Programmierpraktikum
EscapeBot

orcid.org/0009-0008-3023-1228

Wintersemester 2021

Fachrichtung	**Informatik**
Martikelnummer	**104662**
Fachsemester	**4**
Verwaltungssemester	**4**

Inhaltsverzeichnis

Kapitel 1

Benutzerhandbuch

1.1 Ablaufbedingungen

Für den Ablauf des kompilierten Spiels gibt es keine speziellen Hardwareanforderungen. Es existieren folgende Softwareanforderungen:

Name	Version	Beschreibung
Java Runtime Environment (Java)	17.0.1	Java Runtime Environment, oder JRE, wird auf dem Betriebssystem eines Computers ausgeführt und stellt, unter anderem die Bibliotheken, die ein Java-Programm zur Ausführung benötigt bereit.

1.2 Programminstallation/Programmstart

Die .jar-Datei an einem gewünschten Dateipfad sichern.
Um das Programm zu starten muss die .jar-Datei mit der JRE ausgeführt werden.

1.3 Bedienungsanleitung

> "EscapeBot" ist ein leicht abgewandelter Klon von LightBot, bei dem ein Bot korrekt *programmiert* werden muss, um einen Raum zu verlassen. Die Programmierung wird dabei in einem Hauptprogramm und maximal zwei Unterprozeduren anhand einfacher Schritte wie *laufen* oder *nach links/rechts drehen* abgelegt.

- Aufgabenstellung Programmierpraktikum

1.3.1 Ziel

Das Ziel des Spiels ist es den Bot[3.1.1] von seiner Startposition[3.7.9] zur Tür[3.7.7] zu führen. Wenn in dem Level Münzen[3.7.6] existieren, müssen diese vorher eingesammelt werden. Um den Bot zu führen müssen Anweisungen in Prozeduren programmiert werden. Wenn das Ziel nicht erreicht wird, kannst du das an der Animation und Position des Spielercharakter erkennen und durch die Fehlermeldung in der Infoleiste[1.3.5] erfahren, was schief gelaufen ist.

Abbildung 3.1.1: Bot

1.3.2 Regeln

Münzen einsammeln

Du benötigst Münzen, um das Spiel erfolgreich beenden zu können. Um Münzen einzusammeln, musst du den Bot auf das Feld mit der Münze navigieren. Das Einsammeln der Münze passiert dann automatisch.

Abbildung 3.2.2: Münze

Springen

Die Sprung[3.8.13]-Anweisung kann unter bestimmten Bedingungen genutzt werden, um ein einzelnes Abgrund[3.7.5]-Feld zu überspringen. Diese Bedingungen sind:

- Bot steht in Richtung des Abgrunds.
- Feld direkt vor dem Bot ist ein Abgrund.
- Feld nach dem ersten Abgrund ist ein normales[3.7.8] Feld oder eine Münze[3.7.6].

Exit

Um das Spiel zu beenden musst du einige Bedingungen erfüllen. Wenn in dem Level Münzen vorhanden sind, müssen diese alle vorher eingesammelt werden. Dann musst du auf einem Feld vor der Tür stehen und in Richtung der Tür gucken. Wenn diese Bedingungen erfüllt sind kannst du die Exit[3.8.15]-Anweisung verwenden um das Level erfolgreich zu beenden.

1.3.3 Bedienung

Übersicht

Nach dem Programmstart sieht der Nutzer auf der linken Seite ein geladenes Spielbrett mit einem Beispiel-Level und auf der rechten Seite drei leere Gitter und eine Auswahl an Anweisungskarten.

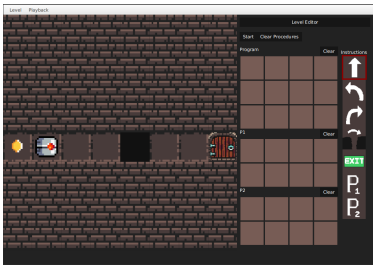


Abbildung 3.3.3: Nach dem Programmstart

Prozeduren erstellen

Die Seitenleiste zum Erstellen von Prozeduren besteht aus drei Gittern und einer Auswahl von Anweisungen. Das "Programm"-Gitter hat zwölf Felder und die Subprozedurgitter `P1` und `P2` haben acht Felder. Die Auswahl von Anweisungen kannst du mit einem Linksklick bedienen.

Mit der Wahl von genau einer Anweisung aus der Auswahl neben der Gitter kann man auf die leeren Gitter-Felder drücken und die Anweisungen einfügen. Du kannst nur, beginnend oben links, das erste leere Feld mit einer Anweisung füllen. Es gibt drei Möglichkeiten der Bearbeitung einer Prozedur:

Einsetzen Um eine Anweisung an einer bestimmten Stelle einzusetzen, wähle die gewünschte Anweisung und linksklicke auf das gewünschte Feld im Gitter. Dabei kannst du nur in das nächste freie oder ein bereits belegtes Feld klicken.

Anfügen Um eine Anweisung ans Ende anzufügen, wähle die gewünschte Anweisung und klicke mit der mittleren Maustaste auf ein bereits belegtes oder das nächste freie Feld.

Löschen Um eine Anweisung zu löschen, rechtsklicke auf das Feld mit der Anweisung.

Ausführung starten

Die Ausführung der Prozeduren wird mit dem Drücken des "Start"-Knopfes begonnen. Danach bewegt sich die "Roboter"-Figur entsprechend der gesetzten Anweisungen über das Spielbrett. Während die Prozeduren ausgeführt werden, ist die Steuerung gesperrt. Rechts in den Prozedur-Gittern wird die Anweisung hervorgehoben die gerade ausgeführt wird.

1.3.4 Karten-Editor

Du kannst auch eigene Level erstellen. Um zum Karten-Editor zu gelangen, drücke den großen Knopf "Editor" in der Sidebar. Du hast nun keinen Zugriff mehr auf die Prozeduren, und kannst diese erst wieder verändern wenn du zurück zum Spiel gehst. Anstatt der Auswahl von Anweisungen siehst du jetzt eine Auswahl von Feld-Typen. Wenn du davon eine auswählst kannst du direkt auf dem Spielbrett auf das gewünschte Spielfeld drücken. Dieses Feld wird dann den ausgewählten Feld-Typen annehmen.

Felder ändern

Um ein Feld des Spielbretts zu verändern musst du das gewünschte neue Spielfeld auswählen und dann auf die gewünschte Position im Spielbrett drücken. Beachte dass, mindestens ein Start[3.7.9]-Feld und mindestens ein Tür[3.7.7]-Feld vorhanden sein muss. Die Platzierung von Münzen ist optional.

Startposition und Richtung

Um die Startposition des Bots, zu verändern musst du das Start[3.7.9]-Feld auswählen und an die gewünschte Position setzen. Die alte Position wird automatisch entfernt. Um die Richtung der Startposition zu verändern musst du das Startfeld auswählen und mit einer beliebigen Maustaste auf ein bereits existierendes Startfeld im Spielbrett drücken. Mit jedem Mausklick dreht sich der Bot auf dem Startfeld um 90°.

Lösbarkeit

Um zu überprüfen, ob das Level, das du designed hast, auch lösbar ist, kannst du auf den Knopf "Verify Level" drücken. Das Spielbrett wird dann analysiert, und danach wird dir das Ergebnis angezeigt.

Spielanalyse Popup

1.3.5 Spiel-Status-Leiste

Unter dem Spielbrett wird unter bestimmten Bedingungen eine Statusleiste angezeigt. Diese gibt dir, der Spieler:in, zusätzliche Informationen über den aktuellen Spielablauf.

Die Leiste erscheint, wenn du das Level und deine Prozeduren ausführst. Solange die Leiste weiß ist, wird das Spiel ausgeführt. Du kannst das Spiel abbrechen mit der Betätigung des "Abort"-Knopfes.

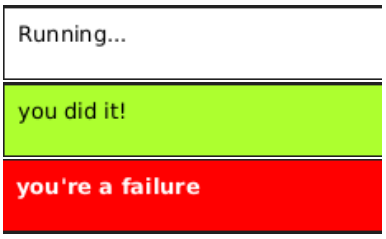


Abbildung 3.5.4: Status Leiste

Wenn das Level beendet ist, zeigt dir die Leiste den Erfolg an: Grün, wenn das Level erfolgreich beendet wurde und rot, wenn die Ausführung nicht zum Ziel geführt hat. Wenn du einen neuen Versuch startest, oder ein anderes Level auswählen willst, musst du in der Statusleiste auf "Next Try" drücken.

1.3.6 Menüleiste

Am oberen Rand des Programms befindet sich die Menüleiste. Das Menü hat zwei Optionen.

Level

Level Auswahl *Choose Level*
Unter diesem Menüpunkt findest du ein Untermenü mit fünf Beispielleveln. In dem du auf das gewünschte Level klickst wird dieses geladen.

Level laden *Load Level*
Unter diesem Menüpunkt kannst du ein gespeichertes Level laden. Nach dem Betätigen öffnet sich ein Dateidialog in dem du die gewünschte Datei auswählen kannst. Nach der Auswahl wird das Level als aktuelles Level geladen.

Leeres Level erstellen *Create Empty Level*
Unter diesem Menüpunkt kannst du ein leeres Level (1 Startfeld, sonst Mauer) laden. Nach dem Betätigen wird das leere Level als aktuelles Level geladen.

Level speichern *Save Current Level*
Unter diesem Menüpunkt kannst du das aktuelle Level speichern. Nach dem betätigen öffnet sich ein Datei Dialog in dem du den gewünschten Speicherort auswählen kannst. Die Level werden im JSON Dateiformat gespeichert.

Playback

Unter diesem Menü hast du eine Auswahl von vier Optionen. Die ersten drei Elemente ändern die Geschwindigkeit (*Slow*, *Normal* & *Fast*) der Animationen. Die letzte Option ("Skip Animation") schaltet die Animationen komplett aus und zeigt direkt das Endergebnis bei der Ausführung der Animationen an.

1.3.7 Feldtypen

Abgrund

Abyss

Abbildung 3.7.5: Abgrund

Kann vom Bot nicht betreten werden. Ein exakt ein Feld breiter Abgrund kann aber übersprungen werden.

Münze

Coin

Abbildung 3.7.6: Münze

Münzen können vom Bot eingesammelt werden, das Feld wird dadurch ein normales[3.7.8] Feld. Erst wenn alle Münzen eingesammelt wurden, kann die Tür geöffnet werden.

Tür

Door

Abbildung 3.7.7: Tür

Mit dem Öffnen der Tür ist das Level gewonnen. Es gibt genau eine Tür pro Level. Nur betretbar, wenn der Bot sich in direkter Ausrichtung vor der Tür befindet und keine Münzen mehr vorhanden sind. Zum Betreten muss die Anweisung "Exit"[3.8.15] verwendet werden.

Normal

Normal

Abbildung 3.7.8: Normal

Ein einfaches, leeres Feld. Kann vom Bot betreten werden.

Start

Start

Abbildung 3.7.9: Start

Auf dem Startfeld beginnt der Bot das Level in einer festgelegten Ausrichtung. Es gibt genau ein Startfeld pro Level.

Wand

Wall

Abbildung 3.7.10: Mauer

Ein nicht betretbares Feld.

1.3.8 Anweisungen

Gehen

Abbildung 3.8.11: 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

Links, Rechts

(a) Links

(b) Rechts

Der Bot dreht sich um 90° nach links (a) oder rechts (b).

Springen

Abbildung 3.8.13: 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[3.7.5] ist und das Feld danach Normal[3.7.8], eine Münze[3.7.6] oder das Startfeld[3.7.9] ist.

Prozedur 1, Prozedur 2

(a) Prozedur 1

(b) Prozedur 2

Subprozedur wird aufgerufen und alle Anweisungen darin ausgeführt. Nach Abarbeitung der Prozedur geht es beim Aufrufer (Programm oder andere Subprozedur) mit der nächsten Anweisung weiter. Eine Prozedur darf sich nicht selbst aufrufen. Ebenso dürfen sich die beiden Subprozeduren nicht gegenseitig aufrufen (Endlosrekursion).

Exit

Abbildung 3.8.15: Exit

Diese Anweisung muss benutzt werden um das Tür[3.7.7]-Feld zu betreten und das Level zu beenden. Es dürfen keine Anweisungen nach dieser Folgen. Sonst gelten die Prozeduren als ungültig.

1.4 Fehlermeldungen

1.4.1 Karten-Editor

Wenn man ein *kaputtes* Level auf Lösbarkeit überprüft oder vom Level-Editor zum Spiel wechselt erscheint ein Fenster mit Informationen über das Spielfeld. Diese sind nicht als Fehlermeldung sondern als Warnungen bzw. Hinweise für den Nutzer zu verstehen. Mehr Informationen findet man im Kapitel: 1.3.4

1.4.2 Spiel ausführen

Fehlermeldung	Ursache	Behebungsmaßnahme
Procedures are invalid	Die Prozeduren erfüllen nicht alle benötigten Anforderungen	Wurde eine benötigte Anweisung vergessen? z.B. die `EXIT` Anweisung? Überprüfen und reparieren.
The procedures contain a illegal recursion.	Die programmierten Prozeduren enthalten eine unerlaubte Aufrufkette	Prozeduren überprüfen und unerlaubte Aufrufe entfernen.

1.4.3 Level laden/speichern

Fehlermeldung	Ursache	Behebungsmaßnahme
File doesn't exist	Ausgewählte Datei existiert nicht	Überprüfen ob der ausgewählte Pfad korrekt ist und zugreifbar
The level file is missing required data	Bei der ausgewählten Datei fehlen benötigte Level-Informationen	Überprüfe ob die richtige Datei ausgewählt wurde. Bei Bedarf Datei reparieren.
While reading the File we expected ...	Die ausgewählte Datei enthält unerwartete Datentypen	Überprüfe ob die richtige Datei ausgewählt wurde. Bei Bedarf Datei reparieren.
The level file has corrupted data	Das Level konnte nicht verifiziert werden.	Wahrscheinlich eine kaputte Level-Datei. Überprüfe ob die richtige Datei ausgewählt wurde. Bei Bedarf Datei reparieren.
Selected file is not a json file ...	Die ausgewählte Datei entspricht nicht dem JSON-Format.	Überprüfe ob die richtige Datei ausgewählt wurde.
Board of level file is missing required field ...	Die ausgewählte Datei enthält ein Spielbrett, dem ein benötigter Feldtyp fehlt	Überprüfe ob die richtige Datei ausgewählt wurde. Bei Bedarf Datei reparieren.

Kapitel 2

Programmierhandbuch

2.1 Entwicklungskonfiguration

2.1.1 Virtual Machine Options

Das Programm verwendet die Library JavaFX für die grafische Benutzeroberfläche. Damit das Programm ausgeführt werden kann, müssen folgende Optionen für die virtuelle Maschine (JRE) gesetzt werden:

VM Options:

```
--module-path lib/ --add-modules "javafx.controls,javafx.fxml"
```

2.1.2 Umgebungsvariabeln

Weitere optionale Programmparameter können als Umgebungsvariablen gesetzt werden.

Name	Möglicher Wert	Beschreibung
VERBOSE	false (Standard)	Aktiviert die Ausgabe von Debug-Informationen in die Konsole (STD_OUT und STD_ERR)
	true	
THEME	dungeon (Standard)	Basierend auf dem Dungeon Tileset
	classic	Basierend auf den Bildern, ausgegeben von der FH Wedel

2.2 Problemanalyse und Realisation

2.2.1 Räumliche Berechnungen

Problemanalyse

Das Spielbrett von "EscapeBot" umfasst zwei Dimensionen. In diesen zwei Dimensionen soll sich eine Spielfigur bewegen und mit dem Spielbrett interagieren. Um Positionen zu bestimmen sind Berechnungen in einem zweidimensionalen Raum notwendig.

Realisationsanalyse

Für Berechnungen in einem zweidimensionalen Raum bietet sich die Vektorrechnung aus der linearen Algebra an. Anstatt jede Koordinate einzeln zu berechnen, können bei der Vektorrechnung mehrere Attribute mit der gleichen Anweisung verarbeitet werden.

Das Spielbrett repräsentiert zwar einen zweidimensionalen Raum, allerdings ist eine Implementierung eines dreidimensionalen Vektorraums sinnvoller. Zwar ist es trivial zweidimensional in einem Unterraum des dreidimensionalen Vektorraums zu rechen; umgekehrt ist das aber nicht der Fall. Um so also für potentielle Anforderungen vorbereitet zu sein die einen dreidimensionalen Raum benötigen, ist es angebrachter diesen direkt zu implementieren.

Realisationsbeschreibung

Mit der Klasse `Vector` repräsentieren wir einen zwei- bzw. dreidimensionalen Vektor. Dieser besteht aus drei immutablen Attributen (`x`, `y`, `z`). Letztendlich nutzen wir nur `x` und `y`, aber wie in der Analyse bereits beschrieben stellt das kein Problem dar. Wir erstellen Vektoren mit einem zweidimensionalen Konstruktor (`Vector(int, int)`) der `z` 0 setzt, und für den Entwickler ist es von einem zweidimensionalen Vektorsystem nicht zu unterscheiden.

Die benötigten Rechenoperationen zwischen den Vektoren werden als Methoden in der `Vector`-Klasse implementiert. Da die `Vector`-Instanzen immutable sind, wird als Ergebnis einer Rechenoperation ein neuer Vektor zurückgegeben, der das Ergebnis repräsentiert.

2.2.2 Logging

Problemanalyse

Logging ist ein wichtiges Werkzeug, um die Wartung und Entwicklung von Software einfacher und genauer zu machen. Allerdings ist, aufgrund der Anforderungen an das Programm, die Ausgabe von Debug-Informationen in die Konsole im normalen Programmablauf nicht möglich.

Außerdem bietet die Ausgabe mit `System.out.println()` wenig Möglichkeiten und Informationen. Die wiederholte Ausgabe von unterschiedlichen Informationen in die Konsole führt zu einer unübersichtlichen Ansammlung an Informationen ohne Kontext.

Realisationsanalyse

Schalter für Konsolen-Ausgabe Um die Ausgabe von Debug-Informationen in die Konsole im normalen Programmbetrieb zu verhindern, bietet es sich an, diese Funktion hinter einer manuellen Aktivierung zu verbergen. Dafür eignen sich unterschiedliche Möglichkeiten:

Flag beim Programmstart Eine Möglichkeit, einen Schalter unterzubringen, wäre als sogenannte `Flag` beim Programmstart. Also durch das übergeben von Programmparametern in einer Konsolen Umgebung. Allerdings erfordert dies, dass man das Programm in einer Konsolenumgebung startet, was im gewöhnlichen Entwicklungsprozess oder Programmbetrieb nicht häufig passiert. Es gibt zwar auch die Möglichkeit über die "Run-Configurations" von "Intellji Idea" diese Programmparameter zu übergeben, dies ist aber nicht auf alle Editoren standardisiert.

Schalter im GUI Es bietet sich an den Schalter als tatsächlichen Schalter in der grafischen Oberfläche unterzubringen, z.B. in einem "Einstellung"-Menüpunkt. Der damit verbundene Aufwand ist aber nicht nur in der Erstimplementierung erheblich aufwendiger als andere Lösungen, sondern fordert auch eine aktive Wartung bzw. Anpassung an das restliche Programm, wenn sich dieses ändert.

Umgebungsvariable Umgebungsvariablen werden außerhalb des Programms gesetzt und müssen auch nicht bei jedem Programmstart neu gesetzt werden. Mit dieser Lösung entfällt der Aufwand, eine grafische Oberfläche programmieren und warten zu müssen. Auch der Konfigurationsaufwand, um das Programm immer mit einem bestimmten Programmparameter starten zu müssen, wird so umgangen. Zusätzlich ist die Java-Schnittstelle, um Umgebungsvariablen abzufragen, sehr einfach und verlässlich aufgebaut.

Basierend auf den analysierten Möglichkeiten ist es offensichtlich, dass Umgebungsvariablen für die Konfiguration am besten geeignet sind.

Kontext Informationen für Konsolenausgabe Wie bereits in der Problemanalyse[2.2.2] angesprochen ist der übliche Weg, Debug-Informationen in die Konsole auszugeben, unzureichend. Es ist also erforderlich eine eigene Funktionalität zu implementieren, die zusätzlich zur eigentlichen Nachricht, die ausgeben werden soll, auch Kontext-Informationen hinzufügt. Um die Ausgabe trotz zusätzlicher Informationen übersichtlich zu halten, müssen die unterschiedlichen Teile der Ausgabe visuell getrennt und unterscheidbar sein.

Realisationsbeschreibung

Es bietet sich an, beide Probleme, die konfigurierbare Ausgabe und das Hinzufügen von Kontextinformationen, zentral in einer Klasse zu implementieren. Diese Klasse (`Log`) verwaltet eine globale statische Variable die von der Umgebungsvariable `VERBOSE`[2.1.1] abhängig ist. Die implementierten Ausgabefunktionen dieser Klasse würden dann bedingt von der statischen Variable ihre Ausgabe unterdrücken.

Das Hinzufügen, beziehungsweise das automatische Erstellen von zusätzlichen Kontextinformationen, wird auch in dieser Klasse implementiert. Man kommt an viele nützliche Informationen, wenn man eine neue `Exception` erstellt (ohne diese zu werfen). Mit dieser Methode kommen wir an Informationen wie den `StackTrace` oder auch, an die Datei und Codezeile, in der die Ausgabe getätigt wurde. In Kombination mit der Funktionalität von Intellji Idea, die ermöglicht auf `File Name:Line Number` zu klicken und direkt zu dieser Stelle geführt zu werden, führt dies zu einem deutlich schnelleren Nachvollziehen des Programmablaufs.

Die Platzierung der Kontextinformationen, beziehungsweise die strukturierte Ausgabe, wird definiert durch mehrere Konstanten, die Format-`Strings` enthalten. Diese werden dann von den implementierten Funktionen genutzt, um ihre Ausgabe entsprechend zu formatieren. An dieser Stelle werden auch weitere visuelle Anpassungen der Ausgabe eingefügt. Wenn die verwendete Konsole dies unterstützt, werden bestimmte Ausgaben farbig oder in anderen Schriftgewichten ausgegeben.

Die Kontextausgabe beinhaltet folgende Informationen:

- mikrosekundengenaue Zeitangabe
- Datei und Codezeile der Log-Ausgabe
- Package, Klasse und Funktion, in der diese Ausgabe aufgerufen wurde

2.2.3 Spielbrett

Problemanalyse

Die Anforderungen definieren ein beliebig großes Spielbrett mit sechs unterschiedlichen Feldtypen. Diese Feldtypen haben jeweils Eigenschaften die ihr Verhalten bestimmen, bzw. das Verhalten mit ihnen beeinflussen.

Realisationsanalyse

Für die Repräsentation eines zweidimensionalen Spielbrett bieten sich zwei Datenstrukturen an.

Map<Vector[2.2.1], FieldType[1.3.7]> Da wir bereits einen Datentypen implementiert haben, der eine Position beschreiben kann, bietet es sich an, diesen auch zur Verwaltung von örtlich bezogenen Informationen zu verwenden. Allerdings führt die Verwendung einer *Map* zu weiterer Komplexität auf Code Ebene und während der Laufzeit.

FieldType[][] Ein eher klassischer Ansatz ist die Verwendung eines mehrdimensionalen Arrays. Zwar sind Arrays an sich etwas *unbeholfen* in Java, und meist werden `Lists` bevorzugt, aber da Arrays wenig Overhead während der Laufzeit haben sind sie sehr schnell. Da diese Datenstruktur sowieso in einer Wrapper-Klasse implementiert wird, kann man die Arrays abstrahieren und die Interaktionen mit dem Array absichern.

Da dieses Spielbrett bereits den größten Umfang der hypothetischen Spielwelt verwaltet, bietet es sich an, die Verwaltung der Position des Bot-Charaters in diese Klasse zu legen. Somit spart man zusätzliche Komplexität an anderer Stelle, und man hat eine zentrale Klasse, die die Wahrheit der Spielwelt repräsentiert.

Realisationsbeschreibung

Um das Spielbrett möglichst effizient zu halten nutzen wir das zweidimensionale `Array` um das Spielbrett zu repräsentieren. Dieses wird aber komplett abstrahiert und es existieren Get- und Set-Methoden um einen abgesicherten Zugriff auf das Array zu ermöglichen.

Wir nutzen das Spielbrett als SSOT[1] für alle mit der Spielwelt assoziierten Informationen. Daraus folgend übernimmt die Spielbrett-Klasse auch zusätzliche Aufgaben, unter anderem die Verwaltung des Spielercharakter (Position und Blickrichtung) und den Zugriff auf Funktionen zur Spielbrettverifikation[2.2.6] und der Generation einer Lösung[2.2.7].

Eine Besonderheit bei der Grundimplementierung dieser Get- und Set-Methoden sind statische Funktionen. Diese Funktionen ermöglichen die Operationen auf beliebigen zweidimensionalen `FieldTypeArrays` anzuwenden. Das kommt bei der Implementierung der Level-Verifikation[2.2.6] zum Einsatz.

2.2.4 Bot (Spielercharakter)

Problemanalyse

Der zentrale *Gameplay Loop* des Spiels besteht daraus, Anweisungen anzuordnen, um den Spielercharakter zu *programmieren*, diese Anweisungen auszuführen, dann den Ausgang zu bewerten und entweder die Anweisungen anzupassen oder ein anderes Level zu starten. Der Spielercharakter bzw. der Bot muss in der Lage sein folgende Dinge zu erfüllen:

- Fähigkeiten vom Spielercharakter implementieren
  - Bewegungen *Siehe: Kapitel 1.3.8*
  - Interaktionen mit Spielbrett
    (z.B. gegen Wände laufen, in Abgründe fallen oder ins Ziel gehen.)
- Prozeduren ausführen
  - Zuordnung von Anweisung zu Funktionen
  - Auflösung geschachtelter Prozeduraufrufe
- Ereignisse sammeln (Siehe Realisation: 2.2.5)
  (z.B. wenn ein Bot über eine Münze läuft, muss eine Bewegung und das Einsammeln der Münze protokolliert werden)

Realisationsanalyse

Um den Code modular und sauber zu halten bietet es sich an, die Fähigkeiten als Verantwortung des Spielercharakters als eigene Klasse zu implementieren. Da das Spiel einen strikt linearen Ablauf hat (*Programmieren* → *Ausführen* → *Ereignisse werden abgespielt*), kann jede Ausführung von einem Start-Zustand (`GameLevel`) generiert werden. Somit ist es nicht notwendig, einen Spielercharakter außerhalb von dieser Ausführung zu erhalten, bzw. die Instanz des Spielercharakters kann mit jedem Spielstart neu erstellt werden.

Mit dem vom Nutzer initialisierten Start werden alle *programmierten* Prozeduren übergeben und der Charakter kann diese umsetzen. Um diese Prozeduren umsetzen zu können müssen aber die geschachtelten Aufrufe aufgelöst werden, also die Stelle der Prozeduraufruf-Anweisung mit der entsprechenden Prozedur. Wichtig hier zu beachten ist auch, dass per Anforderung das GUI wissen muss welche Ereignisse zu welchen Anweisungen zuzuordnen sind.

Die Fähigkeiten des Spielercharakter müssen in Übereinstimmung mit dem Spielbrett stehen, also muss der Bot in der Lage sein direkt auf das Spielbrett zugreifen zu können.

Realisationsbeschreibung

Der Bot ist als eigene Klasse implementiert Um mit dem Spielbrett interagieren und seine eigene Position verwalten zu können, benötigt er eine aktuelle Kopie des Spielbretts. Die Methoden des Bots sind dafür verantwortlich zu überprüfen, ob die ausgeführte Aktion vom Bot zulässig ist und entsprechend als `Action` zu protokollieren.

[1] Single Source Of Truth

Prozedur auflösen Da der Spieler in der Lage ist, Prozeduraufrufe zu verwenden die andere Prozeduren aufrufen, müssen diese *aufgelöst* werden. Da die Liste von Anweisungen eher eine Warteschlange ist, die einzeln abgearbeitet werden kann, bietet sich an, dies auch für die Prozeduraufrufe zu tun. Wenn wir auf einen Prozeduraufruf in der Warteschlange treffen, springen wir zu der Anweisungsliste der aufgerufenen Prozedur, führen diese aus und springen danach wieder in die Hauptliste nach dem Prozeduraufruf. Das gleiche Prinzip funktioniert auch für verschachtelte Aufrufe.

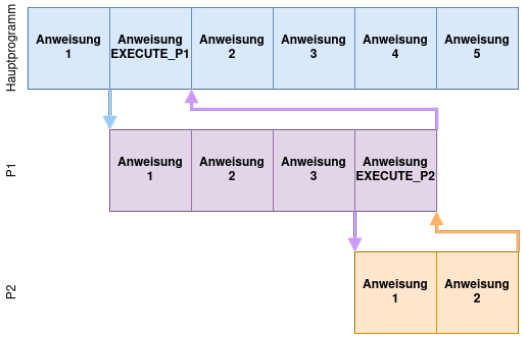


Abbildung 2.4.1: Prozeduren auflösen

2.2.5 Von Anweisungen zu Aktionen

Problemanalyse

Im Spiel baut man Prozeduren zusammen. Diese bestehen aus Anweisungen für den Spielercharakter. Wenn man das Spiel startet, werden die Anweisungen an die Logik übergeben. Um die Verarbeitung der Anweisungen zu visualisieren, brauchen wir eine Auflistung aller Ereignisse, die bei der Ausführung der Anweisungen auftreten.

Realisationsanalyse

Das Spiel hat einen strikt linearen Ablauf ohne parallele Prozesse und wird jedes mal von einem fest definierten Zustand gestartet. Somit bietet sich auch eine lineare Verarbeitung an. Die drei vom Benutzer erstellten Prozeduren werden in ihrer Ursprungsform verarbeitet und an die Bot-Implementierung[2.2.4] übergeben. Der Bot ist somit verantwortlich für die Erstellung der Zusammenfassung der Ereignisse.

Realisationsbeschreibung

Im Bot werden neben der eigentlichen Implementierung der möglichen Bot Interaktionen auch Funktionen zur Verarbeitung bzw. Abarbeitung der vom Nutzer erstellten Prozeduren benötigt. Diese nehmen die Prozeduren in ihrer Ursprungsform an und implementieren die Auflösung der Prozeduraufrufe und die Zuteilung der Anweisungen zu den verantwortlichen Funktionen.

Action Für die Zusammenfassung der Ereignisse, die bei der Verarbeitung der Anweisungen auftreten, wird eine Klasse benötigt die ein solches Ereignis repräsentiert. Dies wird mit der `Action`-Klasse umgesetzt. Eine Action besteht aus einem `Type` (*Was ist passiert?*), einen "Schnappschuss" des Charakterzustand (*Wo ist und war der Charakter?*) und Meta-Informationen zum Ablauf der Prozedur (*Welche Anweisung und Prozedur wird momentan ausgeführt?*). Die Funktionen des Bots geben dann eine Liste von dieser Klasse mit denen in der Ausführung aufgetretenen Ereignissen zurück. Diese werden zusammengetragen und nach der vollständigen Ausführung an das GUI zurückgegeben.

2.2.6 Verifikation und Serialisierung von Level-Dateien

Problemanalyse

Da der Benutzer in der Lage ist Level als Datei zu laden und zu speichern, ist es erforderlich Spieldaten zu serialisieren und bei der deserialisierung den Inhalt zu verifizieren. Ein Level muss bestimmte Anforderungen erfüllen, um vom Programm genutzt werden zu können.

Anforderungen an Level-Datei:

- `REQUIRED` Darstellung des Spielbrett als zweidimensionales Array
  - Mindestgröße: 1x1
  - Muss Startfeld enthalten
  - Spielbrett kann beliebige Größe haben
  - FieldType[1.3.7] in Ordinal-Repräsentation

- `REQUIRED` BotRotation in Ordinal-Repräsentation
- `OPTIONAL` Name des Levels

Anmerkungen

- Anforderungen an Level-Dateien unterscheiden sich zu den Anforderungen an ein gültiges Level (2.2.7)
- `OPTIONAL` Ist optional, muss nicht vorhanden sein.
- `REQUIRED` Anforderungen müssen existieren und verifiziert werden. Ohne diese Daten kann das Level nicht geladen werden.

Realisationsanalyse

Für das Serialisieren von Spieldaten wird die Library *GSON*[2] verwendet. Diese kann in unterschiedlichen Umfängen verwendet werden, primär zu unterschieden in welchem Schritt man die Verifikation durchführt.

Verifikation und Konvertierung vor und nach (De-)Serialiserung In der einfachsten Form kann GSON ein beliebiges POJO[3] in einen JSON-String umwandeln. Es bietet sich an, die Konvertierung von internen Datentypen in die von der Anforderung erwarteten Ordinal-Repräsentationen, in simple Funktionen zu packen. Diese können dann vor und nach GSON aufgerufen werden. Bei der Deserialiserung kann dann nach dem einlesen der JSON die Ordinal-Repräsentation wieder in die interne Datenstruktur konvertiert werden. Nach der Konvertierung kann diese Datenstruktur überprüft werden.

Diese Möglichkeit erscheint auf den ersten Blick zwar simpel, kann aber schnell zu einer unübersichtlichen Verkettung von unintuitiven Abhängigkeiten unterschiedlicher Klassen und Funktionen führen. Dies würde der Modularisierung und der Lesbarkeit des Codes schaden.

Verifikation und Konvertierung während (De-)Serialiserung Eine andere Möglichkeit, die GSON bietet, ist die Implementierung von sogenannten JSON-Adaptern. Diese ermöglichen, bestimmte Klassen und Datenstrukturen beim Serialisieren/Deserialisieren zu konvertieren und in gewünschte Formate zu bringen. Die Konvertierung der Daten wird so vom Entwickler wegabstrahiert und er muss sich nach der initialen Implementierung der Adapter keine Gedanken mehr über ihren Nutzen machen. Dies erlaubt eine sauberere Modularisierung des Codes, als die erste Möglichkeit und führt zu mehr lesbarem und übersichtlicherem Code.

Trotz der zusätzlichen Komplexität bei der Implementierung der JSON-Adapter überwiegen die Vorteile die Nachteile.

Realisationsbeschreibung

Die Level-Datei besteht aus drei Schlüssel-Wert-Paaren, ein optionales und zwei erforderliche:

- **name** Name des Level.
- **field** Daten des Spielfeld. Aufgebaut als zweidimensionales Array, Feldtypen in Ganzzahl-Repräsentation.
- **botRotation** Die Blickrichtung des Bot beim Start (als Ganzzahl).

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

Der Aufbau der Level-Datei steht in direktem Konflikt mit den internen Datenstrukturen des Programms. Um diese Konflikte zu überbrücken, wird eine Datenklasse benötigt, die zwischen den beiden Strukturen adaptiert. Diese Klasse besteht aus der Implementierung des Spielbretts und einem extra Attribut, um die Startrichtung des Bots zu speichern. Bot-bezogene Informationen werden

[2]Eine Java-Serialiserungsbibliothek, um Java Objekte in JSON zu konvertieren. github.com/google/gson
[3]Plain old java object

intern eigentlich im Spielbrett verwaltet. Um die Anforderungen der Level-Datei zu erfüllen wird die Startrichtung aber redundant in der Datenklasse und im Spielbrett gespeichert. Dies wird in der Implementierung der Konstruktor und Getter der Datenklasse sichergestellt.

@SerializedName Statt `field` wird das Spielbrett intern als `board` bezeichnet, ähnlich bei der Richtung des Bots. Um trotzdem intern den Code einheitlich zu halten, können wir die `@SerializedName` Annotation von GSON benutzen. Diese erlaubt einen abweichenden Namen für die Serialisierung zu spezifizieren.

@Required Da nicht immer alle Attribute in einem Level erwartet werden brauchen wir einen Weg diese zu unterscheiden. Realisiert wird das mit einer selbst implementierten Annotation: `@Required`. Diese funktioniert zusammen mit dem JSON-Adapter für die Datenklasse und sucht in dem deserialisierten Objekt nach den Attributen mit der Annotation. Wenn diese nicht vorhanden sind, wird ein Fehler geworfen und die Deserialisierung abgebrochen.

```
public class GameLevel {
    String name;

    @Required
    @SerializedName("field")
    Board board;

    @Required
    @SerializedName("botRotation")
    Direction botDirection;
}
```

Abbildung 2.6.2: Daten-Klasse

Adapter-Implementierung Neben der bereits erwähnten Implementierung eines JSON-Adapters in der Datenklasse gibt es weitere JSON-Adapter für die Aufzählungstypen `FieldType` und `Direction`, um diese in ihre Ordinal-Werte zu konvertieren, sowie auch für das Spielbrett.

Name Das `name`-Attribut ist nicht Teil der Spezifikation der Anforderung und ist deshalb optional. Es kann genutzt werden um einen Levelnamen abweichend vom Dateinamen zu spezifizieren.

2.2.7 Level Lösbarkeit

Problemanalyse

Da der Benutzer in der Lage ist, eigene Level zu erstellen und externe Level zu importieren, ist nicht davon auszugehen, dass diese alle Anforderungen eines gültigen Level erfüllen. Somit ist eine automatische Verifikation der Level notwendig, um entweder die Ausführung zu verhindern oder den Benutzer auf die ungültigen Level hinzuweisen.

Anforderungen an ein gültiges Level:

a) 1 Startfeld

b) 1 Türfeld

c) 0..n Münzen

d) Alle Münzen von Startfeld erreichbar

e) Tür von Startfeld erreichbar

f) *Lösungsweg kann generiert werden*

Realisationsanalyse

Feldanzahl Für die Anforderungen a) und b) ist eine Zählung der Felder im Spielbrett ausreichend. Anforderung c) benötigt keine Zählung, da das Münzfeld[3.7.6] optional ist.

Erreichbarkeit Anforderungen d) und e) benötigen etwas mehr algorithmischen Aufwand. Empfiehlt es sich, den Floodfill-Algorithmus auf das Spielbrett von der Start[3.7.9]-Position aus anzuwenden. Nach der *Flutung* durch den Algorithmus sind alle Felder erreichbar von der Startposition markiert. Wenn alle Münzen und das Tür[3.7.7]-Feld markiert sind, sind diese auch vom Start aus erreichbar.
Floodfill in seiner gewöhnlichen Ausführung[4] verbreitet sich nur über seine direkten Nachbarn. Da

[4]Wikipedia Artikel: wikipedia.org/wiki/Floodfill

der Charakter in diesem Spiel auch in der Lage ist, über den direkten Nachbarn hinaus zu springen sind Anpassungen an den Algorithmus notwendig.

Lösungsweg Das Suchen eines Lösungswegs wird in einem eigenen Kapitel behandelt: 2.2.8

Realisationsbeschreibung

Zur Analyse des Spielfelds fassen wir die in der Realisationsanalyse vorgestellten Methoden zusammen in eine Methode der `Board`-Klasse. Diese gibt dann eine Liste von `Record`[5]-`Board.Problem` zurück.

Die Resultate sollen dem Nutzer mit den wichtigsten Informationen für die Lösung des Problems als Popup angezeigt werden. Dafür benutzen wir ein `Modal` von JavaFx und nutzen diese neue Stage um die Probleme aufzulisten. Die Liste wird generiert mit den in der Analyse angesprochenen Methoden. Diese werden in einer Analyse-Funktion zusammengefasst und in einer Liste von `Board.Problem`-Elementen zurückgegeben. Wenn keine Probleme zurückgegeben werden, ist das Level als konform mit den Anforderungen anzusehen.

Floodfill Anpassungen Da der normale Floodfill-Algorithmus für unsere Anforderungen unzureichend ist, müssen wir ihn erweitern. Speziell geht es um die Möglichkeit einen Abgrund zu überspringen. Wir nutzen eine rekursive Implementierung von Floodfill. Dieser ruft rekursiv die nächsten vier Richtungen auf, an dieser Stelle erweitern wir mit einem boolschen Ausdruck, um bedingt, wenn ein Sprung möglich ist, das nächste Feld einer bestimmten Richtung auch fluten zu können.

Anmerkung Im GUI ist diese Funktion nicht als "Level-Lösbarkeit" definiert, sondern als "Lösung finden", deshalb weicht die Rückgabe von Informationen von den Anforderungen ab. Wenn das Level lösbar ist, wird dies nicht extra dem Spieler gesagt, da das Füllen der Prozeduren dem erwarteten Erfolgsfall entspricht.

2.2.8 Pathfinding

Problemanalyse

Anforderung an das Programm ist, bei der Lösbarkeits-Analyse[2.2.7] auch eine mögliche Lösung (drei Prozeduren mit Anweisungen) zu generieren. Um diese generieren zu können ist ein Lösungsweg vom Startfeld[3.7.9], wenn vorhanden über alle Münzen[3.7.6], bis zur Tür[3.7.6] notwendig.

Realisationsanalyse

Um die Liste von Anweisungen generieren zu können, die später einen korrekten Weg zum Ziel darstellen sollen, brauchen wir die aneinander liegenden Felder im Spielbrett dieses Weges. Es gibt unterschiedliche Pathfinding-Systeme die man implementieren kann, einige davon wurden in Betracht gezogen.

Floodfill Bereits in den Anforderungen als Beispiel genannt ist der Floodfill-Algorithmus. Dieser sehr simple Algorithmus wird genutzt, um eine Menge von aneinander liegenden Pixeln zu *fluten*, z.B. um eine Fläche mit einer Farbe zu füllen. Eigentlich ist dieser Algorithmus nicht geeignet, um als Pathfinding genutzt zu werden. Theoretisch ist dies aber möglich. Wenn wir das Feld `FieldType.START` als Startpunkt verwenden und solange das Spielbrett fluten bis wir auf `FieldType.DOOR` treffen, können wir dem Weg folgen, den der Floodfill-Algorithmus gegangen ist. Zu beachten ist, dass dies zu extrem ineffizienten Lösungen führen kann. Zusätzlich ist die Grundimplementierung von Floodfill nicht darauf ausgelegt, sich an ihren Weg zu erinnern. Nach ersten Versuchen, diesen Algorithmus umzusetzen, habe ich die Implementierung abgebrochen, da ich den Aufwand, der durch

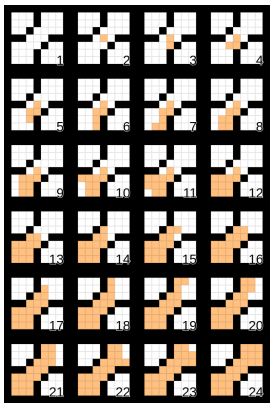


Abbildung 2.8.3: Floodfill
By (Java Doc) Record ermöglichen es By André Karwath aka Aka - Own work, CC BY-SA 2.5, wikimedia.org/w/index.php?curid=481651

[5]Record sind eine eingeschränke Deklaration von class, eingeführt mit Java 15, um Datenklassen in sehr kurzer Schreibweise zu deklarieren.

die umfangreichen Modifikationen notwendig ist, für unverhältnismäßig bewerte.

Anmerkung Floodfill hat dennoch Verwendung im Programm an anderer Stelle gefunden. Mehr dazu im Kapitel: 2.2.7

A* Der A-Stern-Algorithmus ist sehr ähnlich zu Dijkstra, wird aber als *informed search algorithm* bezeichnet. A* gilt als deutlich effizienter für große Datenmengen, z.B. Video-Spiel-Karten. Für das sehr kleine Spielbrett dieses Programms ist der zusätzliche Aufwand mehr Nachteil als die effizientere Laufzeit Vorteil ist.

Dijkstra Der Dijkstra-Algorithmus ist speziell dafür entwickelt den kürzesten Weg von einem Startknoten in einem gewichteten Graph zu finden. Um Dijkstra auf unserem Spielbrett anwenden zu können ist es notwendig dieses erst zu einem Graphen zu konvertieren. Das Spielbrett hat eine sehr simple Geographie. Zu direkten Nachbarn (nicht diagonal, mit einem Abgrund zwischen) bewegt man sich mit einer Anweisung bzw. einem Schritt.

Realisationsbeschreibung

Erzeugung eines Graphen In der Analyse haben wir bereits festgestellt das das Spielbrett eine sehr simple Geographie repräsentiert. Das ermöglicht uns die Komplexität des Graphen start zu verrigern in dem wir die Gewichtung der Kanten weg lassen, bzw. alle Kanten gleich gewichten. Trotzdem ist die Erzeugung eines Graph vom Spielbrett ein mehrstufiger Prozess.

1. Feld → Knoten Im ersten Schritt generieren wir für jedes Feld im Spielbrett einen Knoten. Dieser wird gespeichert in einer `Map`, als Schlüssel wird die Position im Spielbrett verwendet.

2. Verbinde benachbarte Knoten Als zweiter Schritt verbinden wir alle Knoten mit ihren Nachbarn. Dieser Schritt wird nur durchgeführt wenn das Feld begehbar ist. Knoten gelten als benachbart wenn ihre Position direkt neben dem Feld liegt das gerade untersucht wird, oder wenn genau ein Abgrund zwischen dem untersuchten Feld und dem Nachbarsfeld liegt.

Anmerkung zu Wendungen Wendungen des Bot benötigen extra Aufwand. Ein Pfad mit weniger *Kurven* wäre also zu Bevorzugen, allerdings ist die Berücksichtigung der aktuellen Position des Bot und die relative Position des nächsten Feld ein nicht trivialer Mehraufwand, weshalb ich mich entschieden habe dies nicht zu berücksichtigen.

Suche nach dem richtigen Pfad Wie in der Analyse bereits geklärt benutzen wir eine eigene Implementierung des Dijkstra-Algorithmus um den optimalen Weg zwischen Start und Tür zu finden. Der Weg ist aber nicht immer direkt, wenn das Level Münzen[3.7.6] enthält müssen diese zuerst eingesammelt werden bevor man das Exit-Feld betreten kann. Die Lösung ist uns alle Münzen aus dem Graphen zu ziehen und anhand ihrer Distanz vom Startfeld zu sortieren und dann diese Felder nacheinander abzugehen. Dieser Ansatz setzt voraus die die Level so gestaltet sind das man die Münzen in einer intuitiven Reihenfolge ablaufen kann und trotzdem zum Exit kommt. Je nachdem ob Münzen vorhanden waren wird also der optimale Weg vom Start oder von der letzten Münze berechnet.

Knoten → Anweisungen Es werden zwei Knoten betrachtet, wenn deren Positionen zwei Felder auseinander liegen wird ein Sprung eingefügt, wenn sie ein Feld auseinander liegen wird eine `Forward`-Anweisung angefügt. Da die Knoten ohne Richtungs-Information gespeichert werden, ist es Notwendig bei der Konvertierung zu Anweisungen diese zu erzeugen. Dafür wird bei jedem Schritt die relative Richtung zwischen zwei Position überprüft und mit der aktuellen Position verglichen. Wenn sich diese Richtung ändert wird eine `TURN`-Anweisung eingefügt. Wenn das nächste Feld die Tür[3.7.7] ist wird die `EXIT`-Anweisung[3.8.15] gesetzt und der restliche Pfad geleert.

2.2.9 Anweisungen auf Prozeduren verteilen

Problemanalyse

Wenn ein Lösungsweg gefunden und in Anweisungen übersetzt wurde haben wir eine Liste mit Anweisungen die für fast alle Level nicht in das Hauptprogramm (Kapazität: 12 Anweisungen) passen wird.

Realisationsanalyse

Es ist notwendig, die gesamten zur Verfügung stehenden Kapazitäten zu nutzen:

- Hauptprogramm: 12 Anweisungen
- P1: 8 Anweisungen
- P2: 8 Anweisungen

Die Subprozeduren `P1` und `P2` können mit den Anweisungen `EXECUTE_P1` und `EXECUTE_P2` im Hauptprogramm und in der jeweils anderen Subprozedur aufgerufen werden. Sich wiederholende Sequenzen in der Liste von Anweisungen können in diese Subprozeduren gepackt und in der Liste von Anweisungen für das Hauptprogramm mit den Prozedur-Aufrufen (z.B. `EXECUTE_P1`) ersetzt werden.

Realisationsbeschreibung

Die Funktion `Procedure#enhancedOptimize(...)` nimmt eine Liste mit Anweisungen (Ohne Subprozeduraufrufe) an und gibt drei Listen zurück. Diese drei Listen repräsentieren in folgender Reihenfolge die Programme für den Nutzer: "Hauptprogramm", "P1" und "P2". Diese Funktion benutzt `searchRepeatingSequences(...)` um sich wiederholende Sequenzen zu finden. Die Suche nach den Wiederholungen und die Auswahl der am besten passenden Sequenz haben den größten algorithmischen Aufwand.

Wiederholungen finden Um Wiederholungen finden zu können, müssen wir wissen, was sich wiederholen soll. Um etwas untersuchen zu können, nehmen wir unsere komplette Liste von Anweisungen und schneiden in einer verschachtelten Schleife jeweils an den gegenüberliegenden Enden der Liste Anweisungen ab. Die daraus resultierende kürzere Sequenz kann dann in der kompletten Liste gesucht werden. Wenn ein Vorkommen gefunden wird, wird dieses mit der Länge und Position gespeichert.

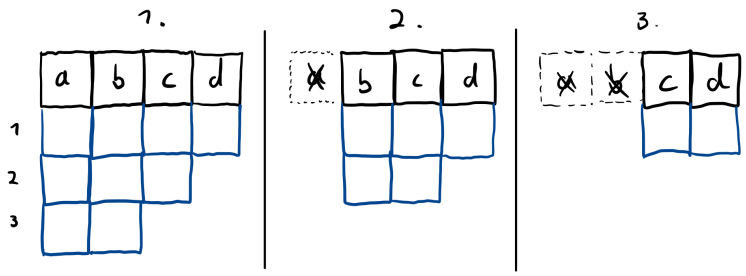


Abbildung 2.9.4: Anweisungen abschneiden

In der Abbildung 2.9.4 sieht man den Ablauf des "Abschneidens" visualisiert. Auf der horizontalen Achse wird der Ablauf der äußeren Schleife abgebildet und auf der vertikalen Achse der Ablauf der inneren Schleife. Die resultierenden Sequenzen (in blau) werden dann in der ursprünglichen Liste gesucht.

Anweisungen Optimieren Nachdem die Liste der gefunden Sequenzen generiert wurde, kann diese anhand der Länge und Häufigkeit der gefundenen Sequenzen bewertet werden. Die Sequenz die die meisten Anweisungen einspart, wird als erste Subprozedur gewählt. Im Hauptprogramm werden die Stellen mit dem Subprozeduraufruf ersetzt. Danach wird alles auf dem neuen Hauptprogramm erneut durchgeführt, um Subprozedur 2 zu füllen.

2.2.10 Komponenten basierte Benutzeroberfläche

Problemanalyse

Für die Umsetzung der geforderten Benutzeroberfläche werden mehrere komplexere Funktionalitäten nötig, z.B. Wahl einer als Bild dargestellten Anweisung, oder das Wechseln zwischen zwei Spielbrettern mit unterschiedlichem und teils inkompatiblem Funktionsumfang. Um diese komplexen Anforderungen zu erfüllen, sind umfangreiche Erweiterungen von bestehenden JavaFx-Komponenten und ein teils aufwändiges State-Management notwendig.

Realisationsanalyse

Die Umsetzung dieser Anforderungen in einer klassischen Controller-Struktur würde schnell zu langem unübersichtlichen Code führen und aufwändiges State Management in einer einzelnen Klasse erfordern. Um die Komplexität übersichtlich zu halten, werden die einzelnen Elemente aufgeteilt und unabhängig voneinander gehalten. Diese neuen Elemente, ab hier als "Komponenten" bezeichnet, sind Klassen die bestehende Javafx Klassen erweitern und diese Erweiterungen so weit wie möglich intern verwalten. Nur notwendige Funktionen und Attribute werden nach außen gereicht.

Diese Komponenten müssen soweit mit JavaFx-Konventionen übereinstimmen, dass sie nahtlos in bestehende JavaFx-Strukturen wie z.B. einem .FXML-Dokument integriert werden können.

Realisationsbeschreibung

Dieser Ansatz einer "Component-First"-Umgebung gibt uns die Möglichkeit, das Programm mit einem einzelnen FXML Dokument und dem dazugehörigen Controller umzusetzen, ohne dass diese zu komplex oder unübersichtlich werden würden. Der Controller dient damit als Klebstoff bzw. Verbindungsstück zwischen den Komponenten und verwaltet die Kommunikation zwischen Komponenten, Benutzerinteraktionen und Logik.

Konstruktor Argumente Wenn ein Argument bestimmte Konstruktor Parameter benötigt um initialisert zu werden, können diese mit der Annotation `@NamedArg` im FXML-Dokument verfügbar gemacht werden. Diese haben allerdings immer den Datentyp `String`. Wenn man Werte eines anderen Datentypen übergeben will, müssen diese geparst oder serialisiert werden.

JavaFx Properties JavaFx Properties sind Klassen bzw. Objekte die es ermöglichen, mit einem `ChangeListener` auf Veränderungen von ihrem internen Wert zu reagieren. Um dies im Rahmen der JavaFx-Konventionen umzusetzen, implementiert man bei Klassen die diese Properties verwenden, zusätzlich zu einer Get-Methode eine Methode, die die Property selbst zurückgibt. So ist es möglich in anderen Programmteilen auf Veränderungen von dieser Property zu reagieren.

2.2.11 CSS-basiertes Styling

Problemanalyse

Das Programm soll sofort erkennbar sein und eine eindeutige Wiedererkennung gewährleisten können. Daraus folgt, dass das Programm von den Standard-Styles der JavaFx-Library abweichen soll.

Realisationsanalyse

JavaFx hat eine eigene Version des CSS3 Standards implementiert. Dieses kann man benutzten, um mit einigen der gängigen CSS-Funktionalitäten das Aussehen von JavaFx-Elementen zu verändern. Dies geht über mehrere Möglichkeiten.

Node#setStyle(String) Jede Node bietet die Möglichkeit an einen CSS-Style-String direkt für diese Node zu setzen. Das ignoriert und überschreibt alle anderen Formen des Style Inputs und ist somit eine Möglichkeit, Style zu überschreiben oder zu erzwingen.

CSS Stylesheets Ähnlich wie in der Web-Entwicklung bietet JavaFx die Möglichkeit CSS-Dateien als Stylesheets zu importieren. Dies tut man in der FXML Datei der obersten Stage (Die Stage, von denen alle anderen Stages und Elemente ausgehen). In dieser Datei kann man wie üblich CSS-Klassen schreiben, die man mit CSS-Selektoren bestimmten Elementen zuordnet.

Realisationsbeschreibung

Die Implementierung nutzt ein einzelnes Stylesheet, das in die `shell.fxml` geladen wird. In diesem Stylesheet befinden sich Styles um die programmeigenen Komponenten zu gestalten. Abgesehen davon werden Standard-Styles überschrieben und erweitert.

Anmerkung: Es hat sich gezeigt, dass diese Methode nicht zuverlässig ist. So hat die Klasse `GameInfoBar` Probleme, ihren Style mit Klassen zu ändern. Deshalb wurde in dieser Klasse auf die `Node#setStyle(String)`-Methode gesetzt um eine Umgestaltung zu erzwingen.

2.2.12 Texturen Verwaltung

Problemanalyse

Bestimmte Komponenten im Spiel benötigen Rastergrafiken um korrekt dargestellt zu werden. Diese Komponenten existieren an undefinierten Stellen der Benutzeroberfläche und müssen zu undefinierten Zeitpunkten erstellbar sein.

Realisationsanalyse

Es ist extrem ineffizient eine bestimmte Textur bei jedem Zugriff, von der Datei laden zu lassen. JavaFx hat das auch erkannt und trennt die eigentlichen Bilddaten (`Image`) von der visuellen Darstellung (`ImageView`). Es lassen sich beliebig viele `ImageView`s von einem Image erstellen, und das Image wird im Arbeitsspeicher gelassen. Das ermöglicht kurze Ladezeiten und (wenig Verkehr) der Festplatte.

Es bietet sich somit an eine zentrale Stelle zur Verwaltung dieser Texturen bzw. `Image`s zu haben die zur Erstellung von ImageViews genutzt werden kann. Weitere Vorteile, die sich aus dieser zentralen Verwaltung ergeben, sind die Möglichkeiten die Images einheitlich zu konfigurieren (z.B. Höhe und Breite), sowie technischere Einstellungen wie das Ausschalten von Antialiasing tätigen zu können. Antialiasing ist praktisch bei natürlichen Subjekten in Fotos, um diese auch bei niedrigeren Auflösungen gut aussehen zu lassen, führt aber zu sehr verschwommenen Darstellungen bei Pixelart bzw. Bitmap-Bildern.

Realisationsbeschreibung

Implementiert wird die zentrale Verwaltung als Aufzählungstyp im `gui`-Package. Die einzelnen Werte des Aufzählungstypen werden nach den korrespondierenden Feldtypen oder Anweisungen benannt und enthalten jeweils ein initialisiertes `Image`. Dieser Aufzählungstyp besitzt wiederum Hilfsmethoden die erlauben aus den `Image`s `ImageView`s zu generieren und auch die Zuweisung der logischen Daten zu ihren Texturen mit `Texture#of(FieldType)` ist unkompliziert zu benutzen.

Texturen mit unterschiedlichen Richtungen Das Startfeld hat, anders als die anderen Felder, die Möglichkeit basierend auf einem internen Zustand (Start Richtung) das Aussehen zu ändern (Richtung in die der Spielercharakter auf der Textur guckt). Eine einfache Rotation fällt hier weg, weil die gewählten Texturen aus dem Dungeon-Tileset[2.4.2] einen Hintergrund haben, die die Ausrichtung des Hintergrunds erkennen lassen. Die Darstellung des Bots und die Darstellung des Hintergrunds müssen von einander getrennt rotiert werden. Der Aufwand verbunden mit der Implementierung eines Systems eigens für diese Trennung wäre unverhältnismäßig. Deswegen wird das bestehende System erweitert und die Rotation des Startfeld auf vier Texturen aufgeteilt, die dann je nach Zustand mit `Texture#of(FieldType, Direction)` geladen werden können. Wenn bei `Texture#of(FieldType, Direction)` kein Startfeld übergeben wird, wird die Richtung nicht beachtet.

Theme wechseln Im Kapitel 2.1.2 wird beschrieben wie man zwischen den unterschiedlichen Mengen von Texturen (von hier an als `Textureset` bezeichnet) wechseln kann. Um dies möglich zu machen, ist eine saubere Pflege der Dateien notwendig. Die Unterscheidung zwischen den `Texturesets` wird durch Ordnerpfade gemacht. Die Dateinamen in den Ordnern müssen dem Aufzählungstypen entnommen werden um den jeweiligen Texturen zugeordnet werden zu können.

2.2.13 Auswahl eines Enum-Elementes

Problemanalyse

In den Anforderungen ist die Möglichkeit definiert, wahlweise Anweisungen oder Feldtypen aus einer visuellen Zusammenstellung zu wählen. Diese beiden Möglichkeiten sind von einander unabhängig in unterschiedlichen Kontexten, aber mit gleicher Funktionalität.

Realisationsanalyse

Ein Problem mit mehreren Vorkommen und unterschiedlichen Daten fordert eine möglichst abstrakte implementierung einer Komponente. Diese Komponente sollte möglichst flexibel einsetzbar sein und nur die wichtigsten Informationen (Auswahl des `Enum`-Elementes) nach außen tragen.

Eine Auswahl aus einer Menge ist eine typische Problemstellung für Formulare. Normalerweise wird dies mit einem `RadioButton` gelöst. Die Standard-Implementierung erfüllt aber nicht alle Anforderungen. Der `RadioButton` muss aus einem Bild bestehen und der visuelle Hinweis für die Markierung soll ein Rahmen um das Bild sein.

Diese `Radio`-Auswahl korrespondiert in der Logik mit zwei Aufzählungstypen (`FieldType` und `Instruction`) und soll auch als Element dieser Aufzählungstypen weiterverwendet werden.

Realisationsbeschreibung

Für die Realisierung brauchen wir zwei Komponenten: einmal die eigentliche Auswahl-Komponente, die eine Menge von Enum-Elementen darstellt und uns zurückgibt was ausgewählt ist, und die Button-Komponente mit der Textur des Enum-Elements, die wie ein `RadioButton` funktioniert.

RadioButton Erweiterung Wir implementieren eine eigene Klasse (`TextureCard`) und erweitern damit `javafx...RadioButton`. `RadioButton` gibt uns bereits eine Menge der Grundfunktionalitäten die wir brauchen, zum Beispiel, ein Element aus einer Gruppe auszuwählen. Wichtig für unsere Anforderungen ist die Darstellung der Textur anstatt der normalen Darstellung des `RadioButton`s.

Um das zu realisieren entfernen wir das Standard-Styling der `RadioButton`-Klasse und nutzen unser eigenes Styling. Dieses besteht nur aus der angezeigten `Graphic` und einem Rahmen, wenn die Komponente gerade ausgewählt ist.

Auswahl-Komponente Die Auswahl-Komponente ist so aufgebaut, dass wir ein `EnumSet` bei der Initialisierung übergeben. Dieses Enum muss Texturen assoziert haben. Die Komponente hat eine `Property`, die das gerade ausgewählte Enum nach außen trägt. Da die Komponente auf Java-Generics basiert sind wir sehr flexibel im Einsatz. So ist es möglich die Komponente für die Auswahl von Anweisungen im Spiel, oder der Auswahl von Feld-Typen im Editor zu nutzen.

2.2.14 Spielbrett darstellen

Problemanalyse

Die beiden Hauptfunktionalitäten des Programms sind der Spielablauf und das Bearbeiten der Level. Beides benötigt ein `Grid` aus Feldtypen bzw. ihren Texturen. Für den Spielablauf wird eine animierbare Spielfigur benötigt. Diese muss sich frei über das Feld bewegen können und auch mit diesem interagieren können (z.B. Münze aufheben). Für den Editor wird die Funktionalität benötigt, den FeldTypen eines Feldes im Spielbrett zu ändern.

Realisationsanalyse

Die Problematik ist sehr gut dafür geeignet mit einer Vererbung gelöst zu werden. Wir haben die Basis-Funktionalitäten:

- Texturen von Feldtypen in einem Raster anzeigen
- Die Funktionalität des Spiels: Spielfigur animieren
- Die Funktionalität des Editor: Spielbrett bearbeiten (Feldtypen auswechseln)

Animation Für die Animation der Spielfigur ist das `Transition`-System von JavaFx geeignet. Es baut auf das interne Animationssystem von JavaFx auf und bietet Bausteine an, um Animationen zusammen zu bauen. Am wichtigsten hier ist die `SequentialTransition`, mit der es möglich ist, mehrere `Transitions` bzw. Animationen nacheinander abzuspielen und die `TranslateTransition`, die eine Node in eine Richtung bzw. zu einer Position schiebt.

Realisationsbeschreibung

TextureGrid (Basis) Das `TextureGrid` ist die Basisklasse und implementiert das eigentliche Raster von Texturen. Dieses wird aufgebaut anhand der internen Kopie des Spielbretts. Diese Implementierung bietet auch die Möglichkeit, das Raster jederzeit mit einem einzelnen Funktionsaufruf (`TextureGrid#draw()`) neu zu erstellen. Das ist notwendig, wenn sich die Felder vom Spielbrett verändern, da dies nicht automatisch erkannt wird.

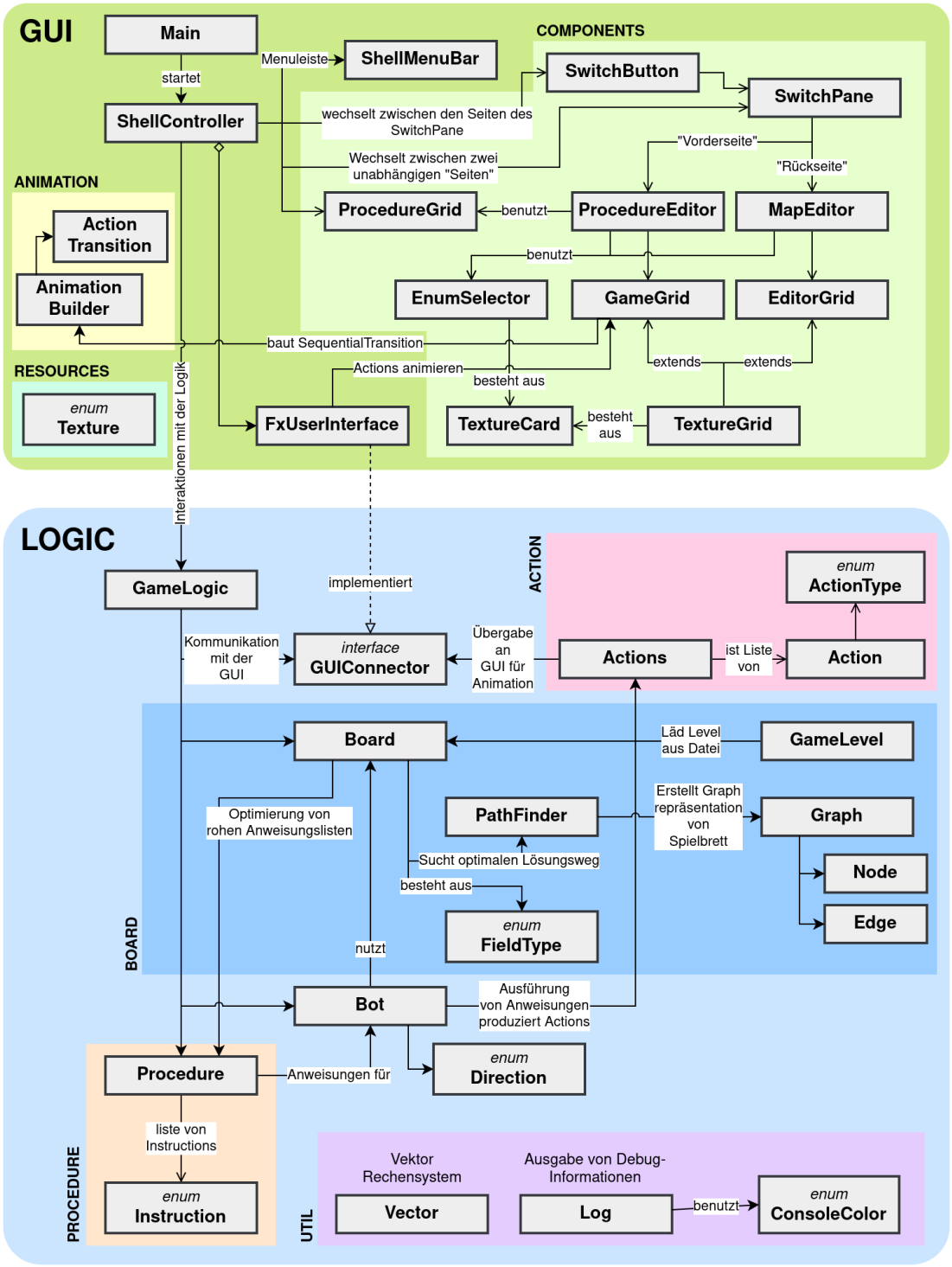
GameGrid (Erweiterung von TextureGrid) Diese Implementierung erweitert `TextureGrid` und fügt die Möglichkeit hinzu, eine Spielfigur zu animieren. Wie in der Analyse angesprochen verwenden wir JavaFx `Transitions` um, die Spielfigur zu animieren. Von der Logik bekommt das GameGrid eine Liste von `Actions`[2.2.5]. Daraus soll dann die Animation gebaut werden. Zusätzlich zu den Bewegungen des Spielercharakters sollen auch Veränderungen am Spielbrett durchgeführt werden.

AnimationBuilder Um das Zusammenbauen der Animation zu vereinfachen implementieren wir eine Hilfsklasse. Dieser `AnimationBuilder` ist inspiriert vom `StringBuilder` und bietet in erster Linie die Möglichkeit, vordefinierte Animationen zusammenzuhängen und als `SequentialTransition` zu exportieren.

Spielfeld verändern Bei `Actions` wie `COLLECT_COIN` ist zusätzlich zu der Animation der Spielfigur eine Veränderung des Spielfelds notwendig (z.B. `FieldType.COIN` → `FieldType.NORMAL`). Dies kann durch die Möglichkeit einen `onFinished-EventHandler` bei einer Animation setzen zu können, realisiert werden. Dieser `EventHandler` ist eine Funktion die aufgerufen wird, wenn die Animation beendet ist. Bei `COLLECT_COIN` können wir beim Event `onFinished` das Feld, auf dem sich der Spielercharakter befindet, zu einem normalen Feld verändern.

EditorGrid (Erweiterung von EditorGrid) Diese Implementierung erweitert `TextureGrid` und fügt die Möglichkeit hinzu, Felder mit Mausklick zu verändern. Diese Funktionalität wird in die einzelnen `TextureGrid.Field` bzw. `EditorGrid.Field`-Elemente implementiert. Ein Field repräsentiert eine einzelne Textur im `Grid` und besteht vor allem aus einer `TextureCard`. Im `EditorGrid` überschreiben wir das `Field` mit einer Implementierung, die bei Mausklick das aktuelle Feld im `Board` vom `TextureGrid` verändert und ein `TextureGrid#draw()` Aufruf macht um die Veränderung anzuzeigen.

2.3 Programmorganisationsplan



2.3.1 Anmerkungen

`gui.resources.Texture`
Um sicherzustellen das die Bilddateien nur einmal initialisert werden, werden die Referenzen statisch mit dem Enum verwaltet

`logic.util.Vector`
Implementiert 3D Vektor mit Rechenoperationen.

`logic.util.Log`
Implementiert ein simples aber ausführliches Logging System das sich mit Umgebungsvariabeln steuern lässt.

2.3.2 Programm Übersicht

Package	Klasse	Beschreibung
gui	Main	Diese Klasse ist der Einstiegspunkt für die JRE. Außerdem werden hier unerwartete Exceptions aufgefangen und das GUI initialisiert.
	FxUserInterface	Ist die Interface-Implementierng des GUI-Connector für die Logik. Primäres Error-Handling ist hier implementiert.
gui .shell	ShellController	Controller der `Shell` Scene. Implementierung von User Interaktionen.
	ShellMenuBar	Implementierung der Menüleiste am oberen Rand des Programm.
	shell.fxml	FXML Markdown Datei, beschreibt die Struktur der GUI.
gui .resources	Texture	Zentrales Texturen Verwaltung. Initialisiert die Images von den lokalen Dateien und stellt sie dem Programm zur Verfügung
gui .control	EnumSelectorGroup	Komponente die dem User eine Auswahl an Bildern gibt. Diese Bilder repräsentieren eine Enum-Konstante. Die Auswahl des Users kann über eine JavaFX-Property gelesen werden
	SwitchButton	Eine Erweiterung des Normalen Buttons. Ermöglicht die Nutzung von zwei verschiedenen Aktionen abhängig vom internen State
	SwitchPane	SwitchPane verwaltet Zwei virtuelle Seiten die einzeln Sichtbar werden können.
	TextureCard	Erweiterung des normalen JavaFX-Radiobutton um Bilder bzw. Texturen anzuzeigen
gui .components	TextureGrid	Stellt eine Visuelle Repräsentation eines Spielbretts dar, wie es in der Logik definiert ist. Nutzt intern eine Kopie des momentanen "Board"
	MapEditor	Enthält die Kontroll-Elemente um das EditorGrid zu bearbeiten
	EditorGrid	Erweiterung des TextureGrid um die Funktionalität einzelne Felder zu ändern
gui .components .game	GameGrid	Erweiterung des TextureGrid um die Funktionalität den Spielercharakter zu animieren
	GameInfoBar	Info-Leiste die unter dem Spielbrett eingeblendet wird
	ProcedureEditor	Enthält die Kontrollelemente um die Prozeduren für den Bot zu "programmieren"
	ProcedureGrid	Ein einzelnes Grid aus Feldern die mit Anweisungen gefüllt werden können

	ProcedureField	Ein einzelnes Feld das eine Anweisung enthalten kann
gui .animation	AnimationBuilder	Eine Builder-Klasse die nach dem Vorbild des `StringBuilder` eine `SequentialTransition` aus mehreren angefügten Animationen erstellt
	ActionTransition	Eine `Transition` Erweiterung die keine Animation enthält aber ein `Runnable` stattdessen ausführt. Genutzt um TextureGrid Veränderungen sequentiell durchzuführen
logic	GameLogic	Verwaltet Spielbrett und Spielercharakter, führt Prozeduren mit Bot auf SpielBrett aus. Gibt resultierende Aktionen an die GUI
	GameLevel	Spiellevel basiert auf vorgegebenem Dateiformat. Enthält alle Level relevanten Informationen
	Direction	Auflistungstyp von möglichen Richtungen. Enthält Hilfsmethoden um die Verwedung von `Direction` einfacher zu machen.
	Bot	Enthält Ausführungslogik für die Anweisungen
	GUIConnector	Interface für "Logik-zu-GUI" Interaktionen
logic .action	Action	Repräsentiert ein Ereignis in der Level Ausführung. Zum Beispiel Bewegung des Bots oder Auswahl von Anweisungskarten
	Actions	Wrapperklasse für eine Liste von Actions. Enthält nützliche Hilfsmethoden zum einfacheren arbeiten mit Anweisungslisten
	ActionType	Auflistungstyp von unterschiedlichen Action Arten
logic .board	Board	Logik Repräsentation des Spielbretts
	FieldType	Auflistungstyp von Feldtypen.
logic .board .graph	Pathfinder	Sucht Pfäde im Spielbrett und sucht optimalen Lösungsweg
	Graph	Graphenrepräsentation des Spielbretts
	Edge	Rerpäsentation einer Kante im Graph
	Node	Rerpäsentation eines Knoten im Graph
logic .procedure	Procedure	Wrapperklasse um eine Liste von Anweisungen. Enthält Hilfsmethoden zur optimalen Verteilung von Anweisungen auf Unterprozeduren
	Instruction	Aufzählungstyp von Anweisungen
logic .util	Log	Hilfsklasse für erleichterte Konsolenausgabe oder das abschalten dieser Ausgabe.
	ConsoleColor	Aufzählungstyp von ANSI Konsolen Farben
	Vector	Implementierung eines 3D Vektor basierten Rechensystem

2.4 Datein

2.4.1 Level

Pfad src/gui/resources/resources/level

Dateiname	Beschreibung
level0.json	Eigenes Beispiellevel. Bietet die Möglichkeit viele Szenarien zu testen
level1.json	Das erste Beispiellevel, vorgegeben von den Anforderungen
level2.json	Das zweite Beispiellevel, vorgegeben von den Anforderungen
level3.json	Das dritte Beispiellevel, vorgegeben von den Anforderungen
level4.json	Das vierte Beispiellevel, vorgegeben von den Anforderungen
level5.json	Das fünfte Beispiellevel, vorgegeben von den Anforderungen

2.4.2 Texturen

“Classic” Texturen

- Dateipfad: src/gui/resources/resources/classic

“Dungeon” Texturen

- Dateipfad: src/gui/resources/resources/dungeon
- Quelle: 0x72.itch.io/dungeontileset-ii

Dateiname	Beschreibung
AGehen.png	gerader Pfeil nach oben
ALinks.png	runder Pfeil nach links
ARechts.png	runder Pfeil nach rechts
ASpringen.png	runder Pfeil über einen Abgrund
AExit.png	Text “EXIT”
AProzedur1.png	Text “P1”
AProzedur2.png	Text “P2”
FAbgrund.png	komplett schwarz
FMauer.png	braune Ziegel Wand
FMünze.png	gelbe Münze auf braunen Grund
FNormal.png	braunes Feld mit dunkel braunen Rand
FTür.png	rotbraune “Holz”-Tür silbener Kreis als klinke
FBot.png	silbener Kreis mit roten Auge in Blickrichtung (3.1.1)
FStart_North.png FStart_East.png FStart_South.png FStart_West.png	Bot[3.1.1] (Spielercharakter) auf mit normalen Feld als Hintergrund. Das Feld ist auf vier Texturen aufgeteilt um den Hintergrund nicht mit zu rotieren.

2.5 Programmtests

Eine Auflistung von Tests für die graphische Oberfläche des Spiels. Logik wird von Komponententests automatisch getestet.

Im Spielablauf

Testfall	Ergebnis
Das Prozedur-Gitter über die Kapazität hinaus füllen.	Prozedur sollte nur Anweisungen ausführen die Im Prozedurgitter sichtbar sind.
Führe eine gültige liste von Anweisungen aus	Der Spielercharakter sollte sich entsprechend der Anweisungen bewegen und reagieren
Drücke während ein Level ausgeführt wird auf "Abort" in der Spiel Status Leiste	Die Ausführung sollte sich beenden.

Datei Laden

Testfall	Ergebnis
Versuche eine nicht-JSON Datei zu laden	Es muss eine Fehlermeldung erscheinen.
Level Datei mit größeren Feld laden	Das Spielbrett muss sich der neuen Größe anpassen
Level Datei mit kleineren Feld laden	Das Spielbrett muss sich der neuen Größe anpassen

Level Editor

Testfall	Ergebnis
Öffne den "Level Editor"	Das Spielbrett sollte mit Koordinaten angezeigt werden, eine Auswahl von Feld-Typen erscheinen und die Prozedur-Gitter disabled werden.
Wähle den Tür-Feldtyp aus	Drücke an zufällige Stellen im Spielbrett. Die vorherige Position der Tür sollte mit einer Wand ersetzt werden, so dass nur eine Tür existiert.
Wähle den Start-Feldtyp aus	Drücke an zufällige Stellen im Spielbrett. Die vorherige Position des Start sollte mit einem normalen Feld ersetzt werden, so dass nur ein Start existiert.
Wähle das Startfeld aus und drücke mehrmals auf das Startfeld im Spielbrett	Das Startfeld im Spielbrett sollte sich drehen.
Baue ein ungültiges Level und drücke auf "Solve Level"	Es sollte eine Liste mit Problemen erscheinen
Baue ein ungültiges Level und drücke auf "Back To The Game"	Es sollte eine Liste mit Problemen erscheinen
Lade ein gültiges Level und drücke auf "Solve Level"	Die Prozeduren sollten sich mit Anweisungen füllen
Level Datei mit kleineren Feld laden	Das Spielbrett muss sich der neuen Größe anpassen