

Programmierpraktikum

FloodPipe

Barth, Philip
Fachrichtung: Informatik
inf104861
5. Fachsemester
eingereicht am: 04.02.2023

Inhaltsverzeichnis

1	Benutzerhandbuch	4
1.1	Ablaufbedingungen	4
1.2	Programminstallation/Programmstart	4
1.3	Bedienungsanleitung	5
1.3.1	Spielaufbau	5
1.3.2	Spielverlauf	7
1.3.3	Spielende	8
1.3.4	Editor	9
1.3.5	Menüinteraktion	11
1.4	Fehlermeldungen	13
1.4.1	Im Editor	13
1.4.2	Beim Speichern/Laden	14
2	Programmiererhandbuch	16
2.1	Entwicklungskonfiguration	16
2.2	Problemanalyse und Realisation	16
2.2.1	Darstellung des Spielfelds	16
2.2.2	Darstellung des Editors	18
2.2.3	Auswahl der Spieloptionen	20
2.2.4	Darstellung des Füllstatus	21
2.2.5	Laden und Speichern von Dateien	24
2.2.6	Aufbau der Spiellogik	25
2.2.7	Interne Speicherung einer Spielsituation	27
2.2.8	Interne Speicherung der einzelnen Felder	29
2.2.9	Generierung einer Spielsituation	30
2.2.10	Modifizierung der Spielsituation	31
2.2.11	Konvertierung zwischen Spielsituation und Datei	32
2.2.12	GUI-Schnittstelle	33
2.3	Algorithmen	34
2.3.1	Spielfeldgenerierung	34
2.3.2	Ermittlung der Füllstände	37
2.4	Programmorganisationsplan	39
2.4.1	Paketzuordnung	41
2.4.2	Erläuterung des Programmorganisationsplans	41
2.5	Dateien	41
2.6	Programmtests	42

Abbildungsverzeichnis

1	Programmstart Terminal	4
2	Startbildschirm FloodPipe	5
3	Optionsfenster	6
4	Gerade	7
5	Kurve	7
6	T-Stück	7
7	Endstück	7
8	Quellenmarkierung auf einer Geraden	7
9	Beispielhafter Zwischenstand	8
10	Gelöstes Spielfeld	9
11	Editormodus	10
12	Dateimenü	12
13	Animationsmenü	12
14	Fehlermeldung beim Speichern einer Datei	13
15	Programmorganisationsplan gui	39
16	Programmorganisationsplan logic	40

Literatur

- [1] Erklärung Breitensuche <http://www.burgnetz.de/otg/informatik/graphen/breitensuche.html>.
Aufgerufen am: 23.01.2023
- [2] Erklärung Tiefensuche <http://www.inf.fu-berlin.de/lehre/SS10/infb/dfs.pdf> Aufge-
rufen am 23.01.2023
- [3] Erklärung FloodFill-Algorithmus <https://www.javatpoint.com/computer-graphics-flood-fill-algorithm> Aufgerufen am 23.01.2023
- [4] Erklärung JSON <https://www.datenbanken-verstehen.de/lexikon/json/> Aufgerufen
am: 18.01.2023
- [5] Aufgabenstellung FloodPipe <https://lms.fh-wedel.de/mod/assign/view.php?id=25424>
Aufgerufen am: 18.01.2023

1 Benutzerhandbuch

1.1 Ablaufbedingungen

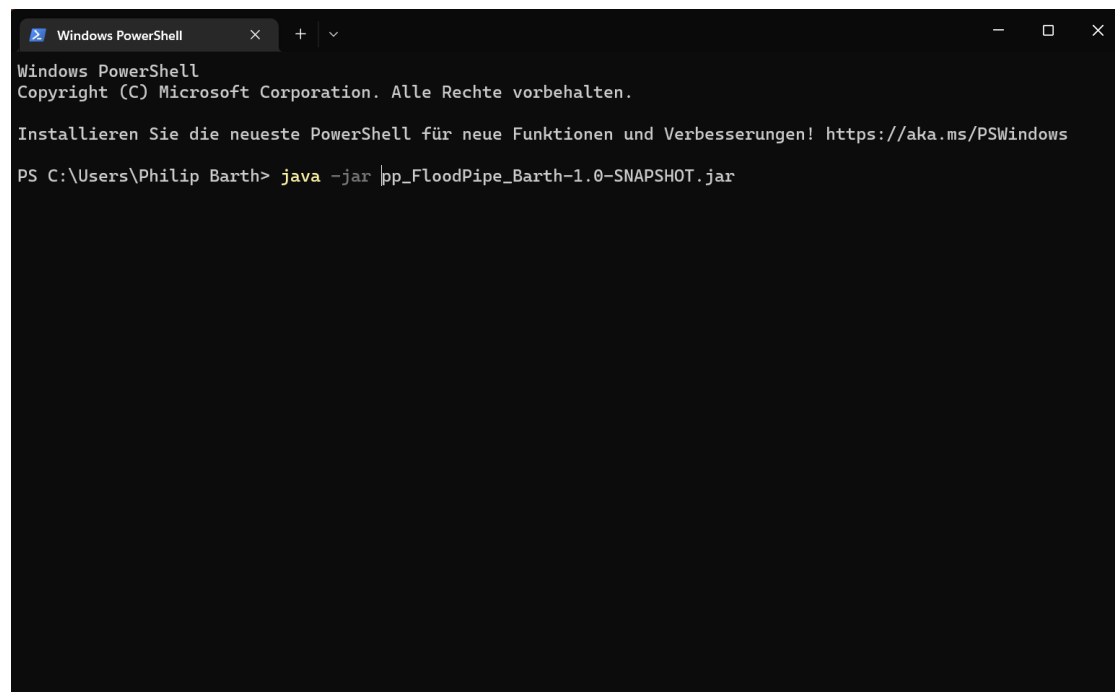
Für den Ablauf des kompilierten Programms werden bestimmte Softwarekomponenten benötigt. Eine erfolgreiche Ausführung mit abweichenden Spezifikationen kann ebenfalls möglich sein, diese kann aber nicht garantiert werden. Es folgt eine Auflistung dieser empfohlenen Komponenten:

1. Java Development Kit 17
2. Betriebssystem: Windows

1.2 Programminstallation/Programmstart

Im Folgenden wird beschrieben, welche Schritte durch den Nutzer durchzuführen sind, um das Spiel zu installieren und anschließend zu starten.

Der Name der Spieldatei lautet "pp_FloodPipe_Barth-1.0-SNAPSHOT.jar". Diese Datei muss weder entpackt noch installiert werden. Der Start des Programmes erfolgt über einen Doppelklick mit der linken Maustaste. Eine weitere Möglichkeit das Programm zu starten ist, in dem Ordner, in dem die Spieldatei gespeichert ist, ein Terminal zu öffnen und den Befehl "java -jar pp_FloodPipe_Barth-1.0-SNAPSHOT.jar" auszuführen. Diese Methode des Programmstarts sollte dann genutzt werden, wenn das Öffnen per Mausklick nicht funktionieren sollte.



```
Windows PowerShell
Copyright (C) Microsoft Corporation. Alle Rechte vorbehalten.

Installieren Sie die neueste PowerShell für neue Funktionen und Verbesserungen! https://aka.ms/PSWindows

PS C:\Users\Philip Barth> java -jar pp_FloodPipe_Barth-1.0-SNAPSHOT.jar
```

Abbildung 1: Programmstart Terminal

1.3 Bedienungsanleitung



Abbildung 2: Startbildschirm FloodPipe

Bei dem Spiel "FloodPipe" handelt es sich um ein Puzzle-basiertes Computerspiel, bei dem der Spieler zu Beginn ein zufällig erstelltes Spielfeld dargestellt bekommt, auf dem Rohrstücke in unterschiedlichen Ausrichtungen platziert sind. In den folgenden Kapiteln wird auf die Regeln des Spiels eingegangen, der Spielablauf und das Ziel des Spiels charakterisiert, sowie zusätzliche Bedienungsmöglichkeiten aufgezeigt.

1.3.1 Spiel Aufbau

Nach dem Start des Spiels wird der Startbildschirm angezeigt (siehe Abbildung 2). Durch einen Klick auf das dort dargestellte Bild gelangt der Spieler anschließend in ein Optionsfenster, bei dem die Rahmenbedingungen des Spiels zu definieren sind. Es besteht die Möglichkeit, ein Spielfeld zu erzeugen, welches zwischen zwei und 15 Spalten sowie Reihen besitzt. Dabei können die Spalten und Reihen unabhängig voneinander angepasst werden, wodurch auch nicht-quadratische Spielfelder erzeugt werden können. Außerdem kann eingestellt werden, zu welchem prozentualen Anteil Mauern maximal auf dem

Spielfeld vorhanden sein können. Dies stellt einen Maximalwert dar, es können also beispielsweise auch keine Mauern vorhanden sein, obwohl ein maximaler Maueranteil von 100 Prozent definiert wurde. Des Weiteren besteht die Möglichkeit, auszuwählen, ob das Spielfeld mit "Überlaufmodus" erstellt werden soll.

Dies ist ein fortgeschrittener Spielmodus, bei dem eine direkte Verbindung zwischen oberem und unterem, bzw. linkem und rechtem Spielfeldrand besteht, wodurch der Schwierigkeitsgrad erhöht werden kann. Standardmäßig voreingestellt ist ein Spielfeld mit zehn Spalten, zehn Reihen, einer maximalen Mauerbelegung von zehn Prozent und deaktiviertem "Überlaufmodus".

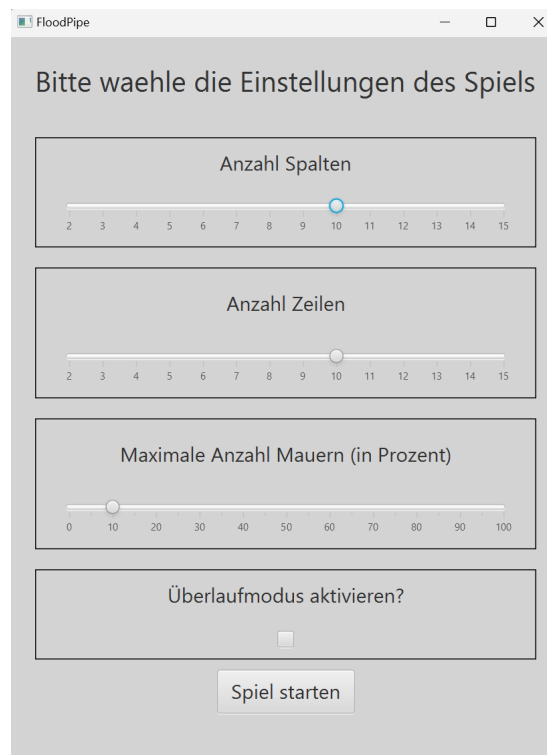


Abbildung 3: Optionsfenster

Ein Mausklick auf den Button "Spiel starten" erzeugt ein zufälliges gelöstes Spielfeld mit den vorgegebenen Rahmenbedingungen und genau einem Rohrstück, welches als Quelle markiert ist (Siehe beispielhaft Abbildung 8). Es existieren vier unterschiedliche Rohrtypen, die auf dem Spielfeld platziert werden können: Endstück, Kurve, Linie, T-Stück. Ein Endstück besitzt lediglich eine Öffnung, eine Kurve und eine Linie jeweils zwei Öffnungen, wobei diese bei der Linie gegenüber platziert sind und bei der Kurve benachbart und ein T-Stück, welches drei Öffnungen besitzt. Anschließend wird das Spielfeld gemischt, also jedes Rohrstück durch eine jeweils zufällige Anzahl von Drehungen rotiert. Diese Spielsituation ist durch den Spieler zu lösen.



Abbildung 4: Gerade

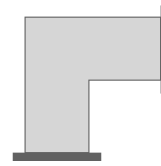


Abbildung 5: Kurve

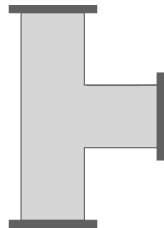


Abbildung 6: T-Stück

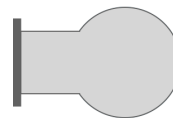


Abbildung 7: Endstück



Abbildung 8: Quellenmarkierung auf einer Geraden

1.3.2 Spielverlauf

Das Ziel des Spiels ist es, alle Rohrstücke auf dem Spielfeld so zu verbinden, dass jedes Rohrstück mit der Quelle verbunden ist und keine offenen Verbindungen vorliegen. Durch einen Klick auf eine Zelle des Spielfelds wird das darin befindliche Rohrstück rotiert. Hierbei wird zwischen einem Links- und einem Rechtsklick unterschieden. Ein Linksklick sorgt für eine Rotation des Rohrstücks um 90° nach links, ein Rechtsklick für eine Rotation um 90° nach rechts. Nach jedem Spielzug werden Felder, die durch die Aktion des Spielers nicht mehr mit der Quelle verbunden sind, geleert und neu verbundene Rohrstücke gefüllt. siehe Abbildung 9).

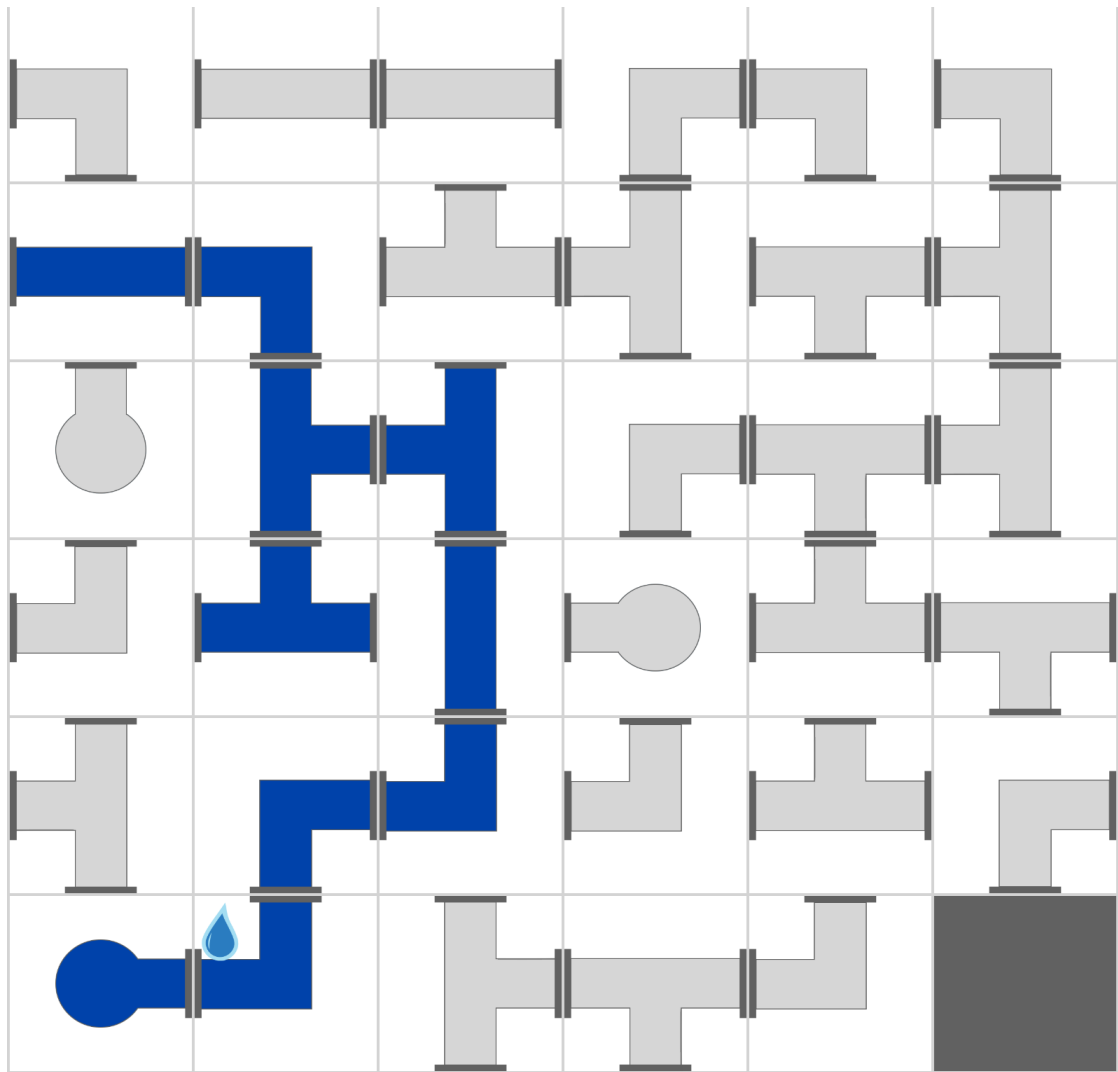


Abbildung 9: Beispielhafter Zwischenstand

1.3.3 Spielende

Hat der Spieler alle Rohrstücke derart rotiert, dass keine offenen Enden mehr vorhanden und alle Rohrstücke mit der Quelle verbunden sind, wird dem Spieler mitgeteilt, dass das Spiel gelöst ist. Hierfür wird am unteren Spielfeldrand eine entsprechende Meldung erzeugt und außerdem die Anzahl benötigter Spielzüge angezeigt (siehe Abbildung 10). Jeder Mausklick stellt hierbei einen Spielzug dar. Das Spielfeld wird mit Ende des Spiels deaktiviert, es können also keine weiteren Züge ausgeführt werden. Anschließend kann über einen Menüpunkt ein neues Spiel gestartet werden, oder das Spielfeld im Editor geöffnet werden.

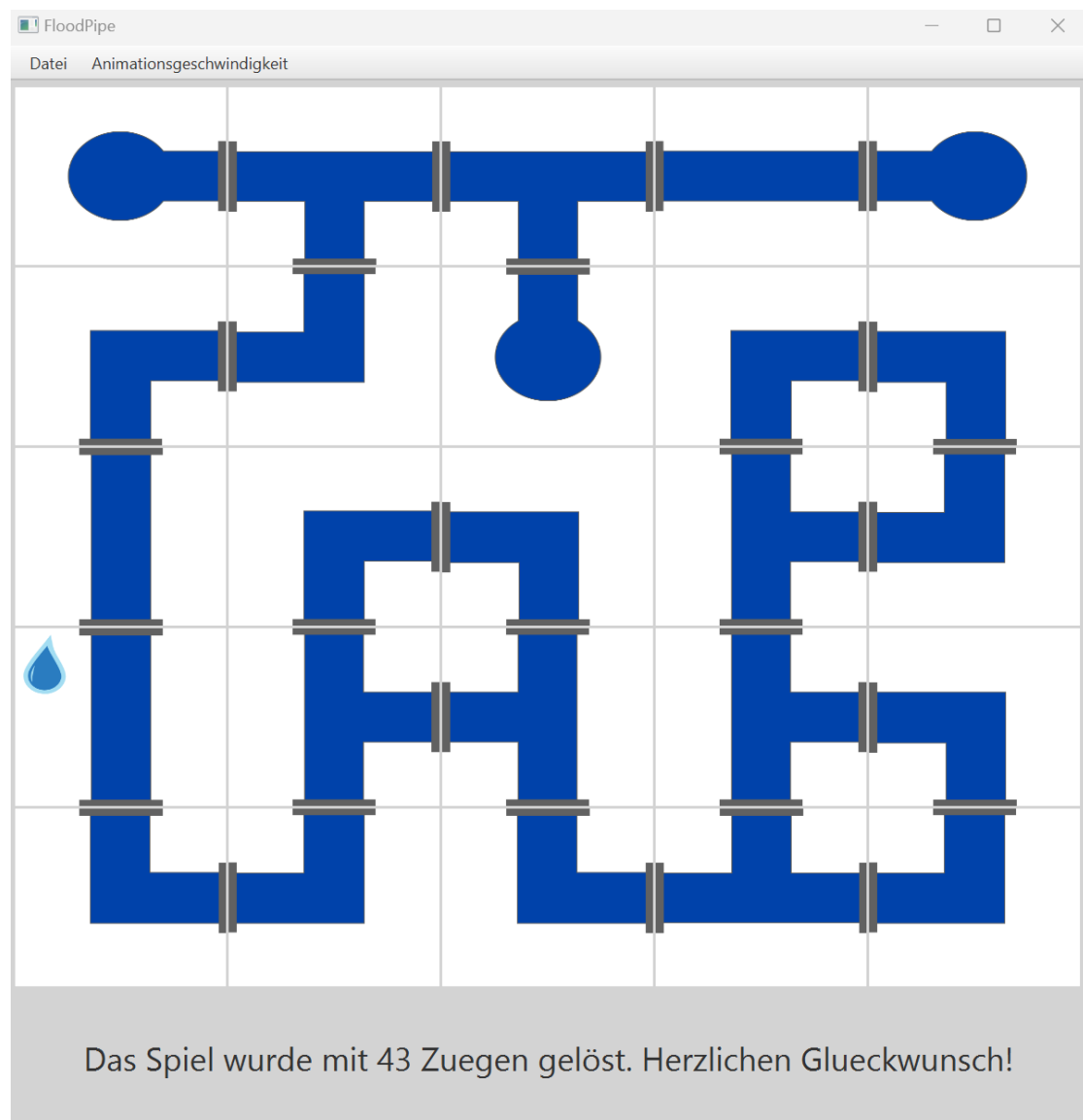


Abbildung 10: Gelöstes Spielfeld

1.3.4 Editor

Über einen Menüpunkt ist es für den Spieler möglich, in einen Editormodus zu schalten, um ein benutzerdefiniertes Spielfeld zu erzeugen, oder ein bestehendes Spielfeld zu verändern. In diesem wird die aktuelle Spielfeldbelegung übernommen, es kann aber auch mittels eines Mausklicks auf den Button "Neues Spielfeld" jederzeit ein neues Spielfeld erstellt, oder aber auch ein initial mit Mauern belegtes Spielfeld geladen werden (siehe Abbildung 11). Die Einstellungen des vorigen Spielfelds bleiben vorerst erhalten. Es wird also ein neues Spielfeld mit den Maßen der vorherigen Spielfeldsituation erstellt, bei dem jede Zelle mit einer Mauer belegt ist.

Neben dem Spielfeld werden die einsetzbaren Rohrstücke, das Mauerstück und eine Quelle dargestellt. Mittels Drag And Drop können die Rohrstücke und das Mauerstück auf ein beliebiges Feld gezogen werden. Durch den Drop wird die Belegung der Zelle anschließend überschrieben. Die Rohrstücke werden hierzu mit ihrer standardmäßigen Ausrichtung auf dem Feld dargestellt. Durch Links- bzw. Rechtsklick können diese ge-

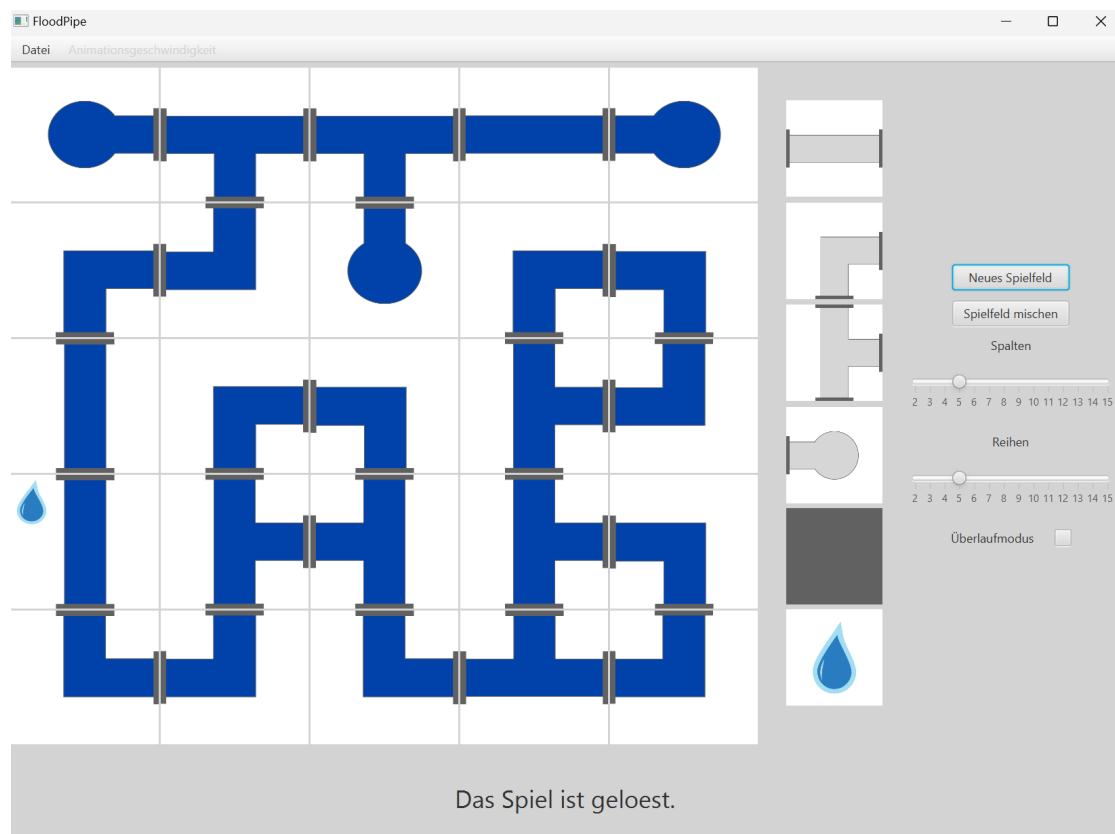


Abbildung 11: Editormodus

dreht werden. Sollte ein Mauerstück auf ein Feld, welches mit einer Quelle gekennzeichnet ist, gezogen werden, so wird die darauf befindliche Quelle entfernt und das Mauerstück abgelegt. Die Quelle kann lediglich auf Rohrstücke platziert werden, wobei jeweils nur eine Quelle zur Zeit auf dem Spielfeld existieren kann. Bei Belegung eines Feldes mit einer Quelle wird die zuvor gesetzte Quelle entfernt. Dem Spieler wird, wie auch im Spielmodus, angezeigt, wenn das Spielfeld gelöst ist. Anders als im Spiel wird das Spielfeld allerdings nicht deaktiviert und die Züge nicht angezeigt. Zusätzlich zur Anpassung einzelner Zellen des Spielfelds besteht die Möglichkeit, die Anzahl an Reihen und Spalten zu verändern. Die Zellenbelegung der noch vorhandenen Zellen innerhalb dieser Reihen und Spalten bleibt bestehen, neue Reihen und Spalten werden mit Mauerstücken aufgefüllt. Außerdem kann das Feld jederzeit gemixt werden, indem auf den Button "Spielfeld mischen" geklickt wird. Entscheidet sich der Benutzer dafür, dass das Editieren des Spielfelds abgeschlossen ist, so kann er entweder in den Spielfeldmodus wechseln, oder aber die Spielsituation speichern. Es muss allerdings darauf geachtet werden, dass eine Quelle auf dem Spielfeld definiert ist.

1.3.5 Menüinteraktion

Das Spiel zeigt eine Menüleiste, die dem Benutzer mehrere Auswahlmöglichkeiten in Untermenüs bietet. Im Folgenden wird erläutert, wofür diese unterschiedlichen Menüs genutzt werden können.

Datei In diesem Menü finden sich Möglichkeiten, um ein neues Spiel zu initialisieren, ein Spiel zu speichern oder zu laden, die Szene zu wechseln, oder das Spiel zu beenden (siehe Abbildung 12):

- **Neues Spiel**
Initialisiert ein neues Spiel. Hierzu wird dem Spieler erneut das Optionsfenster dargestellt (siehe Abbildung 3). Die Einstellungen des vorherigen Spiels sind voreingestellt und der Nutzer erhält die Möglichkeit, ein neues Spiel nach seinen Vorstellungen zu starten.
- **Spiel speichern**
Öffnet einen Dialog, um die Spielfelddatei zu speichern. Diese wird im JSON-Format abgespeichert und besitzt zwei Koordinaten für die Position der Quelle auf dem Spielfeld, einen Wert, der den Überlaufmodus als aktiviert oder deaktiviert symbolisiert und eine Repräsentation des Spielfelds. Sollte keine Quelle auf dem Spielfeld vorhanden sein, kann das Spielfeld nicht gespeichert werden.
- **Spiel laden**
Öffnet einen Dialog, um eine Spielfelddatei zu laden. Diese Spielfelddateien müssen ebenfalls im JSON-Format vorliegen, wobei auch nur passende Dateien angezeigt werden. Es wird empfohlen, lediglich zuvor gespeicherte Spielfelddateien, die aus einem Spiel oder einem mittels Editor erstellten Spielfeld resultieren, zu laden und diese nicht händisch zu manipulieren. Andernfalls können unterschiedliche Fehlersituationen entstehen (siehe 1.4).
- **Zum Editor wechseln/Zum Spiel wechseln**
Dieser Menüpunkt dient dazu, um zwischen Spiel und Editor zu wechseln. Zu beachten ist, dass nur zum Spielfeld gewechselt werden kann, wenn eine Quelle auf dem Spielfeld vorhanden ist. (siehe Abschnitt 1.3.4)
- **Spiel beenden**
Menüpunkt, der das Spiel beendet. Es wird ein Alert angezeigt, bei dem der Benutzer bestätigen muss, dass er das Spiel wirklich beenden möchte. Der Spielstand wird ohne diesen zu speichern verworfen.

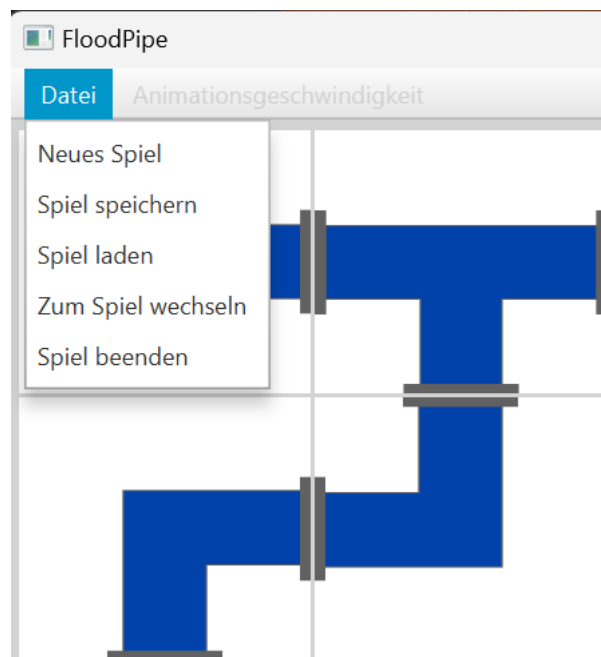


Abbildung 12: Dateimenü

Animationsgeschwindigkeiten Dieses Menü bietet dem Benutzer die Möglichkeit, die Geschwindigkeit der Animation, die für das Füllen der Rohrstücke sorgt, in drei unterschiedlichen Stufen einzustellen (siehe Abbildung 13), wobei standardmäßig "1x" ausgewählt ist. Bei ausgewählter "2x" Option wird die Geschwindigkeit verdoppelt, bei "3x" wird diese verdreifacht. Außerdem hat der Benutzer die Möglichkeit, die Animation auszuschalten. Dies sorgt dafür, das Füllen der verbundenen Rohrstücke ohne Verzögerung auszuführen.

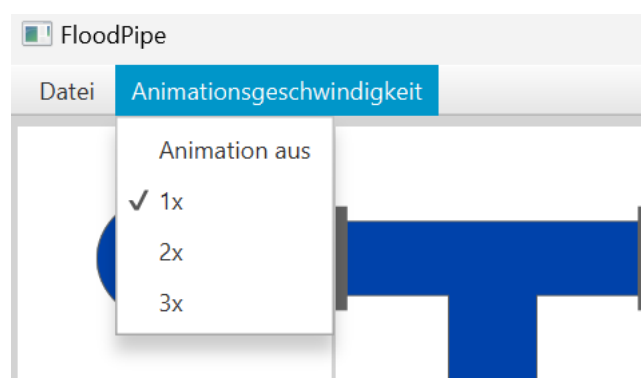


Abbildung 13: Animationsmenü

1.4 Fehlermeldungen

Bei der Ausführung des Programms können unterschiedliche Fehlersituationen auftreten. Die meisten dieser Fehler treten beim Laden einer Datei auf. Im Folgenden werden die möglichen Fehler nach deren Auftreten gruppiert beschrieben, deren Ursachen aufgeführt und Möglichkeiten zur Behebung geboten.

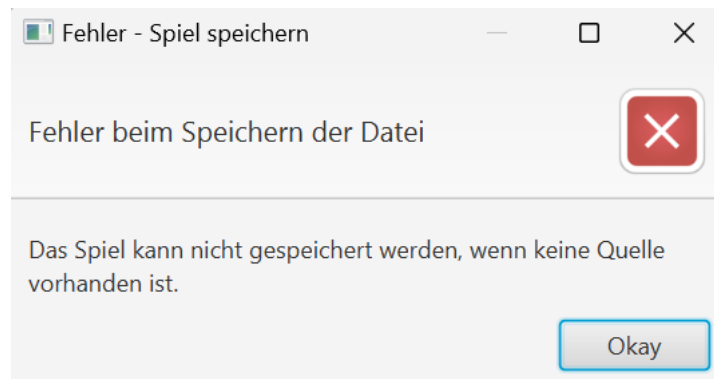


Abbildung 14: Fehlermeldung beim Speichern einer Datei

1.4.1 Im Editor

Fehlermeldung	Ursache	Behebungsmaßnahme
Es kann nicht in den Spielmodus gewechselt werden, wenn keine Quelle auf dem Spielfeld vorhanden ist.	Es kann nicht vom Editormodus in den Spielmodus gewechselt werden, wenn keine Quelle vorhanden ist.	Es muss eine Quelle auf dem Spielfeld platziert werden.
Das Spiel kann nicht gespeichert werden, wenn keine Quelle vorhanden ist.	Die Spielsituation kann nicht gespeichert werden, wenn auf dem Spielfeld keine Quelle vorhanden ist.	Es muss eine Quelle auf dem Spielfeld platziert werden.

1.4.2 Beim Speichern/Laden

Fehlermeldung	Ursache	Behebungsmaßnahme
Die ausgewählte Datei existiert nicht.	Die ausgewählte Spielstandsdatei ist nicht vorhanden.	Es ist eine andere Spielstandsdatei auszuwählen.
Die Datei ist leer.	Es wurde eine leere Spielstandsdatei ausgewählt.	Es ist eine andere Spielstandsdatei auszuwählen.
Es wurde keine Position für die Quelle angegeben.	Die Spielstandsdatei besitzt keine Angaben über die Position der Quelle.	Die Position der Quelle muss hinterlegt, oder eine andere Datei ausgewählt werden.
Es wurde kein oder ein falscher Wert für den Überlauf angegeben.	Der Überlaufmodus wurde nicht korrekt angegeben.	Es ist "true" oder "false" für den Überlaufmodus zu hinterlegen.
Negative Koordinate für die Position der Quelle angegeben.	Eine oder beide Koordinaten der Quelle sind negativ.	Die Werte sind anzupassen, oder eine andere Spielstandsdatei auszuwählen.
Die Position der Quelle liegt nicht innerhalb des Spielfelds.	Es wurde eine Koordinate außerhalb des Spielfelds hinterlegt.	Die Koordinaten sind anzupassen, oder eine andere Datei auszuwählen.
In der Datei ist kein Spielfeld vorhanden.	Die Spielstandsdatei besitzt keine Angabe zum Spielfeld.	Die Spielfeldbelegung ist hinzuzufügen, oder eine andere Datei auszuwählen.
Das Spielfeld enthaelt nicht zwischen 2 und 15 Spalten.	Es wurden zu wenige oder zu viele Spalten in der Datei hinterlegt.	Die Feldbelegung ist anzupassen, oder eine andere Spielstandsdatei auszuwählen.
Das Spielfeld enthaelt nicht zwischen 2 und 15 Reihen.	Es wurden zu wenige oder zu viele Reihen in der Datei hinterlegt.	Die Feldbelegung ist anzupassen, oder eine andere Spielstandsdatei auszuwählen.

Die Position der Quelle darf nicht auf einem Feld mit einer Mauer liegen.	Die Position für die Quelle liegt auf einem Mauerstück.	Die Position ist anzupassen, oder eine andere Spielstandsdatei auszuwählen.
Die Spalten des Spielfelds haben nicht die gleiche Anzahl an Feldern	Die Spalten der Spielstandsdatei haben eine unterschiedliche Anzahl an Elementen	Die Feldbelegung ist anzupassen, oder eine andere Spielstandsdatei auszuwählen.
Es wurde ein falscher Wert für den Rohrtypen angegeben (nicht zwischen 2 und 15).	Es wurde ein falscher Wert für ein Rohrstück angegeben.	Die Feldbelegung ist anzupassen, oder eine andere Spielstandsdatei auszuwählen.
Es wurden nicht zwei Koordinaten für die Position der Quelle angegeben.	Bei der Position der Quelle in der Spielstandsdatei fehlt eine oder beide Koordinaten.	Die Koordinaten sind anzupassen, oder eine andere Spielstandsdatei auszuwählen.

2 Programmiererhandbuch

2.1 Entwicklungskonfiguration

Im Folgenden werden die Konfigurationen aufgelistet, mithilfe derer das Programm entwickelt wurde. Die Entwicklung wurde auf einem Rechner begonnen und auf einem weiteren Rechner fortgeführt. Dies liegt daran, dass auf dem ersten Rechner ein macOS Betriebssystem installiert ist, bei dem die Darstellung der Systemfonts beim Start des Programms nicht funktionierte. Daher sind zwei unterschiedliche Konfigurationen aufgelistet:

Softwarekomponenten Rechner 1		
Art	Name	Version
Betriebssystem	macOS	Ventura 13.0.1
Compiler	Java Development kit	17.0.5
Entwicklungsumgebung	IntelliJ IDEA Ultimate Edition	2022.3.2
FXML-Designer	SceneBuilder	19.0.0

Softwarekomponenten Rechner 2		
Art	Name	Version
Betriebssystem	Windows	11 Professional
Compiler	Java Development kit	17.0.5
Entwicklungsumgebung	IntelliJ IDEA Community Edition	2022.3.2
FXML-Designer	SceneBuilder	19.0.0
Bildbearbeitung	Microsoft Paint	11.2210.4

2.2 Problemanalyse und Realisation

In diesem Kapitel erfolgt für wesentliche Problemstellungen, die zur Implementierung des Spiels FloodPipe zu lösen sind, jeweils eine Analyse des Problems, also der Anforderungen an das Spiel, eine Realisationsanalyse, bei der einer oder mehrere Lösungswege diskutiert und gegeneinander abgewägt werden, um schließlich zu der umgesetzten Implementierung zu gelangen. Diese wird mittels Realisationsbeschreibung dargestellt, um Gedankengänge, die bei der Erstellung eingeflossen sind, darzulegen.

2.2.1 Darstellung des Spielfelds

Problemanalyse Das Spielfeld bei FloodPipe besteht aus einer Tabelle variabler Größe, bei der in jeder Zelle eines der Rohrstücke oder eine Mauer platziert werden kann. Außerdem wird auf einem der Felder eine Markierung für die Position der Quelle dargestellt. Der Zug eines Spielers wird durch einen Klick in eine der Zellen, in der sich kein Mauerstück befindet, ausgeführt. Je nachdem, ob ein Links- oder Rechtsklick erfolgt ist, wird das Rohrstück nach links oder rechts rotiert. Wird die Größe des Fensters verändert, so soll ich auch das Spielfeld automatisch in seiner Größe anpassen.

Realisationsanalyse Für die Darstellung des Spielfelds eignet sich ein *GridPane*. Dieses besitzt Zeilen und Spalten, in denen weitere Komponenten angeordnet werden können, die die einzelnen Felder des Spielfelds repräsentieren. Die einzelnen Felder des Spielfelds werden als Bilder dargestellt (siehe beispielsweise Abbildung 5). Es werden also jeweils zwei Bilder für jedes Rohrstück benötigt, die dieses im leeren und gefüllten Zustand darstellen. Außerdem wird ein Bild für ein Mauerstück und die Markierung einer Spielfeldzelle als Quelle benötigt.

Um diese Bilder in den Zellen des Feldes zu platzieren, müssen diese in ein *ImageView* eingebunden werden. Mithilfe dieses können die Bilder dargestellt und einfach ausgetauscht werden, wodurch ein Wechsel eines Rohres und des Füllstands ermöglicht wird. Der Nachteil eines *ImageViews* ist in diesem Fall, dass die dargestellten Bilder bei Änderung der Fenstergröße verzerrt dargestellt sein könnten. Außerdem muss darauf geachtet werden, dass sich das *ImageView* nach der Rotation durch eine Benutzeraktion wieder zentriert innerhalb der Spielfeldzelle befindet und diese ausfüllt.

An der Position der Quelle ist zusätzlich zu dem Bild des Rohrtypen das Bild der Quelle zu platzieren und darzustellen. Hierfür wird ein weiteres *ImageView* benötigt, welches in der Darstellung über dem anderen *ImageView* platziert sein sollte. Es wird also eine Markierung benötigt, die eindeutig signalisiert, dass es sich bei diesem Feld um eine Quelle handelt, das darunter befindliche Rohrstück allerdings nicht verdeckt (siehe Abbildung 8). Außerdem muss eine Entscheidung darüber getroffen werden, ob die Markierung der Quelle beim Drehen des Rohrstücks ebenfalls rotiert wird.

Realisationsbeschreibung Das Spielfeld wird als Klasse *Field* implementiert, welche als Attribute ein *GridPane*, ein zweidimensionales Array von *FieldCell*-Instanzen und ein *ImageView* enthält. Das *GridPane* kann mit variabler Anzahl an Spalten und Zeilen dargestellt werden. In jeder Zelle des *GridPanes* wird eine *FieldCell* Instanz abgelegt, welche als Container für das *ImageView* dient, das das Rohrstück repräsentiert. Das zusätzliche *ImageView* stellt die Markierung für die Quelle dar (siehe Abbildung). Dies kann variabel einer Zelle der *GridPanes* zugeordnet werden. Beim Rotieren des Feldes wird lediglich das darin befindliche Rohrstück rotiert, die Quelle verbleibt in ihrer aktuellen Position. Die *ImageView*-Elemente werden in einem zweidimensionalen Array gespeichert. Wird eine Zelle des Spielfelds mit linker oder rechter Maustaste angeklickt, werden über eine dafür implementierte *EventHandler*-Methode über den *GameController* Funktionalitäten der *GameLogic* aufgerufen und anschließend die Drehung des Feldes erwirkt und dargestellt (siehe Listing 1). Außerdem werden Methoden angeboten, um einzelne Bilder, oder die Maße des Spielfelds anzupassen.

Listing 1: Field - setOnMouseClicked

```

1  private void setOnMouseClicked(GameController controller) {
2      for (FieldCell[] fieldCell : gameField) {
3          for (FieldCell cell : fieldCell) {
4              cell.setOnMouseClicked((MouseEvent event) -> {
5                  int clickedX = GridPane.getColumnIndex(cell);
6                  int clickedY = GridPane.getRowIndex(cell);
7                  MouseButton btn = event.getButton();
8                  if (btn == MouseButton.PRIMARY || btn ==
9                      MouseButton.SECONDARY) {
10                     // Turn clockwise, if the secondary
11                     // MouseButton was clicked
12                     controller.turn(btn == MouseButton.SECONDARY,
13                                     new Position(clickedX, clickedY));
14                 }
15                 event.consume();
16             });
17         }
18     }
19 }

```

2.2.2 Darstellung des Editors

Problemanalyse Im Editormodus sollen neben dem Spielfeld die einsetzbaren Rohrstücke, ein Mauerstück und eine Quelle angezeigt werden. Mittels *Drag&Drop* sollen diese Elemente auf das Feld gezogen werden, um die Belegung der Zelle durch den *Drop* zu verändern. Eine Quelle darf dabei nur auf einem Feld mit einem Rohrstück platziert werden. Wenn bereits eine Quelle vorhanden ist, soll diese entfernt werden. Ebenfalls wird die Quelle entfernt, wenn ein Mauerstück auf das Feld, auf dem sich diese befindet, gezogen wird.

Zusätzlich dazu kann die Anzahl der Spalten und Zeilen und der Überlaufmodus verändert, ein neues Spielfeld erzeugt und das Spielfeld gemischt werden (Siehe Abbildung 11). Bei Veränderung der Spalten- und Zeilenanzahl soll die noch bestehende Feldbelegung beibehalten werden und neue Zeilen oder Spalten mit Mauern gefüllt werden.

Realisationsanalyse Für die Darstellung der Bilder eignen sich *ImageViews* in einem *GridPane* (siehe Abschnitt 2.2.1). Die Anpassungsmöglichkeiten der Spalten und Zeilen kann durch folgende Komponenten ermöglicht werden:

1. *Spinner*

Ein *Spinner* besitzt ein Textfeld, in dem dessen aktueller Wert dargestellt wird. Zusätzlich dazu kann dieser Wert über einen Klick auf die Pfeiltaste unten oder oben inkrementiert oder dekrementiert werden. Da eine Änderung der Zeilen- bzw. Spaltenanzahl sofort erfolgen soll, wird bei jedem Klick eine neue Reihe oder Spalte erzeugt. Dies kann ein Nachteil sein, wenn die Erweiterung bzw. Verkleinerung der Datenstruktur einen hohen Aufwand darstellt.

2. *Slider*

Ein *Slider* bietet die Möglichkeit, seinen Wert durch Verschieben zu ändern. Vorteil hierbei ist die schönere Darstellung und die Möglichkeit, mehrere Zeilen und Spalten auf einmal hinzuzufügen oder zu entfernen. Nachteilig ist, dass ein *Slider* Werte vom Typ *Double* liefert, wodurch eine Typumwandlung erforderlich sein könnte.

Um ein neues Spielfeld zu erstellen oder das bestehende Spielfeld zu mischen, eignen sich besonders *Buttons* und für die Auswahl des Überlaufmodus ist eine *CheckBox* die optimale Lösung, da diese analog zum Überlaufmodus entweder ausgewählt oder nicht ausgewählt sein kann.

Das Ändern der Feldbelegung und Quellenposition kann als *DragEvent* implementiert werden. Es sind Werte für die einzelnen Editorfelder zu definieren, die bei einem *Drop* auf eine Zelle Funktionalitäten in der *GameLogic* anstoßen könnten, um die Feldbelegung anzupassen. Außerdem können diese farblich markiert werden, wenn das *DragEvent* gestartet ist. Es gibt außerdem die Möglichkeit, während des Events eine Darstellung des Bildes anzuzeigen, das auf die entsprechende Zelle gezogen wird. Allerdings verdeckt diese die darunter befindlichen Zellen, wodurch das Platzieren des *Drops* auf die richtige Zelle erschwert wird.

Die Zellen des Spielfelds können ebenfalls farblich hervorgehoben werden, wenn sich der Mauszeiger während des *DragEvents* über diesen befindet. Zusätzlich dazu sollte ein *Drop* der Quelle auf einem Mauerstück verhindert werden.

Realisationsbeschreibung Der Editor wird im *GameController* erzeugt.

Die Darstellung der Rohrstücke, der Mauer und der Quelle im Editor wird durch ein *GridPane* repräsentiert. In jeder Zelle des *GridPanes* befindet sich ein *ImageView*, in dem eines dieser Bilder gespeichert ist.

Per *Drag* wird die Zelle des Elements blau markiert und der Wert dessen in diesem *DragEvent* gespeichert (Siehe Listing 2). Bei Abschluss des Events wird die Markierung wieder entfernt. Die Zellen des *Fields* erhalten ebenfalls ein *DragEvent*, bei dem das Feld, über dem der Mauszeiger aktuell platziert ist, blau markiert wird, wenn ein *Drop* möglich ist. Wird der *Drop* ausgeführt, so wird über die *GameLogic* das Ändern der Feldbelegung angestoßen und diese anschließend dargestellt. Das Mischen des Spielfelds und die Neuerstellung eines Spielfelds ist als *Button* implementiert. Über diese wird in der *GameLogic* entweder ein Rotieren aller Felder in zufälliger Ausrichtung oder ein Belegen aller Felder mit Mauerstücken ausgelöst.

Die Anpassungsmöglichkeiten der Spalten- und Zeilenanzahl wird durch *Slider* realisiert. Diese werden mit einer *EventHandler*-Methode ausgestattet, die beim Loslassen der Maustaste über die Logik eine Anpassung der Zeilen- oder Spaltenanzahl ermöglicht. Die *Checkbox* erhält ebenfalls eine *EventHandler*-Methode, die über die *GameLogic* den Wert für den Überlaufmodus anpasst. Anschließend werden die gefüllten Zellen neu berechnet und entsprechend dargestellt.

Listing 2: GameController - setDrag

```

17 private void setDrag(ImageView[][] editorImageViews) {
18     // Values of the Strings to transfer
19     Queue<String> nameOfImage = new LinkedList<>(Arrays.asList(
20         PipeType.LINE.name(), PipeType.CURVE.name(),
21         PipeType.T_PIPE.name(), PipeType.DEAD_END.name(),
22         PipeType.WALL.name(), "SOURCE"));
23     for (ImageView[] row : editorImageViews) {
24         for (ImageView cell : row) {
25             String value = nameOfImage.poll();
26             // Put Values to Clipboard and set effect
27             cell.setOnDragDetected((MouseEvent event) -> {
28                 Dragboard db = cell.startDragAndDrop(TransferMode
29                     .ANY);
30                 ClipboardContent content = new ClipboardContent()
31                     ;
32                 content.putString(value);
33                 db.setContent(content);
34                 event.consume();
35                 cell.setEffect(new InnerShadow(10.0, Color.BLUE))
36                     ;
37             });
38             // Remove Effect
39             cell.setOnDragDone((DragEvent event) -> cell.
40                 setEffect(null));
41         }
42     }
43 }

```

2.2.3 Auswahl der Spieloptionen

Problemanalyse Es soll für den Spieler möglich sein, über ein Menü während eines laufenden Spiels die Spieloptionen anzupassen (siehe Abbildung 12) und anschließend ein neues Spiel zu starten. Die Spieloptionen werden außerdem bei Programmstart erfragt. Standardmäßig ausgewählt sollen zehn Spalten, zehn Mauern, zehn Prozent maximaler Maueranteil und ein abgeschalteter Überlaufmodus sein (siehe Abbildung 3). Beim Wechsel vom Spielmodus in die Optionen sollen allerdings die derzeitigen Optionen des laufenden Spiels übernommen und angezeigt werden.

Realisationsanalyse Die Spieloptionen können entweder in einem Menü anpassbar sein, oder es kann eine neue *Szene* erstellt werden, die geladen wird, wenn auf einen Menüpunkt "Neues Spiel" geklickt wird (siehe Abbildung 12). Für die Anpassung der Optionen direkt im Menü spricht, dass diese innerhalb des Spielbildschirms erfolgen kann und kein Sprung zu einem Optionsfenster und zurück erfolgen muss. Umzusetzen wäre dies beispielsweise über ein *MenuItem*, in dem die Auswahlmöglichkeiten der Optionen dargestellt sind. Ein Nachteil ist, dass hierdurch eventuell unabsichtliche Änderungen an der Spielsituation erfolgen können, indem der Spieler nach der Anpassung aus Versehen aus dem Menü klickt.

Eine Umsetzung in einer weiteren Szene hat den Nachteil, dass eine *Controller*-Kommunikation benötigt wird, um die Daten zu übertragen. Der Vorteil liegt allerdings in der besseren Handhabung und der klaren Trennung vom Spielbildschirm. Außerdem kann diese *Szene* so ebenfalls beim Start des Spiels angezeigt werden, um die initialen Optionen des Spiels festzulegen, bevor dieses angezeigt wird.

Die Elemente des Optionsmenüs sollten analog zur Darstellung im Editor als *Slider* für die Spalten-, Zeilen- und Maueranzahl und als *CheckBox* für den Überlaufmodus dargestellt werden (siehe Abbildung 11). Außerdem wird ein *Button* benötigt, der den Start des Spiels initiiert.

Realisationsbeschreibung Für die Spieloptionen wurde eine *fxml*-Datei erstellt, die die Elemente zur Anpassung dieser darstellt (Siehe Abbildung 3). Hierbei werden die Änderungsmöglichkeiten der Spalten-, Zeilen- und Maueranzahl als *Slider* implementiert. Die Minimal-, Maximal-, sowie Standardwerte werden in der Logik definiert und dienen dazu, die *Slider* zu initialisieren. Außerdem bietet der *SettingScreenController* der *fxml*-Datei eine Methode, um mittels eines Parameters die derzeitigen Werte der Elemente zu setzen. So kann aus dem Spiel heraus eine Startbelegung der Elemente übergeben werden, die der derzeitigen Spielsituation gleicht. Durch den Klick auf den *Button* zum Starten des Spiels, wird der *GameController* initialisiert und die gewählten Einstellungen an die *GameLogic* übergeben, um daraus eine passende Spielsituation zu erstellen.

2.2.4 Darstellung des Füllstatus

Problemanalyse Die Darstellung des Füllstatus soll im Spiel mittels einer *Animation* der zu füllenden Rohrstücke erfolgen. Während dieser *Animation* können weitere Spielzüge vorgenommen werden. Die *Animation* soll anschließend entsprechend angepasst werden. Es soll für den Spieler geeignete Auswahlmöglichkeiten für die Anpassung der Geschwindigkeit dieser *Animation* geben (Siehe Abbildung 13). Im Editormodus soll keine *Animation* genutzt, sondern der Füllstatus direkt angezeigt werden. Bei einem Wechsel zurück in den Spielmodus soll die zuvor gewählte Animationsgeschwindigkeit angezeigt sein. Sollten alle Rohrstücke gefüllt und keine offenen Enden vorhanden sein, so soll das Spielfeld deaktiviert werden und eine Meldung unterhalb des Spielfelds dargestellt werden, dass das Spiel gelöst ist. Befindet man sich im Spielmodus, so wird zusätzlich die Anzahl der benötigten Züge ermittelt.

Realisationsanalyse Es soll nur das Füllen von Rohrstücke animiert werden. Bereits gefüllte oder zu leerende Rohrstücke werden direkt als solche dargestellt. Für die Darstellung der Füllanimation kann eine *Timeline* verwendet werden. Diese bietet die Möglichkeit, *KeyFrames* zu speichern und diese anschließend in definiertem zeitlichem Abstand als *Animation* abzuspielen. Es werden also die Positionen der nicht gefüllten und der gefüllten Rohrstücke benötigt. Anschließend sind die Rohrstücke zu ermitteln, bei denen sich der Füllstatus von leer auf voll ändert. Diese werden bei der *Animation* berücksichtigt. Es gibt drei unterschiedliche Möglichkeiten, diese *Animation* durchzuführen:

1. Die *Animation* wird nacheinander für jedes *Position* durchgeführt. Nachteil bei dieser Variante ist, dass sie sehr zeitaufwendig sein kann, da immer nur ein Feld zur gleichen Zeit gefüllt wird. Außerdem ist diese Art der Animation unnatürlich, da das Wasser bei gleicher Entfernung gleichzeitig in die jeweiligen Rohre dringen würde. Vorteil dieser Variante ist, dass sie sich einfach umsetzen lässt und die *Animation* den Nutzer nicht überfordert, wenn zu viele Felder auf einmal animiert werden.
2. Die *Animation* wird bei gleicher Entfernung zur Quelle gleichzeitig durchgeführt. Der Vorteil dieser Variante gegenüber der Ersten ist, dass die *Animation* weniger Zeit benötigt und zudem natürlicher scheint. Ein Nachteil bei dieser Variante ist,

dass nur die Entfernung zur Quelle berücksichtigt wird, nicht aber, ob beispielsweise ein Nachbarfeld bereits gefüllt ist. Das Füllen des Feldes erfolgt erst, wenn kein zu füllendes Feld mehr näher an der Quelle ist.

3. Die *Animation* erfolgt auf Basis der Nähe zu einem gefüllten Feld. Diese Variante stellt die natürlichste Art der *Animation* dar. Allerdings ist die Implementation dieser Option deutlich aufwendiger.

Zur Anpassung der Animationsgeschwindigkeit sollte im *GameController* ein Menü genutzt werden, welches unterschiedliche *RadioMenuItems* in einer *ToggleGroup* vereint, sodass nur eines dieser ausgewählt sein kann. Außerdem wird eine Funktionalität benötigt, um die Animationsgeschwindigkeit während einer laufenden *Animation* anzupassen.

Realisationsbeschreibung Die *JavaFXGUI* ist für die Animation verantwortlich. Felder, die nach einer Benutzeraktion nicht mehr mit der Quelle verbunden sind, werden ohne Animation sofort geleert (siehe Listing 3). Hierfür wird das Bild des entsprechenden *ImageViews* angepasst. Bereits gefüllte Rohrstücke werden nicht verändert. Die *Animation* der zu füllenden Rohrstücke ist als *Timeline* implementiert, die als Attribut in der *JavaFXGUI* gespeichert wird. Um die Änderungsmöglichkeit der Geschwindigkeit dieser *Timeline* zu ermöglichen, kann über das Menü Animationsgeschwindigkeiten ausgewählt werden, mit welcher Geschwindigkeit die *Animation* ausgeführt wird. Es ist aus drei unterschiedlichen Geschwindigkeiten auszuwählen. Alternativ kann die Animation ausgeschaltet werden. Hierfür wird für die Elemente eine *EventHandler*-Methode definiert, welche den im jeweiligen Element als *userData* gespeicherten Wert mittels der *setRate*-Methode der *Timeline* setzt und dadurch die Geschwindigkeit entsprechend vorgegebener Werte anpasst. Dadurch kann ein Wechsel der Animationsgeschwindigkeit auch dann erfolgen, wenn gerade eine *Animation* ausgeführt wird. Das Ausschalten der *Animation* ist technisch gesehen nur ein Setzen der Geschwindigkeit auf einen hohen Wert, wodurch die *Animation* allerdings für den Nutzer nicht sichtbar ist. Diese Option wird beim Wechsel in den Editor gesetzt. Wird zurück in den Spielmodus gewechselt, so wird geprüft, welche Animationsgeschwindigkeit gerade ausgewählt ist und diese in der *Timeline* gesetzt. Bei einer Benutzerinteraktion, die zu einem veränderten Füllstatus des Spielfelds führt, wird mittels der *GameLogic* eine *Map* geliefert, die jeweils Schlüssel für die unterschiedlichen Entfernungen der mit der Quelle verbundenen Positionen liefert und diese Positionen in einer Liste speichert.

Anschließend kann geprüft werden, ob an der jeweiligen *Position* bereits ein Bild eines gefüllten Rohrstücks vorliegt. Falls nicht, wird der *Timeline* ein *KeyFrame* hinzugefügt, der zeitgleich mit den weiteren Positionen mit gleicher Entfernung zur Quelle ausgeführt wird und den Füllstatus anpasst. (siehe Listing 4). Es wurde also die Variante implementiert, dass alle Felder, die sich im gleichen Abstand zur Quelle befinden, zeitgleich gefüllt werden. An dieser Stelle besteht noch eine Optimierungsmöglichkeit, die Animation auf Basis der Nähe zu einem gefüllten Feld zu realisieren.

Es wird außerdem ein Zähler übergeben, der symbolisiert, ob das Spiel gelöst ist. Falls dies der Fall ist, so wird das Spielfeld bei Beendigung der Animation deaktiviert und je nachdem, ob der Benutzer sich derzeit im Editor oder Spiel befindet, eine entsprechende Meldung angezeigt. Nicht verbundene Positionen werden ebenfalls von der *GameLogic* übergeben. Diese werden ohne zeitliche Verzögerung geleert.

Listing 3: JavaFXGUI - displayFieldWithoutAnimation

```

38 public void displayFieldWithoutAnimation(Pipe[][] gameField, Position
    sourcePosition,
39                                     Set<Position>
                                        reachablePositions,
                                        boolean solved) {
40     // Stop Timeline and clear keyValues
41     stopTimeline();
42     int cols = gameField.length;
43     int rows = gameField[0].length;
44     Image img;
45     PipeType type;
46     Position pos;
47     for (int x = 0; x < cols; x++) {
48         for (int y = 0; y < rows; y++) {
49             pos = new Position(x, y);
50             type = gameField[x][y].getType();
51             // Get filled or empty image
52             img = reachablePositions.contains(pos) ?
                    getFilledImageFromPipeType(type)
53                 : getEmptyImageFromPipeType(type);
54             field.setImageAt(pos, img);
55             field.setRotationAt(pos, getRotation(gameField[x][y].
                    getRotation()));
56         }
57     }
58     field.setSourcePosition(sourcePosition);
59     showGameDoneIfSolved(solved, 0);
60 }

```

Listing 4: JavaFXGUI - displayFieldWithAnimation

```

61 public void displayFieldWithAnimation(Pipe[][] gameField, Map<
    Integer, List<Position>> connectedPositions,
62                                     Set<Position>
                                        unconnectedPositions,
                                        Integer counter) {
63     stopTimeline();
64
65     // Set empty Pipe Images on Positions, which are not
        connected to the source
66     for (Position position : unconnectedPositions) {
67         field.setImageAt(position, getEmptyImageFromPipeType(
            gameField[position.x()][position.y()].getType()));
68     }
69
70
71     double dur = 0;
72     List<KeyValue> fieldsToFillAtTheSameTime;
73     KeyFrame keyFrame;
74     // Add Fill animations to the timeline if the image is
        currently not filled. The distance to the source
75     // represents the order of fields to be filled
76     for (int i = 0; i < connectedPositions.size() - 1; i++) {
77         // Get KeyValues for all the fields to be filled at the
            same time

```

```

78         fieldsToFillAtTheSameTime =
            getListOfFieldsToFillInTheSameDistanceToSource(
                gameField, connectedPositions.get(i));
79
80         // Check if fields in the distance "i" need to be changed
            and change the duration accordingly
81         if (!fieldsToFillAtTheSameTime.isEmpty()) {
82             keyFrame = new KeyFrame(Duration.seconds(++dur),
                fieldsToFillAtTheSameTime.toArray(new KeyValue[0])
            );
83             this.timeline.getKeyFrames().add(keyFrame);
84         }
85     }
86     // if game is done, show the Label
87     if (counter != null) {
88         this.timeline.setOnFinished(event -> showGameDoneIfSolved
            (true, counter));
89     }
90
91     // Filling animation
92     this.timeline.play();
93 }

```

2.2.5 Laden und Speichern von Dateien

Problemanalyse Der Nutzer soll die Möglichkeit haben, Spielstände zu speichern und zu laden. Das Speichern eines Spielstands soll nur dann möglich sein, wenn eine Quelle auf dem Spielfeld vorhanden ist. Beim Laden des Spiels können nur syntaktisch und logisch korrekte json-Dateien geladen und dargestellt werden. Die Einstellungen des zu ladenden Spiels sind, mit Ausnahme der maximalen Mauerbelegung, im Editor und Optionsmenü entsprechend darzustellen. Sollte eine Spielsituation geladen werden, die bereits gelöst ist, so ist das Spielfeld im Spielmodus zu deaktivieren und eine entsprechende Meldung zu erzeugen.

Realisationsanalyse Das Speichern und Laden von Dateien kann als *MenuItems* umgesetzt werden und sollte einem passenden *Menu* zugeordnet sein.

Es ist außerdem denkbar, dass das Laden einer Datei ebenfalls im Startbildschirm ermöglicht sein kann, damit der Spieler nicht erst ein Spielfeld erzeugen muss, bevor eine Spielfelddatei geladen werden kann.

Für die Serialisierung und Deserialisierung dieser Spielfelddateien bietet sich die Bibliothek *Gson* an. Es ist außerdem abzuwägen, wie detailliert und mit welcher Möglichkeit die Spielfelddateien beim Ladevorgang auf logische Fehler untersucht werden sollten. Hierfür bieten sich unterschiedliche Möglichkeiten an:

1. Die Deserialisierung deckt logische Fehler auf
Diese Möglichkeit würde dafür sorgen, dass bei der Deserialisierung einer Spielfelddatei bereits logische Fehler erkannt werden und eine *JsonParseException* geworfen wird. Hieraus können dann entsprechende Fehlermeldungen erzeugt werden. Vorteil dieser Möglichkeit ist, dass nicht erst ein Datenobjekt mit Spielfelddaten erstellt werden muss, um dieses anschließend zu validieren. Fehler werden bereits beim Parsevorgang entdeckt und zurückgemeldet. Es ist allerdings abzuwägen, ob dies einen deutlichen Vorteil bringt, da ein benutzerdefinierter *JsonParser* benötigt wird.

2. Bei der **Deserialisierung** wird nur auf syntaktische Fehler in der Dateistruktur geprüft.

Diese Möglichkeit ist deutlich einfacher zu implementieren, da *Gson* bereits die Möglichkeit bietet, die Dateien auf syntaktische Fehler zu untersuchen. Nachteilig daran ist, dass gewisse Fehlersituationen sich nicht, oder nur umständlich prüfen lassen. Beispielsweise werden primitive Datentypen, sollten sie in der json-Datei nicht angegeben sein, automatisch mit deren Standardwert belegt. Dies kann zu Inkonsistenzen beim Einlesen der *Position* der Quelle und des Überlaufmodus führen. Logische Fehler können so nämlich erst anschließend an die Deserialisierung ermittelt werden und werden dann nicht mehr erkannt.

Außerdem sind Fehler zu beachten, die sich daraus ergeben können, dass eine Datei nicht vorhanden ist, oder fehlende Schreibberechtigungen vorliegen. Syntaktische und logische Fehler in der Spielfelddatei können mittels eines *Alerts* für den Nutzer angezeigt werden.

Realisationsbeschreibung Das Laden und Speichern von Dateien wurde in zwei Menüpunkten implementiert (Siehe Abbildung 12). Mittels einer EventHandler-Methode wird je nachdem, ob das Speichern oder Laden einer Datei gewünscht ist, ein *Dialog* zur Auswahl einer Datei angezeigt. Dieser filtert alle Dateien im derzeitigen Ordner und zeigt lediglich json-Dateien an. Sollte ein Fehler beim Laden oder Speichern dieser Datei auftreten, so übernimmt der Dialog die Darstellung einer Fehlermeldung.

1. Speichern

Es wird vor der Darstellung des Dialogs geprüft, ob eine Quelle auf dem *GameField* vorhanden ist. Ist dies nicht der Fall, so wird mittels *JavaFXGUI* eine Fehlermeldung angezeigt und der Speichervorgang abgebrochen. Andernfalls wird in der *GameLogic* eine Methode angestoßen, die die derzeitige Spielfeldbelegung in ein durch *Gson* serialisierbares Format konvertiert und dieses liefert. Anschließend wird die Datei gespeichert. Sollte während des Schreibvorgangs ein Fehler auftreten, so wird ebenfalls eine Fehlermeldung angezeigt.

2. Laden

Beim Laden der Datei wird ein Dialog zum Öffnen einer Datei dargestellt, in dem eine passende Datei auszuwählen ist. Anschließend wird diese Datei deserialisiert und der Ladevorgang in der *GameLogic* angestoßen. Hierfür wird die erzeugte Datenstruktur validiert und an den *GameController* gemeldet, ob das Laden der Datei erfolgreich war. Sollte dies der Fall sein, so wird die Darstellung des Spielfelds an die Einstellungen des zu ladenden Spiels angepasst und das *Field* angezeigt. Sollte die Datei logische Fehler aufweisen, teilt die *GameLogic* dies der GUI mit, damit eine entsprechende Fehlermeldung erzeugt werden kann.

2.2.6 Aufbau der Spiellogik

Problemanalyse Die Logik soll für die Erstellung und interne Speicherung einer Spielsituation unter Einhaltung gewisser Rahmenbedingungen verantwortlich sein. Es soll auf Benutzerinteraktionen mit der GUI reagiert werden können und eine Schnittstelle geboten werden, um die Kommunikation mit der GUI und damit einhergehend die Anzeige der Spielsituationen zu realisieren.

Realisationsanalyse Die erste zu treffende Entscheidung ist, wo die Erzeugung einer Spiellogik erfolgt. Es existieren zwei unterschiedliche Ansätze, um dies zu realisieren:

1. Erzeugung durch den *GameController* Die Spiellogik kann durch den GameController erzeugt werden, indem bei einem Klick auf den Button zum Starten des Spiels im Optionsbildschirm eine neue Instanz der *GameLogic* erzeugt wird, die die ausgewählten Eigenschaften übergeben bekommt. Außerdem wird im GameController eine Instanz der Klasse *GUIContector* erzeugt, die der Logik übergeben werden soll. Vorteil dieser Variante ist, dass die Logik direkt im Controller gespeichert werden kann, um diese bei Benutzerinteraktion direkt ansprechen zu können.
2. Erzeugung durch die *JavaFXGUI* Die Spiellogik kann aus der JavaFxGUI heraus erzeugt werden. Bei dieser Variante ist es notwendig, dass der GameController die Einstellungen an die GUI übergibt und diese anschließend eine *GameLogic* erzeugt und sich selbst als Parameter an diese übergibt. Bei Benutzereingaben spricht der GameController die JavFXGUI an, um Änderungen an der *GameLogic* durchzuführen. Nachteilig an dieser Variante ist, dass der Controller nicht direkt mit der Logik kommunizieren kann. Außerdem wird die GUI dann nicht nur noch primär dafür genutzt, um auf Änderungen in der Logik zu reagieren.

Für die interne Speicherung der Spielsituation gibt es folgende Lösungswege:

1. Speicherung als Attribute der *GameLogic*
Bei dieser Option besteht der Nachteil, dass sämtliche Methoden zur Erstellung, Manipulation und Konvertierung eines Spielfelds innerhalb der Spiellogik vorliegen. Dadurch entsteht eine Klasse, die unterschiedliche Zwecke erfüllt und dadurch unübersichtlich wird.
2. Auslagerung in eine eigene Klasse für das Spielfeld
Methoden zur Erzeugung und Manipulation des Spielfelds werden in einer eigenen Klasse gebündelt und als Attribut der *GameLogic* hinterlegt. Diese kann auf Informationen des Spielfelds zugreifen und dessen Struktur verändern. Dadurch werden die Funktionalitäten zur Erzeugung und Manipulation des Spielfelds in dieser Klasse gebündelt und die Logik bietet lediglich Methoden zur Kommunikation mit GameController und GUI.

Außerdem wird ein Zähler benötigt, der die Anzahl der Spielzüge des aktuellen Spiels zählt. Dieser kann als primitiver Datentyp *int* vorliegen.

Realisationsbeschreibung Die Erzeugung der Spiellogik erfolgt direkt aus dem GameController heraus. Hierfür werden die gewählten Einstellungen des Spielers und zusätzlich eine Instanz der Klasse *GUIContector* an die Klasse *GameLogic* übergeben. Diese definiert Methoden zur Darstellung der Zustände der Spiellogik. Nach Erzeugen der *GameLogic* wird eine neue *GameField*-Instanz erzeugt, die die Repräsentation des Spielfelds darstellt. Dieses Spielfeld ist für die Erzeugung einer neuen gelösten Spielsituation verantwortlich, welche durch das Erzeugen der Instanz angestoßen wird. Anschließend wird das Spielfeld rotiert und über die Verbindung zur GUI dargestellt. (Siehe Listing 5). Die *GameLogic* bietet weiterhin Funktionalitäten, die vom Controller aus aufgerufen werden können, um Benutzereingaben auf Seiten der Logik auszuführen. Diese werden validiert, an das Spielfeld weitergeleitet und anschließend auf der Oberfläche angezeigt. Außerdem liegt ein Zähler vor, der die Spielzüge zählt und aus dem Controller heraus zurückgesetzt werden kann.

Listing 5: GameLogic - Konstruktor

```

94 public GameLogic(int cols, int rows, int maxPercentageWalls, boolean
    overflow, GUIConnector gui) {
95     this.gui = gui;
96     this.gameField = new GameField(cols, rows, maxPercentageWalls
        , overflow);
97     gameField.rotateRandomly();
98     displayField();
99 }

```

2.2.7 Interne Speicherung einer Spielsituation

Problemanalyse Es soll möglich sein, eine Spielsituation zu erzeugen. Es sollen Rahmenbedingungen definiert sein, die die Maße des Spielfelds und den zulässigen Bereich der Maueranzahl vorgeben. Außerdem soll die Position der Quelle gespeichert werden können.

Realisationsanalyse Die zulässigen Rahmenbedingungen einer Spielsituation sollten als Konstanten vorliegen, die öffentlich abrufbar sind. Dadurch ist es für die GUI möglich, die Elemente zur Anpassung der Optionen an diese Konstanten anzupassen.

Für die Realisation des Spielfelds ist eine zweidimensionale Struktur notwendig, um die einzelnen Zellen in der jeweiligen Spalte und Zeile abzulegen. Der Zugriff auf diese Elemente soll indiziert über die Spaltennummer und anschließend die Reihenummer erfolgen, um die vorgegebene Zugriffsfunktionalität zu erreichen. Hierbei sind zwei unterschiedliche Lösungsvarianten zu unterscheiden. Es kann eine **statische Speicherstruktur** genutzt werden, bei der die Größe dieser Struktur bei der Erstellung festgelegt wird und nicht mehr nachträglich verändert werden kann. Eine andere Möglichkeit liegt in der Nutzung einer **dynamischen Speicherstruktur**, deren Größe variabel vergrößert oder verkleinert werden kann. Im Folgenden werden die Vor- und Nachteile dieser Speicherstrukturen miteinander verglichen, um eine für diese Aufgabenstellung optimale Lösung zu evaluieren.

Statische Speicherstruktur Es kann ein zweidimensionales Array verwendet werden, in dem die einzelnen Spielfeldzellen gespeichert werden. Der größte Vorteil eines Arrays im Vergleich zu dynamischen Speicherstrukturen liegt darin, dass ein Array mehrdimensional sein kann und somit eine simple und äußerst effektive Möglichkeit bietet, das Spielfeld abzubilden. Im Vergleich zu dynamischen Speicherstrukturen hat ein Array außerdem weniger Overhead, da beispielsweise kein Zähler für die Größe des Arrays geführt werden muss. Außerdem ist ein direkter indizierter Zugriff auf einzelne Elemente über die Spalten- und Zeilennummern möglich. Des Weiteren werden Feldbelegungen beim Speicher- und Ladevorgang in zweidimensionalen Arrays gespeichert. Diese Konvertierung ist bei der Umsetzung mittels Arrays einfach zu realisieren.

Der Nachteil eines Arrays ist, dass eine Änderung des Speicherbereichs nicht möglich ist. Um also neue Spalten und Zeilen hinzuzufügen oder zu entfernen, ist die Initialisierung eines neuen Arrays mit den geänderten Maßen des Spielfelds erforderlich. Um diese Neuinitialisierung zu vermeiden, könnte initial ein Array mit den maximal möglichen Maßen des Spielfelds erzeugt werden und über Zugriffsmethoden und Attribute definiert werden, auf welche Indizes des Arrays zugegriffen werden kann. Diese Attribute könnten

entsprechend erweitert oder verringert werden, wenn die Spalten- oder Zeilenanzahl anzupassen ist. Diese Funktionalität bieten dynamische Speicherstrukturen allerdings von Natur aus.

Dynamische Speicherstruktur Es gibt unterschiedliche dynamische Speicherstrukturen, die in Java genutzt werden können. Diese können über das *Java Collection Framework* abgerufen und genutzt werden. Auf eine weitere Erläuterung der einzelnen Bereiche des Frameworks wird verzichtet, da davon ausgegangen wird, dass diese bereits bekannt sind. Im Folgenden werden die unterschiedlichen Bereiche auf deren Eignung für die Anforderung geprüft und bewertet.

1. Liste Es könnte eine Listenstruktur verwendet werden, die wiederum indiziert auf eine weitere Listenstruktur zugreift. Besonders geeignet für diesen Anwendungsfall ist eine *ArrayList*, da diese einen indizierten Zugriff auf die darin befindlichen Elemente ermöglicht. Eine *LinkedList* ist nicht zu empfehlen, da diese keinen indizierten Zugriff auf die darin befindlichen Elemente bietet, sondern die Listenstruktur bei jedem Zugriff bis zu dem gesuchten Element traversiert werden muss. Der Vorteil der *ArrayList* im Vergleich zu einem Array liegt darin, dass das Erweitern der jeweiligen Spalte oder Zeile durch die Methoden `add()` bzw. `remove()` ermöglicht ist. Nachteilig im Vergleich zu einem Array ist, dass die Methoden `add()` bzw. `remove()` einer *ArrayList* jeweils mehrfach aufgerufen werden müssen, um mehrere Spalten oder Zeilen hinzuzufügen oder zu löschen. Nutzt man ein Array, so können die weiterhin bestehenden Elemente einfach in das neue Array kopiert werden. Außerdem ist das Erzeugen einer *ArrayList* aus einer Spielfelddatei mit bisher bekannten Konzepten aufwändiger, da die Werte vor dem Einfügen in die *ArrayList* noch konvertiert werden müssen.
2. Warteschlange und Menge Warteschlangen und Mengen sind für diese Problemstellung zu vernachlässigen, da sie keinen indizierten Zugriff auf die darin befindlichen Elemente bieten.
3. Verzeichnis Es könnte ein Verzeichnis genutzt werden, um die Felder des Spiels zu speichern. Die Schlüssel-Wert-Paare könnten aus einer Positionsangabe als Schlüssel und dem Element als Wert bestehen. Neue Spalten und Reihen werden hinzugefügt, indem entsprechende neue Positionen als Schlüssel generiert werden. Um Spalten und Zeilen zu löschen, sind alle Einträge des Verzeichnisses zu traversieren und deren Koordinaten mit den zu erwarteten Maßen des Spielfelds zu vergleichen. Da dies einen deutlichen Mehraufwand im Vergleich zum indizierten Zugriff bedeuten würde, wird auch diese Möglichkeit verworfen.

Realisationsbeschreibung Es wird eine Klasse *GameField* implementiert, die öffentliche Konstanten für die vorgegebenen Rahmenbedingungen des Spielfelds definiert. Das Spielfeld wird als zweidimensionales Array als Attribut dieser Klasse gespeichert, in dem die einzelnen Spielfeldzellen abgelegt werden können. Der Zugriff auf diese erfolgt indiziert über die Angabe der Spalten- und Zeilennummer. Der Zugriff auf die interne Struktur des Spielfelds wird über Getter- und Settermethoden realisiert. Die Getter-Methoden liefern lediglich Kopien der Spielfeldelemente. Für die Speicherung der Quelle bietet sich die Implementierung der Klasse *Position* an, in der als x-Koordinate die Spaltennummer und als y-Koordinate die Zeilennummer der Quelle gespeichert werden kann.

2.2.8 Interne Speicherung der einzelnen Felder

Problemanalyse Die Elemente des Spielfelds sollen einem Rohrstück zugeordnet werden können und zusätzlich die Ausrichtung dieses Rohrstücks liefern können. Außerdem soll es möglich sein, die Felder sowohl nach links, als auch nach rechts und zufällig oft zu drehen.

Realisationsanalyse Um für jedes Element des Spielfelds dessen Rohrtyp und Ausrichtung zu bestimmen, gibt es unterschiedliche Möglichkeiten:

1. separate Speicherung der Rotation und des Rohrtyps
Die Eigenschaften der Spielfeldelemente könnten in zwei unterschiedlichen Arrays gespeichert werden, wobei eines den Rohrtypen und eines die Ausrichtung repräsentiert. Nachteil an dieser Implementierung ist, dass dann lediglich die Daten gespeichert werden. Funktionalitäten zur Änderung der Ausrichtung müssten im Spielfeld implementiert werden. Außerdem können Inkonsistenzen auftreten, da die Arrays unabhängig voneinander verändert werden könnten.
2. Implementierung einer neuen Klasse
Es kann eine neue Klasse implementiert werden, deren Attribute der Rohrtyp und dessen Ausrichtung ist. Dadurch können in dieser Klasse zusätzliche Funktionalitäten implementiert werden, die für eine Rotation des Elements sorgen, oder dieses beim Lade- und Speichervorgang der Spielsituation in/aus einem dafür geeigneten Format zu konvertieren.

Für die Rotation und den Rohrtypen bietet es sich an, einen Aufzählungstypen zu verwenden, um den Quellcode übersichtlich und für einen Menschen einfach lesbar zu halten.

Realisationsbeschreibung Für die Speicherung der einzelnen Elemente des Spielfeldes wurde die Klasse Pipe implementiert. Diese ermöglicht die Erzeugung einer Spielfeldzelle mittels direkter Übergabe eines Rohrtypen und einer Rotation, oder eines Zahlenwertes, der die Anzahl und Richtung der Öffnungen widerspiegelt. Daraus kann anschließend der Rohrtyp und die Rotation ermittelt werden. Für den Rohrtypen wurde ein Aufzählungstyp definiert, der die unterschiedlichen Rohrstücke und das Mauerstück abdeckt (Siehe Listing 6). Die Rotation wird ebenfalls als Aufzählungstyp gespeichert (Siehe Listing 7). Außerdem kann die Rotation nachträglich angepasst werden, um das Drehen einer Feldzelle zu ermöglichen. Zusätzlich kann ein EnumSet ermittelt werden, welches ausgehend von Rohrtyp und Rotation die Verbindungen des Rohrstücks liefert. Dies kann genutzt werden, um den Füllstatus des Spielfelds zu ermitteln.

Listing 6: Aufzählungstyp - PipeType

```
100 public enum PipeType {  
101     LINE,  
102     CURVE,  
103     T_PIPE,  
104     DEAD_END,  
105     WALL  
106 }
```

Listing 7: Aufzählungstyp - Rotation

```
107 public enum Rotation {  
108     NORMAL,  
109     RIGHT,  
110     INVERTED,  
111     LEFT;  
112 }
```

2.2.9 Generierung einer Spielsituation

Problemanalyse Es soll möglich sein, eine zufällige Spielsituation durch die Logik generieren zu lassen. Geraden, Kurven und T-Stücke sollen zunächst mit gleicher Wahrscheinlichkeit eingesetzt werden. Endstücke werden nur dann eingesetzt, wenn sonst kein anderes Rohrstück passt. Die Quelle wird auf einem zufälligen Rohrstück platziert. Sind weniger oder gleich viele Zellen nach Belegung der Felder unbelegt, so werden in diesen Mauerstücke platziert.

Realisationsanalyse Zunächst ist ein Spielfeld mit den gewählten Maßen zu erstellen. Hier sollte eine Validierung der Eingabeparameter erfolgen. Anschließend sind Überlegungen erforderlich, wie die Belegung der Zellen erfolgen soll. Hierfür gibt es mehrere Möglichkeiten:

1. Start am Anfang des Spielfelds
Das Spielfeld kann von der ersten Position des Spielfelds aus gefüllt werden. Nachteilig an dieser Variante ist, dass alle Spielsituationen im ersten Feld des Spielfelds beginnen und dadurch meistens ähnliche Spielfelder erzeugt werden. Außerdem verhindert dies den Fall, dass in diesem Feld jemals Mauerstücke oder Endstücke gesetzt werden können.
2. Start an zufälliger Position
Wenn die Spielfelderzeugung an einer zufälligen Position beginnt, können die generierten Spielfelder deutlich variabler ausfallen. Außerdem ist dadurch sichergestellt, dass alle Feldtypen an jeder Position des Feldes eingesetzt werden können.

Analog können unterschiedliche Zeitpunkte zur Festlegung der Position der Quelle gewählt werden:

1. Festlegung zu Beginn der Spielfelderzeugung
Diese Variante besitzt deutliche Nachteile. Wenn die Quellposition zu Beginn der Spielfelderzeugung, also auf dem ersten erzeugten Rohrstück, platziert wird, ist es nicht möglich, dass diese sich jemals auf einem Endstück befindet. Außerdem wird die Quelle, sollte man sich für einen festen Startpunkt bei der Spielfelderzeugung entscheiden, immer an der gleichen Stelle platziert werden.
2. Festlegung nach Erstellung des Spielfelds
Bei dieser Variante wird die *Position* der Quelle erst dann ermittelt, wenn die Generierung des Spielfelds bereits abgeschlossen ist. Dadurch ist es möglich, diese auf jedem möglichen Rohrstück zu platzieren und außerdem unabhängig von der Implementierung des Algorithmus zufällige Quellpositionen zu erhalten.

Es wird ein komplexer Algorithmus für die Erzeugung eines Spielfelds benötigt. Dieser sollte in der Lage sein, Nachbarpositionen zu bestimmen, abhängig davon, ob der Überlaufmodus aktiviert ist und entsprechend passende Rohrstücke zu erzeugen. Es

sollte ein Algorithmus gewählt werden, der ausgehend von einer Position ein Spielfeld erzeugt, indem ein Rohrstück platziert wird und alle Nachbarfelder, zu denen eine Verbindung erforderlich ist, anschließend mit einem Rohrstück versieht, bis keine offenen Verbindungen mehr vorliegen.

Anschließend eignet sich eine Traversierung des Feldes, um nicht belegte Zellen zu identifizieren und diese mit Mauerstücken zu belegen. Außerdem sind Validierungen zu implementieren, die prüfen, ob sich an die maximale Maueranzahl gehalten wurde, alle Rohrstücke mit der Quelle verbunden sind und keine offenen Enden existieren.

Sollte eine dieser Validierungen fehlschlagen, ist die Spielsituation zu verändern, bis die Bedingungen erfüllt sind. Für die Anpassung der Spielsituation im Falle fehlgeschlagener Validierungen gibt es grundsätzlich zwei Varianten:

1. Das Spielfeld wird verworfen und ein neues wird erstellt.
Der Vorteil dieser Variante liegt in der Einfachheit der Implementierung und dem damit einhergehenden geringen Fehlerpotenzial.
2. Es werden ausgehend vom zuletzt gesetzten Rohrstück rekursiv oder iterativ Rohrstücke ausgetauscht, bis eine gelöste Spielsituation erzeugt wurde.
Wie zuvor angedeutet ist diese Implementierungsmöglichkeit deutlich komplexer und fehleranfälliger. Außerdem bietet sie keine nennenswerten Vorteile gegenüber einer Neuerstellung.

Realisationsbeschreibung Der Konstruktor der Klasse *GameField* validiert die Einstellungen, anhand derer ein Spielfeld erzeugt werden soll. Sollten die Einstellungen nicht valide sein, so wird eine Exception ausgelöst. Anschließend wird bei validen Einstellungen ein Algorithmus zur Spielfelderzeugung genutzt (Siehe Abschnitt 2.3.1).

2.2.10 Modifizierung der Spielsituation

Problemanalyse Es soll möglich sein, einzelne Felder des Spielfelds oder die Position der Quelle auszutauschen. Außerdem kann die Spalten- und Reihenanzahl des Spielfelds innerhalb definierter Grenzen angepasst werden und ein neues Spielfeld mit Mauerfeldern initialisiert werden.

Realisationsanalyse Der Austausch einzelner Felder auf dem Spielfeld kann durch eine Setter-Methode erfolgen. Die einzige Besonderheit liegt darin, dass bei der Platzierung eines Mauerstücks auf der Position der Quelle diese entfernt wird. Dies kann realisiert werden, indem die GUI eine Position an die GameLogic übergibt, an der das Feld ausgetauscht werden soll. Mittels equals-Methode der Position kann dann verglichen werden, ob es sich um die Position der Quelle handelt.

Der Austausch der Position der Quelle kann ebenfalls mittels Setter-Methode erfolgen. Dabei ist darauf zu achten, dass die Quelle nicht auf ein Mauerstück gesetzt wird.

Die Anpassung der Spalten- und Reihenanzahl sollte durch eine Neuinitialisierung des Arrays erfolgen. Um die vorherige Feldbelegung beizubehalten, werden noch vorhandene Zellen in das neue Array kopiert. Zusätzliche Spalten und Reihen werden mit Mauerstücken belegt. Eine alternative Möglichkeit wäre, bei der Verringerung der Spalten- und Zeilenanzahl keine Veränderung der Größe des Arrays vorzunehmen. Dadurch kann sich der Initialisierungsaufwand verringern. Dies würde allerdings voraussetzen, dass ein Zähler vorliegt, der die Anzahl an Spalten und Reihen führt. Somit stellt dies keine optimale Lösung dar.

Realisationsbeschreibung In der Klasse `GameField` werden Methoden implementiert, um die Position der Quelle zu setzen und einzelne Spielfeldzellen auszutauschen. Die `GameLogic` stößt diese Funktionalitäten an, nachdem eine Validierung der übergebenen Position erfolgt ist. Das Spielfeld wird entsprechend angepasst und auf der Oberfläche angezeigt. Bei Änderung der Spalten- und Zeilenanzahl wird das in `GameField` gespeicherte Array neu initialisiert und noch vorhandene Werte kopiert. Außerdem wird gegebenenfalls die Quelle entfernt, wenn diese sich nicht mehr innerhalb des Spielfelds befindet und anschließend das Spielfeld auf der Oberfläche dargestellt.

2.2.11 Konvertierung zwischen Spielsituation und Datei

Problemanalyse Die Attribute eines Spielfelds sollen vor dem Lade- und Speichervorgang aus/in Datentypen konvertiert werden, die Bestandteil einer json-Datei sein können (Siehe Abschnitt 2.5). Beim Ladevorgang ist die Datei auf logische Fehler zu untersuchen. Eine valide json-Datei besteht aus einem Objekt, das die x- und y- Koordinate der Quellposition angibt, einem Boolean für den Überlaufmodus und einem Array mit den einzelnen Belegungen der Spalten und Zeilen.

Realisationsanalyse Es wird eine Klasse zur Abbildung der json-Datei benötigt. Es sind Attribute für die Angabe der Quellposition, des Überlaufmodus und der Feldbelegung erforderlich. Der Überlaufmodus kann entweder als primitiver boolean-Wert definiert werden, oder es kann der Wrapper-Datentyp `Boolean` genutzt werden. Beim Parsen eines json-Strings werden fehlende Werte bei primitiven Datentypen mit dem Standardwert initialisiert. Ein `Boolean` besitzt also den Vorteil, dass geprüft werden kann, ob in der json-Datei ein Wert für den Überlaufmodus angegeben wurde. Diese Überlegung sollte ebenfalls für die Koordinaten der Quelle berücksichtigt werden, um zu verhindern, dass nur eine oder keine Koordinate angegeben wird.

Realisationsbeschreibung Die Klasse `GameFieldData` wird implementiert, um Spielsituationen zu speichern und zu laden. Diese besitzt eine Funktionalität zur Validierung der erstellten Daten, welche beim Laden einer Spieldatei ausgelöst wird. Diese liefert den Aufzählungstypen `FieldError` (siehe Listing 8), der angibt, ob ein Fehler in der Datei vorliegt und wenn ja welcher. Die Spielsituation wird nur dann geladen, wenn die Ausprägung `ERR_NULL` vorliegt. Dann wird mittels der `GameLogic` über eine Verbindung zur Oberfläche eine Darstellung des Spielfelds angestoßen. Andernfalls wird über diese ein Fehlertyp mitgeteilt und anschließend angezeigt.

Listing 8: Aufzählungstyp - FieldError

```

113 public enum FieldError {
114
115     ERR_NULL,
116     ERR_EMPTY,
117     ERR_NO_SOURCE,
118     ERR_NO_OVERFLOW,
119     ERR_NEGATIVE_SOURCE,
120     ERR_SOURCE_OUT_OF_BOUNDS,
121     ERR_NO_BOARD,
122     ERR_INVALID_COLS,
123     ERR_INVALID_ROWS,
124     ERR_SOURCE_POSITION_WALL,
125     ERR_COLS_UNEVEN,
126     ERR_SOURCE_ONLY_ONE_VALUE,
127     ERR_WRONG_PIPE_VALUE
128 }

```

2.2.12 GUI-Schnittstelle

Problemanalyse Es soll eine Schnittstelle definiert werden, über die die Logik Änderungen der Spielsituation auf der Oberfläche anzeigen kann. Es soll möglich sein, ein Spielfeld mit den derzeitigen Füllständen darzustellen. Dies soll, wenn ein Spielfeld neu initialisiert wird, ohne Animation ablaufen. Bei Änderungen der Spielsituation soll eine Animation erfolgen. In der Logik wird keine Unterscheidung zwischen Editor- und Spielmodus vorgenommen. Zudem sollen die Felder des Spielfelds angepasst, gedreht und die Position der Quelle verändert werden können. Außerdem sollen Fehler, die in der Logik auftreten, auf der Oberfläche dargestellt werden können.

Realisationsanalyse Es ist ein Interface in der Logik zu definieren, das Methoden für die oben genannten Funktionalitäten bietet.

Für die Darstellung des Spielfelds kann entweder die Spielfeldinstanz, oder die benötigten Informationen übergeben werden. Der Vorteil der Nutzung einer Spielfeldinstanz liegt darin, dass die Parameterliste übersichtlicher ist. Nachteilig ist allerdings, dass in der Oberfläche die für die Darstellung benötigten Daten erst berechnet werden müssen, um diese anschließend zu verwenden.

Wenn eine Animation des Spiels erfolgt, der Spieler also einen Zug getätigt hat, wird zudem ein Zähler benötigt, der, sollte das Spiel gelöst sein, die entsprechende Meldung erzeugt. Alternativ könnte eine separate Methode erstellt werden, die das Darstellen der Meldung am Spielende übernimmt. Allerdings müsste diese auch genutzt werden, wenn das Spiel nicht gelöst ist, um die Animation zu starten. Sollte das Starten der Animation aus der Methode zur Darstellung des Spielfelds erfolgen, so kann es sein, dass diese bereits abgeschlossen ist, wenn die Meldung dargestellt werden soll.

Für das Drehen eines Feldes, das Ändern der Position der Quelle und das Ändern eines Feldes wird die jeweilige Position übergeben. Um die Belegung eines Feldes darzustellen, wird zusätzlich der Rohrtyp benötigt. Die Rotation ist nicht notwendig, da ein Feld immer im Ausgangszustand platziert werden soll.

Realisationsbeschreibung Die Schnittstelle zur Oberfläche wird als Interface GUI-Connector implementiert, welches Funktionalitäten für das Darstellen des Spielfelds und die Manipulation der einzelnen Spielfeldelemente bietet. Diesen Methoden werden lediglich die Parameter übergeben, die sie zur Darstellung der veränderten Spielsituation

benötigen. Auf die Übergabe einer Spielfeldinstanz wird verzichtet, damit die Berechnung dieser Daten ausschließlich in der Logik erfolgt. Um von der GameLogic übergebene Positionen mit Positionen zur Repräsentation einzelner Felder der Oberfläche zu vergleichen, wird in der Klasse *Position* die equals-Methode überschrieben, sodass zwei Positionen dann gleich sind, wenn sie die gleichen Koordinaten haben.

2.3 Algorithmen

In diesem Kapitel werden komplexe Algorithmen erläutert, die für die Implementierung genutzt werden. Außerdem werden bekannte Algorithmen, sollte auf diese zurückgegriffen werden, erklärt und begründet, warum diese genutzt werden. Bei dem Spiel FloodPipe werden komplexe Algorithmen für folgende Funktionalitäten benötigt:

- Erstellung eines zufällig generierten Spielfelds
- Ermittlung der Füllstände einzelner Spielfeldzellen

2.3.1 Spielfeldgenerierung

Anforderungen Erste Überlegungen zu den Rahmenbedingungen des Algorithmus wurden bereits angestellt (siehe Abschnitt 2.2.9). Auf Basis derer werden nun Rahmenbedingungen erarbeitet, die für die Auswahl eines geeigneten Algorithmus dienen:

- Zufälliger Startpunkt für die Erstellung eines Spielfelds
- Die Position der Quelle wird nachträglich festgelegt
- Nachbarfelder, die keine Verbindung zum aktuellen Feld benötigen, werden nicht belegt.
- Das generierte Spielfeld wird verworfen und ein neues erstellt, wenn dies nicht valide ist.

Es wird ein Algorithmus benötigt, der mittels übergebener Spieloptionen ein zufälliges Spielfeld generiert. Ausgehend von einer Startposition sollen rekursiv alle Nachbarn, in deren Richtung das auf der Startposition platzierte Rohrstück Verbindungen hat, untersucht und gefüllt werden. Für eine solche Implementierung eignet sich eine Variante des FloodFill-Algorithmus.

FloodFill-Algorithmus Es handelt sich um einen Algorithmus, der ausgehend von einer Startposition rekursiv alle Positionen einer Struktur besucht und deren Wert unter bestimmten Bedingungen ändert. Anschließend wird dieser Algorithmus rekursiv mit benachbarten Positionen aufgerufen, um die gesamte Struktur zu traversieren.

Vorteile dieses Algorithmus liegen darin, dass er einfach zu implementieren ist und schnell ausgeführt werden kann. Für diese Aufgabe muss eine angepasste Variante des FloodFill-Algorithmus genutzt werden, damit nicht kategorisch alle Nachbarfelder einer Ausgangsposition mit Rohrstücken belegt werden. Dies sorgt dafür, dass auch Felder frei bleiben können, um später Mauerstücke zu platzieren. Wenn kategorisch alle Nachbarfelder belegt werden, würde ein Spielfeld ohne jegliche Mauerstücke entstehen. [3]

Umsetzung Die Generierung des Spielfeldes wird begonnen, indem ein leeres Feld erzeugt und eine zufällige Position auf diesem ausgewählt wird. Anschließend wird für diese Position ein passendes Rohrstück ermittelt. Hierfür werden folgende Informationen über die Nachbarfelder benötigt:

- Nachbarfelder, die eine Verbindung in die Richtung der aktuellen Position besitzen
- leere Nachbarfelder

Die Richtungen, in denen sich die Nachbarfelder befinden, werden in *EnumSets* gespeichert. Je nachdem, ob der Überlaufmodus aktiviert ist, können auch Nachbarfelder über die Spielfeldgrenzen hinaus ermittelt werden. Für Nachbarfelder mit einer Verbindung in Richtung der aktuellen Position ist es notwendig, dass das zu platzierende Rohrstück mit dieser verbunden werden kann, während leere Nachbarfelder optional verbunden werden können. Zur Bestimmung der leeren Nachbarfelder wird die Methode `getEmptyNeighbors` genutzt, welcher ein Set mit allen Ausprägungen des Ausprägungstyps *Direction* übergeben wird, um alle Nachbarfelder zu prüfen.

Listing 9: GameField - getEmptyNeighbors

```
129 private EnumSet<Direction> getEmptyNeighbors(Position pos, EnumSet<
    Direction> directions) {
130     EnumSet<Direction> emptyNeighbors = EnumSet.noneOf(Direction.
        class);
131     int cols = getCols();
132     int rows = getRows();
133     for (Direction direction : directions) {
134         Position neighborPos = pos.getNeighborPosition(cols, rows
            , direction, overflow);
135         if (isEmptyField(neighborPos)) {
136             emptyNeighbors.add(direction);
137         }
138     }
139     return emptyNeighbors;
140 }
```

Ermittlung eines Rohrstücks Anschließend kann anhand dieser beider *EnumSets* ermittelt werden, welche Rohrtypen für die aktuelle Position in Frage kommen. Ein Endstück kann nur dann gesetzt werden, wenn insgesamt genau eine Öffnung zur Verfügung steht, da dieses nur dann platziert werden soll, wenn kein anderes Rohrstück passend ist.

Anschließend ist zu prüfen, ob eine Kurve oder eine Gerade möglich sind. Dies ist dann der Fall, wenn maximal zwei Öffnungen benötigt werden und mindestens zwei mögliche Öffnungen vorhanden sind.

Anschließend ist zu prüfen, ob die erforderlichen Öffnungen der Nachbarfelder bereits eine der beiden Möglichkeiten ausschließen. Sollte dies nicht der Fall sein, so kann mittels der möglichen Öffnungen geprüft werden, ob eine Gerade oder Kurve gesetzt werden kann. Hierbei ist bei der Implementierung ein Fehler aufgetreten. Grundsätzlich sollte es möglich sein, dass sowohl eine Gerade, als auch eine Kurve verwendet werden kann und diese sich nicht kategorisch ausschließen. Da diese sich allerdings mittels *if*-Verzweigung ausschließen, kann eine Gerade nur dann platziert werden, wenn eine Kurve nicht möglich ist (Siehe Listing 10). Dieser Fehler kann dadurch behoben werden, dass beide Prüfungen in einem *else*-Teil der *if*-Verzweigung separat voneinander erfolgen

und der Liste möglicher Rohrtypen hinzugefügt werden. Anschließend erfolgt noch die Prüfung, ob ein T-Stück platziert werden kann. Anschließend wird eine Liste geliefert, die alle passenden Rohrtypen beinhaltet, die auf diesem Feld gesetzt werden können. Sollte diese leer sein, ist es nicht möglich, ein Rohrstück zu platzieren und das Feld bleibt null. Andernfalls wird zufällig eines dieser Rohrstücke ausgewählt. Anschließend wird dieses so lange rotiert, bis es alle benötigten Öffnungen abdeckt und alle weiteren Öffnungen auf leere Felder gerichtet sind.

Listing 10: GameField - getRandomPipeForPosition

```

141 private Pipe getRandomPipeForPosition(Position position) {
142     // Neighbors that have Openings to this position
143     EnumSet<Direction> mandatoryOpenings = getNeighborsToConnect(
144         position);
145
146     // Empty neighbors
147     EnumSet<Direction> optionalOpenings = getEmptyNeighbors(
148         position, EnumSet.allOf(Direction.class));
149
150     List<PipeType> possibleTypes = getPossiblePipeTypes(
151         mandatoryOpenings, optionalOpenings);
152
153     // DeadEnds can only be set, if no other pipe fits and
154     if (possibleTypes.isEmpty() && (mandatoryOpenings.size() +
155         optionalOpenings.size()) == 1) {
156         possibleTypes.add(PipeType.DEAD_END);
157     }
158
159     // no type possible
160     if (possibleTypes.isEmpty()) {
161         return null;
162         //get a random Pipe from the suitable ones and rotate it
163         //correctly
164     } else {
165         Random rnd = new Random();
166         PipeType chosen = possibleTypes.get(rnd.nextInt(
167             possibleTypes.size()));
168
169         Pipe pipe = new Pipe(chosen);
170         rotateCorrectly(pipe, mandatoryOpenings, optionalOpenings
171             );
172         return pipe;
173     }
174 }

```

Rekursive Füllung des Spielfelds Sobald die Startposition mit einem passenden Rohrstück ausgestattet ist, werden alle Nachbarfelder ermittelt, die mit diesem Rohrstück verbunden werden müssen. Anschließend erfolgt ein rekursiver Abstieg, der für die Generierung zu verbindender Felder sorgt. Die Rekursion wird abgebrochen, sobald für die aktuelle Position kein passendes Rohrstück ermittelt werden konnte, oder keine leeren Nachbarfelder mehr vorhanden sind, die zu verbinden sind. Dadurch können beispielsweise auch Konstellationen entstehen, bei denen ein Spielfeld mit zehn Spalten und zehn Reihen nur vier Rohrstücke besitzt.

Validierung des Spielfelds Wenn ein generiertes Spielfeld erzeugt ist, werden in allen Feldern, die bisher nicht belegt sind, Mauern platziert. Anschließend erfolgt eine Validierung des Spielfelds. Dafür wird die Anzahl der Mauerstücke ermittelt und geprüft, ob diese Belegung erlaubt ist. Außerdem wird für alle Positionen auf dem Spielfeld geprüft, ob die Öffnungen des Rohrstücks korrekt und vollständig mit seinen Nachbarn verbunden sind. Sollte dies der Fall sein, ist die Erzeugung des Spielfelds abgeschlossen und valide. Andernfalls wird so lange versucht, ein Spielfeld zu erzeugen, bis dieses valide ist. Anschließend wird die Position der Quelle bestimmt und die Felder zufällig rotiert.

2.3.2 Ermittlung der Füllstände

Anforderungen Es wird ein komplexer Algorithmus benötigt, um ausgehend von der Quelle die Spielfeldzellen zu ermitteln, die mit dieser verbunden sind. Außerdem sollen diese für die Umsetzung einer zeitgleichen Animation des Füllvorgangs bei gleicher Distanz zur Quelle in einer Map gruppiert werden. Die Quelle selbst soll ebenfalls Teil dieser Map sein, mit einer Distanz von "0". Nicht gefüllte Felder sollen anschließend ermittelt werden können, indem geprüft wird, ob diese in der Map vorhanden sind.

Breitensuche Da die gefüllten Spielfeldzellen gruppiert nach der Distanz zur Quelle geliefert werden sollen, eignet sich besonders das Verfahren der Breitensuche. Bei diesem Suchverfahren werden zunächst alle Positionen besucht, die von einem Ausgangsknoten direkt erreichbar sind. Erst anschließend wird die Suche auf die Folgeknoten ausgeweitet. [1] Daher eignet sich diese Suchvariante besonders für die Gruppierung der Positionen nach deren Distanz zur Quelle. Ein alternatives Suchverfahren ist die Tiefensuche. Diese wäre vorteilhaft, wenn die Animation nicht gleichzeitig stattfinden sollte, sondern die einzelnen Pfade nacheinander gefüllt werden sollten. [2] Es wurde sich allerdings für die zeitgleiche Füllung bei gleicher Distanz zur Quelle entschieden.

Ablauf der Breitensuche Bei der Breitensuche werden folgende Strukturen benötigt:

- Eine Warteschlange, in die alle Positionen eingefügt werden, deren Nachbarfelder zu besuchen sind.
- Eine Menge, in der alle bereits besuchten Positionen gespeichert sind.

Anschließend wird die Startposition in die Warteschlange eingefügt und als besucht markiert. Für diese Position werden dann alle Nachbarn gesucht, diese ebenfalls in die Warteschlange eingefügt und als besucht markiert, sollten diese nicht bereits besucht sein. Erst wenn alle direkten Nachbarfelder besucht wurden, erfolgt ein tieferer Abstieg in die Suche und die Nachbarfelder der besuchten Positionen werden ermittelt. Dies wird so lange ausgeführt, bis alle Knoten besucht wurden, oder das Ziel gefunden wurde.

Umsetzung Um die mit der Quelle verbundenen Positionen nach deren Distanz zu dieser zu gruppieren, wird eine Map verwendet, die zu einem Integer eine Liste mit Positionen speichert. Die Quelle wird dieser Map mit einer Distanz von "0" hinzugefügt, um diese ebenfalls in der GUI füllen zu können. Anschließend wird die Distanz um eins erhöht und alle Nachbarfelder der Startposition überprüft. Dafür wird eine Methode verwendet, die ermittelt, ob die aktuelle Position mit dem Nachbarn in der jeweiligen Richtung verbunden ist. Zunächst ist zu prüfen, ob es sich bei der Nachbarposition um eine Position innerhalb des Spielfelds handelt. Falls dies der Fall ist, wird geprüft, ob das Feld auf der aktuellen Position eine Verbindung in die Richtung des Nachbarfeldes hat. Anschließend ist noch zu prüfen, ob das Nachbarfeld eine Verbindung in Richtung der

aktuellen Position hat. Falls ja, handelt es sich um ein mit der Startposition verbundenes Feld, welches als besucht markiert, der Warteschlange hinzugefügt und in die Map eingefügt wird. Wenn alle Nachbarn des aktuellen Feldes ermittelt wurden, wird die Distanz zur Quelle erhöht und deren Nachbarn werden überprüft. Der Algorithmus endet, wenn alle Felder besucht wurden, oder keine Verbindungen mehr zur Quelle vorhanden sind.

2.4 Programmorganisationsplan

In diesem Kapitel wird eine Übersicht über die Zusammenhänge zwischen den genutzten Klassen gegeben. Dafür wird die Datenstruktur des Projekts anhand eines Programmorganisationsplans erläutert, um einen Überblick über diese zu schaffen.

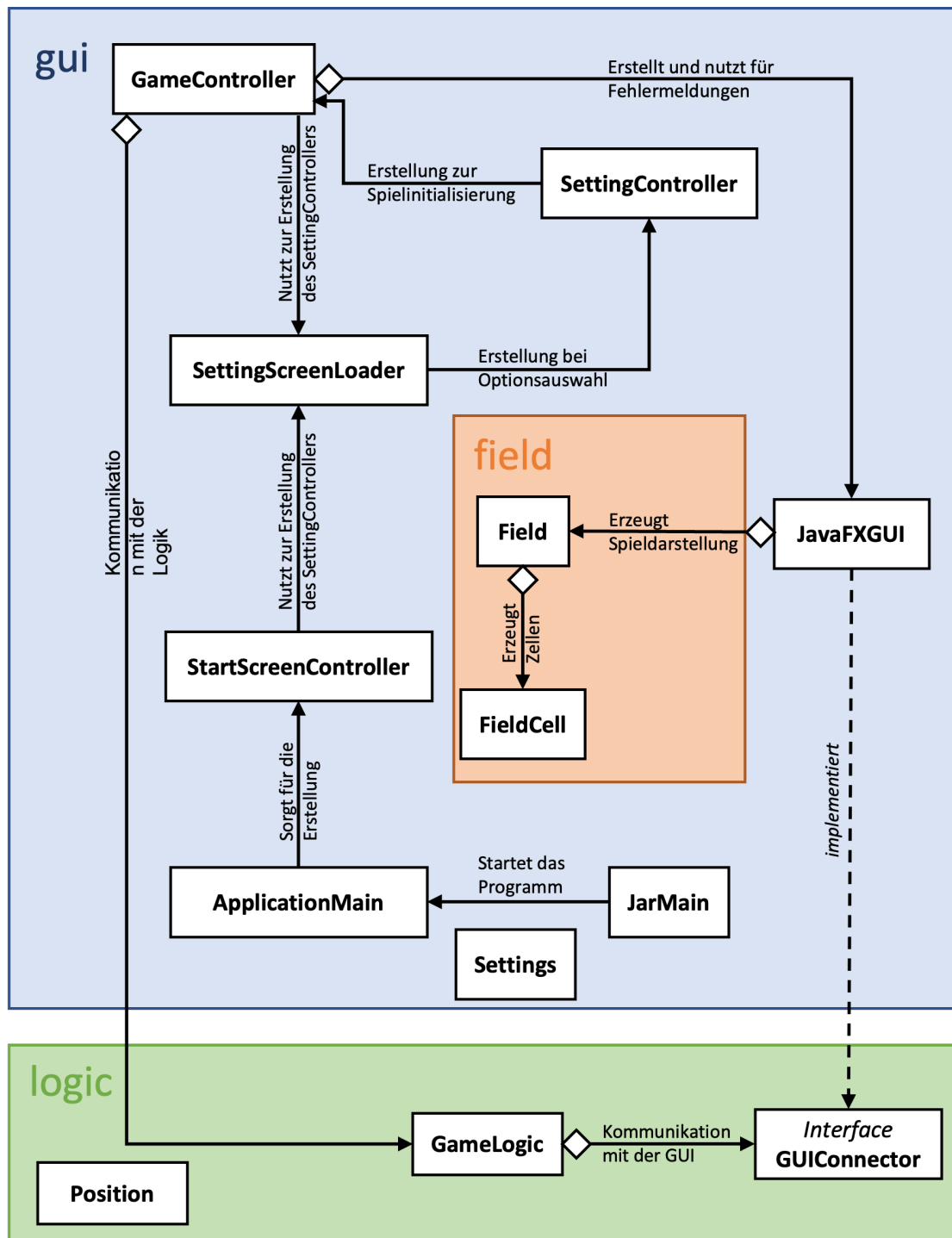


Abbildung 15: Programmorganisationsplan gui

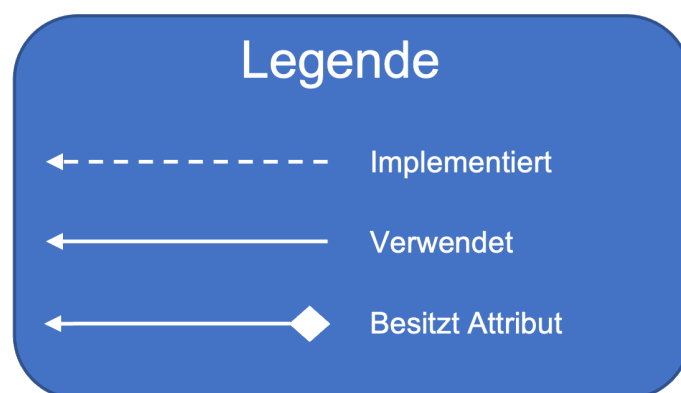
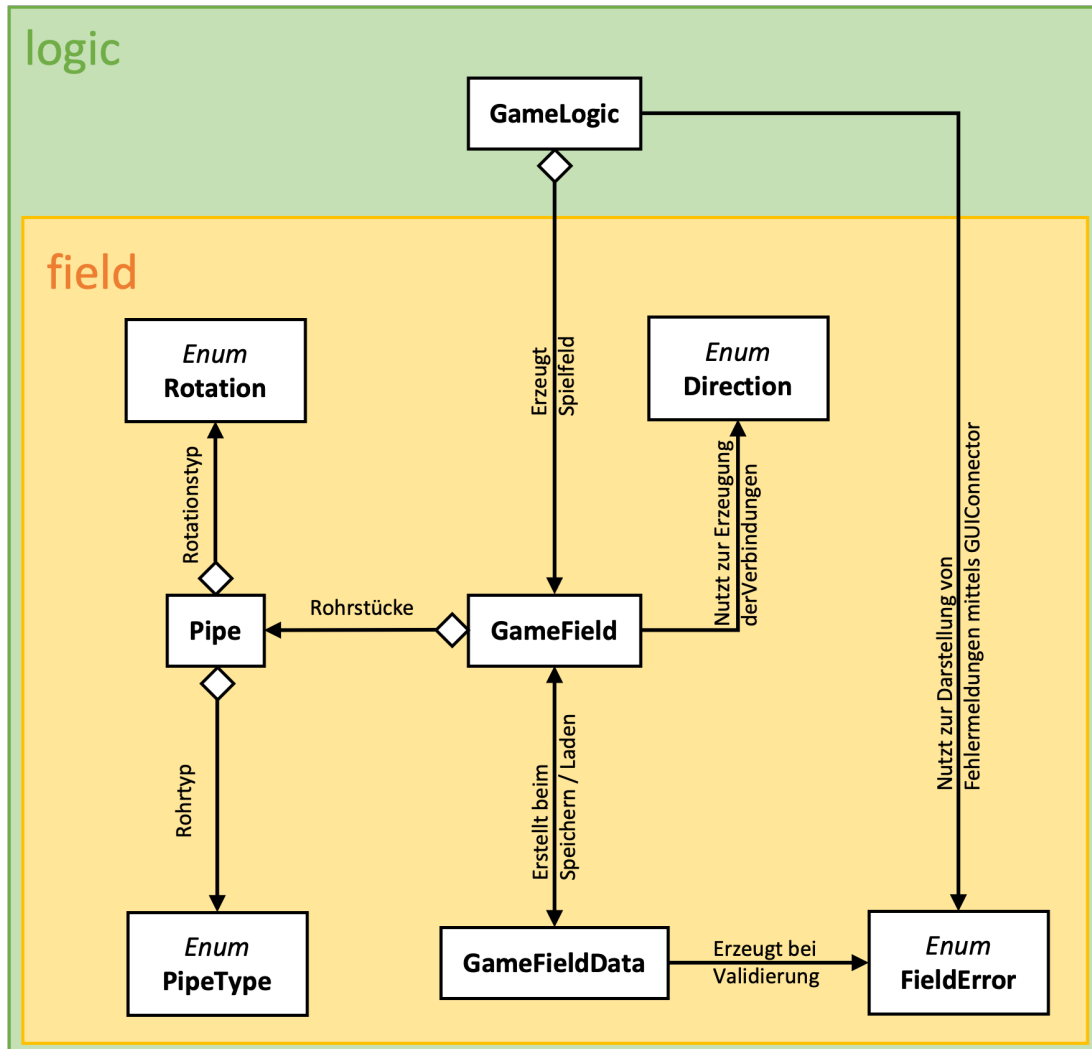


Abbildung 16: Programmorganisationsplan logic

2.4.1 Paketzuordnung

In Abbildung 15, dem Programmorganisationsplan der gui, sind die Datenstrukturen der Oberflächensteuerung dargestellt. Außerdem vorhanden ist die GameLogic, da sie eine wichtige Schnittstelle zur GUI darstellt. Sie kann sowohl auf Benutzerinteraktionen mit der Oberfläche reagieren, als auch mittels der Schnittstelle zur GUI, dem GUIConnector, Oberflächenänderungen erzeugen. Ebenfalls dargestellt ist die Klasse Position, da diese besonders wichtig für den Austausch von Informationen zwischen dem Paket gui und logic ist. Es wurde zudem ein Unterpaket field angelegt, welches für die Klassen Field und FieldCell genutzt wird. Diese stellen die GUI-Darstellung des Spielfelds dar. In Abbildung 16, dem Programmorganisationsplan der logic, ist das Paket logic dargestellt. Dieses beinhaltet die Klassen, die die logische Grundlage des Spiels FloodPipe definieren. In dieser Darstellung wird auf die bereits in Abbildung 15 vorhandenen Klassen GUIConnector und Position verzichtet. Klassen, die für die interne Speicherung des Spielfelds zuständig sind, sind dem Unterpaket field zugeordnet.

2.4.2 Erläuterung des Programmorganisationsplans

Es wurde sich bei der Darstellung der Verbindungen zwischen den einzelnen Klassen auf die grundlegenden Verbindungen beschränkt, um den Programmorganisationsplan übersichtlich zu halten. Es ist Sinn und Zweck dieses Plans, eine Übersicht über die Datenstruktur zu erhalten, welche durch die grundlegenden Verbindungen ermöglicht wird. Diese Verbindungen wurden so gewählt, dass das Implementieren eines Interfaces dargestellt wird, um die Verbindung zwischen logic und gui zu demonstrieren. Außerdem werden Verbindungen dargestellt, bei denen eine Klasse Instanzen einer anderen Klasse erzeugt und besonders gekennzeichnet sind Klassenverbindungen, die daraus bestehen, dass eine Klasse ein Attribut einer anderen Klasse darstellt. Die Klasse *Position* ist eine besondere Klasse, da diese fast ausschließlich als Methodenparameter benutzt wird, um Positionen miteinander vergleichen zu können. Dies ist besonders in der JavaFX-GUI relevant, da so für die Spielfeldzellen geprüft werden kann, ob deren Positionen beispielsweise in einem Set vorhanden sind, das die nicht gefüllten Positionen darstellt.

2.5 Dateien

In diesem Kapitel wird der Aufbau der im Programm verwendeten Dateien aufgeführt. Bei dem Spiel FloodPipe werden Spielstandsdateien im JSON-Format verwendet. Dies ist ein Datenformat, welches die Daten kodiert speichert. Vorteilhaft daran ist, dass dieses Format für den Menschen besonders gut lesbar ist. Ein JSON-Dokument besteht aus einem JSON-Objekt. Dieses besitzt eine Menge von Schlüssel-Wert Paaren. Zu Beginn dieses Objekts steht immer eine öffnende geschweifte Klammer. Beendet wird ein JSON-Dokument mittels schließender geschweifter Klammer. Die Schlüssel werden als String dargestellt. Anschließend wird der zugehörige Wert mittels Doppelpunkt zugewiesen. In unserem Fall wird ein Objekt für die Repräsentation der Position der Quelle, ein Boolean für den Überlaufmodus und ein zweidimensionales Array zur Repräsentation der Spielfeldbelegung benötigt. [4] Weitere Dateien werden nicht benötigt.

Listing 11: vorgegebene JSON Struktur [5]

```

1 {
2   "source": {
3     "x":1,
4     "y":2
5   },
6   "overflow": false,
7   "board": [
8     [6,5,3],
9     [12,7,11],
10    [2,13,9]
11  ]
12 }

```

2.6 Programmtests

Im Folgenden werden Oberflächentests aufgelistet, die die korrekte Funktionsweise des Programms überprüfen. Spieldateien, die hierfür benötigt werden, sind im Ordner "test-files" abgelegt. Spieldateien, die das Laden einer Spielsituation testen, sind im Unterordner "loading" zu finden. Für Tests, bei denen Spieldateien benötigt werden, die das Speichern einer Spielsituation oder das Wechseln vom Editormodus in den Spielmodus testen, sind diese Dateien im Unterordner "saveAndSwitchingToGame" vorhanden.

Testfall	Ergebnis
<i>Laden einer leeren Spieldatei:</i> Es wird versucht, eine Spielfelddatei zu laden, die keinen Inhalt besitzt. Dafür wird der Menüpunkt "Spiel laden" angeklickt und die Datei "empty.json" ausgewählt.	Es wird ein Alert angezeigt mit folgender Fehlermeldung: "Die Datei ist leer."
<i>Laden einer Spielfelddatei ohne Quellposition:</i> Es wird versucht, eine Spielfelddatei zu laden, bei der die Angabe der Quelle fehlt. Dafür wird der Menüpunkt "Spiel laden" angeklickt und die Datei "testNoSource.json" ausgewählt.	Es wird ein Alert angezeigt mit folgender Fehlermeldung: "Es wurde keine Position für die Quelle angegeben."
<i>Laden einer Spielfelddatei ohne Overflow:</i> Es wird versucht, eine Spielfelddatei zu laden, bei der die Angabe des Überlaufmodus fehlt. Dafür wird der Menüpunkt "Spiel laden" angeklickt und die Datei "noOverflow.json" ausgewählt.	Es wird ein Alert angezeigt mit folgender Fehlermeldung: "Es wurde kein oder ein falscher Wert für den Überlauf angegeben."
<i>Laden einer Spielfelddatei mit negativer Quellkoordinate:</i> Es wird versucht, eine Spielfelddatei zu laden, bei der eine Koordinate der Position der Quelle negativ ist. Dafür wird der Menüpunkt "Spiel laden" angeklickt und die Datei "testSourceNegativeValue.json" ausgewählt.	Es wird ein Alert angezeigt mit folgender Fehlermeldung: "Negative Koordinate für die Position der Quelle angegeben."

<i>Laden einer Spielfelddatei mit x-Koordinate außerhalb des Spielfelds:</i> Es wird versucht, eine Spielfelddatei zu laden, bei der die x-Koordinate der Position der Quelle außerhalb des Spielfelds ist. Dafür wird der Menüpunkt "Spiel laden" angeklickt und die Datei "testSourceXOutOfBounds.json" ausgewählt.	Es wird ein Alert angezeigt mit folgender Fehlermeldung: "Die Position der Quelle liegt nicht innerhalb des Spielfelds."
<i>Laden einer Spielfelddatei mit y-Koordinate außerhalb des Spielfelds:</i> Es wird versucht, eine Spielfelddatei zu laden, bei der die y-Koordinate der Position der Quelle außerhalb des Spielfelds ist. Dafür wird der Menüpunkt "Spiel laden" angeklickt und die Datei "testSourceYOutOfBounds.json" ausgewählt.	Es wird ein Alert angezeigt mit folgender Fehlermeldung: "Die Position der Quelle liegt nicht innerhalb des Spielfelds."
<i>Laden einer Spielfelddatei mit zu wenig Spalten:</i> Es wird versucht, eine Spielfelddatei zu laden, bei der weniger als die minimale Anzahl an Spalten vorhanden ist. Dafür wird der Menüpunkt "Spiel laden" angeklickt und die Datei "testNotEnoughColumns.json" ausgewählt.	Es wird ein Alert angezeigt mit folgender Fehlermeldung: "Das Spielfeld enthaelt nicht zwischen 2 und 15 Spalten."
<i>Laden einer Spielfelddatei mit zu wenig Reihen:</i> Es wird versucht, eine Spielfelddatei zu laden, bei der weniger als die minimale Anzahl an Reihen vorhanden ist. Dafür wird der Menüpunkt "Spiel laden" angeklickt und die Datei "testNotEnoughRows.json" ausgewählt.	Es wird ein Alert angezeigt mit folgender Fehlermeldung: "Das Spielfeld enthaelt nicht zwischen 2 und 15 Reihen."
<i>Laden einer Spielfelddatei mit zu vielen Spalten:</i> Es wird versucht, eine Spielfelddatei zu laden, bei der mehr als die maximale Anzahl an Spalten vorhanden ist. Dafür wird der Menüpunkt "Spiel laden" angeklickt und die Datei "testTooManyColumns.json" ausgewählt.	Es wird ein Alert angezeigt mit folgender Fehlermeldung: "Das Spielfeld enthaelt nicht zwischen 2 und 15 Spalten."
<i>Laden einer Spielfelddatei mit zu vielen Reihen:</i> Es wird versucht, eine Spielfelddatei zu laden, bei der mehr als die maximale Anzahl an Reihen vorhanden ist. Dafür wird der Menüpunkt "Spiel laden" angeklickt und die Datei "testTooManyRows.json" ausgewählt.	Es wird ein Alert angezeigt mit folgender Fehlermeldung: "Das Spielfeld enthaelt nicht zwischen 2 und 15 Reihen."
<i>Laden einer Spielfelddatei mit ungleich großen Spalten:</i> Es wird versucht, eine Spielfelddatei zu laden, bei der die Spalten nicht gleich viele Elemente beinhalten. Dafür wird der Menüpunkt "Spiel laden" angeklickt und die Datei "testNotAllColsSameLength.json" ausgewählt.	Es wird ein Alert angezeigt mit folgender Fehlermeldung: "Die Spalten des Spielfelds haben nicht die gleiche Anzahl an Feldern."
<i>Laden einer Spielfelddatei mit negativem Wert für ein Rohrstück:</i> Es wird versucht, eine Spielfelddatei zu laden, bei der ein Rohrstück einen negativen Wert besitzt. Dafür wird der Menüpunkt "Spiel laden" angeklickt und die Datei "testValuePipeTypeNegative.json" ausgewählt.	Es wird ein Alert angezeigt mit folgender Fehlermeldung: "Es wurde ein falscher Wert für den Rohrtypen angegeben (nicht zwischen 0 und 14)."

<i>Laden einer Spielfelddatei mit einem zu hohen Wert für ein Rohrstück:</i> Es wird versucht, eine Spielfelddatei zu laden, bei der ein Rohrstück einen zu hohen Wert besitzt. Dafür wird der Menüpunkt "Spiel laden" angeklickt und die Datei "testValuePipeTypeTooHigh.json" ausgewählt.	Es wird ein Alert angezeigt mit folgender Fehlermeldung: "Es wurde ein falscher Wert für den Rohrtypen angegeben (nicht zwischen 0 und 14)."
<i>Laden einer Spielfelddatei mit nur einer Quellkoordinate:</i> Es wird versucht, eine Spielfelddatei zu laden, bei der nur die x-Koordinate der Quellposition vorhanden ist. Dafür wird der Menüpunkt "Spiel laden" angeklickt und die Datei "testSourceOnlyOneValue.json" ausgewählt.	Es wird ein Alert angezeigt mit folgender Fehlermeldung: "Es wurden nicht zwei Koordinaten für die Position der Quelle angegeben."
<i>Laden einer Spielfelddatei mit einem falschen Wert für Overflow:</i> Es wird versucht, eine Spielfelddatei zu laden, bei der für den Überlaufmodus kein Boolean-Wert eingetragen ist. "Spiel laden" angeklickt und die Datei "testOverflowWrongValue.json" ausgewählt.	Es wird keine Fehlermeldung erzeugt und die Spielsituation dargestellt mit ausgeschaltetem Überlaufmodus. Dies liegt daran, dass hierfür der JsonParser angepasst werden muss. Ansonsten wird der Wert standardmäßig auf false gesetzt.

<i>Platzierung eines Mauerstücks auf der Quelle:</i> Für diesen Test wird die Spielfelddatei "testPlaceWallOnSource.json" geladen. Anschließend wird in den Editormodus gewechselt und ein Mauerstück auf das Feld mit der Quelle platziert. Anschließend wird versucht in den Spielmodus zu wechseln und das Spiel zu speichern.	Die gefüllten Felder werden nach Platzierung der Mauer geleert. Beim Versuch, das Spiel zu speichern, wird ein Alert mit folgender Fehlermeldung angezeigt: "Das Spiel kann nicht gespeichert werden, wenn keine Quelle vorhanden ist." Beim Versuch, in den Spielmodus zu wechseln, wird ein Alert mit folgender Fehlermeldung angezeigt: "Es kann nicht in den Spielmodus gewechselt werden, wenn keine Quelle auf dem Spielfeld vorhanden ist. "
---	---

<p><i>Entfernung der Reihe, auf der die Quelle platziert ist:</i> Für diesen Test wird die Spielfelddatei "testRemoveRowSource.json" geladen. Anschließend wird in den Editormodus gewechselt und die Anzahl an Reihen mittels des Sliders um eins verringert. Anschließend wird versucht in den Spielmodus zu wechseln und das Spiel zu speichern.</p>	<p>Die gefüllten Felder werden geleert, nachdem die Reihe, auf der die Quelle platziert war, entfernt wird. Beim Versuch, das Spiel zu speichern, wird ein Alert mit folgender Fehlermeldung angezeigt: "Das Spiel kann nicht gespeichert werden, wenn keine Quelle vorhanden ist." Beim Versuch, in den Spielmodus zu wechseln, wird ein Alert mit folgender Fehlermeldung angezeigt: "Es kann nicht in den Spielmodus gewechselt werden, wenn keine Quelle auf dem Spielfeld vorhanden ist. "</p>
<p><i>Entfernung der Spalte, auf der die Quelle platziert ist:</i> Für diesen Test wird die Spielfelddatei "testRemoveColSource.json" geladen. Anschließend wird in den Editormodus gewechselt und die Anzahl an Spalten mittels des Sliders um eins verringert. Anschließend wird versucht in den Spielmodus zu wechseln und das Spiel zu speichern.</p>	<p>Die gefüllten Felder werden geleert, nachdem die Spalte, auf der die Quelle platziert war, entfernt wird. Beim Versuch, das Spiel zu speichern, wird ein Alert mit folgender Fehlermeldung angezeigt: "Das Spiel kann nicht gespeichert werden, wenn keine Quelle vorhanden ist." Beim Versuch, in den Spielmodus zu wechseln, wird ein Alert mit folgender Fehlermeldung angezeigt: "Es kann nicht in den Spielmodus gewechselt werden, wenn keine Quelle auf dem Spielfeld vorhanden ist. "</p>
<p><i>Platzierung der Quelle auf einem Mauerstück:</i> Für diesen Test wird die Spielfelddatei "testPlaceSourceOnWall.json" geladen. Anschließend wird in den Editormodus gewechselt und die Quelle per Drag&Drop auf das vorhandene Mauerstück gezogen.</p>	<p>Das Mauerstück wird nicht blau markiert. Außerdem wird nicht angezeigt, dass ein Drop möglich ist. Beim Drop wird die Quelle nicht auf dem Mauerstück platziert und die blaue Markierung der Quelle verschwindet.</p>