UNIVERSITY OF APPLIED SCIENCES

Department of Computer Science

Programmierpraktikum WS2021

"Escape-Bot"

Eingereicht am:

24.01.2022

Eingereicht von:

Timo Peters

Fachrichtung: Informatik

Matrikelnummer: inf104225

Fachsemester: 5

Verwaltungssemester: 5

Referent:

Dipl.-Ing. (FH) Gerit Kaleck

Fachhochschule Wedel

Feldstraße 143, 22880 Wedel

Tel.: (041 03) 80 48-22

E-Mail: klk@fh-wedel.de

Inhaltsverzeichnis

1 Benutzerhandbuch

1.1 Ablaufbedingungen

Mindestangaben zur Nutzung der Software.

Komponente	**Version**
Bildschirm	min. 1024x720
Java Runtime Environment	11
Windows	8/10/11

1.2 Programminstallation/Programmstart

Um das Programm ausführen zu können, müssen die Ablaufbedingungen erfüllt sein. Ist dies der Fall, so kann das Programm über einen Doppelklick auf die .jar-Datei geöffnet werden. Alternativ kann die .jar-Datei auch in der Konsole mit folgendem Befehl ausgeführt werden:

```
java -jar PP_Peters.jar
```

Falls es beim Starten des Programms zu Fehlern kommen sollte, kann der Abschnitt Fehlermeldungen zur Identifikation und Behebung des Fehlers genutzt werden.

1.3 Bedienungsanleitung

1.3.1 Idee und Ziel des Spiels

Bei dem Spiel ”Escape-Bot” steuert der Spieler einen Roboter mithilfe einer Vielzahl von Anweisungen innerhalb eines Spielfeldes von einem Startpunkt zu einem Ziel. Dabei existieren unterschiedliche Anweisungen, aus denen der Spieler sein Programm zusammensetzen kann. Ein Programm kann hierbei maximal 12 Anweisungen enthalten, wobei zusätzlich 2 Prozeduren mit jeweils maximal 8 Anweisungen verwendet werden können. Diese Prozeduren können wie die anderen Anweisungen, sowohl im Programm, als auch in der jeweils anderen Prozedur aufgerufen werden. Sie repräsentieren eine Sequenz aus den in ihr enthaltenen Anweisungen, welche nacheinander ausgeführt werden.

Ziel des Spiels ist es den Roboter mit einem geschickten Programm durch ein aus einer Datei geladenes oder selbst entworfenes Level zu führen um am Ende den Ausgang zu erreichen.

Der Ausgang kann jedoch erst betreten werden, sobald keine Münzen mehr auf dem Spielfeld liegen. Diese müssen also alle auf dem Weg eingesammelt werden.

1.3.2 Verfügbare Anweisungen

Folgende Anweisungen können verwendet werden um ein Level zu lösen:

Anweisung	**Wirkung**	**Aussehen**
Gehen	Der Bot bewegt sich in seiner aktuellen Ausrichtung ein Feld nach vorne. Geht nur, wenn das Zielfeld normal, eine Münze oder das Startfeld ist	gerader Pfeil nach oben
Links	Der Bot dreht sich um 90° nach links.	runder Pfeil nach links
Rechts	Der Bot dreht sich um 90° nach rechts.	runder Pfeil nach rechts
Springen	Der Bot spring in seiner aktuellen Ausrichtung über genau ein Feld mit einem Abgrund. Geht nur, wenn das Feld vor dem Bot ein Abgrund ist und das Feld danach normal, eine Münze oder das Startfeld.	gerader Pfeil nach oben über einen grauen Kasten
Prozedur 1	Prozedur 1 wird aufgerufen und alle Anweisungen darin ausgeführt. Nach Abarbeitung der Prozedur geht es beim Aufrufer (Programm oder Prozedur 2) mit der nächsten Anweisung weiter. Eine Prozedur darf sich nicht selbst aufrufen. Ebenso dürfen sich die beiden Prozeduren nicht gegenseitig aufrufen (Endlosrekursion).	Text ”P1”
Prozedur 2	Prozedur 2 wird aufgerufen und alle Anweisungen darin ausgeführt. Nach Abarbeitung der Prozedur geht es beim Aufrufer (Programm oder Prozedur 1) mit der nächsten Anweisung weiter. Eine Prozedur darf sich nicht selbst aufrufen. Ebenso dürfen sich die beiden Prozeduren nicht gegenseitig aufrufen (Endlosrekursion).	Text ”P2”
Exit	Die Tür wird geöffnet und somit entfernt (das Feld wird dadurch ein normales Feld). Geht nur, wenn der Bot sich direkt in richtiger Ausrichtung vor der Tür befindet und keine Münzen mehr vorhanden sind.	Text ”EXIT”

Kopie aus der Aufgabenstellung

Tabelle 1.1: Verfügbare Anweisungen

1.3.3 Regeln

Wie in der vorherigen Tabelle Tabelle 1.1 beschrieben, gelten ein paar Regeln die bei der Programmerstellung beachtet werden müssen.Allgemein gilt das Programm nur als erfolgreich, wenn nach Abarbeitung **aller** Anweisungen alle Münzen eingesammelt wurden und das Ziel erreicht wurde.

Dabei sind zudem folgende Restriktionen zu beachten:

- Der Spieler kann nicht durch oder gegen eine Wand oder einen Abgrund laufen
- Der Spieler kann nicht aus dem Spielfeld heraus laufen
- Der Spieler kann nicht über Abgründe springen, die sich über mehrere Felder erstrecken

- Der Spieler kann nur über einen Abgrund springen, wenn das Feld dahinter frei ist.
  - Dabei muss das Feld auf dem der Bot landet eine Münze oder ein normales Feld sein
- Zum abschließen des Levels ist eine ”Exit” Anweisung erforderlich, während der Spieler vor dem Ausgang steht
- Um die ”Exit” Anweisung ausführen zu können, muss der Roboter direkt vor der Tür stehen und in dessen Richtung gucken
- Werden nach einem erfolgreichem Aufruf von ”Exit” weitere Anweisungen ausgeführt, so gilt das Level als nicht geschafft.
- Die maximale Programmlänge von 12 Anweisungen zuzüglich zweier Prozeduren mit jeweils maximal 8 Anweisungen darf nicht überschritten werden
- Bei der Programmerstellung dürfen keine Rekursionen eingebaut werden.
  - Als Rekursion gilt dabei
    * das Aufrufen einer Prozedur in der Selben Prozedur
    * das gegenseitige Aufrufen der beiden Prozeduren

1.3.4 Benutzeroberfläche/Bedienung

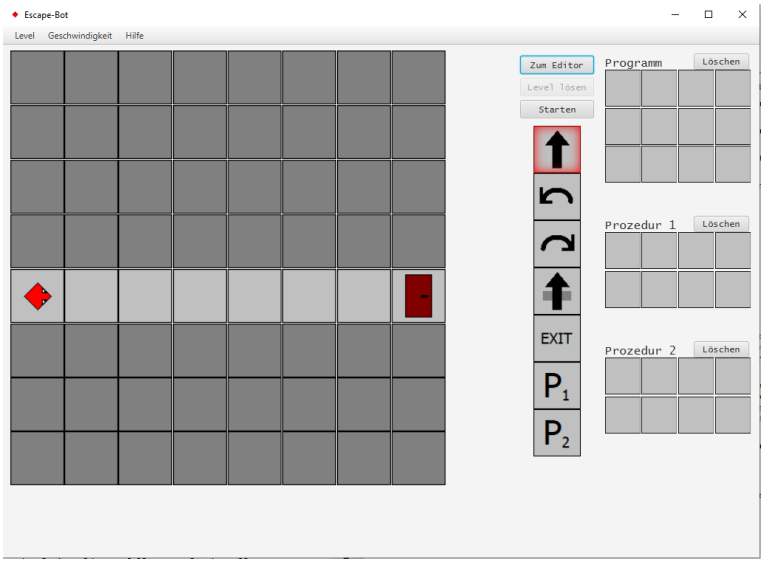

Abbildung 1.1: Benutzeroberfläche

Programmerstellung

Um ein Programm erstellen zu können muss sich der Spieler im Spielmodus befinden. Ist dies nicht der Fall, so kann über den Knopf „Zum Spiel” im mittleren Teil des Fensters (siehe Abbildung 1.1) in den Spielmodus gewechselt werden. Hier gibt es im mittleren Teil des Fensters unter den Aktionsknöpfen eine Liste an verfügbaren Anweisungen.

Der Spieler kann mit der linken Maustaste auf eine der Anweisungen klicken um diese auszuwählen. Die aktuelle Auswahl ist zudem mit einem roten Rahmen markiert. Mit klicken auf ein freies Feld in dem Bereich von Programm und Prozeduren wird die ausgewählte Anweisung hinzugefügt. Dabei ist zu beachten, dass nur das nächste, nicht besetzte, Feld mit einer Anweisung besetzt werden kann. Klickt der Spieler mit der linken Maustaste auf ein besetztes Feld, so wird dieses gegen das momentan ausgewählte ausgetauscht.

Außerdem kann mit einem Rechtsklick auf eine Anweisung, im Programm oder den Prozeduren, diese entfernt werden, wobei nachfolgende Anweisungen aufrücken um das freie Feld auszufüllen. Zum löschen aller Programm- oder Prozeduranweisungen dient der „Löschen" Knopf auf der rechten Seite über dem jeweiligen Anweisungsblock.

Wenn der Spieler mit seiner Programmerstellung glücklich ist, kann das Programm über den Knopf „Starten" im mittleren Fensterbereich gestartet werden.

Anpassung der Animationsgeschwindigkeit

Vor dem Start des Levels kann über die Menüoption „Geschwindigkeit" die Geschwindigkeit eingestellt werden. Zur Verfügung stehen dabei die Geschwindigkeiten x0.5, x1, x1.5 und x2. Das Umstellen der Geschwindigkeit ist auch während einer Animation möglich, gilt dann aber erst für den nächsten Start einer Animation.

Zudem können die Animationen im selben Menü „Geschwindigkeit" über die Menüoption „Animationen auschalten" auch komplett ausgeschaltet werden. Dadurch wird mit starten des Levels automatisch der Endzustand des Spiels, nach Abarbeitung aller Anweisungen, angenommen und es kommt zu einer direkten Meldung über den Erfolg/Misserfolg des Levels.

Anzeige der aktuell ausgeführten Anweisung

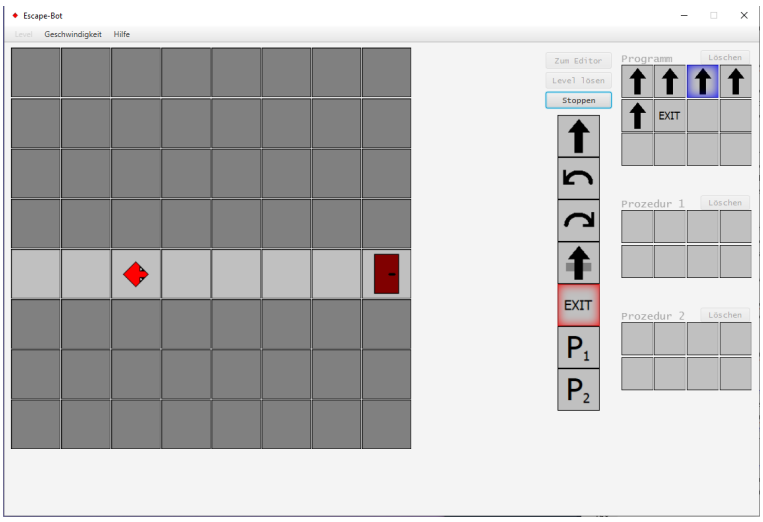

Abbildung 1.2: Benutzeroberfläche bei laufender Animation

Während die Animation des Levels stattfindet, wird die Anweisung, die gerade ausgeführt wird, mit einem blauen Rahmen markiert. So kann jederzeit nachverfolgt werden, bei welcher

Anweisung sich die Animation gerade befindet und wo möglicherweise fehlerhafte Anweisungen vorhanden sind.

Wie man in Abbildung 1.2 sehen kann, sind während eine Animation läuft einige Funktionen nicht verfügbar. Dazu gehört das Öffnen des „Level” Menüs, das Wechseln in den Editor und das Lösen des Levels. Letzteres ist generell nur im Editormodus möglich. Außerdem kann das Programm bei laufender Animation nicht mehr angepasst werden. Zudem lässt sich das Anwendungsfenster währenddessen nicht vergrößern oder verkleinern.

Ende eines Spieldurchlaufes

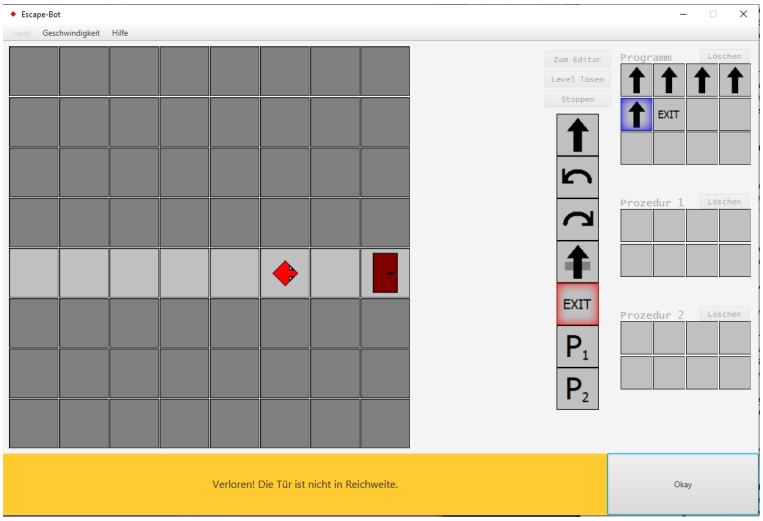

Abbildung 1.3: Meldung nach Beendigung eines Spieldurchlaufes

Ein Spieldurchlauf endet entweder über das manuelle anhalten des Spiels über den „Stoppen” Knopf, oder durch einen vollendeten Programmdurchlauf. Ein Programm kann dabei auf mehrere Weisen beendet werden, was jeweils über einen erscheinenden Meldedialog am unteren Fensterrand zu erkennen ist (siehe Abbildung 1.2).

1. **Fehler:** Das Programm konnte nicht gestartet werden, da die eingegebenen Anweisungen einen Fehler aufweisen. Dazu zählt zum Beispiel das erzeugen einer Endlosrekursion.
2. **Gewonnen:** Das Programm ist erfolgreich durchgelaufen und das Ziel wurde erreicht nachdem alle Münzen eingesammelt wurden.
3. **Verloren:** Das Programm ist erfolgreich durchgelaufen, das Ziel wurde jedoch nicht erreicht oder vor dem Erreichen des Ziels wurden nicht alle Münzen eingesammelt.

Eigene Level Erstellen (Editor-Modus)

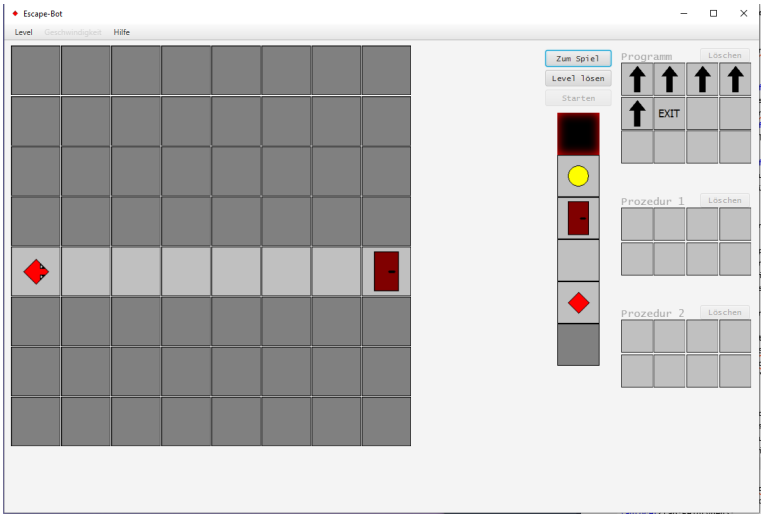

Abbildung 1.4: Editor-Modus

Der Spieler kann über den Knopf „Zum Editor” in eine Level-Editor Ansicht wechseln und dort sein eigenes Level erstellen. Dabei stehen verschiedene Feldtypen zur Verfügung, wobei zwingend genau ein Startfeld und ein Ausgang vorhanden sein muss, damit das Level überhaupt gelöst werden kann.

Das Erstellen eigener Level erfolgt in dieser Ansicht über das Auswählen von Feldtypen aus der Auswahl in der Mitte des Fensters (vgl. Abbildung 1.4) und das setzen dieser mit Linksklick auf das jeweilige Feld innerhalb des Spielerasters. Das erstellte Level kann dann durch einen Wechsel auf den Spielmodus direkt gespielt, oder über den Reiter „Speichern unter” innerhalb des „Level” Menüs in einer Datei abgespeichert werden. Alternativ kann zum Speichern das Tastenkürzel *Strg+S* verwendet werden.

Falls man ein Level von Grund auf neu erstellen möchte, kann über die Menüoption „Neu” innerhalb des „Level” Menüs ein leeres Feld erstellt werden. Alternativ kann diese Funktionalität auch mit dem Tastenkürzel *Strg+N* erreicht werden.

Blickrichtung des Roboters

Die Blickrichtung des Roboters kann im Editor angepasst werden. Hierfür muss der Roboter unter den Feldtypen ausgewählt werden und kann dann mit einem Rechtsklick gedreht werden. Dabei dreht sich der Roboter mit jedem Rechtsklick um 90° nach rechts.

Verfügbare Feldtypen

Insgesamt gibt es im Editor Modus die folgenden sechs Feldtypen Tabelle 1.2:

Feldtyp	Beschreibung	Aussehen
Abgrund	Kann vom Bot nicht betreten werden. Ein genau ein Feld breiter Abgrund kann aber übersprungen werden.	komplett schwarz
Münze	Münzen können vom Bot eingesammelt werden, das Feld wird dadurch ein "normales"Feld. Erst wenn alle Münzen eingesammelt wurden, kann die Tür geöffnet werden.	gelber Kreis auf hellgrauem Grund
Tür	Mit dem Öffnen der Tür ist das Level gewonnen. Es muss genau eine Tür pro Level geben	rotbraunes Rechteck mit schwarzem Strich als Klinke auf hellgrauem Grund
Normal	Ein einfaches, leeres Feld.	komplett hellgrau
Start	Auf dem Startfeld beginnt der Bot das Level in einer pro Level festzulegenden Ausrichtung. Es muss genau ein Startfeld pro Level geben.	rote Raute auf hellgrauem Grund
Mauer	Ein nicht betretbares Feld.	komplett dunkelgrau

Kopie aus der Aufgabenstellung

Tabelle 1.2: Verfügbare Feldtypen

Erstelltes Level auf Lösbarkeit prüfen

Als Hilfe steht dem Spieler eine Prüfung auf Lösbarkeit in der Editor-Ansicht zur Verfügung, womit das Programm automatisch prüft, ob das kreierte Level lösbar ist. Ob das Level lösbar ist, wird über eine Meldung am unteren Fensterrand angezeigt. Ist das kreiert Level lösbar, so wird dem Spieler außerdem eine mögliche Lösung in Programm und erstellt, welche automatisch in die Anweisungsblöcke auf der rechten Seite eingefügt wird.

1.3.5 Beispiellevels laden

Möchte der Nutzer keine eigenen Level erstellen, so kann aus sechs verschiedenen, vordefinierten, Beispiellevels ausgewählt werden. Diese können innerhalb des Menüreiters „Level” geladen werden.

Level aus Dateien laden

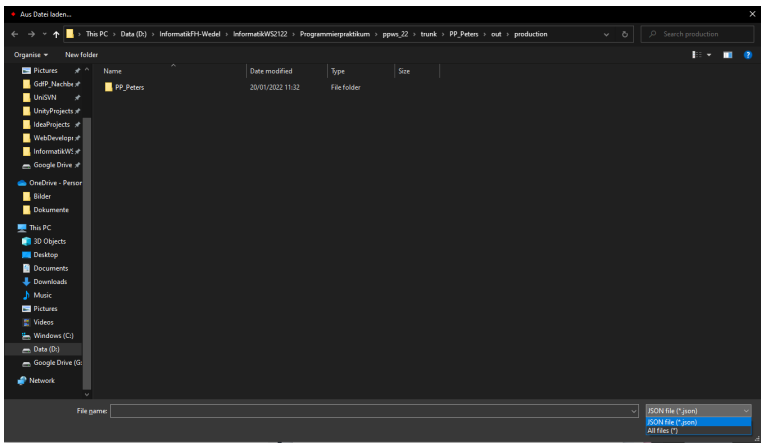

Abbildung 1.5: Datei-Auswahlfenster

Es besteht zudem die Möglichkeit Level aus Dateien zu laden. Dies ist über die Menüoption „Aus Datei laden...” innerhalb des „Level” Menüs möglich. Alternativ kann hierfür auch das Tastenkürzel *Strg+O* verwendet werden.

Wird die entsprechende Menüoption ausgewählt, öffnet sich ein Datei-Auswahlfenster in dem eine Datei auf dem Computer geöffnet werden kann. Standardmäßig ist hierbei der Dateifilter auf .json-Dateien eingestellt. Dieser kann, wie in Abbildung 1.5 zu sehen ist, über die Auswahl am unteren rechten Fensterrand auf „Alle Dateien” geändert werden um auch Dateien zu finden, welche nicht die .json-Dateiendung besitzen.

Ungeachtet ihrer Dateiendung müssen die Leveldateien in einem gültigen JSON-Format vorliegen und folgendendermaßen aufgebaut sein Listing 1.1:

```
{"field": [
    [4, 3, 3, 3, 3, 3, 3, 1],
    [5, 3, 0, 0, 0, 0, 0, 3],
    [2, 3, 0, 0, 0, 0, 0, 3],
    [3, 0, 0, 0, 0, 0, 0, 3],
    [3, 0, 0, 0, 0, 0, 0, 3],
    [3, 0, 0, 0, 0, 0, 0, 3],
    [1, 3, 3, 3, 3, 3, 3, 1]
  ],
  "botRotation": 1}
```

Listing 1.1: Möglicher Aufbau einer Leveldatei

Das Feld wird hierbei mit den numerischen Werten der jeweiligen Feldtypen beschrieben:

0 = Abgrund	3 = Normal
1 = Münze	4 = Start
2 = Tür	5 = Mauer

Die Ausrichtung des Roboters wird repräsentiert als numerischer Wert der Himmelsrichtung angefangen mit Norden:

0 = Norden	2 = Süden
1 = Osten	3 = Westen

1.3.6 Spielanleitung

Innerhalb des Programms ist es außerdem möglich eine Spielanleitung unter dem Menüreiter „Hilfe" afuzurufen. Alternativ kann hierfür das Tastenkürzel *Strg+H* verwendet werden.

Diese Spielanleitung unterscheidet sich von dem jeweiligen Modus, in dem sich der Nutzer gerade befindet. Ist der Nutzer momentan im Spielmodus, so zeigt die Spielanleitung eine Anleitung zum erstellen eines Programms an. Befindet er sich hingegen im Editor, so wird eine Anleitung zum Erstellen eines Levels angezeigt.

1.4 Fehlermeldungen

Abbildung 1.6: Meldedialog eines schwerwiegenden Fehlers

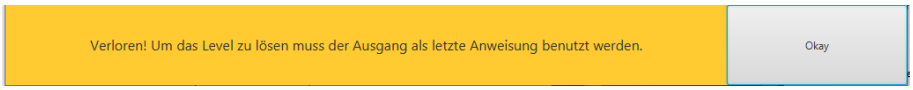

Abbildung 1.7: Meldedialog eines Fehlers

Fehlermeldungen werden am unteren Fensterrand durch ein Dialog dargestellt. Sie sind in zwei verschiedene Arten aufgeteilt, welche durch verschiedene Farbhervorhebungen voneinander unterschieden werden können.

Die erste Art von Fehlermeldungen sind die schwerwiegenden Fehler. Diese treten beispielsweise beim Laden invalider Dateien, oder beim erzeugen einer Endlosrekursion auf. Dargestellt werden diese Fehler mit einem roten Hintegrund (siehe Abbildung 1.6.

Neben einer kurzen Erklärung des aufgetretenen Fehlers, enthält der Meldedialog für Fehler beim Laden eines Levels noch eine genauere Fehlerbeschreibung in dick gedruckter Schrift. Diese ist insbesondere bei invalidem JSON sinnvoll, da hier die Zeile in der Datei beschrieben wird, an der sich der Fehler befindet.

Die zweite Art von Fehlermeldungen sind die Fehler, die im Rahmen des Spielens auftreten können. Dazu gehören Meldungen über Misserfolg beim eigenen Lösen eines Levels. Auch Meldungen zu nicht lösbaren Levels, wird mit dieser Art von Fehlermeldung dargestellt. Zu erkennen sind solche Fehler an deren gelbem Hintergrund.

Des weiteren gibt es auch noch grün hinterlegte Meldungen, welche bei erfolgreichem Lösen eins Levels, beim Stoppen des Levels und bei Erfolgreicher Prüfung eines Levels, angezeigt

werden. Diese Meldungen werden im folgenden nicht aufgelistet, da sie keinen Fehler darstellen.

1.4.1 Programmstart

Fehlermeldung	**Fehlergrund**	**Fehlerbehebung**
Der Befehl "javaïst entweder falsch geschrieben oder konnte nicht gefunden werden	Die Java JRE wurde nicht gefunden	Installiere eine Java JRE, welche mindestens der Version 11 entspricht.
Error: A JNI error has occured, please check your installation and try again	Das Programm wurde mit einer nicht kompatiblen Java JRE-Version ausgeführt	Installiere eine Java JRE, welche mindestens der Version 11 entspricht

1.4.2 Laden von Dateien

Fehlermeldung	Fehlergrund	Fehlerbehebung
Die angegebene Datei kann nicht eingelesen werden. Stelle sicher, dass es sich um Datei im benötigten JSON-Format handelt	Innerhalb der Datei befindet sich ein JSON-Syntaxfehler	Achte darauf, dass die Datei aus valider JSON-Syntax bestehen muss (siehe Abschnitt 1.3.5)
Die ausgewählte Leveldatei ist leer und kann somit nicht eingelesen werden.	Es wurde versucht eine leere Leveldatei zu öffnen	Befülle die Datei mit gültigem JSON in dem in Abschnitt 1.3.5 beschriebenen Format
Die in der ausgewählten Leveldatei angegebene Botrichtung ist invalide. Achte darauf, dass nur eine Botrichtung von 0-3 zulässig ist.	Die Botausrichtung aus der Leveldatei ist invalide	Wähle eine Botausrichtung zwischen 0 und 3 (Norden bis Westen)
Mindestens ein Feld in der angegebenen Leveldatei ist invalide. Achte darauf, dass nur Feldtypen von 0-5 spezifiziert sind.	Mindestens ein Feldtyp ist nicht richtig spezifiziert	Wähle einen Feldtyp zwischen 0 und 5 (siehe Abschnitt 1.3.5)
Das angegebene Spielfeld hat nicht die richtige Größe. Stelle sicher, dass es sich um ein 8x8 Feld handelt.	Es handelt sich nicht um ein valides 8x8 Spielfeld	Stelle sicher, dass die "field-Eigenschaft aus einem 2-dimensionalen Array der Größe 8x8 besteht
Die angegebene Leveldatei besitzt kein Attribut "field".	Das "fieldAttribut fehlt	Ergänze die Datei um ein "field-Attribut gemäß Abschnitt 1.3.5
Die angegebene Leveldatei besitzt kein Attribut "botRotation".	Das "botRotationAttribut fehlt	Ergänze die Datei um ein "botRotationAttribut gemäß Abschnitt 1.3.5
Es ist kein Zugriff auf die gegebene Datei möglich. Überprüfe, ob die Datei existiert und es sich um eine lesbare/schreibbare Datei handelt.	Die Datei konnte nicht gelesen werden	Stelle sicher, dass die nötigen Rechte zum lesen der Datei vorhanden sind

1.4.3 Spielstart

Fehlermeldung	Fehlergrund	Fehlerbehebung
Das erstellte Level ist invalide. Stelle sicher, dass genau eine Tür und ein Startfeld vorhanden ist	Es muss genau eine Tür und ein Startfeld in dem Level vorhanden sein	Füge dem Level genau eine Tür und einen Start hinzu
Fehlerhafte Anweisungen! Eine Prozedur darf sich nicht selbst aufrufen.	Endlosschleife, weil sich eine Prozedur selbst aufruft	Vergewissere dich, dass sich keine der zwei Prozeduren selbst aufruft
Fehlerhafte Anweisungen! Die Prozeduren dürfen sich nicht gegenseitig aufrufen.	Endlosschleife, da sich die Prozeduren gegenseitig aufrufen	Vergewissere dich, dass sich die beiden Prozeduren nicht gegenseitig aufrufen

1.4.4 Level lösen

Fehlermeldung	Fehlergrund	Fehlerbehebung
Verloren! Es wurden keine Anweisungen angegeben.	Keine Anweisungen in den Anweisungsblöcken	Füge Anweisungen zu den Anweisungsblöcken hinzu und starte das Level erneut
Verloren! Die Tür ist nicht in Reichweite.	Es wurde eine „Exit”-Anweisung aufgerufen, obwohl der Roboter nicht direkt vor einer Tür steht	Rufe „Exit” nur dann auf, wenn sich der Roboter unmittelbar vor einer Tür befindet und diese anguckt
Verloren! Das nächste Feld ist blockiert und kann nicht betreten werden.	Es wurde eine „Gehen”-Anweisung aufgerufen, obwohl das Feld vor dem Roboter nicht begehbar ist	Der Roboter kann nur auf leere Felder und auf Münzen laufen
Verloren! Über das nächste Feld kann nicht gesprungen werden.	Es wurde eine „Springen”-Anweisung aufgerufen, obwohl nicht über das nächste Feld gesprungen werden kann	Der Roboter kann nur über einen Abgrund springen
Verloren! Das Feld hinter dem Abgrund kann nicht betreten werden.	Hinter dem Abgrund befindet sich ein Feld, auf dem der Roboter nicht landen kann	Der Roboter kann nur auf leeren Feldern oder Münzen landen
Verloren! Es wurden nicht alle Münzen eingesammelt	Vor dem Aufruf von „Exit” wurden nicht alle Münzen eingesammelt	Der Roboter muss alle Münzen einsammeln, bevor er durch den Ausgang gehen kann
Verloren! Um das Level zu lösen muss der Ausgang als letzte Anweisung benutzt werden.	Die letzte Anweisung des Programms war nicht „Exit”	Das Level muss mit einer „Exit”-Anweisung enden
Verloren! Nach Betreten des Ausgangs dürfen keine weiteren Anweisungen mehr aufgerufen werden.	Es wurden weitere Anweisungen nach „Exit” aufgerufen.	Nach dem ersten „Exit” darf keine weitere Anweisung mehr folgen

1.4.5 Level prüfen

Fehlermeldung	Fehlergrund	Fehlerbehebung
Das Level invalide und daher nicht lösbar. Stelle sicher, dass genau ein Start und eine Tür existiert.	Es wurde ein invalides Level auf seine Lösbarkeit geprüft	Achte darauf, dass das Level vor dem Prüfen lösbar sein muss, d.h. es muss genau eine Tür und einen Start geben.
Das Level ist nicht lösbar, da nicht alle Münzen erreichbar sind.	Mindestens eine Münze ist nicht erreichbar	Ändere das Level, sodass alle Münzen erreichbar sind
Das Level ist nicht lösbar, da die Tür nicht erreichbar ist.	Die Tür ist blockiert und somit nicht vom Roboter erreichbar.	Ändere das Level, sodass die Tür vom Roboter erreichbar ist
Das Level benötigt zu viele Anweisungen um gelöst zu werden. Daher ist keine korrekte Aufteilung in Programm und Prozeduren möglich. Denke daran, dass nur maximal 12 Programmanweisungen und 8 Prodezuranweisungen erlaubt sind.	Der Lösungsalgorithmus konnte keine Lösung finden, die innerhalb der maximalen Längen der Anweisungsblöcke liegen.	Ändere das Level, sodass es lösbar ist

2 Programmiererhandbuch

2.1 Entwicklungskonfiguration

Die folgende Tabelle 2.1 enthält die Konfigurationen mit den das Programm entwickelt wurde:

Komponente	**Version**
Microsoft Windows 10 Home	10.0.19043 Build 19043
Java Runtime Environment Java Development Kit Language Level	11 11.0.4 9

Tabelle 2.1: Entwicklungskonfiguration

Dabei wurden folgende externe Bibliotheken eingebunden und verwendet (Tabelle 2.2):

Bibliothek	**Version**	**Verwendungszweck**
GSON	2.8.5	JSON-Parser
JavaFX	11.0.2	Grafische Benutzeroberfläche

Tabelle 2.2: Bibliotheken

2.2 Problemanalyse und Realisation

Im folgenden werden die Kernaspekte des Spiels „Escape-Bot” aufgearbeitet und mehrere Realisationsmöglichkeiten für diese dargestellt und diskutier. Daraufhin wird die im Projekt gewählte Realisationsmöglichkeit einmal genauer beschrieben.

2.2.1 GUI-Darstellung des Spielfelds und der Anweisungsblöcke

Problemanalyse

Das Spielfeld besteht aus 8x8 Feldern, die jeweils einen eigenen Feldtypen haben können. Der Spieler muss in der Lage sein auf einzelne Felder des Spielfelds zu klicken um an dieser Stelle einen ausgewählten Feldtypen zu setzen. Gleiches Konzept gilt für die Anweisungsblöcke für das Programm und die zwei Prozeduren. Dabei besteht das Programm aus 12 Anweisungen, welche in einem Block von 4x3 angezeigt werden können, während Prozeduren mit 8 Anweisungen in einem Block von 4x2 dargestellt werden können. Genau wie bei dem Spielfeld, kann der

Nutzer auf die einzelnen Zellen der Anweisungsblöcke klicken und so eine Anweisung auf die entsprechende Zelle setzen.

Realisationsanalyse

Für das Spielfeld und die Anweisungsblöcke eignet sich jeweils eine GridPane. Durch dessen Row- und Column-Constraints kann hier einfach ein Feld bestehend aus Row x Column Zellen erstellt werden. Außerdem können Klicks des Nutzers, auf ein GridPane, recht einfach in eine Koordinate umgewandelt werden. Somit lässt sich ziemlich genau bestimmten auf welche Zelle des Grids der Nutzer gerade geklickt hat.

Bei der Darstellung der einzelnen Zellen in dem Spielfeld und den Anweisungsblöcken bleibt etwas mehr Spielraum. Hier würden sich folgende Komponenten anbieten:

1. Buttons: Die einzelnen Feldtypen könnten als Emoji's dargestellt werden. Durch die Nutzung von Buttons hätte man weiterhin den Vorteil, dass auf einen Klick des Nutzers reagiert werden kann. Problematisch wird es jedoch, wenn Bilder in den Button geladen werden sollen, hier wäre dann auf jeden Fall der zweite Ansatz von Vorteil.

   Außerdem ist es wie im vorherigen Absatz beschrieben, schon recht einfach möglich die geklickte Koordinate aus einem Grid auszulesen. Daher würde hier die Nutzung einzelner Buttons für die Reaktion auf einen Klick nur eine weitere Elementebene darstellen, wobei von den Eigenschaften der Buttons außerhalb der Reaktion auf den Klick nicht weiter benötigt wird.

2. ImageViews: Die zweite Variante wäre das Nutzen von ImageViews. Hier könnten dann die von der Aufgabenstellung mitgelieferten Bilder genutzt werden, um die einzelnen Feldtypen und Anweisungen darzustellen. Da die geklickte Koordinate aus dem Grid errechnet werden kann ist hier auch kein EventHandler, wie bei den Buttons, notwendig.

Realisationsbeschreibung

Das Spielfeld wird durch ein GridPane mit 8 Zeilen und Spalten repräsentiert. Innerhalb jeder Zelle des Grids befindet sich ein ImageView, welches das Bild für eines der Feldtypen enthält. Die ImageViews werden dabei an die Größe ihrer Zelle angepasst, wodurch sie sich auch beim vergrößern/verkleinern des Spielfelds mit skalieren. Das Spielfeld behält durch eine preserveRatio Option auch bei anderer Skalierung immer ihre Längen- und Breitenverhältnisse, wodurch jedes Bild auf dem Spielfeld immer schachbrettförmig und in gleicher Größe dargestellt wird.

Die Anweisungsblöcke sind ähnlich aufgebaut. Auch sie werden durch jeweils ein GridPane der Größe 4x4 (Programm) oder 4x2 (Prozeduren) dargestellt. Genau wie die Zellen des Spielfelds, enthalten die Anweisungsblöcke ein ImageView in jeder Zelle, welches mit einem Bild der aktuell zugewiesen Anweisung versehen wird.

Außerdem ist zwischen jeder Zelle ein horizontaler und vertikaler Abstand zu den anderen Zellen, damit der Nutzer die einzelnen Zellen klar voneinander trennen kann und die Unterschiede der einzelnen Felder deutlicher hervorstechen

Sobald auf eine Zelle auf dem Spielfeld oder in einem der Anweisungsblöcke gedrückt wird, wird die Koordinate der aktuellen Zelle berechnet und das Ergebnis an die Logik weitergegeben. Schlussendlich wird dann nach Abarbeitung der Logik die entsprechend ausgewählte Anweisung der der ausgewählte Feldtyp als Bild auf dem Grid dargestellt.

2.2.2 GUI-Darstellung der Auswahl an Anweisungen und Feldtypen

Problemanalyse

Der Nutzer soll die Möglichkeit haben im Spielmodus zwischen verschiedenen Anweisungen auswählen zu können und diese in die Anweisungblöcke zu setzen. Des weiteren soll statt einer Auswahl an Anweisungen im Editor-Modus eine Auswahl an Feldtypen existieren, die der Nutzer auf das Spielfeld setzen kann. Diese Auswahl soll sich in der Mitte des Spielfeldes befinden, wobei die einzelnen Anweisungen/Feldtypen vertikal untereinander angeordnet sind.

Realisationsanalyse

Für die grundlegende Struktur würde sich entweder wieder ein GridPane oder aber eine VBox eignen. Würde man diesen Programmteil mit einem GridPane realisieren, dann würde es sich wegen der vertikalen Anordnung um ein 1x7 oder 1x6 Grid handeln. Da die verschiedenen Größen jedoch in einem GridPane selbst angepasst werden müssen, eignet es sich in diesem Fall nicht.

Die zu präferierende Variante ist hier eine VBox. Diese ist implizit auch wie ein 1xAnzahl Grid zu verstehen, übernimmt jedoch die Skalierung und die Ausrichtung der Elemente von alleine. Außerdem ist eine VBox deutlich flexibler was die Anzahl der enthaltenden Elemente angeht. Dies ist für unser Programm von Vorteil, da so im Spielmodus die Anweisungen in einer VBox angezeigt werden können, während die VBox im Editor-Modus mit Feldtypen befüllt wird. Damit kann der Platz im Programm wiederverwendet werden und es kommt nie zu dem Problem, dass wir uns im Spielmodus befinden, der Nutzer aber Feldtypen auswählt.

Für die einzelnen Anweisungen/Feldtypen einen sich wieder ImageViews am besten, da so die mitgelieferten Bilder verwendet werden können und die Größe der Bilder an die Größe der VBox (geteilt durch die Anzahl der Elemente) angepasst werden kann. Dies hat zudem den Vorteil, dass kenntlich gemacht werden kann, welches der Elemente gerade ausgewählt ist. Dies ist möglich, da jedem ImageView ein Effekt gegeben werden kann, welcher Beispielsweise einen Schatten um das Element erzeugt und es so kennzeichnet.

Realisationsbeschreibung

Die Auswahl der Anweisungen und Feldtypen wird jeweils als VBox mit den möglichen Anweisungen/Feldtypen als ImageView dargestellt. Da es sich um eine VBox handelt, sind die einzelnen Anweisungen/Feldtypen standardmäßig untereinander aufgereiht. Durch die Nutzung von ImageViews kann, wie im vorherigen Absatz erwähnt, ein Effekt auf das angeklickte Element angewandt werden, welcher das jeweilige Element mit einem roten Rand versieht. Dieser wird also über ein einfaches Klick-Event hinzugefügt.

Damit Für die Logik und vor allem für Folgeklicks auf das Spielfeld oder die Anweisungsblöcke bekannt ist, welche Anweisung oder welcher Feldtyp sich hinter dem aktuellen Element befindet, wird jedem ImageView userData mitgegeben. Darin kann für jedes Element gespeichert werden um welchen logischen Feldtypen und welche logische Anweisung es sich handelt.

2.2.3 Weitere zentrale GUI Elemente

Problemanalyse

Neben dem Spielfeld, den Anweisungsblöcken und der Auswahl an Anweisungen/Feldtypen muss es möglich sein ein Spiel zu starten oder zu stoppen, ein Level auf seine Lösbarkeit zu prüfen und zwischen Editor- und Spielmodus zu wechseln.

Realisationsanalyse

Da es sich bei diesen Anforderungen um klare Hauptfunktionalitäten handelt, sollten diese jederzeit in dem Programm sichtbar sein. Da jede dieser Anforderungen eine bestimmte Aktion ausführen soll, bietet sich hier die Nutzung von Buttons an. Diese können im Controller jeweils mit einer Aktion belegt werden, die ausgeführt werden soll, wenn auf sie geklickt wird. Außerdem lassen sich Buttons mit einer einfachen Abfrage zu einer Toggle-Funktionalität umbauen, wodurch zum Beispiel der Wechsel zwischen Editor und Spiel nur mithilfe eines Knopfes dargestellt werden kann.

Realisationsbeschreibung

Realisiert werden die oben genannt Anforderungen wie erwähnt jeweils als eigener Button. Dabei besitzt der Button zum wechseln zwischen Editor und Spiel eine Toggle-Funktionalität, wodurch nur ein Button für zwei Funktionalitäten benötigt wird. Ein Klick auf den entsprechenden Knopf ruft im Controller eine EventHandler-Methode auf, welche dann die gewollte Aktion in Gange bringt.

2.2.4 Menü für weitere Interaktionen

Problemanalyse

Zusätzlich zu den anderen UI Komponenten, ist ein Menü gefordert, von welchem aus Beispiellevel geladen oder auch eigene Level geladen und gespeichert werden können. Außerdem soll es einen Menüpunkt zum anpassen der Animationsgeschwindigkeit geben, in dem zwischen mindestens drei Animationsgeschwindigkeiten gewechselt werden können soll. Zudem soll es die Möglichkeit geben die Animationen komplett auszuschalten

Realisationsanalyse

Da es sich um ein typisches Programmmenü handelt, bietet es sich an dieses an den oberen Teil des Fensters zu platzieren. Durch die Nutzung von Menügruppen ist es außerdem möglich mehrere Interaktionsmöglichkeiten unter einer Menüoption zu „verstecken', wodurch die Leiste beim Starten des Programms recht überschaubar bleibt. Außerdem kann den einzelnen Menüoptionen ein eigenes Tastenkürzel zugewiesen werden, wodurch die dahinter stehende Funktionalität noch schneller ausgeführt werden kann. Ein weiterer Vorteil von JavaFX Menüs ist die Integration von RadioButtons. Somit können unter einem Menüreiter mehrere Optionen erstellt werden, von denen nur eine zur gleichen Zeit ausführbar ist. Dies eignet sich insbesondere gut für das Einstellen der Animationsgeschwindigkeit, da hier immer nur eine Geschwindigkeit zur Zeit aktiv sein kann.

Realisationsbeschreibung

Das Menü wird im oberen Teil des Programms angezeigt und enthält drei Menüreiter. Der erste Menüreiter „Level" enthält Menüoptionen zum erstellen eines neuen, leeren Levels, zum laden von Beispielleveln und zum laden und speichern eigener Level. Außerdem ist eine weitere Menüoption enthalten, mit der das Programm direkt geschlossen werden kann.

Der zweite Reiter „Geschwindigkeit" Enthält eine RadioToggleGroup bestehend aus drei verschiedenen Geschwindigkeiten und der Möglichkeit Animationen komplett auszuschalten. Da es sich jeweils um RadioButtons handelt kann hier also nur eine Option zeitgleich ausgewählt sein.

Sobald eine der Geschwindigkeiten ausgewählt wird, wird sie von dem Controller an die JavaFXGUI weitergeleitet und dort als neue Geschwindigkeit gespeichert. Da die Animationen jedoch alle am Stück erstellt werden, ist zwar das Wechseln der Geschwindigkeit während einer Animation möglich, hat aber erst Auswirkungen auf die nächste Animation.

2.2.5 Angezeigte Texte auf der GUI

Problemanalyse

In der GUI wird eine Vielzahl von Texten gepflegt. Dazu gehören die Labels aller Menüoptionen und Buttons, sowie jede mögliche Statusmeldung. Diese sollen ausschließlich in der GUI erstellt werden.

Realisationsanalyse

Dabei bieten sich zwei Vorgehensweisen an, wie die Texte in der GUI gepflegt werden können:

1. Die verschiedenen Texte können in dem FXML Dokument, dem FXMLController und der JavaFXGUI direkt gepflegt werden. Das hat den Vorteil, dass kein Extra System vorhanden ist die Texte an die richtige Stelle zu bringen, da sie dort einfach direkt statisch hinterlegt werden. Ein Nachteil ist jedoch bei größeren Projekten, dass die einzelnen Texte so in mehreren Dateien verstreut direkt im Quellcode stehen, was das auffinden spezifischer Textpassagen etwas aufwendiger macht

2. Eine zweite Variante wäre die Nutzung einer Zentralen Lokalisationsdatei, welche alle Texte enthält, die in der gesamten GUI auftauchen. Dies hat den Vorteil, dass alle Texte an einem Ort zu finden sind und es somit nur eine Stelle gibt an der eine Änderung gemacht werden muss. Außerdem ist es so möglich die gleichen Texte mehrfach zu nutzen anstatt sie mehrfach zu schreiben. Des weiteren könnte man so einfach eine weitere Lokalisationsdatei hinzufügen, welche die Texte in eine andere Sprache übersetzt und könnte so auch zwischen Sprachen wechseln, wobei die Atttibutschlüssel für die einzelnen Texte, und somit der Quelltext, immer gleich bleiben.

Realisationsbeschreibung

Die Texte werden also in der GUI zentral durch eine .properties Datei gepflegt und müssen dann mit ihrem entsprechenden Attributschlüssel an der jeweiligen Stelle im Quellecode eingefügt werden. Dazu muss in der Hauptklasse ein ResourceBundle erstellt werden, welches auf ein Paket zeigt, das eine .properties Datei mit den jeweiligen Strings enthält. Dieses ResourceBundle kann dann an den FXMLLoader übergeben werden und kann so in der initialize() Methode des FXMLControllers genutzt werden. Will man nun im FXML Dokument, dem Controller oder der JavaFXGUI einen Text hinterlegen, so kann dieser in der .properties-Datei angelegt werden. Auf den Text kann dann mit dem ResourceBundle über dessen Attributschlüssel zugegriffen werden.

2.2.6 Kommunikation zwischen Contoller, Logik und GUI

Problemanalyse

Innerhalb des Projektes muss es ein Möglichkeit geben, dass der Nutzer eine Aktion auf der Benutzeroberfläche ausführt, wodurch etwas in der Logik des Programms ausgeführt wird. Außerdem soll es möglich sein, dass nach Beendigung einer Operation in der Logik eine Ausgabe auf der Benutzeroberfläche erfolgt.

Realisationsanalyse

Das es sich um ein JavaFX-Projekt handelt wird allgemein das Model-View-Controller Pattern verwendet. Hierbei gibt es einen Controller, welcher auf Interaktionen des Nutzers reagiert. Das Model hingegen steht für die interne Repräsentation des Systems, also für die Logik dahinter. Der dritte Baustein ist dann die View, also die Änderung der Benutzeroberfläche. Dabei können Model, View und Controller auf unterschiedliche Weise miteinander kommunizieren.

1. Linear: Die drei Komponenten können linear miteinander Kommunizieren. Dabei reagiert der Controller auf Interaktionen des Benutzers und liefert diese dann an das Model weiter. Dort wird dann die entsprechende Logik ausgeführt, worauf das Model die Informationen an die View weitergibt. Nachteil an diesem linearen Vorgehen ist jedoch, dass das Model auch Aufrufe an die View weiterreichen muss, die in der Logik selber gar nichts bewirken.

2. Zirkulär: Die andere Variante ist, dass der Controller sowohl mit dem Model kommunizieren kann, sofern die Logik von der Interaktion mit betroffen ist oder aber direkt mit der View kommuniziert. Das Model kann währenddessen weiterhin nur von dem Controller Aufrufe empfangen und diese an die View weiterleiten. Dies hat den Vorteil, dass keine Aufrufe mehr an die Logik weitergeleitet werden, welche nur Auswirkungen auf die Benutzeroberfläche haben.

Realisationsbeschreibung

Im Projekt wird die zweite Variante verwendet. Dabei stellt der FXMLController den Controller, die GameLogic das Model und die JavaFXGUI die View dar. Somit kann der FXMLController entweder Aufrufe an die GameLogic weiter leiten, welche wiederum mit der JavaFXGUI kommuniziert, oder der FXMLController leitet Aufrufe direkt an die JavaFXGUI weiter, sofern diese Aufrufe keine Auswirkungen für die Logik haben. Außerdem kommuniziert die GameLogic nur über einen GUIConnector mit der JavaFXGUI, weshalb die Benutzeroberfläche auch ausgetauscht werden kann und das Spiel weiterhin funktioniert, sofern die neue Benutzeroberfläche den GUIConnector implementiert.

2.2.7 Interne Darstellung eines Levels

Problemanalyse

Ein Level besteht aus einem Spielfeld und der Rotation des Roboters. Das Spielfeld ist dabei ein 8x8 Feld und muss Informationen darüber enthalten, in welcher Zelle sich welcher Feldtyp befindet. Außerdem muss es in der Logik möglich sein einzelne Felder oder sogar das gesamte Spielfeld auszutauschen. Zusätzlich ist für die Logik wichtig, in welche Richtung der Roboter gerade guckt.

Realisationsanalyse

Intern gibt es mehrere Möglichkeiten ein Spielfeld darzustellen. Man könnte es

1. als 2-dimensionales Array darstellen. Dies hat den Vorteil, dass ein indizierter Zugriff für einzelne Zellen sehr schnell möglich ist und die Referenz auch ausgetauscht werden kann um ein komplett neues Level zu erstellen. Außerdem kann durch den indizierten Zugriff auch durch alle Felder gegangen werden um bestimmte Feldtypen ausfindig zu machen.
2. als 2-dimensionale Liste darstellen. Dies hätte den Vorteil, dass schon viele Standardmethoden für Listen vorhanden sind, was bei Arrays nicht der Fall ist. Jedoch hat man im Gegensatz zu Arrays auch einen deutlichen Performance-Nachteil, da Listen mit ihren Funktionen einen deutlich größeren Overhead mit sich bringen und Zugriffe auf bestimmte Indizes nur mit schlechterer Laufzeit möglich sind

Für die Darstellung der Bot Rotation und des Feldtypen kommen auch zwei unterschiedliche Möglichkeiten zur Geltung:

1. Zahl: Die Bot Rotation könnte als Zahl von 0 (Norden) bis 3 (Westen) dargestellt werden. Gleichermaßen könnte man die Feldtypen von 0 (Abgrund) bis 5 (Mauer) darstellen. Vorteil dabei ist, dass Zahlen sehr leichtgewichtig sind und dementsprechend vor allem in diesem Zahlenbereich nur eine sehr geringe Speicherauslastung nötig ist. Nachteil ist jedoch, dass einfache Zahlenrepräsentationen im Code typischerweise nicht sehr sprechend sind und so spätestens in Unit-Tests Beispiellevel nicht sehr leicht zu lesen sind.

2. Enum: Die Bot Rotation könnte auch als Enum dargestellt werden, wobei jede Himmelsrichtung einem Enum-Wert entspricht. Das Selbe gilt für die Feldtypen, wobei jeder Feldtyp als ein Enum-Wert dargestellt werden kann. Dies hat den Vorteil, dass es sich intern weiterhin um Zahlen handelt, die aber im Code viel sprechender sind als reine Zahlenrepräsentationen. Außerdem können Enums genau wie normale Klassen weiter Parameter bekommen und es kann gekapselte Methoden für Operationen auf den Enums geben. So kann beispielsweise innerhalb des Enums für die Bot Rotation der Roboter rotiert werden, anstatt das dies eine extra Methode in der Logik ist.

Realisationsbeschreibung

Die Rotation des Bots und die verschiedenen Feldtypen werden in der Logik wegen im letzten Absatz genannten Vorteilen als Enums dargestellt. Dabei besitzt die Bot Rotation mehrere Methoden und Parameter, durch die die Arbeit mit der Rotation erleichtert wird. Ein Level wird in der Logik mit einem Spielfeld, welches ein 2-dimensionales Array aus Feldtypen ist, und einer Bot Rotation dargestellt. Zudem enthält das Level mehrere Methoden um den Roboter zu drehen oder ihn auf dem Spielfeld zu bewegen.

2.2.8 Datenstruktur der Anweisungen

Problemanalyse

In dem Spiel muss es eine Reihe an Anweisungen geben, aus denen der Nutzer welche auswählen und zu seinem Programm und zu den Prozeduren hinzufügen können muss. Dazu gehören auch Prozeduren, welche zum einen Anweisungen beinhalten, zum anderen aber auch im Programm und der jeweils anderen Prozedur als Anweisung verwendet werden können.

Realisationsanalyse

Bei der Darstellung der Anweisungen kommen zwei Konzepte in Frage:

1. Enum: Anweisungen könnten als Enums dargestellt werden.Auch wenn sie wie bereits erwähnt selber Methoden und Parameter bekommen können, ist es mit Enums nur sehr schwer möglich eine so komplexe Struktur aufzubauen, die vor allem für das Konvertieren von Anweisungen in spätere Aktionen von Nöten ist. Da es sich um unterschiedliche Anweisungen handelt, welche alle anders konvertiert werden, ist hier also die Nutzung von Enum-Werten eher ungeeignet, da bis auf den Fakt, dass es sich um Anweisungen handelt, fast keine Gemeinsamkeiten vorhanden sind die sich in Methoden ausdrücken lassen. Ein weiterer Nachteil ist, dass vor allem im Rahmen der Konvertierung jedes Mal

eine Unterscheidung gemacht werden muss um welche Ausprägung des Enums es sich gerade handelt.

2. Klassen: Anweisungen können auch als eigene Klassenstruktur dargestellt werden. Dabei würde es eine abstrakte Oberklasse oder ein Interface geben, welches die Methoden implementiert, die jede Anweisung gemeinsam hat. Außerdem könnten das Programm und die Prozeduren dann als Listen des abstrakten Obertypen realisiert werden, wodurch sie alle möglichen Anweisungen enthalten können. Des weiteren könnte so anders als bei Enums für jede Unterklasse eine eigene Konvertierungslogik erstellt werden, wodurch die GameLogic etwas kleiner wird.

Realisationsbeschreibung

Anweisungen werden als eigene Klassenstruktur dargestellt. Dabei haben sie alle eine gemeinsame Oberklasse Instruction, aus der sie eine Klasse zur späteren Konvertierung in Aktionen erben. Somit kann, egal um welche Anweisung es sich handelt, in der Logik immer die Selbe Methode auf den einzelnen Anweisungen aufgerufen werden. Dabei wird abhängig von der konkreten Anweisung immer andere Ablauflogik zur Konvertierung der Anweisungen realisiert.

Außerdem nutzen die einzelnen Anweisungen das sogenannte Singleton-Muster. Das bedeutet es handelt sich um Klassen, von denen es im gesamten Programm nur eine Instanz geben kann. Realisiert wird dies durch einen Parameter, der die Klasse über einen privaten Konstruktor aufruft und nur über einen Getter nach außen gegeben werden kann. Somit kann in dem Getter sichergestellt werden, dass es immer nur diese eine Instanz der Klasse geben kann. Dies hat den Vorteil, dass beispielsweise bei einer Folge von fünf Lauf-Anweisungen, nur eine Instanz der Klasse Laufen benötigt wird und nicht pro Anweisung eine neue Instanz angelegt werden muss.

Auch die Prozeduren werden als eigene Klassen in der Klassenhierarchie dargestellt. Anders als die anderen Anweisungen können sie jedoch nicht als Singleton realisiert werden, da es von den Prozeduren zwei verschiedene Ausprägungen geben kann. Außerdem gibt es die Möglichkeit diese Ausprägungen im Nachhinein wie bei dem Programm mit Anweisungen zu füllen, weshalb sie unabhängig voneinander existieren müssen. Da die beiden Prozeduren jedoch in ihrer Funktionalität identisch sind, bekommen sie eine abstrakte Oberklasse in der ihre gemeinsame Funktionalität schon vorimplementiert wird. Somit unterschieden Sie sich nur in ihren Referenzen und den Anweisungen, die sie beinhalten.

2.2.9 Unterscheidung zwischen Operationen für Programm und Prozeduren

Problemanalyse

Sowohl zum Programm als auch zu den Prozeduren soll es möglich sein, Anweisungen hinzuzufügen, welche mit vorhandenen auszutauschen und welche zu löschen. Dabei sollten Redundanzen in der Implementierung vermieden werden.

Realisationsanalyse

Zu dieser Anforderung gibt es mehrere Realisierungsmöglichkeiten. Man könnte

1. für jeden Anweisungsblock, also Programm, Prozedur 1 und Prozedur 2 eine separate Implementierung vorsehen, die sich insgesamt nur in der Liste unterscheidet zu der Anweisungen hinzugefügt oder von der welche gelöscht werden. Dies hätte den Nachteil, dass vor allem bei den Prozeduren Programmcode dupliziert wird, da sie sich wirklich nur noch in ihrer Liste unterscheiden.
2. für jeden Anweisungsblock zwar eine eigene Methode im FXMLController erstellen, die bei einem Klick auf den jeweiligen Anweisungsblock ausgeführt wird, die Implementierung dieser Methode jedoch in einer gemeinsamen Methode zusammenfassen. Dies ist beispielsweise mit der Nutzung von Consumern möglich. Dabei handelt es sich um ein Callback-System bei dem einer Methode eine Methode als Parameter mitgegeben werden kann, welche dann wiederum in der Methode mit eigenen Paramtern aufgerufen werden kann. Ein BiConsumer beispielsweise ist eine Methode, die als Parameter übergeben werden. Möchte eine Methode den übergebenen BiConsumer nun aufrufen, so kann dies über parameter.accept() getan werden, wobei als Paramter von accept() die nötigen Parameter der übergebenen Methode aufgerufen werden können.

Realisationsbeschreibung

Um Redundanzen im Code zu vermeiden wird an mehreren Stellen das Consumer-Konzept verwendet. Beispielsweise gibt es jeweils eine Methode im FXMLController, die aufgerufen wird, wenn der Nutzer auf einen der Anweisungsblöcke klickt. Dabei haben diese Methoden jedoch keine eigene Implementierung, sondern rufen lediglich die allgemein Methode auf und übergeben ihr einen BiConsumer, welcher die Methode zum hinzufügen einer Anweisung zur Liste darstellt, und einen Consumer, welcher die Methode zum löschen einer Anweisung aus der Liste darstellt. Die Methode selber ruft dann einfach an gegebener Stelle die .accept()-Methode des als Parameter übergebenen Callbacks auf und kann so zu der richtigen Liste Einträge hinzufügen oder auch wieder heraus löschen.

2.2.10 Datenstruktur für Aktionen

Problemanalyse

Während Anweisungen die Operationen darstellen, die ein Nutzer in seinem Programm aufrufen kann, gibt es noch weitere Aktionen, die nicht direkt von dem Spieler aufgerufen werden können, aber auch wichtig für den Ablauf des Spiels sind. Dies betrifft vor allem spätere Animation und die korrekte Meldung über Erfolg oder Misserfolg eines Levels. Dazu gehört beispielsweise das Aufsammeln von Münzen, was implizit bei einem Sprung oder einem Schritt passiert, oder das Informieren über ein gewonnenes oder verlorenes Level (inklusive der richtigen Nachricht, wenn das Level nicht korrekt absolviert wurde).

Realisationsanalyse

Aktionen könnten generell auf drei Weisen implementiert werden. Man könnte:

1. sie als einfache Zahlen repräsentieren. Somit hätte man zwar eine einfache, nicht sehr speicherintensive Repräsentation, diese hätte jedoch nicht sehr viel Semantik.
2. sie als Enum repräsentieren. Dies hat den Vorteil, dass es sich wie bei Zahlen um ein nicht sehr speicherintensives Medium handelt, wobei Enums zusätzlich eine sehr gute Semantik aufweisen durch die Namen der einzelnen Enum-Werte.
3. sie als Klassen darstellen. Diese Vorgehensweise bringt in unserer Anforderung jedoch recht wenig, da die einzelnen Klassen keine wirkliche eigene Logik besitzen. Somit wäre das Erstellen einer Klasse pro Aktion ein sehr starker Overhead, der nicht gebraucht wird.

Realisationsbeschreibung

Die Aktionen werden in dem Projekt als Enum repräsentiert. Dabei enthält das Enum die einzelnen Anweisungen, wobei die Prozeduren nicht enthalten sind, da diese im Endeffekt auch einfach aus einer Folge der anderen Anweisungen bestehen. Zusätzlich enthält das Enum eine Repräsentation für ein gewonnenes Level und eine Vielzahl an Fehlerindikatoren, welche eine genaue Fehlermeldung bei nicht erfolgreichem Level ermöglichen.

2.2.11 Konvertierung von Anweisungen in Aktionen

Problemanalyse

Da es neben Anweisungen auch noch die im Unterabschnitt 2.2.10 beschriebenen Aktionen gibt, muss es zusätzlich eine Möglichkeit geben, die vom Benutzer genutzten Anweisungen in Aktionen umzuwandeln. Dabei würden beispielsweise aus einer Lauf-Anweisung über eine Münze die zwei Aktionen Laufen und Münze einsammeln, entstehen.

Realisationsanalyse

Die Anweisungen liegen in einer eigenen Klassenstruktur vor. Um diese Konvertieren zu können, könnte man mehrere Datenstrukturen und demnach Vorgehen verwenden. Man könnte

1. Alle Programmanweisungen in ihrer Listenrepräsentation behält und somit iterativ diese durchgehen. Dabei müsste in der Konvertierungsmethode jeder Unterklasse der Klassenstruktur eine Liste an Aktionen zurückgeben. Dies ist zum Beispiel beim Einsammeln einer Münze nötig, da hier aus einer Anweisung zwei Aktionen entstehen können. Außerdem kann es in dem Programm auch zur Ausführung einer Prozedur kommen, in der auch mehrere Aktionen erstellt werden (für jede Anweisung in der Prozedur). Ein Vorteil wäre es, dass die Programmanweisungen und die Prozeduren schon in Listenform vorliegen, hier wäre also keine Umwandlung in eine andere Datenstruktur notwendig. Nachteil ist jedoch, dass durch das iterative Vorgehen zusätzliche Logik erstellt werden muss, um den

Abbruch der Konvertierung einzuleiten, sobald eine Anweisung nicht ausgeführt werden konnte.

2. Eine andere Variante wäre die Nutzung der Java Streams API. Hierbei handelt es sich um den funktionalen Ansatz von Java, der in Java 8 hinzugefügt wurde. Allgemein werden Streams dazu verwendet um eine Sequenz an Elementen einzulesen und auf diesen bestimmte Funktionen auszuführen. Für die Konvertierung von Anweisungen in Aktionen hat dies den Vorteil, dass die Streams API eine Vielzahl vordefinierter Funktionen mit sich bringt, mit denen die Logik des Zusammenführens verschiedener Aktionen und die Logik des Abbruchs mit vordefinierten Funktionen deutlich einfacher erstellt werden kann, als würde man dies alles manuell tun, wie im letzten Ansatz. Außerdem kann jede Liste einfach in einen Stream umgewandelt und ein Stream jederzeit wieder zu einer Liste gemacht werden. Nachteil der Java Streams API ist jedoch, dass es sich um eine sequenzielle Abarbeitung eines Elementstroms handelt, weshalb das gerade ausgeführte Element keine Information darüber hat, welches Element vor ihm kam oder welches Element als nächstes kommt. Dies bringt im Gegensatz zur Nutzung von Listen das Problem, dass der Spezialfall von Anweisungen nach einem „Exit" gesondert behandelt werden muss und nicht mit in der restlichen Logik abgefangen werden kann.

Neben der Entscheidung, welche Datenstruktur zur Konvertierung verwendet werden soll, muss auch noch entschieden werden, in welche Datenstruktur tatsächlich konvertiert wird. Hier sind auch wieder mehrere Entscheidungen möglich:

1. Die einfachste Möglichkeit wäre es, einfach eine Liste an Aktionen zurück zu geben. Für das Animieren der Aktionen reicht jedoch das reine wissen über die momentan auszuführende Aktion noch nicht aus. Daher ist diese minimale Darstellung nicht für die Weiterarbeit insbesondere im Bereich der Animationen geeignet.

2. Eine weitere Möglichkeit wäre es, eine eigene Datenstruktur zu erstellen, die neben der jeweiligen Aktion auch die aktuelle Bot Rotation und Bot Position speichert. Somit sind bei der Erstellung der Animation zusätzlich Informationen darüber gegeben, wo sich der Roboter gerade befindet, wenn die Aktion ausgeführt werden soll. Damit wird die Liste zwar deutlich Komplexer, da es sich nicht mehr um eine Liste aus Enum-Werten, sondern um eine Liste aus Instanzen einer eigenen Klasse handelt. Jedoch sind diese Zusatzinformationen dringend notwendig um eine gute Animation gewährleisten zu können.

3. Die dritte Möglichkeit baut auf der zweiten auf, erweitert diese jedoch noch um ein weiteres Attribut. Hier wird neben der Bot Rotation und der Bot Position auch noch eine Liste für jede Aktion migeführt, die zeigt, in welchem Anweisungsblock die jeweilige Aktion gerade ausgeführt wird. Dadurch kann in der später ausgeführten Animation zusätzlich die aktuell ausgewählte Anweisung in dem jeweiligen Anweisungsblock angezeigt werden, in dem sie sich befindet.

Realisationsbeschreibung

Die Konvertierung von Anweisungen in Aktionen ist durch die Nutzung von Streams realisiert. Dadurch kann in der Logik recht einfach durch die Programm Anweisungen gegangen werrden, wobei über eine .flatMap() Operation jede einzelne Anweisung zu einem Stream an Moves umgewandelt wird. Ein Move ist dabei eine eigene Datenstruktur, in der neben der in diesem

Schritt auszuführenden Aktion auch noch die aktuelle Bot Position, Bot Rotation und eine Liste an Anweisungsblöcken, in denen die Aktion ausgeführt wird, befinden. Als Abbruchlogik dient dann eine von der Streams API definierte .takeWhile() Operation, welche solange Elemente aus einem Stream aufnimmt, bis ein bestimmtes Predikat erfüllt wird. Die Abbruchbedingung ist in dem Fall das Auftreten einer Fehler- oder Erfolgsaktion. Dies beinhaltet das gewinnen der Runde oder das Verlieren,weil zum Beispiel versucht wurde auf ein blockiertes Feld zu laufen.

Die .takeWhile() Operation ist jedoch exklusive definiert. Das heißt, wenn eine solche Aktion gefunden wurde, die ein Resultat repräsentiert, wird die Aufnahme in den Stream abgebrochen und auch das Resultat wird nicht mehr aufgenommen. Deshalb ist an diesem Schritt etwas mehr Logik in dem takeWhile() nötig, damit dieses als inklusives takeWhile() funktioniert.

Nachdem der komplette Stream abgearbeitet wurde und demnach alle möglichen Anweisungen in Aktionen konvertiert wurden, kommt es zu dem schon inder Realisationsanalyse angesprochenen Nachteil von Java Streams. Hier muss der Stream nämlich nach der Konvertierung in eine Liste noch einmal auf das Auftreten von weiteren Anweisungen nach einem Exit geprüft werden. Dieser Fall kann nicht berücksichtigt werden, während der Stream ausgewertet wird, da hier ein Element in dem Stream wissen müsste, was nach ihm kommt. Dementsprechend ist hier noch ein bisschen extra Logik notwendig um das Vorkommen der „Exit” Anweisung zu finden und dann zu prüfen, ob es sich wirklich um die letzte Anweisung handelt.

2.2.12 Dateien einlesen und speichern

Problemanalyse

Der Nutzer muss die Möglichkeit haben Beispiellevel oder eigene Level zu laden. außerdem muss es eine Möglichkeit geben Level abzuspeichern. Dabei muss sowohl eine geladene Datei in ein Level konvertiert werden, als auch ein vorhandenes Level in eine Datei zu konvertieren.

Realisationsanalyse

Im ersten Schritt muss der Nutzer eine Datei auswählen die er laden, oder in die er das momentane Level speichern möchte. Dazu bietet JavaFX einen FileChooser an, welcher den Dateiexplorer des Betriebssystems nutzt um eine Datei auf dem System auszuwählen und eine File-Instanz davon zurückzugeben.

Zum einlesen der Datei bietet Java mehrere Reader an. Beispielsweise kann ein FileReader genutzt werden um Daten aus einer Datei einzulesen. Dieser FileReader funktioniert jedoch beispielsweise nicht auf den Beispieldateien in dem Programm, da es sich bei diesen Beispielen nach dem bauen der .jar-Datei nicht mehr um normale Dateiformate handelt. Hier muss also ein anderer Reader wie zum Beispiel ein InputStreamReader verwendet werden. Dieser arbeitet anders als der FileReader nicht auf einer File-Instanz, sondern direkt auf einer Bytefolge, was nach der Erstellung der .jar-Datei bei den Beispielleveln nötig ist.

Zum Speichern einer Datei gibt es, wie bei Readern zum einlesen, passende Writer. Da wir das momentane Level in eine Datei auf dem Sytem speichern wollen, kann hier ein einfacher FileWriter verwendet werden.

Das eigentliche einlesen und schreiben der des JSON-Formats übernimmt dann aber GSON, ein für JSON entwickelter Parser. Die Nutzung eines solchen Parsers hat den Vorteil, dass die Logik für das einlesen von bestimmten Dateiformaten nicht von Hand geschrieben werden muss. Außerdem wäre eine eigene Implementierung eines solchen Parsers sehr zeitaufwändig und deutlich fehleranfälliger als die Nutzung eines vorhandenen Parsers.

Realisationsbeschreibung

Wie in der Realisationsanalyse beschrieben, wird ein JavaFX FileChooser verwendet um eine bestimmte Datei vom Betriebssystems des Nutzers zu lesen oder in eine Datei auf dessen Betriebssystem zu speichern.

Das lesen und übernimmt dabei GSON, ein JSON-Parser, wobei zum einlesen zum einen ein File Reader und zum anderen ein Input Stream Reader verwendet wird. Der File Reader wird verwendet, wenn es sich um externe Dateien handelt, die geladen geöffnet werden sollen, während der Input Stream Reader interne Beispiellevel lädt.

Dabei bekommt GSON als Parameter die Datei aus der gelesen werden soll und eine Klasse in die das JSON formatiert werden soll. Hierbei wird die JSON-Datei erst einmal in einen SavedState umgewandelt und noch nicht in das schlussendliche Level. Das hat den Vorteil, dass die Level Repräsentationen aus den Dateien erst einmal in ein einfacheres Zahlenformat konvertiert werden können. Danach folgt dann die Logik zur Konvertierung dieser einfachen Repräsentation in die Schlussendliche Datenhaltungsklasse (Level).

Beim speichern nutzt GSON einen einfachen File Writer, welcher das momentane Level in eine Datei schreibt.

Selber Vorgehen wie beim lesen einer JSON-Datei passiert somit auch beim schreiben.Hier wird eine Datei angegeben in die das aktuelle Level gespeichert werden soll und dann wird aus dem Level wieder ein einfacher SavedState erstellt, der dann in ein JSON-Format gebracht werden kann.

2.2.13 Animation des Roboters

Problemanalyse

Sobald der Nutzer sein Programm erstellt hat kann er das Programm starten. Hier sollen nun die von dem Nutzer eingegebenen Anweisungen animiert werden, sodass für den Nutzer jeder gemachte Schritt visualisiert wird.

Realisationsanalyse

Standardmäßig ist der Roboter ein Feldtyp und wird somit als Bild direkt in der Zelle dargestellt. Da die Animation des Roboters flüssig ablaufen soll anstatt aus dem Austauschen der Bilder in den Zellen zu bestehen, müssen JavaFX Transitions verwendet werden. Damit können beispielsweise Bilder in einer spezifizierten Zeit an eine andere Position animiert werden. Diese Animationen beziehen sich jedoch auf das gesamte Bild und würden somit den Bot und seinen Hintergrund, also die gesamte Zelle, animieren. Um dies zu verhindern kann beim Start der

Animation das Bild des Spielers auf der Zelle gegen eine normale Zelle ausgetauscht werden. Über dieses Feld kann man dann ein neues Bild des Spielers legen, welches einen transparenten Hintergrund hat. Dieser kann dann animiert werden und muss zum Schluss nur wieder auf seine Endposition gestellt werden. So ist es auch möglich weitere Animationen wie das Drehen des Spielers zu ermöglichen ohne, dass das darunter liegende Bild des Feldtypen in irgendeiner Weise beeinflusst wird.

Realisationsbeschreibung

Der Roboter auf dem Spielfeld wird also nach Start der Animation gegen ein normales Feld ausgetauscht und als neues Bild über das restliche Spielfeld gelegt. Nun kann der Roboter für jede Anweisung, die er ausführen muss, eine weitere Transition zugewiesen bekommen, wobei die Transitionen in einer SequenqualTransition zusammengesammelt und dann am Ende gebündelt ausgeführt werden.

2.2.14 Animation der aktuell ausgeführten Anweisung

Problemanalyse

Neben der Animation des Roboters, soll der Nutzer auch sehen, welche Anweisung gerade ausgeführt wird. Dazu gibt es in der Aufgabenstellung die Anforderung, dass die Anweisungen, die ablaufen mit einem blauen Rahmen versehen werden sollen.

Realisationsanalyse

Um diese Anforderung erfüllen zu können, sind in der GUI folgende Informationen zur Erstellung der Animation der gerade laufenden Anweisung erforderlich:

1. Die Anweisung/Aktion, die gerade ausgeführt wird
2. Die Anweisungsblöcke in denne gerade etwas ausgeführt wird. Hier können auch mehrere Blöcke gleichzeitig angezeigt werden, wenn beispielsweise eine Prozedur aus dem Programm aufgerufen wird. Dann muss die sowohl die Prozedur im Programm, als auch die gerade laufende Anweisung in der Prozedur, markiert sein. Daher ist es hier sogar möglich, dass in jedem Anweisungsblock zeitgleich eine Anweisung markiert ist.
3. Der Index der zu markierenden Anweisung innerhalb des Anweisungsblocks
4. Eine Referenz auf das Bild der aktuell ausgeführten Anweisung, damit diese im Rahmen der Animation markiert werden kann.

Sind all diese Informationen gegeben, so kann innerhalb der GUI beim erstellen der Animationen für die Anweisungen auch die gerade ausgeführte Anweisung markiert werden. Da es sich bei der Animation des Roboters auf dem Spielfeld um Transitionen handelt, bietet es sich an einfach für jede Markierung eine eigene PauseTransition an die richtige Position in der SequencialTransition zu packen.

Als kleine Optimierung, wäre es außerdem denkbar keine neue PauseTransition nur für die blauen Rahmen zu erstellen, sondern einfach die PauseTranstion, die vor jeder Anweisung kommt

dafür zu verwenden. Diese ist eigentlich dafür da, eine kleine Verzögerung für die Ausführung der Anweisungen einzuführen, es könnte somit aber auf doppelt erstellte PauseTransitions verzichtet werden.

Realisationsbeschreibung

Die Animation der gerade ausgewählten Anweisungen wird wie beschrieben über weitere PauseTransitions dargestellt, die immer jeweils vor die Animation der als nächstes asuzuführenden Anweisung hinzugefügt werden. Somit zeigen die blauen Rahmen immer die letzte valide Anweisung an. Wenn also beispielsweise nach einem Schritt eine weitere Anweisung folgt, die nicht möglich ist, dann bleibt der blaue Rahmen am Ende der Animation auf dem Schritt und nicht auf der Anweisung, die nicht mehr ausführbar war.

Dabei gibt es zahlreiche Sonderfälle, die zusätzlich beachtet werden müssen. Wird eine leere Prozedur aufgerufen und danach folgen noch weitere Anweisungen, dann muss der blaue Rahmen automatisch weiter auf die nächste Anweisung springen. Wird eine Prozedur, die Anweisungen enthält, beendet, dann muss dessen Markierung verschwinden und es muss die Markierung in dem aufrufenden Anweisungsblock weiter gesetzt werden. Wird innerhalb der Animation eine Prozedur von dem Programm aus aufgerufen, welche wiederum die andere Prozedur aufruft, dann müssen drei blaue Markierungen gleichzeitig vorhanden sein (eine Markierung für jeden Anweisungsblock).

2.2.15 Pfadfindung im Lösungsalgorithmus

Problemanalyse

Der Lösungsalgorithmus muss in der Lage sein einen Weg von dem Startfeld zur Tür zu finden und auf dem Weg alle Münzen einzusammeln. Dazu ist ein Algorithmus notwendig, welcher sowohl den Weg zu den Münzen, als auch den Weg zur Tür finden kann.

Realisationsanalyse

Es gibt sehr viele verschiedene Algorithmen zum finden eines Pfades. Beispiele die in der Praxis häufig verwendet werden sind Dijkstra, A* oder auch FloodFill.

Dijkstra und A* sind dabei ziemlich ähnlich, wobei A* im Bereich der Spieleentwicklung deutlich häufiger verwendet wird. Dies hat den Grund, dass A* ein informierter Suchalgorithmus ist. Das bedeutet, dass der A*-Algorithmus Heuristiken (also Schätzungen) verwendet um sich in jedem Schritt im besten Fall immer in Richtung des Ziels zu bewegen. Dijkstra und FloodFill hingegen nutzen keine Heuristiken. Das heißt es handelt sich hier um nicht-informierte Suchalgorithmen, wodurch sie jedes mögliche Feld untersuchen müssen und somit möglicherweise deutlich mehr Schritte bis zum Ziel brauchen als A*. Dafür hat eine Implementierung von A* einen deutlich höheren Aufwand als zum Beispiel FloodFill und der Algorithmus kann auch nur besser sein, wenn die Heuristiken richtig gewählt wurden. Wurden die Heuristiken falsch zugewiesen, dann ist A* im schlechtesten Fall wieder genau so schnell wie Dijkstra.

FloodFill ist hingegen ein deutlich einfacherer Algorithmus, der gut mit dem internen Algorithmus eines Wassereimers in zum Beispiel Photoshop zu vergleichen ist. Hierbei startet der

Algorithmus auf einer bestimmten Zelle und breitet sich jeweils auf seine vier Nachbarn aus. Diese wiederholen dasselbe Prinzip und breiten sich wieder an jeweils ihre vier Nachbarn aus. Somit ist nach einer endlichen Menge an Schritten jedes mögliche Feld asugefüllt. Da sich FloodFill aber nur über seine vier direkten Nachbarn ausbreiten kann, ist keine direkte diagonale Fortbewegung möglich. Durch dieses Vorgehen kann aber schon einmal geprüft werden, ob überhaupt alle Münzen und die Tür auf dem Feld erreichbar sind. Um nun herauszufinden wie weit eine Münze oder eine Tür von dem Startfeld entfernt liegen, können in jeder Iteration die direkten Nachbarn des aktuellen Feldes durchnummeriert werden. Sie erhalten dabei immer die nächsthöhere Zahl von der aktuellen Zelle. Wird eine Zelle besucht, die bereits eine Zahl zugewiesen bekommen hat, so muss diese Zahl nicht mehr geändert werden, da es sich um keine niedrigere Zahl mehr handeln kann.

Nachdem die Entscheidung des Suchalgorithmus gefallen ist, muss noch entschieden werden, wie die schlussendliche Repräsentation des Spielfeldes nach Anwendung des Suchalgorithmuses aussehen soll. Dazu kann beispielsweise ein zweidimensionale Array verwendet werden. Dies hat den Vorteil, dass für die spätere Nutzung der Repräsentation ein schneller indizierter Zugriff möglich ist. Ein Nachteil würde sich nur ergeben, wenn die Repräsentation häufig kopiert werden müsste, was in unserer Anforderung nicht der Fall ist.

Alternativ könnte man auch eine zweidimensionale Liste verwenden. Der Vorteil davon wären die bereits vordefinierten Funktionen einer Liste, die beispielsweise genutzt werden können um den Index eines bestimmten Elementes zu finden. Auf der anderen Seite sorgt die Nutzung von einer Liste jedoch wieder für einen sehr großen overhead, da nur wenige der Funktionalitäten in unserer Anforderung überhaupt benötigt werden. Außerdem ist anders als bei Arrays ein indizierter Zugriff nicht in O(1), also in konstanter Zeit möglich, weshalb hier Einbußen in der Laufzeit zu erwarten sind.

Realisationsbeschreibung

In dem Projekt wurde zum finden eines Pfades zu den jeweiligen Münzen und zur Tür der FloodFill Algorithmus verwendet. Dabei wurden, wie im letzten Teil erklärt, die Nachbarn mit ihrer Distanz zum Start beschriftet und so ein Weg bis zum Ziel ausgemacht. Als interne Repräsentation wurde hierbei ein zweidimensionales Array verwendet. Der Start wurde dabei immer mit 0 versehen, wodurch für später direkt klar ist, wo der FloodFill-Algorithmus gestartet hat. Auf dem Weg zum Ziel wurden die begehbaren Nachbarn also immer inkrementell durchnummeriert, wobei nicht erreichbare Felder mit einer -1 indiziert wurden. Am Ende jedes FloodFill-Durchlaufes wurde dann noch gespeichert, an welcher Koordinate sich nun das Ziel befand, da dies nicht mit einem zweidimensionalen Array aus Zahlen gespeichert werden konnte.

Ein Nachteil der bei diese Implementierung aufgekommen ist ist die Lösbarkeit bestimmter Level. Da der FloodFill-Algorithmus immer die nächste Münze sucht und danach zur Tür geht, kann es in bestimmten Levels sein, dass zu viele Anweisungen nötig sind um das Level zu lösen. Das ist insbesondere Der Fall, wenn eine Münze, die eigentlich weiter Weg ist als eine andere, früher eingesammelt werden muss, damit im Nachhinein eine bessere Aufteilung der Anweisungen möglich ist. Für solche Fälle findet der FloodFill-Algorithmus dann also möglicherweise keine Lösung, die innerhalb der maximalen Anzahl an Anweisungen lösbar ist. Somit ist auch die Reihenfolge in der die Nachbarn geprüft werden entscheidend. In meiner Implementierung werden die Nachbarn in der Reihenfolge unten, oben, links, rechts geprüft.

Somit wird immer die untere Münze vor der rechten gefunden, was möglicherweise wieder zu einer zu großen Menge an Anweisungen führen kann. Genaueres zu diesen Fällen ist in Abschnitt 2.5 beschrieben.

2.2.16 Umwandlung des gefundenen Pfades in Anweisungen

Problemanalyse

Nachdem der Lösungsalgorithmus einen lösbaren Pfad gefunden hat, muss er in der Lage sein diesen in eine Folge von Anweisungen zu übersetzen.

Realisationsanalyse

Hier muss also die Repräsentation des Suchalgorithmus verwendet werden um am Ende aus dem Weg eine Folge von Anweisungen zu entwickeln. Da hier bei richtiger Implementierung des Suchalgorithmus stets ein möglicher Weg zu einer Münze oder einer Tür dargestellt werden kann, gibt es auch auf jeden Fall eine endliche Menge an Anweisungen, die diesen Weg darstellen können. Außerdem ist zu diesem Zeitpunkt das eigentliche Level nicht mehr relevant, da über das Ergebnis des Suchalgorithmus direkt betretbare Koordinaten erstellt werden können, welche dann nur noch miteinander verglichen werden müssen.

Bei dem Erstellen dieser Koordinaten kann jedoch auf verschiedene Weisen vorgegangen werden:

1. Es kann an der Startposition begonnen werden und dann der Weg zum Ende analysiert werden. Dabei müssten jedoch auch Wege geprüft werden, die möglicherweise gar nicht zum Ziel führen. Somit müsste hier zusätzlich eine Liste darüber geführt werden, welche Wege schon gegangen wurden und aus welchen Koordinaten schlussendlich der richtige Weg besteht.
2. Die zweite Variante ist es von dem Ziel aus zu starten und zurück zum Anfang zu gehen. Da hier jeweils einfach die nächstkleinere Zahl zu der aktuellen gesucht werden muss, ist sichergestellt, dass der Algorithmus auf jeden Fall immer auf dem richtigen Weg zum Startpunkt ist. Somit können die begehbaren Koordinaten auf dem Weg mitgespeichert werden und es ist keine Unterscheidung in Gänge, die zum Ziel führen und welche die dies nicht tun, erforderlich.

Im zweiten Teil müssen die gefundenen Koordinaten nun miteinander verglichen werden um daraus dann entsprechende Anweisungen erstellen zu können. Hier ist jedoch nur eine Vorgehensweise vom Start bis zum Ende sinnvoll, da so die Liste der entstehenden Anweisungen auf jeden Fall in der richtigen Reihenfolge erstellt wird.

Realisationsbeschreibung

Zur Übersetzung des Pfades in Anweisungen startet der Algorithmus an der Zielposition und geht rückwärts vor bis zum Start. Dabei entscheidet er sich immer für den Nachbarn, der mit der geringsten Zahl aus der Flood Fill Repräsentation indiziert wurde. Somit kann sichergestellt werden, dass bei diesem Vorgehen immer der richtige Pfad zum Start gefunden wird. In jedem Schritt wird dabei die Koordinate des besten Nachbarn in einer Warteschlange gespeichert. Würde man exakt das Selbe Vorgehen vom Start zum Ziel machen, so müsste man viele Wege entlang gehen, die möglicherweise gar nicht zum Ziel führen, weshalb hier der Weg vom Ziel zum Start zu bevorzugen ist.

Nachdem nun alle betretbaren Koordinaten in der Warteschlange gesammelt wurden, wird die Warteschlange von der Anderen Seite aus (also von vorne nach hinten) durchgegangen. Dabei wird nun immer die vorherige Koordinate mit der nächsten Verglichen. Ist die Differenz der Beiden Koordinaten 1, so handelt es sich bei der nächsten Anweisung um einen Schritt, während es sich bei einer Differenz von 2 um einen Sprung handeln muss. Das liegt daran, dass bei dem Sammeln der einzelnen Koordinaten ein Abgrund nicht mit gespeichert wird, da sich der Roboter nicht auf diesen stellen kann.

Zudem muss noch ermittelt werden, wie häufig sich der Roboter vor der Anweisung drehen muss. Dazu werden wieder die beiden Koordinaten mit einander Verglichen und es wird geschaut, ob die aktuelle Koordinate mit dem aktuellen Richtungsvektor der Bot Rotation die neue Koordinate ergibt. Ist dies der Fall, so muss keine Drehung stattfinden und die Anweisung kann direkt ausgeführt werden. Kommt dabei jedoch nicht die richtige Koordinate heraus, muss erst einmal die Anzahl an nötigen Drehungen errechnet werden, wobei diese dann als extra Anweisung vor dem Schritt oder dem Sprung zu der Anweisungsliste hinzugefügt werden.

Je nachdem, ob es sich bei dem Ziel um eine Tür oder eine Münze handelt muss außerdem die letzte Anweisung ausgewählt werden. Handelt es sich um eine Tür, so wird als letzte Anweisung ein „Exit” ausgeführt. Handelt es sich stattdessen um eine Münze, so kann eine „Gehen”-Anweisung ausgeführt werden um auf diese Münze zu gehen und diese somit einzusammeln.

2.2.17 Aufspalten der Anweisungen in Programm und Prozeduren

Problemanalyse

Nachdem die Anweisungen vom Start zum Ziel erstellt wurden, müssen diese möglicherweise noch in Programm und Prozeduren aufgespaltet werden. Dies ist der Fall, sobald mehr als 12 Anweisungen (also mehr als in den Anweisungsblock vom Programm passen) aufgerufen werden müssen, um das Ziel zu erreichen.

Realisationsanalyse

Dazu ist es erforderlich jegliche auftretende Folge von Anweisungen zu speichern und dessen Vorkommen zu zählen. Außerdem müssen diese Folgen mit ihren Vorkommen in einer Datenstruktur gespeichert werden, wo dann mit ihnen weitergearbeitet werden kann. Dazu sind mehrere Datenstrukturen denkbar:

1. Es kann eine klassische Map verwendet werden, bei der der Schlüssel die Anweisungsfolge und der Wert die Vorkommen repräsentiert. Dies hat den Vorteil, dass beide Werte direkt zusammen gespeichert werden können und diese durch die vordefinierten Funktionen einer Map auch gut auszulesen sind.
2. Es kann auch eine Liste an Paaren genutzt werden. Dabei würde wieder ein Wert des Paares der Anweisungsfolge und der andere Teil den Vorkommen entsprechen. Da dieses Vorgehen jedoch sehr stark einer Map ähnelt und es standardmäßig keine Implementierung eines Paares oder eines Tupels gibt, ist diese Möglichkeit nicht so effizient.
3. Man könnte sich auch eine Liste an Indizes speichern, wobei die Indizes das Vorkommen einer Folge darstellen. Hier ist jedoch extra Logik nötig um dann die schlussendliche Anweisungsfolge, die jetzt die beste war wieder zu entziffern.

Nachdem die Vorkommen gezählt wurden muss nun entschieden werden, welche zwei Anweisungsfolgen am häufigsten vorkommen (da es zwei Prozeduren gibt). Hierbei ist es jedoch nicht sinnvoll auch einelementige Vorkommen mit zu zählen, da eine 1 zu 1 Umsetzung einer Prozedur in eine Anweisung keine Verkürzung der gesamten Anweisungsliste mit sich bringt.

Zum bestimmen der besten Anweisungsfolge ist es typisch eine Kombination aus folgenden zwei Kriterien zu verwenden:

1. Was ist die Längste Anweisungsfolge?
2. Welche Anweisungsfolge kommt am häufigsten vor?

Eine Anweisungsfolge hat die höchste Effizienz, wenn damit die größte Anzahl an Anweisungen abgedeckt wird. Dementsprechend muss hier immer die Größe der Anweisungsfolge mit dessen Vorkommen verrechnet werden um dessen Effizienz zu bestimmen. Dabei wird bei gleichem Ergebnis stets die Folge bevorzugt, die aus mehr Anweisungen besteht, da so die Prozedur möglichst stark gefüllt werden kann.

Außerdem muss noch beachtet werden, dass eine Anweisungsfolge nur so groß sein darf wie die maximal mögliche Anzahl an Anweisungen in einer Prozedur.

Realisationsbeschreibung

Zum erstellen aller möglichen Anweisungsfolgen in der übergebenen Liste an Anweisungen nutze ich ein einfaches sliding-window Prinzip. Dabei startet der Algorithmus mit dem erstellen einer Subliste einer gegebenen Länge, das sogenannte window. Danach wird das windows dann immer um einen Index weiter verschoben, bis jedes Element in mindestens einer Subliste enthalten ist. Dieses Vorgehen muss nun für eine window size von 2 bis zur maximalen Größe einer Prozedur ausgeführt werden. Der Start bei zwei stellt sicher, dass keine einelementigen Sublisten mitgezählt werden, da diese nicht sinnvoll ausgelagert werden können.

Von diesen Sublisten kann nun in der Liste an Anweisungen jeweils das Vorkommen gezählt und in einer Map gespeichert werden. Dabei bestehen die Einträge der Map wie in der Realisationsanalyse beschrieben aus der Anweisungsfolge als Schlüssel und dessen Vorkommen als Wert.

Nun können aus der Map mit Folgen und Vorkommen die besten zwei Folgen herausgesucht werden um diese dann als Prozeduren zu verwenden. Dabei wird immer zur Ermittlung der

besten Subliste dessen Größe mit dessen Vorkommen verrechnet. Gibt es dabei mehrere Anweisungsfolgen die gleich auf sind, so gewinnt die Anweisungsfolge, die aus mehr Anweisungen besteht.

Bei der Auswahl der besten Anweisungsfolge verwende ich in meiner Implementierung wieder die Java Streams API und zwei verschiedene Komparatoren. Der erste Komparator vergleicht die einzelnen Entries der Map nach dessen errechneten Effizizenz, also der Anzahl an Anweisungen mal der Anzahl an Vorkommen. Der zweite Komparator vergleicht die Anweisungsfolgen dann noch einmal nur nach dessen Größe. Um dort dann die beste Entry zu finden, nutze ich die .max() Funktion der Streams API. Das Ergebnis muss dann auf einem Optional gespeichert werden. Dies ist ein Datentyp, der anstatt null zurück zu liefern „no result" liefert. Dies wird vor allem dann verwendet, wenn das zurückgeben von null möglicherweise zu Fehlern führen kann. Daher muss das Ergebnis einmal auf isPresent() geprüft werden um dann an das Ergebnis heran zu kommen.

Nachdem eine Subliste als effizienteste gefunden wurde, wird diese nun der Prozedur hinzugefügt und es werden alle Vorkommen dieser Folge in der Anweisungsliste gegen die Prozedur ausgetauscht. So kann dann im zweiten Durchlauf eine neue effizienteste Liste für die zweite Prozedur gefunden werden, die zeitgleich möglicherweise auch Aufrufe der ersten Prozedur enthalten kann. Auch hier müssen am Ende dann wieder alle Vorkommen der zweiten Prozedur mit der Prozedur selber ausgetauscht werden.

Sobald dies geschehen ist werden die restlichen verbleibenden Anweisungen in der Anweisungsliste als Programmanweisungen verwendet.

2.2.18 Rückgabewert des Lösungsalgorithmus

Problemanalyse

Nachdem der Lösungsalgorithmus eine Lösung gefunden hat, muss diese an die Logik zurück gegeben werden. Dabei müssen sowohl die Programmanweisungen, als auch die beiden Prozeduren zurückgegeben werden. Außerdem muss es eine Rückmeldung darüber geben, ob das Level nun lösbar war oder nicht. Zudem muss im nicht lösbaren Fall eine genau Meldung erfolgen, warum das Level nicht lösbar war.

Realisationsanalyse

Da sowohl Programmanweisungen als auch zwei Prozeduren zurückgegeben werden müssen, muss hier eine eigene Datenstruktur angelegt werden, welche diese drei Werte zurück gibt. Zusätzlich muss es einen Weg geben, die Logik darüber zu informieren, ob das Level lösbar war. Die einfachste Variante wäre hier die Rückgabe von null im Fehlerfall. Dies hat jedoch den Nachteil, dass nicht klar ist, warum genau das Level nicht lösbar war.

Die nächste mögliche Variante ist somit die Nutzung eines Enum, welches einen Status für ein lösbares Level und mehrere Status für unlösbare Level enthält. Dieser Status muss dann in der Datenstruktur mit übergeben werden, damit die Logik nun prüfen kann, ob das Level lösbar war oder nicht.

Realisationsbeschreibung

Das Ergebnis des Lösungsalgorithmus wird mit der Klasse SolveResult repräsentiert. Diese Klasse enthält eine Liste an Programmanweisungen, die beiden Prozeduren und einen Status darüber, ob das Level lösbar war oder nicht. Dieser Status ist als Enum SolveStatus realisiert und enthält einen Werte für ein lösbares Level, und mehrere Unterscheidungen für unlösbare Fälle.

Im Falle eines nicht lösbaren Levels wird demnach der jeweilge SolveStatus mit angegeben und für die Programmanweisungen und die Prozeduren wird null zurückgegeben. Somit muss in der Logik einmal geprüft werden, ob es sich um ein lösbares Level handelt, bevor mit dem Ergebnis weiter gearbeitet wird.

2.3 Programmorganisationsplan

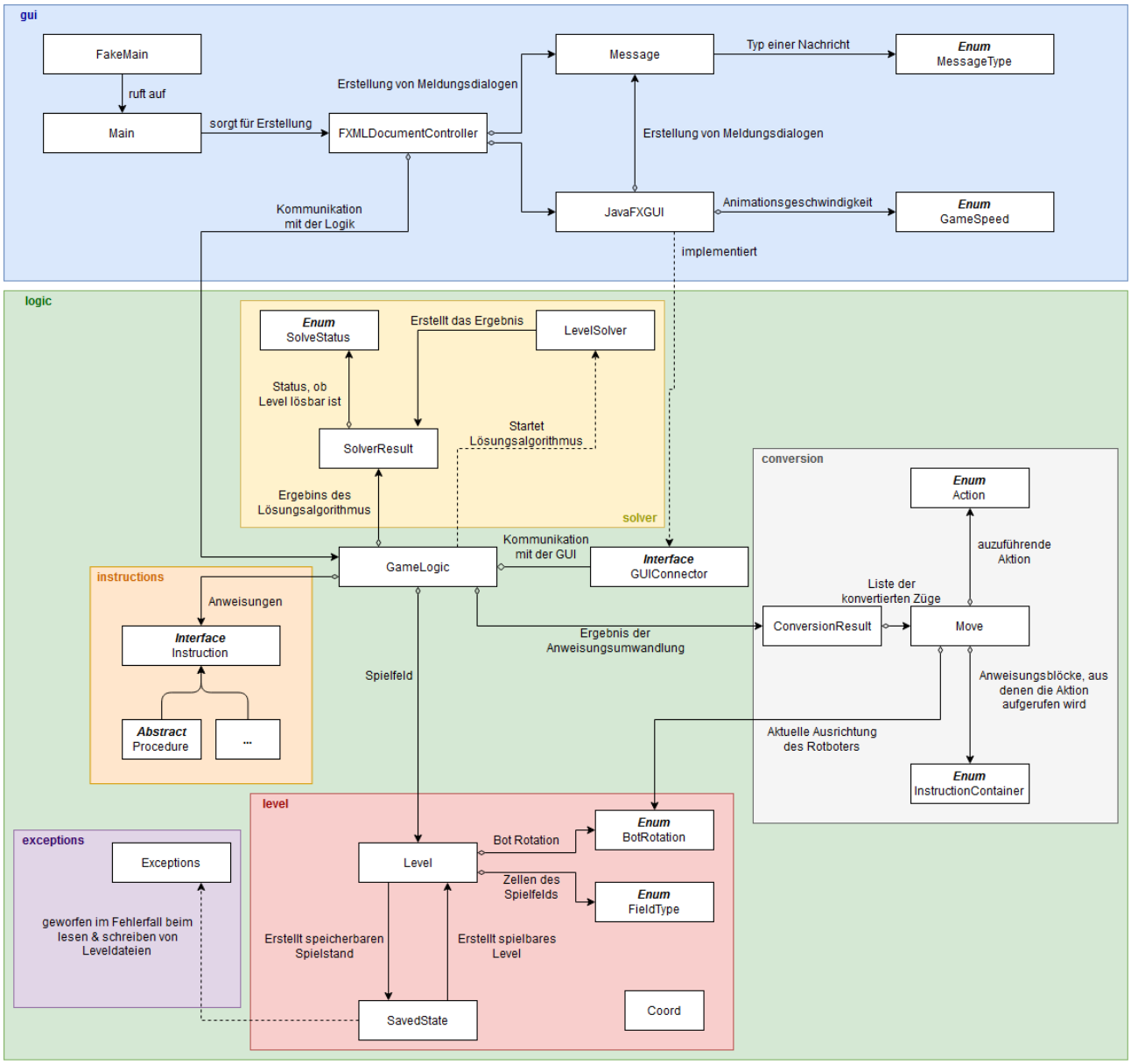

Abbildung 2.1: Programmorganisationsplan erstellt mit draw.io

Die Abbildung 2.1 enthält eine Auflistung aller wichtigen Klassen des Projekts, wobei diese grundlegend in GUI und Logik aufgeteilt sind. Die GUI kommuniziert dabei nur über eine GameLogic-Instanz mit der Logik. Auf der anderen Seite besitzt die GameLogic keinen direkten Zugriff auf Klassen der GUI, sondern kommuniziert mit dieser nur über das GUIConnector-Interface.

In der GUI befindet sich zusätzlich zu dem Controller noch die Klassen Main und FakeMain, welche für das Ausführen der Anwendung verantwortlich sind. Über die Klasse JavaFXGUI können Änderungen an der Benutzeroberfläche vorgenommen werden. Zusätzlich nutzt die GUI noch eine Message-Klasse über die Nachrichten erstellt werden können, die dann auf der Benutzeroberfläche angezeigt werden.

Die Logik enthält dann alle Klassen, die für die Logik des Spiels benötigt werden. Diese sind in weitere packages Aufgeteilt. In dem package instructions befindet sich eine Klassenstruktur aller Anweisungen, worunter auch die beiden Prozeduren Fallen. Diese repräsentieren die Programm- und Prozeduranweisungen die aktuell in der GameLogic gespeichert sind.

Das package level enthält alle Dateien, die die Erstellung eines Levels betreffen. Dies beinhaltet die Klasse Level selbst, sowie die Enums die innerhalb des Levels zur Repräsentation eines Spielfeldes und der Botausrichtung verwendet werden. Außerdem ist ne weitere Klasse SavedState enthalten, welche als Zwischenrepräsentation verwendet wird um Leveldateien laden und speichern zu können. Kommt es bei dem laden von Leveldateien zu einem Fehler, dann werden die Exceptions aus dem exceptions package verwendet.

Eine besondere Rolle nimmt jedoch die Coord-Klasse ein. Diese beschreibt eine Koordinate und besitzt keine Kantenverbindung zu anderen Klassen. Das liegt daran, dass diese Klasse sowohl in der Logik verwendet wird um Punkte auf dem Spielfeld zu referenzieren, als auch in der GUI in der sie sowohl für das Spielfeld als auch für die einzelnen Anweisungsblöcke verwendet wird.

Innerhalb des conversion packages befinden sich alle nötigen Klassen um Anweisungen in Aktionen zu konvertieren. Dabei wird die Move Klasse als Wrapper der einzelnen Aktionen verwendet um weitere Zusatzinformationen wie zum Beispiel die Anweisungsblöcke, in denen die Aktion aufgerufen wird, zu der Aktion zu speichern.

Das letzte package solver bündelt alle Klassen, die zum Erstellen einer Lösung von einem gegebenen Level benötigt werden. Dabei wird die statische solve()-Methode der LevelSolver-Klasse aufgerufen, um den Lösungsalgorithmus zu starten. Als Ergebnis wird dann ein SolveResult an die Logik zurückgegeben in der neben den Programm- und Prozeduranweisungen ein SolveResult gespeichert ist, welches auf Erfolg bzw. Misserfolg des Level lösens hinweist.

2.4 Dateien

Der Aufbau einer Leveldatei wird im Benutzerhandbuch in dem Abschnitt 1.3.5 erläutert. Die Beispiellevel, die geladen werden können befinden sich dabei mit in dem gui package des Projekts, sind also innerhalb der .jar-Datei direkt verfügbar.

2.5 Programmtests

Im Folgenden werden Tests der Benutzeroberfläche aufgelistet. Dabei wird in jedem Test eine definierte Situation beschrieben, die ein genau definiertes Ergebnis erwartet. Ist das tatsächliche Ergebnis abweichend von dem erwarteten Ergebnis, so wird dies in der dritten Spalte „Abweichung" angemerkt. Ist dies nicht der Fall, so bleibt die dritte Spalte leer. Die bei den Programmtests verwendeten Leveldateien sind hierbei innerhalb des Unterverzeichnisses „Programmtests" innerhalb des final-binaries Ordners zu finden. Sofern für den Test eine bestimmte Leveldatei verwendet wurde, ist dessen Name in Klammern mit bei dem Testfall angegeben.

2.5.1 Level laden

Testfall	**Erwartetes Ergebnis**
Beim Laden aus einer Datei wird eine leere Leveldatei angegeben (empty.json)	Das aktuelle Level soll nicht abgeändert werden und der Nutzer wird darauf hingewiesen, dass das Level nicht geladen werden konnte
Beim Laden aus einer Datei wird eine Leveldatei mit invalider JSON-Syntax angegeben (invalidJson.json)	Die Datei wird nicht geladen und der Nutzer wird auf den Syntaxfehler hingewiesen
Beim Laden aus einer Datei wird eine Leveldatei ohne „field"-Attribut angegeben (noField.json und fieldNull.json)	Die Datei wird nicht geladen und der Nutzer wird darauf hingewiesen, dass das „field"-Attribut nicht vorhanden ist.
Beim Laden aus einer Datei wird eine Leveldatei mit mindestens einem invaliden Feldtypen angegeben (invalidFieldType.json)	Die Leveldatei wird nicht geladen und der Nutzer wird darauf hingewiesen, dass ein Feldtyp nicht valide ist
Beim Laden aus einer Datei wird eine Leveldatei mit invalider Feldgröße angegeben (invalidFieldSize.json und invalidFieldSize_OneCellMissing.json)	Die Datei wird nicht geladen und der Nutzer wird auf die falsche Feldgröße aufmerksam gemacht
Beim Laden aus einer Datei wird eine Leveldatei ohne „botRotation"-Attribut angegeben (noBotRotation.json und botRotationNull.json)	Die Datei wird nicht geladen und der Nutzer wird darauf hingewiesen, dass die Botausrichtung nicht vorhanden ist.
Beim Laden aus einer Datei wird eine Leveldatei mit invalider Botausrichtung angegeben (invalidBotRotation.json)	Die Datei wird nicht geladen und der Nutzer wird darauf hingewiesen, dass die Botausrichtung fehlerhaft war.

2.5.2 Erstellung eines Programms

Testfall	**Erwartetes Ergebnis**
Der Nutzer startet das Programm ohne Angabe von Anweisungen	Die Animation wird nicht gestartet und der Nutzer wird darauf hingewiesen, dass keine Anweisungen angegeben wurden
Der Nutzer versucht durch eine Tür zu gehen, die sich nicht in Reichweite des Roboters befindet	Das Programm endet und der Nutzer wird darauf hingewiesen, dass die Tür nicht erreichbar ist.
Der Nutzer versucht auf ein Feld zu gehen, welches blockiert ist	Das Programm endet und der Nutzer wird informiert, dass auf das nächste Feld nicht gegangen werden kann
Der Nutzer versucht über ein Feld zu springen, dass kein Abgrund ist	Das Programm endet und der Nutzer wird informiert, dass er nicht über das nächste Feld springen kann
Der Nutzer versucht auf einem Feld hinter einem Abgrund zu landen, welches nicht begehbar ist	Das Programm endet und der Nutzer wird informiert, dass er nicht auf dem Feld hinter dem Abgrund landen kann
Der Nutzer versucht die Tür zu betreten ohne alle Münzen eingesammelt zu haben	Das Programm wird beendet und der Nutzer wird darauf hingewiesen, dass er nicht alle Münzen eingesammelt hat.
Der Nutzer versucht weitere Anweisungen nach einem Exit anzugeben	Das Programm endet mit der Meldung, dass weitere Anweisungen nach dem Exit angegeben wurden. Dies ist nicht der Fall, wenn ein Exit aufgerufen wird, obwohl noch nicht alle Münzen eingesammelt wurden. Hier wird der Nutzer auf die nicht eingesammelten Münzen hingewiesen und nicht auf die Anweisungen nach dem Exit.
Der Nutzer ruft eine Prozedur in sich selbst auf	Das Programm kann gar nicht erst gestartet werden und der Nutzer bekommt eine Fehlermeldung, dass er eine Endlosrekursion erzeugt hat.
Der Nutzer lässt die beiden Prozeduren sich gegenseitig aufrufen	Das Programm kann gar nicht erst gestartet werden und der Nutzer bekommt eine Fehlermeldung, dass er eine Endlosrekursion erzeugt hat.
Der Nutzer verwendet eine leere Prozedur im Programm	Es wird bei der nächsten Anweisung fortgesetzt

2.5.3 Level auf Lösbarkeit überprüfen

Testfall	Erwartetes Ergebnis	Tatsächliches Ergebnis
Das angegebene Level ist lösbar (validLevel.json)	Das Level wird gelöst, die resultierenden Anweisungen werden in die Anweisungsblöcke geladen und der Nutzer bekommt eine Meldung, dass das Level lösbar ist	
Das angegebene Level ist nicht vom lösungsalgorithmus lösbar, da es sich um ein invalides Level handelt (invalidLevel.json)	An den Anweisungsblöcken wird nichts geändert und der Nutzer bekommt eine Meldung, dass das Level invalide ist und somit nicht gelöst werden kann	
Das angegebene Level ist nicht vom Lösungsalgorithmus lösbar, da nicht alle Münzen erreichbar sind (unreachableCoin.json)	Die Anweisungsblöcke ändern sich nicht und der Nutzer wird darauf hingewiesen, dass nicht alle Münzen erreichbar sind	
Das angegebene Level ist nicht vom Lösungsalgorithmus lösbar, da die Tür nicht erreichbar ist (unreachableDoor.json)	Die Anweisungsblöcke werden nicht verändert und der Nutzer wird darauf hingewiesen, dass die Tür nicht erreichbar ist.	
Das angegebene Level ist nicht vom Lösungsalgorithmus lösbar, da zu viele Anweisungen nötig sind und diese nicht in Programm und Prozeduren überführt werden können (notSolvable.json)	Die Anweisungsblöcke werden nicht verändert und der Nutzer bekommt eine Meldung, dass das Level durch zu viele benötigte Anweisungen nicht lösbar ist.	Dieses verhalten kann möglicherweise auch bei eigentlich lösbaren Levels auftreten. Genauere Spezialfälle werden in den nächsten zwei Testfällen genannt.
Spezialfall bei dem mein Algorithmus das Level nicht lösen kann, obwohl es eigentlich lösbar ist. Dabei handelt es sich um ein Level, bei welchem eine spätere Münze zuerst eingesammelt werden muss, damit das Level überhaupt lösbar sein kann (laterCoin.json)	Das Level wird von meinem Lösungsalgorithmus als nicht lösbar erkannt, obwohl es eigentlich lösbar ist.	Da der Algorithmus immer zu der nächsten Münze geht, kann dieses Level nicht gelöst werden. Dafür müsste der Roboter erkennen, dass vorerst eine Münze eingesammelt werden muss, die sich weiter weg befindet.
Spezialfall bei dem mein Algorithmus das Level nicht lösen kann, obwohl es eigentlich lösbar ist. Dabei handelt es sich um ein Level, bei dem von oben rechts diagonal 6 Münzen nach unten gesetzt werden (diagonalCoins.json).	Das Level wird als nicht lösbar erkannt, da hier die Reihenfolge der geprüften Münzen entscheidet ist. Wird hierbei zuerst die rechte Münze geprüft, dann wird dies die erste sein zu der der Roboter geht. Bei mir wird aber zuerst die untere geprüft, weshalb dieses Level nicht lösbar sein kann.	Das Level ist also nicht lösbar, da es sich hier um einen Sonderfall handelt bei dem die Reihenfolge der geprüften Felder eine Rolle spielt. In anderen Fällen, in denen Münzen anders diagonal gesetzt werden wäre mein Algorithmus dann aber wieder im Vorteil.

2.5.4 Weitere Tests

Testfall	**Erwartetes Ergebnis**
Der Nutzer ändert während eine Animation läuft die Animationsgeschwindigkeit	Die Animationsgeschwindigkeit ändert sich erst, wenn der Nutzer eine weitere Animation startet

Abbildungsverzeichnis

List of Listings