen zwei Türme auf verschiedenen Seiten der Karte immer abwechselnd neu gebaut und später wieder abgerissen wurden, sodass die Kreaturen immer von einem Ende zum anderen laufen mussten.

- Türme können gar nicht abgerissen werden, oder nur, wenn kein lebender Creep diesen erkundet hat. Dies schränkt die Möglichkeiten der Spieler zwar ein, ist aber besser verständlich und bleibt dem Spielprinzip treu.

1.3 Reflexion der Projektdefinition

Neben der oben ausführlich beschriebenen Projektidee formulierte die Projektdefinition bereits Details zur Umsetzung des fertigen Spiels aus User-Sicht und einige Mindestanforderungen sowie mögliche Erweiterungen. Im Folgenden soll diese Beschreibung noch einmal wiedergegeben werden, da sie ähnlich einem Lastenheft den Leitfaden für das Projekt bot.

> Die Spielansicht zeigt die Leben und das Geld des Spielers, eine Zeitlinie der nächsten Gegnergruppen, sowie die Labyrinth-Ebene. In Letzterer kann der Spieler Mauern bauen und dann diese mit Türmen aufrüsten. Es stehen verschiedene Türme zur Verfügung, die zudem während des Spiels durch Upgrades verbessert werden können.

Diese Anforderung wurde exakt wie gefordert umgesetzt

> In mehreren Wellen werden entsprechend der Zeitlinie Gegner generiert, die mit unterschiedlichen Strategien versuchen, den Ausgang des Labyrinths zu erreichen. Sie kommunizieren untereinander, vermeiden also z.B. Sackgassen, die andere bereits erkundigt haben. Die Einheiten ändern somit während dem Spiel dynamisch ihre Wegfindung.

Auch diese Anforderung ist umgesetzt worden und wird in den nächsten Kapiteln noch genauer beschrieben.

> Mindestanforderungen:
>
> 1. Desktop-Anwendung (erfüllt)
> 2. Grundfunktionalitäten eines Mazing-TD-Spiels (Leben, Geld, Mazing-Funktionalitäten, Pfadsuche der Computergegner) (erfüllt)
> 3. Verschiedene Türme mit mehreren Upgradestufen (erfüllt)
> 4. Verschiedene Gegnerarten (erfüllt)
> 5. Kommunikation der Gegner untereinander bei Sackgassen (erfüllt)
> 6. Rudimentäre Grafikgestaltung in JavaFX (Top Down Perspektive, einfaches Design) (übertroffen)

Neben den Mindestanforderungen, die ein spielbares und interessantes Spiel beschrieben, wurden noch einige mögliche Erweiterungen beschrieben, die nur zum Teil umgesetzt wurden. Dafür wurden bei der Entwicklung auf andere Sachen größeren Wert gelegt.

Erweiterungen

1. Kampagnenmodus (Steigender Schwierigkeitsgrad, Story) (nicht umgesetzt)
2. Tutorial für Spielsteuerung (nicht umgesetzt)
3. Mehrere Karten (Andere Szenarie, mehrere Zugänge, ...) (nicht umgesetzt)
4. Kompliziertere Strategien der Gegner (z.B. Auf Mauern klettern und andere Einheiten dirigieren) (z.T. umgesetzt)
5. Strategiesteuerung für Türme (Angreifen des schwächsten/stärksten/-nächsten/... Gegners) (z.T. umgesetzt)
6. Anspruchsvollere Grafik (z.B. isometrische Perspektive) (nicht umgesetzt)
7. Android-Anwendung (erfüllt)
8. Mehrspielermodus (Große Karte, in der jeder Spieler einen Teil ausbaut) (nicht umgesetzt)

Die Android-Anwendung wurde auf Kompatibilität bis Android 4.2.2 (Jelly Bean) getestet. Dafür waren einige Laufzeitoptimierungen notwendig, die systematisch gemessen und implementiert wurde, wie in den folgenden Kapiteln auch beschrieben wird. Neben diesem Feature wurde in das Projekt eine (De-) Serialisierung mit JSON eingebaut, sodass das aktuelle Spiel abgespeichert und neu geladen werden kann. Ein weiteres nicht genanntes Element bei der Entwicklung, das nicht in der Projektdefinition erwähnt war, ist das ausführliche UnitTesting, das in Kapitel 2.3 genauer beschrieben wird.

2 Entwicklungsprozess

Bei dem Entwicklungsprozess wurde auf mehrere professionelle Elemente besonderen Wert gelegt. Diese sollen im Folgenden kurz erläutert werden.

2.1 Werkzeuge

Versionsverwaltung

Eine der grundlegendsten Dinge bei der Softwareentwicklung ist ein Versionskontrollsystem. In diesem Projekt wurde dafür *Git* verwendet und in einem Repository auf Github (`https://github.com/Knorrke/MazeRunner`) veröffentlicht. Dies ermöglichte zum einen unkompliziert das Arbeiten auf mehreren Geräten. Zum anderen konnte durch das Erstellen neuer Branches seperat an Features entwickelt werden, ohne die Hauptversion während dem Entwicklungsprozess zu verändern. Über Pull Requests wurden die Features dann in den Hauptzweig gemergt.

Dependency Management

Um nicht jedes Mal das Rad neu zu erfinden, ist es oft sinnvoll, existierende Bibliotheken zu verwenden. Damit das Projekt leicht auf anderen PCs eingerichtet werden kann, sollte das Einbinden von solchen Abhängigkeiten durch ein Verwaltungssystem geschehen. Dafür wurde *Maven* verwendet, das ebenfalls eine Buildautomatisierung ermöglichte.

Continuous Integration

Mit einem Continuous Integration Service (in diesem Fall *Travis CI*) konnte sichergestellt werden, dass die Tests nicht nur auf dem eigenen Gerät funktionieren, sowie dass Pull Requests vollständig funktionsfähig sind, bevor sie in den Hauptzweig eingebunden werden. Des Weiteren ließ sich dadurch das Testcoverage Tool *JaCoCo* integrieren (Genaueres siehe 2.3).

2.2 Agile Softwareentwicklung

Die Entwicklung und Planung der Software lief agil ab. Es wurde also in kleineren Etappen (Sprints) gearbeitet, die mit einer Auswahl der wichtigsten Issues aus Kundensicht begannen und an deren Ende immer funktionsfähige Produkte standen. Wichtiger Bestandteil davon war das Issue-System von Github, sowie die Zuordnung zu Meilensteinen und Releases für die Zwischenversionen. Eine Ausbauung hätte das Kanban-Board (Github Projects) ermöglicht, in dem die Issues gefiltert und priorisiert in Spalten abgelegt werden können. Das hätte noch mehr Ähnlichkeit zum Sprint Backlog gehabt, der jetzt nur unpriorisiert in Meilensteinen umgesetzt wurde. Die inkrementellen Produkterweiterungen ermöglichten frühzeitige Usertests, die auch die Priorisierung für die weitere Entwicklung festlegten.

2.3 Testgetriebene Entwicklung

Für die Gewährleistung der Funktionalität und Absicherung für Änderungen sind Unit-Tests enorm hilfreich. Sie verkürzen die Zeit zur Fehlersuche, da Fehler direkt mit einem Knopfdruck angezeigt werden können. Außerdem geben sie dem Entwickler die Sicherheit bei größeren Strukturänderungen, dass das System danach noch die Anforderungen erfüllt. Das Projekt wurde komplett testgetrieben entwickelt, was bedeutet, dass nicht mit der Implementierung begonnen wird und dann ein Test dazu geschrieben wird, sondern umgekehrt: Zuerst definiert ein Test, wie sich der Code verhalten soll, anschließend wird dieser Test so einfach wie möglich erfüllt. Dadurch wird sichergestellt, dass nur Code geschrieben wird, der auch wirklich notwendig ist. In einer dritten Phase, dem Refactoring, wird der Code anschließend nochmal bzgl Code-Qualität überarbeitet. Dabei hilft der bereits definierte Test zu überprüfen, ob die Funktionalität nach dem Refactoring immer noch gegeben ist.

Ein Testcoverage Tool ermöglicht anschließend zu überprüfen, ob auch alle Teile des Codes durch Tests abgedeckt sind. In diesem Projekt liegt die Testabdeckung zur Zeit der Abgabe bei 98%. Die restlichen 2% betreffen überwiegend das Loggen von evtl. auftretenden `IOExceptions`, die nicht extra für Tests erzeugt wurden.

Um die Tests eines Codeteiles besser von dem Verhalten des restlichen Codes trennen zu können, wurde zudem das Mocking Framework *Mockito* verwendet. Das ermöglicht es, für Klassen und Interfaces eine Art Attrappe zu erzeugen, für die dann konfiguriert werden kann, wie er auf bestimmte Methodenaufrufe mit den korrekten Argumenten reagiert. Zum Einen kann dadurch der zu testende Code besser isoliert werden, weil keine Abhängigkeiten zu anderen Klassen existieren. Es kann aber auch überprüft werden, ob die Aufrufe richtig stattgefunden haben, ob also die Interaktion mit dem Mock-Objekt so ablief, wie erwartet. Dadurch lässt sich auch die Integration in den restlichen Softwareaufbau gut testen.

```
1  public class MazeTest {
2    MazeModelInterface maze;
3    PlayerModelInterface playerMock;
4    int x, y;
5
6     @Before
7     public void setup() {
8        playerMock = Mockito.mock(Player.class);
9        maze = new Maze();
10       maze.setPlayerModel(playerMock);
11       x = 2;
12       y = 3;
13    }
```

```
14
15    @Test
16    public void sellShouldEarnMoneyTest() {
17       ObservableList<Wall> walls = maze.getWalls();
18       Wall wall = maze.buildWall(x, y);
19       assertTrue("maze contains built wall", walls.contains(wall));
20       maze.sell(wall);
21       assertFalse("Maze shouldn't contain the wall after sell",
             walls.contains(wall));
22       Mockito.verify(playerMock,
             Mockito.times(1)).earnMoney(wall.getCosts());
23    }
24 }
```

Listing 1: Testbeispiel mit Mockito. Das Maze verwendet ein Mock Objekt von `PlayerModelInterface` (Zeile 8) und testet, ob das Verkaufen einer Mauer die Mauer wirklich entfernt (Zeile 21) und dem Spieler den richtigen Geldwert gibt (Zeile 22).

2.4 JavaFX

Für die Umsetzung der Benutzeroberfläche wurde JavaFX verwendet, das durch die Abwandlung des Observer-Patterns auch viel Einfluss auf den Softwareentwurf hatte (siehe Kapitel 3.2). Das Framework ermöglichte eine modern wirkende Oberfläche durch viele vorgefertigte Komponenten und zahlreiche verfügbare Erweiterungen, wie z. B. ein Popup-Menü. Dabei wird die Komplexität von Swing für ansprechende Layouts deutlich reduziert und das meiste geschieht für den Programmierer unsichtbar im Hintergrund. Zudem existiert für JavaFX Anwendungen das Projekt *javafxports*, das ermöglicht, den gesamten Code ohne große Anpassungen direkt in Mobile Apps zu packen.

Bei der testgetriebenen Entwicklung ist die View-Programmierung meistens eine Herausforderung, da sie schwer Unit-Testbar ist. Hierfür konnte jedoch mit *TestFX* (`https://github.com/TestFX/TestFX`) ein gutes Framework verwendet werden, das eine schnelle und anschauliche Implementierung von Unit-Tests für JavaFX ermöglicht. Dadurch konnte auch die korrekte Reaktion auf (erwartete) Userinteraktionen überprüft werden.

```
1 @Test
2 public void buildWallOnClick() {
3   clickOn("#maze");
4   assertEquals("There should be a wall now", 1, maze.getWalls().size());
```

```
5   verifyThat("#maze", NodeMatchers.hasChildren(1, ".wall"), collectInfos());
6   verifyThat(".wall", NodeMatchers.isVisible(), collectInfos());
7   verifyThat("#maze .wall", NodeMatchers.hasChildren(1, ".wall-image"),
        collectInfos());
8   // second click: on already existing wall
9   clickOn(MouseButton.PRIMARY);
10  assertEquals("There should still be only one wall", 1,
        maze.getWalls().size());
11  verifyThat("#maze", NodeMatchers.hasChildren(1, ".wall"), collectInfos());
12
13  // third click: somewhere else
14  moveBy(200,0);
15  clickOn(MouseButton.PRIMARY);
16  assertEquals("There should be two walls now", 2, maze.getWalls().size());
17  verifyThat("#maze", NodeMatchers.hasChildren(2, ".wall"), collectInfos());
18 }
```

Listing 2: Beispielcode von TestFX. Das Setup wurde aus Platzgründen nicht aufgeführt, ist aber im Code in der Abstrakten Oberklasse aller ViewTests `AbstractViewTest` zu finden. TestFX ermöglicht die Simulation von Mausklicks (Zeile 3, 9 und 15), Bewegungen (Zeile 14) und ermöglicht Überprüfungen der Auswirkung (z.B Zeilen 5-7). Die Methode collectInfos() sammelt bei Fehlschlagen der Überprüfung bisherige Events und erstellt einen Screenshot der Situation für leichtere Problembehandlung.

Die Tests konnten dabei mit dem von *TestFX* angebotenen *Monocle* Build und geeigneten Systemvariablen in einem „headless“ Zustand ausgeführt werden. Das heißt, dass die Tests nicht tatsächlich die Maus des ausführenden Nutzers bewegen, sondern in einem unsichtbaren Fenster im Hintegrund ausgeführt werden.[1]

3 Softwaredesign

Um den Code leicht erweitern zu können und Fehlern vorzubeugen, ist es wichtig, den Aufbau der Software gut zu strukturieren. Dabei gibt es einige typische Probleme, für die bereits etablierte Lösungen existieren. Diese Design Pattern helfen nicht nur dabei, den Code wartbar zu halten, sondern helfen auch in der Kommunikation mit anderen Entwicklern bei strukturellen Problemen. In diesem Kapitel soll ein Highlevel-Überblick über die Softwarestruktur und verwendete Design Muster gegeben werden.

[1] Die Systemvariablen sind `-Dtestfx.robot=glass -Dtestfx.headless=true -Dprism.order=sw -Dprism.text=t2k -Djava.awt.headless=true -Dtestfx.setup.timeout=10000`

3.1 Model View Controller

Eines der wohl am weitesten verbreitete Entwurfsmuster ist das Model View Controller Pattern, das auch in dieser Arbeit umgesetzt wurde. Es trennt die verschiedenen Aufgaben einer Software in drei Bereiche auf: Daten und Spiellogik bilden das Modell, das in sich funktioniert und nach außen Schnittstellen bietet, die dann interne Spielprozesse auslösen. Getrennt davon ist die View, deren Aufgabe einzig darin besteht, die Daten aus dem Modell (evtl. gefiltert) dem Spieler anzuzeigen. Diese View ist mit JavaFX geschrieben, wobei die Trennung vom restlichen Code teilweise durch die Verwendung von *FXML* besonders deutlich wird. Die View muss dabei Änderungen aus dem Modell mitbekommen und daraufhin entsprechend anzeigen. Die View stellt zudem Schnittstellen zur Verfügung, um auf bestimmte Useraktivitäten zu reagieren. Diese Schnittstellen verwendet der Controller. Er setzt also Listener auf Komponenten der View für verschiedene User Interaktionen wie Mausklicks, Hovering oder Tastatureingaben. Die Listener sind größtenteils als anonyme Klassen implementiert und rufen bei einer Aktion die entsprechenden Schnittstellen des Modells auf.

3.2 Observer Pattern

Im Zusammenspiel mit MVC ist das Observer Pattern eines der wichtigsten. Es ermöglicht eine lose Kopplung zwischen View und Modell, indem sich die View beim Modell als Observer registriert und das Modell bei Änderungen alle Observer benachrichtigt. Dadurch ist gewährleistet, dass das Modell nicht explizit die View kennen muss und so kann die View leichter ausgetauscht werden oder weitere Views (z.B. auch Sounds) angebunden werden, ohne in das Modell einzugreifen. In JavaFX funktioniert die Umsetzung ein wenig anders, nämlich mittels „Bindings“: Das Modell bietet Properties an, auf denen Listener registriert werden können, die aber auch direkt an Viewausgaben gebunden werden können. Beispielsweise stellt die Geldanzeige im Spiel einen formatierten String der `IntegerProperty money` des `PlayerModel` dar. Da immer das aktuelle Geld angezeigt werden soll, kann hierfür ein Binding verwendet werden, statt den Umweg über einen Listener zu nehmen.

3.3 Game Loop

Eine Game Loop löst das Problem, die Spielzeit mit der Realzeit zu synchronisieren, statt ihren Ablauf von User-Interaktionen (wie bei klassischen EVA), abhängig zu machen oder von der Prozessorgeschwindigkeit abhängig zu sein [Nys14]. Die Implementierung richtet sich nach [Imp15b, Imp15a] und verwendet drei Elemente:

- **Synchronisation mit der Realzeit**: Messe wie viel Realzeit seit der letzten Spielzeit Simulation vergangen ist und simuliere ungefähr diese Zeitspanne (Einschränkungen siehe die nächsten zwei Punkte).
- **Feste Zeitschritte**: Eine Frame Rate über 60 Bilder pro Sekunde ist nicht notwendig, daher wird versucht diese Zielrate ungefähr zu erreichen. Kürzere Zeitschritte werden also zusammengefasst, bis 1/60 Sekunde zusammenkommen, größere Zeitschritte werden in mehrere 1/60 Sekunden Schritte zerteilt und einzeln simuliert. Dadurch bleibt das Spiel deterministisch – im Gegensatz zur Verwendung von variablen Zeitschritten.
- **Begrenzte Zeitschritte**: Ist der Prozessor zu langsam, um eine Iteration in $1/60\,s$ auszuführen, ist bei der nächsten Iteration mehr Zeit vergangen, wodurch er mehr Zeitschritte machen muss, die wieder länger dauern, etc. Um diese Spirale zu verhindern, wird die vergangene Zeit auf eine bestimmte Länge beschränkt. Die Nachteil daran ist, dass das Spiel auf zu langsamen Prozessoren langsamer läuft.

Interpolation der Zeitschritte wurde nicht eingebaut, da diese Komplexität sich in alle Komponenten fortsetzt, die von der Game Loop synchronisiert werden sollen.

Wegen JavaFX wurde die `GameLoop` als Spezialisierung des `AnimationTimer` erstellt, der automatisch durch den JavaFX-Thread aufgerufen wird. Durch das Binding konnten die Zeitupdates der View aus der Game Loop entfernt werden, da sie bei Änderungen des Modells bereits ihre internen Werte ändern und vom JavaFX-Thread aktualisiert werden.

3.4 Strategy Pattern

Das Strategy Pattern wird verwendet, um den Ablauf eines Algorithmus zur Laufzeit ändern zu können. Man erstellt Objekte, die die Implementierung des Algorithmus beinhalten und verwendet dann Komposition der Objekte. Dadurch können auch viele unterschiedliche Algorithmen für die selbe Klasse implementiert werden, ohne Unterklassen zu verwenden. Dies entspricht auch dem Prinzip der Objekt Orientierung „composition over inheritance“, durch das eine lose Kopplung erreicht werden kann. Verwendet wurde dieses Pattern bei der Wegfindung der Creeps. Die Klasse `RandomMovement` implementiert eine völlig zufällige Bewegung und Berücksichtigt nur Wände. Diese wird von den einfachsten Kreaturenarten verwendet, die nur am Anfang auftauchen. Dagegen sucht `NoSightMovement` mit einer abgewandelten Dijkstra-Suche das nächste unbekannte Feld (bezieht also bereits besuchte Felder ein) und bewegt sich zu diesem hin. Dabei werden jedoch nur Felder auf besucht gesetzt, die tatsächlich abgelaufen wurden. Eine weitere mögliche Klasse, die nicht mehr implementiert wurde,

ist `SightMovement`, die alle Felder auf besucht setzt, die von der aktuellen Position aus gesehen werden können und erst dann nach einem unbekannten Ziel sucht. Ebenso wäre eine Klasse `PerfectMovement` denkbar, die dem kürzesten Weg zum Ziel durch das Labyrinth verfolgt, wie es normalerweise in Tower Defense Spielen üblich ist.

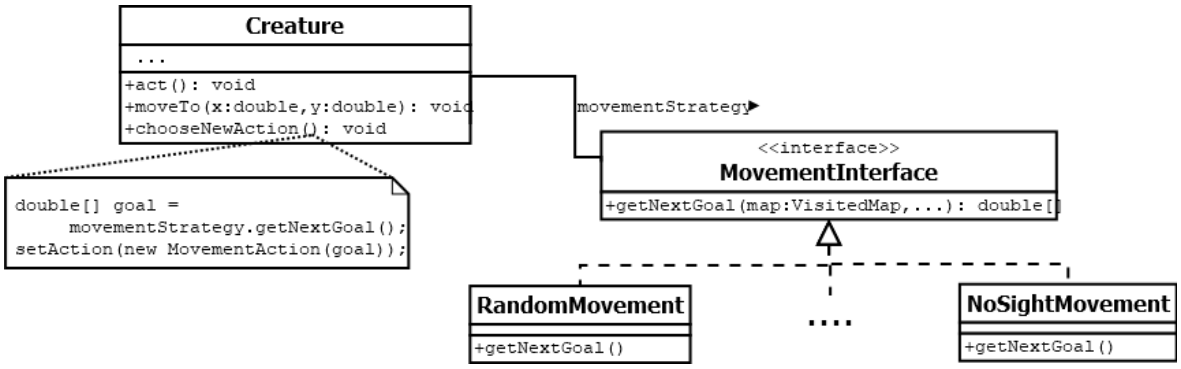

Abbildung 4: Strategy Pattern bei der Wegfindung

Dass diese Algorithmen zur Laufzeit ersetzt werden können, bietet die Möglichkeit für interessante Kreaturen-Interaktionen und Tower-Arten: Mit dem `AmnesiaTower` wurde ein Turm implementiert, der die Creeps durch seine Schüsse verwirrt. Dabei ersetzt er für eine kurze Zeit den eigentlichen Wegfindungsalgorithmus der Kreatur durch `RandomMovement`. Genauso könnte man Creeparten implementieren, die anderen beim Kommunizieren bessere Strategien beibringen, also z.B. `RandomMovement` der einfachsten Kreaturen durch `SightMovement` ersetzen. Ebenfalls angedacht sind Kommandeure, die – statt selbst durch das Labyrinth zu rennen – auf Türme klettern und von dort mit Überblick über das gesamte Labyrinth die anderen Creeps durchlotsen können, also die Bewegung durch `PerfectMovement` ersetzen.

Auch Algorithmusimplementierungen, die gar nicht darauf abzielen das Labyrinth zu durchqueren, wären möglich. Beispielsweise könnten intelligentere Kreaturen an geschützten Stellen warten oder priorisiert zu anderen Kreaturen laufen, um diesen Informationen weiterzugeben. Das Projekt bietet hier wie man sieht noch einige Erweiterungen, die bisher noch nicht implementiert werden konnten. Die Grundfunktionalität, um solche Features umzusetzen, sind allerdings mit dem Strategy Pattern gelegt und am Beispiel des Amnesie-Turms auch verwendet.

3.5 Static Factory Method

Gemeint ist hier nicht das Factory Method Pattern, sondern eine statische Methode zur Erzeugung eines Objektes einer Klasse. Im Gegensatz zum Konstruktor hat die Static Factory Method die Möglichkeit auch Objekte von Unterklassen zu erzeugen oder ein bereits erzeugtes Objekt zurückzugeben [Blo08, 5 ff.]. In diesem Sinne verwendet z.B.

auch das Singleton Pattern eine Static Factory Methode. Dieses Vorgehen wurde sowohl für Türme verwendet, bei denen der Aufruf konkrete Objekte der Unterklassen erzeugt, sowie für die Kreaturen. Bei letzteren ist die Erzeugung etwas umfangreicher und mit verschiedenen Parametern möglich und wurde zur Übersichtlichkeit daher in eine eigene Klasse `CreatureFactory` ausgelagert. Diese Factory kann dann z. B. anhand des Types und des Zeitfortschritts die Leben und den Wert der Creeps, sowie deren Bewegungsstrategie bestimmen und entsprechend den Konstruktor der `Creature` aufrufen. Dadurch wird die Erzeugung der Figuren besser gekapselt.

4 Überblick der wichtigsten Programmteile

Im Folgenden wird auf die wichtigsten Teile des Programms nochmal genauer eingegangen und einzelne relevante Algorithmen und Datenstrukturen beschrieben und reflektiert.

4.1 Model

Das Model teilt sich in drei Hauptkomponenten auf: Die Spielerverwaltung (Leben und Geld) im Paket `model.player`, die Levelverwaltung (Zeitlicher Ablauf der Wellen) im Paket `model.level` und die Spielwelt an sich im Paket `model.maze` mit Kreaturen und Türmen. Das `PlayerModelInterface` und die implementierende Klasse `Player` sind sehr einfach (eigentlich nur Resourcen erhöhen / verringern) und werden daher nicht genauer betrachtet. Auf die anderen beiden Teile wird dagegen im Folgenden noch genauer eingegangen, wobei die Spielwelt aufgrund der Komplexität noch weiter unterteilt wird. Grundlegend für beide sind Aktionen, die zunächst kurz erklärt werden.

4.1.1 Akteure und Aktionen

Alle Objekte, die im Rahmen der Spielzeit agieren, implementieren das `ActorInterface` – ein Funktionales Interface, das nur eine Methode `void act(double dt)` beinhaltet. Alle Akteure können beliebig auf das Fortschreiten der GameZeit reagieren. Häufig ist die Entscheidung, was als nächstes getan wird, jedoch nicht bei jedem Tick zu treffen, sondern kann in eine gewisse Spanne andaurnde Aktionen untergliedert werden. Die Klasse `Action` bietet eine abstrakte Implementierung eines Akteurs, der solange eine bestimmte Aktion durchführt, bis eine Abbruchbedingung erfüllt ist. Konkrete

Implementierungen davon sind `CountdownAction`, die als Abbruchbedingung das Ablaufen einer Zeitspanne hat, sowie `MoveAction`, die als Abbruchbedingung das Erreichen eines gewissen Punktes oder beweglichen Objektes hat. Diese Klassen bieten die Grundlage für alle wiederholenden Aktionen.

4.1.2 Das Paket `level`

Die Aufgabe des Level Pakets ist wie gesagt die zeitliche Steuerung der Creep-Wellen. Sie bietet nach außen für die View ein Binding, wie viel Prozent des Spiels schon vergangen sind. Nach einem gewissen Countdown erstellt sie – entsprechend der Konfiguration des aktuellen Levels – eine Liste von Kreaturen in der `CreatureFactory` und beauftragt das `MazeModelInterface` diese in das Spiel einzufügen. Anschließend wird der Countdown neu gestartet.

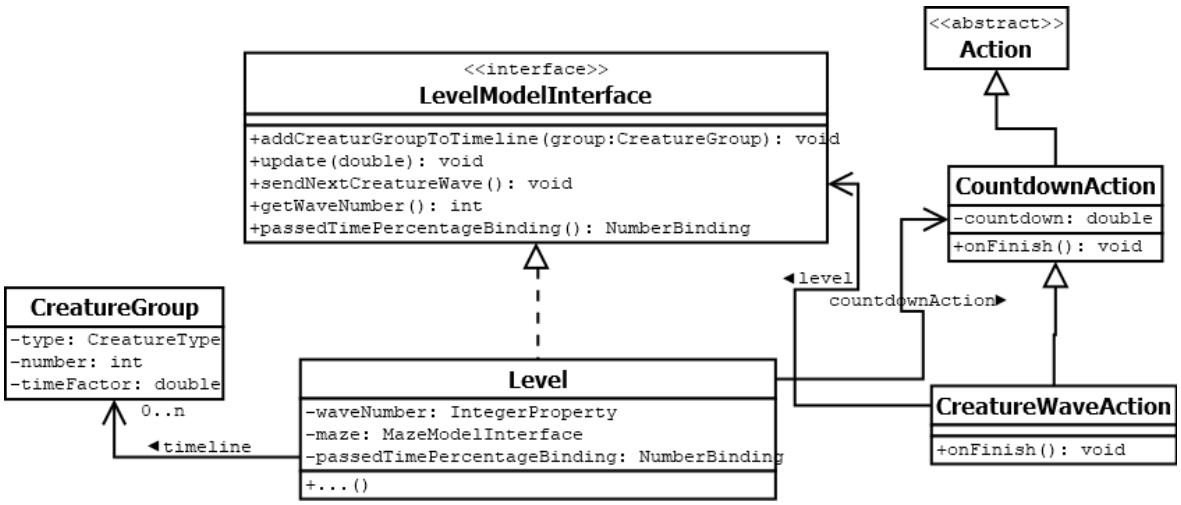

Abbildung 5: Das Klassendiagramm des `level` Packages (vereinfachte Darstellung).

4.1.3 Das Paket `maze` – Kreaturen

Vieles zu den Kreaturen wurde bereits in den vorherigen Kapiteln (insb 3.4) erklärt. Im Folgenden wird sich daher auf bisher noch nicht erwähnte Strukturen fokusiert.

Es gibt verschiedene Kreaturenarten, die in einem Enum `CreatureType` spezifiziert sind. Dieses Enum enthält auch die Default-Werte der Typen für die Leben, Geschwindigkeit, Wert und Bewegungsalgorithmus. Ausgehend von diesen Default-Werten werden die Creeps in der `CreatureFactory` mit einem bestimmten Zeitfaktor abgewandelt (Kreaturen werden mit der Zeit stärker, geben aber auch etwas mehr Geld).

```
1 public enum CreatureType {
2   DUMB(3, 10, 1, new RandomMovement()),
3   NORMAL(1, 10, 1, new NoSightMovement()),
4   TOUGH(0.8, 30, 3, new NoSightMovement());
```

```
5     //... Attribute mit Gettern
6     private CreatureType(double defaultVelocity, int defaultLifes, int
          defaultValue, MovementInterface defaultMovement) {
7      //... Zuweisung
8     }
9   }
```

Listing 3: Enum der Kreaturen Arten mit Konfiguration der default Werte (gekürzt).

Die Kreaturen sind Akteure und werden daher von der GameLoop aktualisiert. Sie besitzen eine aktuelle `Action`, die z.B. eine `MoveAction` zu einem bestimmten Ziel ist. Ist dieses Ziel erreicht, sucht die Kreatur mit dem gesetzten Bewegungsalgorithmus das nächste Ziel und erstellt dafür eine neue MoveAction. Zusätzlich wird bei jeder Positionsänderung überprüft, ob Kreaturen in der Reichweite sind, die andere Informationen über die Spielwelt haben (`VisitedMap`). Falls das der Fall ist, wird die Bewegungsaktion durch eine `CountdownAction` vorübergehend unterbrechen, um sich mit der anderen Kreatur zu unterhalten. Danach wird die Bewegung fortgesetzt.

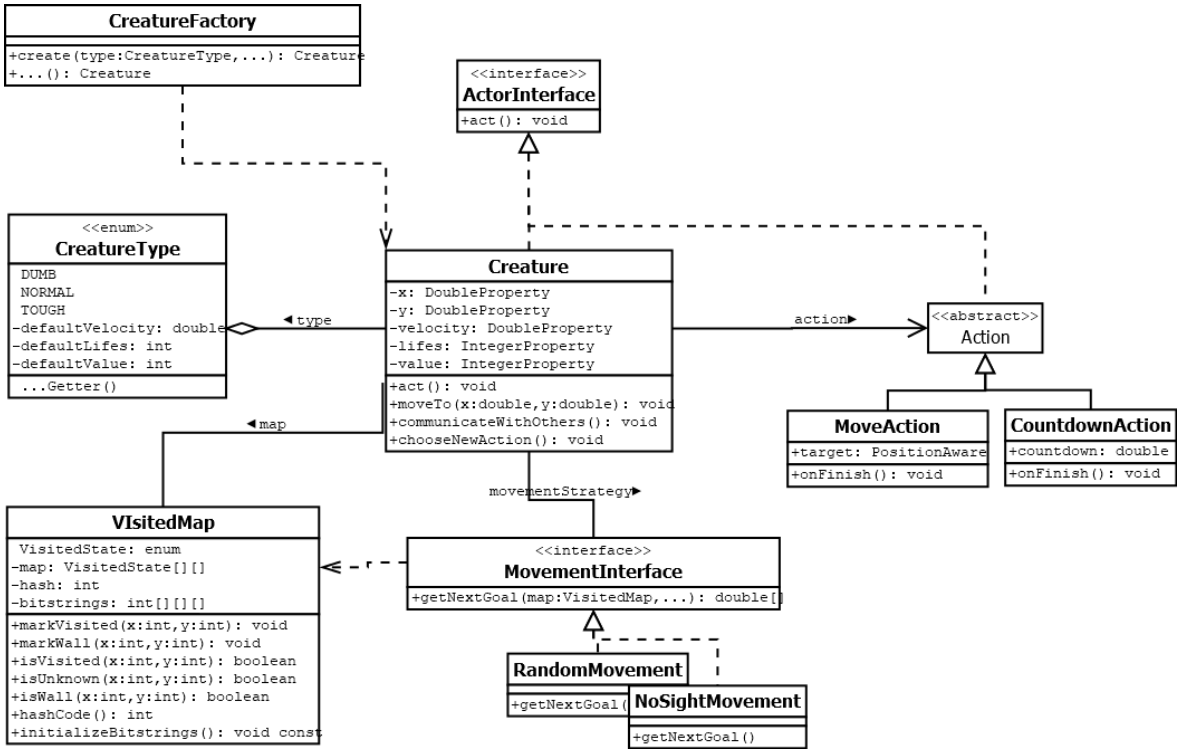

Abbildung 6: Das Klassendiagramm der Creature Klassen (vereinfachte Darstellung). Anm.: Die Darstellung der `MovementInterface` Implementierungen veranschaulicht das Strategy-Pattern.

VisitedMap

Wie bereits in Kapitel 1.2 erwähnt, ist die VisitedMap zentraler Bestandteil der scheinbaren Intelligenz der Kreaturen. Da diese sehr oft verglichen wird, war hier eine effizi-

ente Implementierung notwendig. Die Speicherkomplexität der Map liegt in $O(c \cdot n \cdot m)$, wobei c die Anzahl der Creeps, und $n \times m$ die Abmessung der Spielwelt ist (im aktuellen Spiel 10×20). Sind jedoch alle c Kreaturen nah zusammen, versucht jede Kreatur mit jeder anderen zu kommunizieren, also $c \cdot (c-1)$ und jeder Vergleich der Arrays benötigt $O(n \cdot m)$ Schritte $\Rightarrow O(c^2 \cdot n \cdot m)$. Eine Optimierung gelingt dadurch, dass für die Arrays ein Hash-Wert erstellt wird und für den Vergleich herangezogen wird. Nur wenn die Hash-Werte unterschiedlich sind, wird über das gesamte zweidimensionale Array iteriert und die Informationen zusammengefügt. Dadurch muss in dem gleichen Fall nur eine Kreatur mit allen anderen den Zustand synchronisieren – alle anderen Vergleiche sind unabhängig von der Spielfeldgröße $\Rightarrow O(c \cdot n \cdot m + (c-1) \cdot (c-1) \cdot 1) = O(c \cdot n \cdot m + c^2)$. Da sich die Figuren in der Regel eine längere Zeit in der Gruppe bewegen, unterscheidet sich der State auch seltener, wodurch der Vorteil des Hashwertes noch deutlich wird. Nicht beachtet wurde dabei bisher die Komplexität zur Berechnung des Hash-Wertes. Zunächst wurde die Methode `Arrays.deepHashCode(Object[] a)` zur Berechnung verwendet, die allerdings in Benchmarks immer noch eine hohe Auslastung anzeigte. Stattdessen wurde anschließend das Zobrist Hashing [Zob70] verwendet (genaueres siehe nächster Abschnitt). Das ermöglichte, den Hash-Wert bei Änderungen direkt mit einer einfachen Bit-Operation anzupassen, statt ihn mit dem gesamten Array neu zu berechnen. Dies führte zu einem vernachlässigbaren Aufwand zur Hashberechnung und der Vergleich war effizient genug, dass dies kein Problem mehr darstellte. Die tatsächlich gemessenen Laufzeitergebnisse sind in Abbildung 7 dargestellt.

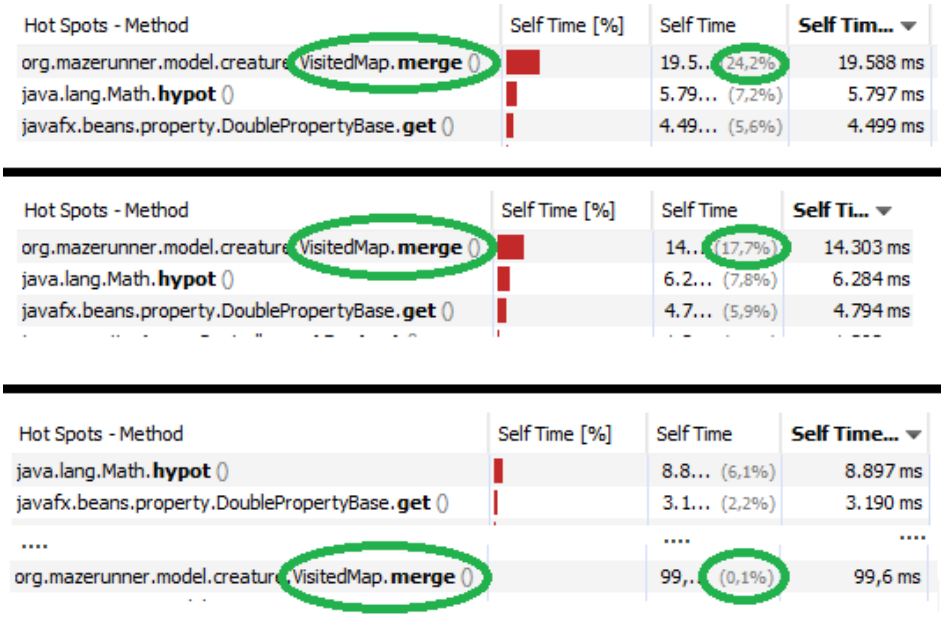

Abbildung 7: Laufzeitanalyse der drei Implementierungen der Merge-Methode mit JavaVisualVM. Die Daten zeigen einen Ausschnitt der rechenintensivsten Methoden gemessen zur Laufzeit mit 600 Kreaturen und dem gleichen Labyrinth Setup. Die erste Tabelle zeigt die Laufzeit ohne Hashing, die zweite mit `Arrays.deepHashCode`, die dritte die letztliche Programmierung mit Zobrist Hashing.

Ablauf des Zobrist Hashing Beim Zobrist Hashing werden einmalig für jedes Feld in jedem möglichen Zustand ein Bitmuster (in Form eines zufälligen Integers) erzeugt.

```
public class VisitedMap {
  private static int[][][] bitStrings;
  public enum VisitedState { UNKNOWN, VISITED, WALL }

  private void initializeBitStrings(int maxX, int maxY) {
    if (bitStrings == null) {
      bitStrings = new int[maxX][maxY][VisitedState.values().length];
      for (int x = 0; x < maxX; x++) {
        for (int y = 0; y < maxY; y++) {
          for (int j = 0; j < VisitedState.values().length; j++) {
            bitStrings[x][y][j] = new Random().nextInt();
          }
        }
      }
    }
  }
  /* ... */
}
```

Der Hash wird dann pro `VisitedMap` Objekt (also pro Kreatur) einmal gebildet, indem ausgehend von dem Hash 0 für jedes Feld der VisitedMap das entsprechende Bitmuster mit *XOR* mit dem Hash verknüpft wird.

```
  private VisitedState[][] map;
  private int hash = 0;

  public VisitedMap(int maxX, int maxY) {
    initializeBitStrings(maxX, maxY);
    map = new VisitedState[maxX][maxY];
    for (int x = 0; x < map.length; x++) {
      for (int y = 0; y < map[x].length; y++) {
        map[x][y] = VisitedState.UNKNOWN;
        hash ^= bitStrings[x][y][VisitedState.UNKNOWN.ordinal()];
      }
    }
  }
```

Änderungen können dann einfach vorgenommen werden, indem zuerst mit *XOR* der alte Wert aus dem Hash rausgenommen wird (denn *XOR* ist seine eigene Umkehrfunktion)

und der neue Wert mit *XOR* hinzugefügt wird.

```
1 private void setNewStateOnPosition(int x, int y, VisitedState newState) {
2   VisitedState old = map[x][y];
3   hash ^= bitStrings[x][y][old.ordinal()] ^
        bitStrings[x][y][newState.ordinal()];
4   map[x][y] = newState;
5 }
```

Wegsuche

Von der konkreten Algorithmik interessant ist bei der Pfadfindung nur die Klasse `NoSightMovement`. Das Vorgehen wurde schon in Kapitel 3.4 kurz beschrieben. Die Aufgabe des Algorithmus ist es, aus der aktuellen `VisitedMap` und der aktuellen Position ein nächstes Ziel zu bestimmen. In Betracht gezogen wurden dafür alle noch unbekannten Felder, also ist das naheste, unbekannte Feld gesucht. Die einfachste Implementierung war, eine Breitensuche wie im Dijkstra-Algorithmus zu beginnen und bei dem ersten unbekannten Feld abzubrechen. Da diese Berechnung nicht in jedem Tick ausgeführt wird, sondern nur wenn eine neue Aktion berechnet wird (bei den normalen Kreaturen also einmal pro Sekunde), ist die Performanz hiervon auch kein Problem gewesen. Tatsächlich ist eine Verbesserung hier gar nicht so leicht: Überlegt war zum Beispiel, ob der Algorithmus beschleunigt werden kann, indem für das gesamte Labyrinth einmalig z.B. mit dem Floyd-Warshall-Algorithmus von allen Punkten aus der kürzeste Weg zu allen anderen Punkten berechnet wird. Statt für jede Aktion den Dijkstra-Algorithmus erneut auszuführen, müsste so nur bei einer Änderung des Labyrinths neuberechnet werden. Das Problem ist allerdings, dass den Kreaturen diese Tabelle nicht so viel nützt, weil sie ihr Ziel nicht kennen, sondern anhand der bereits bekannten Felder ein weiteres suchen wollen. Ebenso ist die Suche mit Dijkstra für diese Aufgabe sehr gut geeignet. Eine algorithmische Verbesserung in diesem Bereich schien also bisher nicht notwendig und konnte von mir auch noch nicht gefunden werden.

Datenstruktur der Kreaturen

Ein Bereich, der im Rahmen des Projektes noch nicht optimiert wurde, ist die Datenstruktur der Kreaturen. Diese sind in einer `ObservableList` gespeichert, was der View erlaubt einen `ChangeListener` darauf zu registrieren. Die Datenstruktur wurde verwendet, da die Implementierung am einfachsten war, jedoch ist sie ein Engpass für die Berechnungen im Model und aktuell der langsamste Teil im Projekt. Grund dafür ist, dass besonders häufig Bereichsanfragen auf die Kreaturen gestellt werden,

beispielsweise wenn ein Tower zum Schießen bereit ist, oder wenn Kreaturen miteinander Kommunizieren wollen. Dementsprechend wäre wahrscheinlich eine Speicherung in einem k-d Baum oder R-Baum effizienter. Das Problem daran ist, dass die Kreaturen sich sehr viel bewegen und somit entweder häufig die aufwändigen Update-Operationen auf den Bäumen ausgeführt werden müssen, oder die Bäume zu jedem Tick mit Bulkload neu erstellt werden müssen. Das stellt einen hohen Anspruch an die Effizienz der Bäume dar.

4.1.4 Das Paket `maze`: Türme

Grundbausteine zum bilden des Labyrinths sind zunächst Mauern. Diese haben sehr geringe kosten und blockieren die Creeps. Jede Mauer hat eine `ObjectProperty` vom generischen Typ `AbstractTower`, kann also durch Polymorphie ein beliebiges Objekt der Unterklassen von `AbstractTower` besitzen. Diese Unterklassen sind die konkreten Türme, die eigene Upgrades definieren und eine Methode `void shoot()` implementieren müssen.

Um ständige `null` Überprüfungen zu vermeiden, wurde auch eine Unterklasse `NoTower` implementiert, die keinerlei Aktivität übernimmt. Die weiteren Turmarten sind:

- `NormalTower`: Ein Turm, der nur einmal pro Sekunde schießt, dafür aber relativ stark ist und der billigste Turm ist. Upgrades erhöhen vor allem die Stärke und Reichweite des Turms. Dieser Turm ist die Orientierung für die Stärke der anderen Türme.
- `FastTower`: Schießt schneller, aber schwächer und kostet etwas mehr. Upgrades sind dafür billiger und verstärken vor allem die Geschwindigkeit und Stärke.
- `SlowdownTower`: Macht keinen Schaden, sondern verlangsamt die getroffene Kreatur. Schießt selbst langsamer und kostet etwas mehr. Upgrades erhöhen die Geschwindigkeit und verstärken den Slowdown-Effekt.
- `AmnesiaTower`: Macht keinen Schaden, sondern „verwirrt“ die getroffene Kreatur, indem eine Zeit lang die Wegfindung der Kreatur durch eine zufällige Wegwahl ersetzt wird. Der Turm versucht beim schießen Kreaturen zu finden, die er bisher noch nicht getroffen hat und kann so eine größere Menge an vorbeilaufenden Kreaturen treffen. Es ist der teuerste Turm, jedoch existieren keine Upgrades.

Der normale Turm ist auch die Orientierung für Game Balancing Überlegungen wie „*Wie oft muss ein Turm eine normale Kreatur treffen?*“, und „*Wie viele Schüsse kann ein Turm auf eine vorbeilaufende Kreatur abgeben?*“, nach denen sich die Leben der Kreaturen richten. Die Werte aller anderen Türme werden ebenfalls an den Daten des normalen Turms orientiert festgelegt. Eine automatisierte Anpassung basierend

auf Nutzerdaten wurde nicht implementiert, wäre aber grundsätzlich möglich und eine interessante Erweiterung.

Die Implementierung der Türme verwendet eine `CountdownAction`, die das Schießen nach dem entsprechenden Intervall triggert. Da wird zunächst mit einer Bereichsanfragen auf die Kreaturen ein Ziel ausgewählt. Anschließend wird eine Kugel erzeugt, die sich wiederum mit einer `MoveAction` zunächst zum Ziel bewegen muss, bevor sie ihre Wirkung entfaltet. Vor allem Letzteres ist wichtig, um den Effekt der Kugel bis zum tatsächlichen Treffen verzögern zu können und nicht direkt in der Schießen-Methode auszuführen. Dadurch ist der Ablauf für den User besser sichtbar und nachvollziehbar.

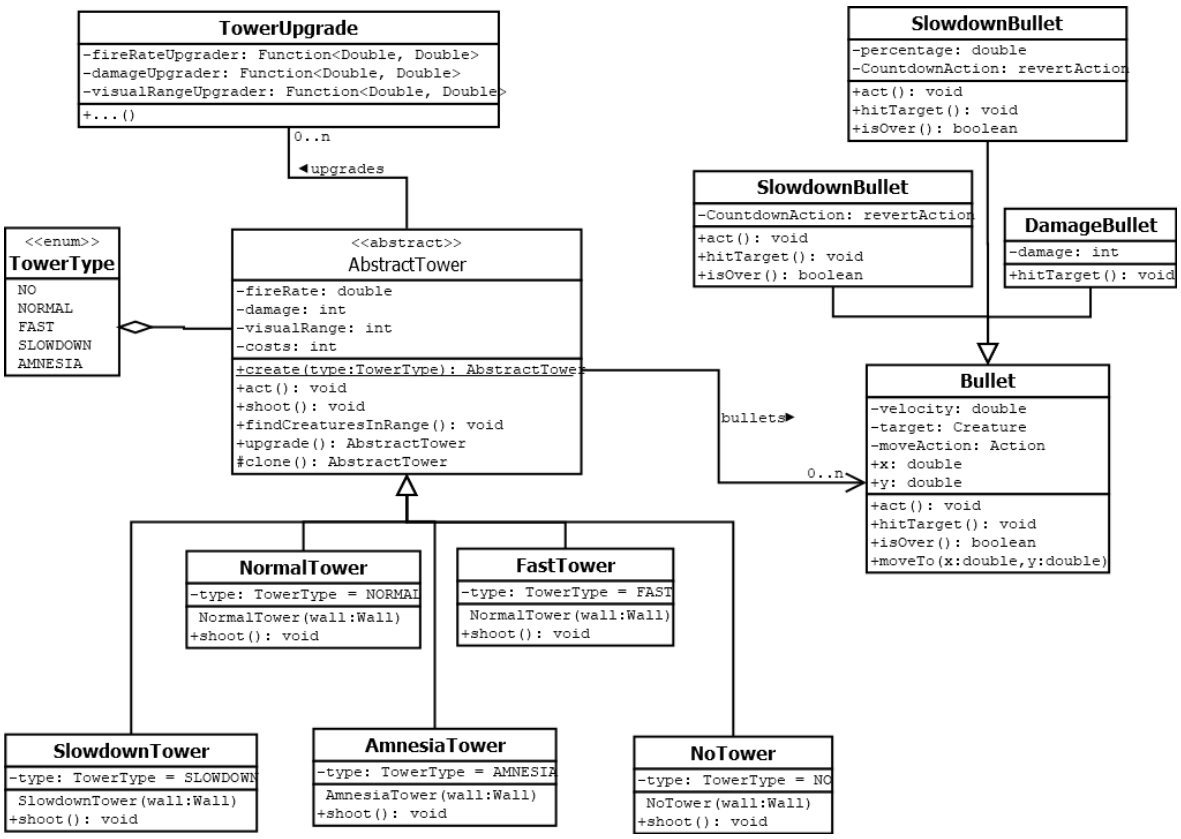

Abbildung 8: Das Klassendiagramm der Tower Klassen (vereinfachte Darstellung).

Upgrades

Die Upgrades werden definiert über drei Implementierungen eines Funktionalen Interfaces, das die Schussgeschwindigkeit, den Schaden bzw. den Sichtradius anpasst. Um die `ObjectProperty` zu nutzen, wird nicht der aktuelle Turm verändert, sondern ein Klon erzeugt, dessen Werte angepasst und der anschließend an den Aufrufer zurückgegeben wird.

```java
public class Wall implements ActorInterface {
  /* ... */
  public void upgradeTower() {
    setTower(getTower().upgrade());
  }
}

public abstract class AbstractTower implements ActorInterface, Cloneable {
  /* ... */
  public AbstractTower upgrade() {
    if (upgrades.size() > getLevel()) {
      TowerUpgrade upgrade = upgrades.get(getLevel());
      try {
        AbstractTower clone = (AbstractTower) this.clone();
        clone.fireRate = upgrade.getFireRateUpgrader().apply(getFireRate());
        clone.damage = upgrade.getDamageUpgrader().apply(getDamage());
        clone.visualRange =
            upgrade.getVisualRangeUpgrader().apply(getVisualRange());
        clone.costs = this.costs + upgrade.getCosts();
        clone.level = level +1;
        return clone;
      } catch (CloneNotSupportedException e) {
        LOG.log(Level.SEVERE,"couldn't clone tower", e);
      }
    }

    return this; //if something went wrong
  }
}
```

Datenstruktur der Mauern

Die Datenstruktur im `Maze`, die die Mauern hält, ist wie bei den Kreaturen eine `ObservableList`. Dadurch kann die View wieder einen Listener auf die Änderungen registrieren. Da die Kreaturen oft anfragen müssen, ob auf einem Feld eine Mauer den Weg blockiert, wurde zusätzlich noch ein zweidimensionales Boolean-Array gespeichert, das allerdings nur intern verwendet wird. Der zusätzliche Speicheraufwand ist zu vernachlässigen, da nur ein einzelnes Maze existiert und somit nur ein Boolean-Array in der Größe des Spielfeldes gespeichert wird. Die Anfragezeit, ob ein Feld belegt ist, konnte dagegen dadurch auf konstante Zeit $O(1)$ im Gegensatz zu linearer Zeit $O(w)$ (w Anzahl der `Wall` Objekte) reduziert werden.

4.1.5 Die Klasse `Game`

Das Interface `GameModelInterface` und die implementierende Klasse `Game` ist der Einstiegspunkt im Model, der das Player-, Level- und Maze-System initialisiert und miteinander verbindet. Zusätzlich verwaltet es einen `GameState`, der beschreibt ob

das Spiel noch in der BauPhase vor dem Start ist, gerade läuft, oder bereits beendet (gewonnen / verloren) ist.

4.2 View und Controller

Das Grundlayout der View wurde in *FXML* erstellt und besteht aus den folgenden Bereichen: Links Die Level Timeline mit den Kreaturen Wellen, daneben ein weiterer unterteilter Bereich mit den Spielerinformationen (oben), der Labyrinthfläche (Mitte) und den Kontrollbuttons (unten).

Abbildung 9: Grundaufbau der Oberfläche, wie in fxml spezifiziert. Die Aufteilung gelang mit zwei geschachtelten `BorderLayout` s.

4.2.1 CreatureTimeline

Die CreatureTimeline hat einen interessanten Aspekt, nämlich das scrollen mit der Spielzeit. Dafür verwendet sie das `passedTimePercentageBinding` aus dem LevelModelInterface und bindet an die `translateYProperty` aus der VBox, die die Kreaturen Gruppen enthält. Beim Klicken auf eine Kreatur der Timeline wird ein Popup mit den entsprechenden Informationen über die Kreaturen der Welle erzeugt

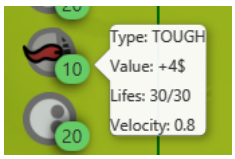

Abbildung 10: Popup mit Informationen über kommende Kreaturen.

4.2.2 PlayerView

Die PlayerView ist ebenfalls in fxml geschrieben und bindet die `lifesProperty` und die `moneyProperty` (als Geld-Text formatiert) an die Labels.

```
1 public void initModel(PlayerModelInterface player) {
2   this.player = player;
3   money.textProperty().bind(Util.moneyString(this.player.moneyProperty()));
4   lifes.textProperty().bind(Bindings.max(0,
        this.player.lifesProperty()).asString());
5 }
```

4.2.3 Controls

Die Controls sind direkt in der `GameView.fxml` enthalten, da sie keinerlei View-Aufgaben haben. Drücken des Play/Pause-Buttons ruft eine Methode auf dem GameController auf, der die GameLoop startet bzw. stoppt. Ebenso rufen die Bauen- und Info Buttons Methoden auf, die den Modus entsprechend wechseln. Im Bauen Modus werden alle Aktionen von den Controllern als Bau-aktionen interpretiert (Klick in Labyrinthfläche baut Mauer), im Informationsmodus werden keine Bauaktivitäten durchgeführt sondern wenn möglich weitere Informationen angezeigt (Klick auf Turm zeigt Informationen statt Buttons zum Abreisen).

Abbildung 11: Klick auf Turm im Bauen Modus (links) und Informationsmodus (rechts)

4.2.4 MazeView

Die MazeView enthält eine `Pane` für die Kreaturen zur beliebigen Positionierung und eine `StackPane` für die Türme. Dies ist leider nicht optimal, da dann entweder die KreaturenEbene unter der Turmebene oder andersrum liegen muss. Dadurch wird entweder die Lebensanzeigen und Sprechblasen der Kreaturen (in Kreaturenebene) von

Türmen verdeckt werden, oder die Kugeln der Türme (in Turmebene) unter den Kreaturen durch fliegen. Hier wäre eine Neustrukturierung gut.

Von der Implementierung her arbeiten die beiden Ebenen recht ähnlich: Sie haben einen Listener auf der Liste der Creeps bzw. Mauern im Model und erstellen bzw. entfernen dementsprechend `CreatureView` bzw. `WallView`, die sie als Kinder verwalten.

```
1  public class CreaturesView extends Pane implements
       Bindable<MazeModelInterface> {
2      ListChangeListener<Creature> listener =
3        (c) -> {
4          while (c.next()) {
5            if (c.wasAdded()) {
6              createCreatures(c.getAddedSubList());
7            } else if (c.wasRemoved()) {
8              removeCreatures(c.getRemoved());
9            }
10         }
11       };
12
13   @Override
14   public void bind(MazeModelInterface maze) {
15     /* ... */
16     creatures.addListener(listener);
17   }
18   /* ... */
19 }
```

Wird eine Kreatur entfernt, erstellt die Kreaturenebene ein Label für das gewonnene Geld, zeigt es kurz an und entfernt es dann wieder.

CreatureView

Die Klasse `CreatureView`, die für die Anzeige einer einzelnen Kreatur zuständig ist, hat folgende interessante Aspekte:

- Ein Listener auf den Leben der Kreatur zeigt einen Fortschrittsbalken an, sobald die Kreatur Schaden nimmt.
- Ein Listener auf der Position wird verwendet um aus der Änderung die aktuelle Drehung der Kreatur zu berechnen.

- Ein Listener auf der Aktion der Kreatur zeigt eine Sprechblase an, sobald die Kreatur eine `TalkAction` anfängt.

TowerView

Die `WallView` ist für die Anzeige einer Mauer zuständig und verwendet intern nochmal eine `TowerView` für die Anzeige des Turms und der Kugeln. Der Code ist sehr einfach, das einzige spannende ist das Menü zum Turmbau und -upgrade, wenn man eine Mauer anklickt. Dafür wurde das `CirclePopupMenu` aus der Bibliothek *jfxtras* verwendet, das die Elemente gleichmäßig in einem Kreis anordnet und auf- und zuklappen animiert.

Abbildung 12: Popup-Menü mit der externen Klasse `CirclePopupMenu` zum Bauen von Türmen.

4.3 Weitere Klassen

Für das Projekt wurden noch einzelne Hilfsklassen programmiert, die größtenteils statisch verwendet werden. Beispielsweise bietet `Util` Methoden zur Berechnung der Distanz zwischen zwei Punkten, oder zur Berechnung des Winkels eines Vektors. Besonders elegant ist der `ImageLoader`. Dieser bietet einen statischen Zugriff auf die Bilder und verhindert so, dass diese mehrmals geladen werden oder die Logik zum Laden von Bildern sich im Code verteilt. Die Bilder werden dort versucht zu laden, wobei nicht existierende Bilder durch einen Platzhalter ersetzt werden. Das ermöglichte die Grafiken unabhängig von den Features zu entwickeln.

```
1 public class ImageLoader {
2   private static final String basePath = "images/";
3   private static Image placeholder = loadImage("placeholder.png");
4   private static Image normalCreature;
5   /* ... */
6
7   public static void loadGameImages() {
8     normalCreature = loadImage("creatures/normal.png"); /* ... */
9   }
```

```
10
11   private static Image loadImage(String src) {
12     InputStream resourceStream =
13         ImageLoader.class.getClassLoader().getResourceAsStream(basePath +
               src);
14     if (resourceStream != null) {
15       return new Image(resourceStream);
16     } else if (placeholder != null) {
17       return placeholder;
18     }
19
20     LOG.severe("Image loading unsuccessfull!");
21     throw new Error("Failed loading Image " + src);
22   }
23 }
```

5 Reflexion und Ausblick

Insgesamt ist das Projekt sehr weit so verlaufen, wie ich es mir vorgestellt hatte und das Ergebnis finde ich auch durchaus gelungen. Einige Erweiterungen, die ich in dem Spiel gerne haben würde, um das Spielerlebnis noch einzigartiger zu gestalten, konnten leider noch nicht umgesetzt werden, aber der Grundstein dafür ist gelegt. Die wichtigsten Features sind dabei die intelligenteren Kreaturenarten bzw. andere Wegsuchealgorithmen, wie sie in 3.4 beschrieben wurden. Des Weiteren steht noch die Implementierung eines Kampagnenmodus und eines Tutoriallevels an, um so einen gewissen Fortschritt durch das Spielen zu bekommen. Eine größere Herausforderung ist, ein Game-Balancing-System umzusetzen, da dafür eine Menge Daten und Datenverarbeitung nötig wären. Im Moment sind die Schwierigkeitsüberlegungen alle manuell fest einprogrammiert.

Eine persönliche Weiterbildung gelang vor allem in der View-Programmierung, da es mein erstes JavaFX Projekt ist. Das Framework hat viele Dinge vereinfacht, mit denen ich mich bei Swing sonst lange herumschlagen musste, unter anderem Layouting mit fxml und Drehungen von Elementen und Animationen. Insbesondere die Bindings sind großartig und entlasten das MVC-Konzept (was man an den kleinen Controller-Klassen sieht). Auch die Kompatibilität zu Android ist ein großes Plus und macht es auch für Programmierprojekte im Schulunterricht nochmal interessanter.

Die konsequent testgetriebene Entwicklung benötigte während der Implementierung deutlich merklich mehr Zeit. Jedoch hatte ich dadurch während des ganzen Projektes auch bei größeren Änderungen nie Probleme Fehler zu finden. Außerdem gab die große

Testabdeckung eine gewisse Sicherheit, sodass ich vor den größeren Umstrukturierungen auch nicht verunsichert war, ob hinterher noch alles funktioniert. Diese Sachen haben viel Zeit wieder eingespielt und ich denke, dass dadurch auch eine höhere Code-Qualität zustande gekommen ist.

Negativ ist zu bemerken, dass ich die Javadoc-Dokumentation sehr habe schleifen lassen. Die konsequente Kommentierung des Codes kann auf jeden Fall noch verbessert werden. Dafür wurde versucht möglichst sprechenden Code (Variablen-, Methoden- und Klassennamen) zu schreiben und lange Methoden oder sogar Klassen in kleinere zu unterteilen.

Insgesamt lässt sich aber festhalten, dass das Projekt durchaus gelungen umgesetzt wurde und ohne größere Komplikationen verlief.

Literatur

[Blo08] Bloch, Joshua: *Effective Java*. Boston : Addison-Wesley Professional, 2008 `https://books.google.de/books?id=ka2VUBqHiWkC&pg=PA5`. – ISBN 978–0–132–77804–6

[Imp15a] Impe, Steven V.: *Game loops – Applying the theory to JavaFX*. `http://svanimpe.be/blog/game-loops-fx`. Version: Januar 2015

[Imp15b] Impe, Steven V.: *Game loops – Basic theory*. `http://svanimpe.be/blog/game-loops`. Version: Januar 2015

[Nys14] Nystrom, Robert: *Game Loop*. `http://gameprogrammingpatterns.com/game-loop.html`. Version: 2009-2014

[Zob70] Zobrist, Albert L.: A new hashing method with application for game playing. In: *ICCA journal* 13 (1970), Nr. 2, S. 69–73

Eidesstattliche Versicherung

Erklärung zur Hausarbeit gemäß §29 (Abs.6) LPO I

Hiermit erkläre ich, dass die vorliegende Hausarbeit von mir selbstständig verfasst wurde und dass keine anderen als die angegebenen Hilfsmittel benutzt wurden. Die Stellen der Arbeit, die anderen Werken dem Wortlaut oder Sinn nach entnommen sind, sind in jedem einzelnen Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht.

Diese Erklärung erstreckt sich auch auf etwa in der Arbeit enthaltene Zeichnungen, Kartenskizzen und bildliche Darstellungen.

.....	.....
Ort, Datum	Name