

# Programmierpraktikum

Hase und Igel

*Frahm, Lukas, Computer Games Technology, 105751*

## INHALTSVERZEICHNIS

<b>1</b>	<b>Benutzerhandbuch</b>	<b>2</b>
1.1	Ablaufbedingungen	2
1.2	Programminstallation/Programmstart	2
1.3	Bedienungsanleitung	3
1.3.1	Allgemeine Spiel-Information	3
1.3.2	Spielstart	3
1.3.3	Spielablauf	3
1.3.4	Feldtypen und ihre Aktionen	5
1.3.5	Hasenkarten und ihre Wirkung	7
1.4	Fehlermeldungen	9
1.4.1	Fehlermeldungen mit Benachrichtigung	9
1.4.2	Andere Fehler	9
<b>2</b>	<b>Programmiererhandbuch</b>	<b>10</b>
2.1	Entwicklungskonfiguration	10
2.2	Problemanalyse und Realisation	10
2.2.1	Logik des Spiels	10
2.2.2	GUI-Darstellung des Spielmenüs	14
2.2.3	GUI-Darstellung des Spielbretts	15
2.3	Algorithmen	19
2.4	Programmorganisationsplan	20
2.5	Dateien	21
2.6	Programmtests	22

# 1 BENUTZERHANDBUCH

## 1.1 ABLAUFBEDINGUNGEN

### BETRIEBSSYSTEM: WINDOWS 11

Entwickelt und getestet auf Windows 11. Das Programm sollte auf allen modernen Windows-Versionen lauffähig sein. Plattformübergreifende Kompatibilität ist gegeben, ist aber nicht explizit getestet worden.

### JAVA-VERSION: JAVA 17

Das Programm benötigt zur Ausführung mindestens Java SE 17. Für eine optimale Kompatibilität wird die Verwendung von Java Temurin-17.0.9 empfohlen.

### NETZWERKANFORDERUNGEN: KEINE

Keine Internetverbindung erforderlich.

## 1.2 PROGRAMMINSTALLATION/PROGRAMMSTART

### PROGRAMMDATEI (JAR):

Das Programm wird als ausführbare JAR-Datei (Java Archive) bereitgestellt. Es ist nicht notwendig, Archive zu entpacken oder spezielle Installationsprozesse durchzuführen.

### PROGRAMM AUSFÜHREN:

Das Programm wird durch Doppelklicken auf die JAR-Datei gestartet. Alternativ kann auch über die Kommandozeile oder ein Terminalfenster mit dem Befehl „`java -jar pp_haseundigel_frahm.jar`“ gestartet werden.

### ERSTELLUNG DER LOG-DATEI:

Bei der Ausführung des Programmes wird unter Umständen eine log.txt Datei im gleichen Verzeichnis wie die JAR-Datei automatisch erstellt. Diese Log-Datei dient der Aufzeichnung des Spielverlaufs und kann zur Fehleranalyse, insbesondere im Falle eines Programmabsturzes, nützlich sein.

### UMGANG MIT DER LOG-DATEI:

Falls die log.txt-Datei regelmäßig auf dem Desktop erscheint, so kann diese problemlos ohne Beeinträchtigung des Programmes gelöscht werden. Die Datei wird vom Programm bei Bedarf neu erstellt. Um zu vermeiden, dass die log.txt-Datei wiederholt auf dem Desktop erscheint, wird empfohlen, die JAR-Datei in einen separaten Ordner zu verschieben. Dadurch wird sichergestellt, dass die log.txt-Datei in diesem neuen Verzeichnis generiert wird.

## 1.3 BEDIENUNGSANLEITUNG

### 1.3.1 ALLGEMEINE SPIEL-INFORMATION

**Spieleranzahl:** Das Spiel kann von 2 bis 6 Spielern gespielt werden.

**Spielziel:** Ziel ist es, mit seiner eigenen Spielfigur vor den anderen Spielern das Ziel zu erreichen.

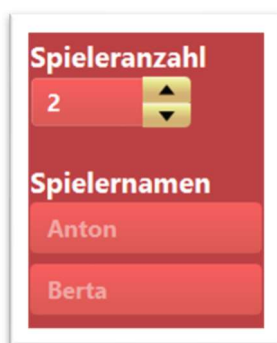
### 1.3.2 SPIELSTART

**Spieleranzahl einstellen:**

Die Anzahl der Spieler kann mit den Pfeil-Knöpfen zwischen 2 und 6 beliebig erhöht oder verringert werden.

**Spielernamen:**

Unterhalb der Spieleranzahlauswahl werden die Spielernamen angezeigt. Standardmäßig sind diese hellgrau und vorgegeben, können aber durch Anklicken des jeweiligen Feldes individualisiert werden, diese erscheinen dann in Weiß.



**Spielerreihenfolge:**

Die Reihenfolge der Spieler richtet sich nach Anordnung der Namen (oberstes Feld = erster Spieler).

**Spielerfarben:**

Jeder Spieler hat eine zugeordnete Farbe:

1. Spieler: Blau
2. Spieler: Gelb
3. Spieler: Grün
4. Spieler: Orange
5. Spieler: Rot
6. Spieler: Weiß



### 1.3.3 SPIELABLAUF

**Rundenbasiertes Spiel:**

Das Spiel wird in Runden gespielt, wobei jeder Spieler für gewöhnlich ein Feld auswählt, auf welche er seine Spielfigur bewegen möchte.





### Karotten als Währung:

Jeder Spieler beginnt mit einer Anzahl an Karotten, die von der Spieleranzahl abhängt:

-  weniger als 5 Spieler: 68 Karotten
-  5 oder 6 Spieler: 98 Karotten

Diese Karotten werden verwendet, um sich auf dem Spielfeld nach vorne zu bewegen.

### Bewegungskosten:

-  1 Feld vorwärts = 1 Karotte
-  2 Felder vorwärts =  $1 + 2 = 3$  Karotten
-  3 Felder vorwärts = 3 (*Kosten für 2 Felder*) + 3 = 6 Karotten
-  Für jede weitere Bewegung addieren sich die Kosten progressiv.

### Berechnung der Karottenkosten:

Die Berechnung der Karottenkosten für größere Distanzen kann komplex werden. Beispielsweise kostet die Bewegung über 13 Felder bereits 91 Karotten. Mit einem Startvorrat von 98 Karotten können Spieler sich maximal 13 Felder weit bewegen, bevor ihre Karotten aufgebraucht sind. Damit die Kosten nicht jedes Mal ausgerechnet werden müssen, unterstützt das Programm den Spieler.

### Feldauswahl und -anzeige:

Grüne Umrandung: Zeigt an, dass genügend Karotten für das Erreichen des Feldes vorhanden sind und es sich um ein Feld handelt, welches der Spieler betreten darf. Der Spieler kann sich durch Linksklick auf das grün umrandete Feld bewegen.

Rote Umrandung: Indiziert, dass nicht genügend Karotten vorhanden sind oder das Feld aus anderen Gründen nicht betreten werden darf. Diese Umrandung erscheint, wenn der Spieler mit dem Mauszeiger über das Feld fährt. Bei einem Linksklick passiert hier nichts.

### Karottenanzeige:

Die Anzahl der Karotten, die ein Spieler besitzt, wird oben rechts unter dem Spielernamen angezeigt. Nach dem Symbol „->“ wird die Anzahl der Karotten angezeigt, die ein Spieler noch hätte, wenn er sich auf das Feld bewegt, über dem der Mauszeiger gerade schwebt. Diese Anzeige erfolgt nur, wenn das Feld gültig, also grün umrandet ist.



### Salate als Spielkomponente:

Jeder Spieler startet das Spiel mit 3 Salaten. Diese Salate müssen im Laufe des Spiels abgeben werden, da ein Spieler das Ziel nur erreichen kann, wenn er keine Salate mehr besitzt. Wird ein Salat abgeben, bekommt der Spieler 10 Karotten multipliziert mit der Platzierung des Spielers auf dem Spielbrett.

### Spiefelder:

Das Spielbrett besteht aus einer Reihe von Felder, die jeweils unterschiedliche Typen und Funktionen haben. Diese Felder können das Spiel und die Strategie der Spieler auf verschiedene Weisen beeinflussen. Die Aktionen oder Ereignisse, die mit den verschiedenen Spielfeldern verbunden sind, werden in der Regel erst aktiv, nachdem der Spieler das betreffende Feld betreten hat und anschließend erneut an der Reihe ist.

## 1.3.4 FELDTYPEN UND IHRE AKTIONEN

### KAROTTEN – FELD



**Zugang:** Jederzeit, sofern genügend Karotten vorhanden sind und das Feld nicht belegt ist.

**Aktionen bei Betreten:** Keine sofortige Aktion.

**Aktionen in der nächsten Runde:**

Wahl aus:

- Erhalt von 10 Karotten
- Abgabe von 10 Karotten
- Weiterziehen auf ein neues Feld

### ZAHLN – FELD



**Zugang:** Jederzeit, sofern genügend Karotten vorhanden sind und das Feld nicht belegt ist.

**Aktionen bei Betreten:** Keine sofortige Aktion.

**Aktionen in der nächsten Runde:** Weiterziehen auf ein neues Feld. Abhängig von der aktuellen Position des Spielers im Rennen, kann das Zahlenfeld aber zuerst Belohnungen in Form von Karotten auslösen.

Beispiel: Ein Spieler landet auf dem Zahlenfeld mit der Zahl „2“. In der nächsten Runde, wenn dieser Spieler auf dem zweiten Platz im Rennen steht, erhält er 20 Karotten, andernfalls erhält er keine Karotten. (*erster Platz 10 Karotten, zweiter 20, dann 30, etc.*)

**Flaggenfeld:** Ein spezielles Zahlenfeld mit einer Flagge gilt für den ersten, fünften und sechsten Platz und vergibt Karotten entsprechend der Platzierung des Spielers.

**Einschränkungen:** In Spielen mit weniger Spielern sind einige Zahlenfelder möglicherweise wirkungslos, da nicht genügend Spieler vorhanden sind, um alle Platzierungen zu besetzen.

## SALAT – FELD



**Zugang:** Gleich den vorherigen Feldern, mit der zusätzlichen Bedingung, dass auch noch Salate vorhanden sein müssen.

**Aktionen bei Betreten:** Keine sofortige Aktion.

**Aktionen in der nächsten Runde:** Verzehr eines Salats, wodurch ein Salat abgegeben und die Runde ausgesetzt wird. Spieler erhalten Karotten entsprechend ihrer Platzierung. (Siehe „Salate als Spielkomponente“)

## IGEL - FELD



**Zugang:** Rückwärtsbewegung zum nächsten Igel-Feld, vorwärts ist nicht möglich. Keine Karotten nötig, um rückwärtszuziehen.

**Aktionen bei Betreten:** Erhalt von Karotten, berechnet als Anzahl zurückgelegter Felder mal 10.

**Aktionen in der nächsten Runde:** Weiterziehen auf ein neues Feld. Wenn dies vorwärts geschieht, dann auch wieder mit Karottenkosten.

## HASEN – FELD



**Zugang:** Jederzeit, sofern genügend Karotten vorhanden sind und das Feld nicht belegt ist.

**Aktionen bei Betreten:**

- Automatisches Ziehen: Wenn ein Spieler auf ein Hasenfeld zieht, wird automatisch eine Hasenkarte vom oberen Ende des Kartenstapels gezogen.
- Rotation der Karten: Nach dem Ziehen einer Karte wird diese sofort unterhalb des Stapels wieder eingefügt. Dieser Vorgang führt dazu, dass sich die Reihenfolge der Karten im Laufe des Spiels wiederholt.
- Mehrere Hasenfelder: Es ist möglich, dass eine Karte den Spieler von einem Hasenfeld auf ein anderes bewegt, auch dann wird wieder eine Karte gezogen.

**Aktionen in der nächsten Runde:** In vielen Fällen führt das Ziehen einer Hasenkarte dazu, dass der Spieler das Hasenfeld verlässt und sich auf ein anderes Feld bewegt. Sollte jedoch die gezogene

Hasenkarte den Spieler auf dem Hasenfeld belassen, muss der Spieler in der darauffolgenden Runde einen regulären Zug durchführen und auf ein neues Feld ziehen, es sei denn, die Hasenkarte zwingt den Spieler länger zu bleiben.

### 1.3.5 HASENKARTEN UND IHRE WIRKUNG

#### EINMAL AUSSETZEN

**Aktion:** Der Spieler setzt aus und bleibt für die aktuelle und die nächste Runde auf dem Hasenfeld stehen.

**Anzahl:** 1x im Stapel

#### VORWÄRTS ZUM NÄCHSTEN KAROTTEN-FELD

**Aktion:** Sofortiges Vorwärtsziehen zum nächsten freien Karottenfeld. Sollte kein freies Karottenfeld vorwärts erreichbar sein, bleibt der Spieler auf dem Hasenfeld.

**Anzahl:** 1x im Stapel

#### ZURÜCK ZUM LETZTEN KAROTTEN-FELD

**Aktion:** Sofortiges Rückwärtsziehen zum letzten freien Karottenfeld. Sollte kein freies Karottenfeld rückwärts erreichbar sein, bleibt der Spieler auf dem Hasenfeld.

**Anzahl:** 1x im Stapel

#### ZIEHE NOCHMAL

**Aktion:** Der Spieler darf vom Hasenfeld aus normal weiterziehen. Dies kostet Karotten, es sei denn, es erfolgt ein Rückzug auf ein Igel-Feld.

**Anzahl:** 1x im Stapel

#### LETZTER ZUG KOSTET NICHTS

**Aktion:** Der Spieler erhält alle Karotten zurück, die für das Erreichen des Hasenfeldes ausgegeben wurden.

**Anzahl:** 2x im Stapel

#### 10 KAROTTEN NEHMEN / ABGEBEN

**Aktion:** Der Spieler entscheidet zwischen Nehmen und Abgeben von 10 Karotten, ähnlich wie bei einem Karottenfeld. Als dritte Option kann der Spieler nichts tun, außer weiterzuziehen.

**Anzahl:** 2x im Stapel

#### ESSE EINEN SALAT

**Aktion:** Der Spieler verhält sich, als wäre er auf einem Salatfeld gelandet und isst in der nächsten Runde einen Salat. Hat der Spieler keine Salate mehr, muss er dennoch die nächste Runde aussetzen, aber erhält keine Karotten dafür.

**Anzahl:** 1x im Stapel



## POSITION ZURÜCKFALLEN

**Aktion:** Der Spieler fällt um eine Position zurück und wird ein Feld hinter den Spieler gesetzt, der eine Platzierung hinter ihm ist. Ist dieses Feld nicht betretbar, muss er möglicherweise weitere Felder zurückfallen, bis hin zum Startfeld. Für das zurückfallen auf ein Igel-Feld gibt es jedoch keine Karotten. Falls der Spieler bereits auf dem letzten Platz ist, wird die Karte wirkungslos.

**Anzahl:** 2x im Stapel

## POSITION VORRÜCKEN

**Aktion:** Der Spieler rückt um eine Position vor und zieht ein Feld vor den Spieler, der einen Platz vor ihm ist. Ist dieses Feld nicht betretbar, kann der Spieler noch weitere Felder vorrücken, bis hin zum Ziel. Ins Ziel kann er aber nur ziehen, wenn er auch die Bedingungen dafür erfüllt. Falls der Spieler bereits führt oder es kein betretbares Feld gibt, ist die Karte wirkungslos.

**Anzahl:** 1x im Stapel

## 1.4 FEHLERMELDUNGEN

Kommt es zu Fehlern im Programm, so kann eine Benachrichtigung, ähnlich wie beim Ziehen einer Karte, angezeigt werden, dies ist jedoch meistens nicht der Fall. Es gibt einige Fehler, welche den Spielverlauf nicht weiter beeinflussen und nur im Log eingesehen werden können.

### 1.4.1 FEHLERMELDUNGEN MIT BENACHRICHTIGUNG

Diese Fehlermeldungen verschwinden durch das Klicken auf „OK“ in der Benachrichtigung.

**Speichern fehlgeschlagen:** *„Speichern fehlgeschlagen! Versuche es mit einem anderen Dateinamen erneut.“*

Dieser Fehler tritt auf, wenn die zu speichernde Datei nicht richtig benannt wurde, zum Beispiel der Name einfach leer gelassen wurde, aber versucht wurde zu speichern. Möchte man die Datei speichern, so kann man es mit einem anderen, am besten nicht bereits verwendeten, Namen versuchen.

**Laden fehlgeschlagen:** *„Laden fehlgeschlagen! Stelle sicher, dass die Datei auch ein gültiger Spielstand ist.“*

Dieser Fehler tritt auf, wenn die zu ladende Datei nicht dem richtigen Format entspricht oder die Spieldaten in der Datei kein plausibles funktionierendes Spiel bilden. Dies passiert eigentlich nur wenn versucht wird, einen nicht vom Spiel erzeugten oder durch den Benutzer veränderten Spielstand zu laden.

*Für Interessierte: Möchte man die Datei dennoch laden, muss man diese erst korrigieren. Dafür kann man die Datei mit einer vom Spiel neuerzeugten Speicherdatei vergleichen, um mögliche Format-Fehler zu finden. Behebt das den Fehler immer noch nicht, kann man überlegen, ob die Datei in Hinblick auf die Spielregeln überhaupt valide ist, ein Spieler kann beispielsweise keinen Salat auf dem Startfeld essen oder negative Karotten oder Salate haben. Dies ist aber nur relevant, wenn man eigene Spielstände schreiben möchte.*

### 1.4.2 ANDERE FEHLER

**Das Programm stürzt beim Öffnen ab:** Wenn dieser Fehler auftritt, sollte die JAR-Datei neu heruntergeladen und überprüft werden, ob die richtige Version von Java installiert ist (Siehe Ablaufbedingungen).

**Die Spielsteine sind nur farbige Kreise ohne Bild:** Das Laden der Spielstein-Bilder ist fehlgeschlagen, das Spiel kann dennoch normal weitergespielt werden. Ein Neustart des Programms kann helfen.

**Auf dem Spielbrett bewegt sich nichts mehr und es lassen sich keine Felder auswählen:** Stelle sicher, dass es nicht vielleicht doch ein Feld gibt, auf welches der Spieler ziehen kann und prüfe, dass keine Benachrichtigungen, wie zum Beispiel Hasenkarten, in den Hintergrund verschoben worden. Ist beides nicht der Fall muss ein neues Spiel gestartet werden, dieser Fehler sollte in einem normalen Spiel aber nicht auftreten.

*Im Log lassen sich manchmal noch weitere Fehler in Bezug auf das Laden und Speichern von Spielständen finden, diese sollten jedoch irrelevant für den Spielablauf sein und diesen nicht weiter beeinflussen.*

## 2 PROGRAMMIERERHANDBUCH

### 2.1 ENTWICKLUNGSKONFIGURATION

Kategorie	Tool/Software	Version
Betriebssystem	Windows 11	21H2
Java Development Kit (JDK)	OpenJDK	17.0.9
Integrierte Entwicklungsumgebung (IDE)	IntelliJ IDEA	2021.3.1 (Community Edition)
Java Frameworks und Bibliotheken	JavaFX	17.0.2
	Gson	2.10.1
	JUnit	4.13.2
Versionskontrolle	Subversion (SVN)	1.14.2
Build-Management	Apache Maven	4.0.0
Build-Konfiguration	Maven-Compiler-Plugin	3.8.1
	JavaFX-Maven-Plugin	0.0.6
	Maven-Jar-Plugin	3.1.0
	Maven-Shade-Plugin	3.2.4

### 2.2 PROBLEMANALYSE UND REALISATION

#### 2.2.1 LOGIK DES SPIELS

##### PROBLEMANALYSE

Die Implementationsdetails des Spiels können größtenteils aus dem Benutzerhandbuch abgeleitet werden. Allerdings gibt es spezifische Herausforderungen, die über das Benutzerhandbuch hinausgehen und eine detailliertere Betrachtung erfordern.

- **Echtzeit-Log-Erstellung:** Es muss ein Log in Echtzeit erstellt werden, um den Spielablauf genau nachstellen zu können, insbesondere für Debugging-Zwecke. Dieses Log sollte alle relevanten Spielaktionen, Entscheidungen und Ereignisse chronologisch erfassen. Wenn ein neues Spiel erstellt oder geladen wird, sollte der alte Log überschrieben werden.
- **Speichern und Laden von Spielständen:** Das Spiel muss die Möglichkeit bieten, Spielstände zu laden und zu speichern. Dabei muss speziell der Zustand jedes Spielers, einschließlich seiner Karotten und Salate, Positionen, Namen, sowie geplanter Aktionen für die nächste Runde (wie das Essen eines Salats oder das Aussetzen einer Runde) gesichert werden. Ebenso wichtig ist das Speichern der Reihenfolge der Spieler, die das Ziel erreicht haben. Der Zustand des Kartenstapels muss nicht gespeichert werden, obwohl dies zu unterschiedlichem Verhalten nach dem Laden führen kann.
- **GUI-Interaktionen:** Die Spiellogik muss in der Lage sein, effektiv mit der GUI zu interagieren. Dies beinhaltet das Warten auf und Verarbeiten von Benutzereingaben sowie die Steuerung der GUI zur Anzeige von Informationen (z.B. "Du bekommst 10 Karotten" oder "Du musst um eine Position zurückfallen"). Die Logik sollte auch in der Lage sein, Anweisungen zur Bewegung der Spielerfiguren

zu geben, Feldauswahlen zu ermöglichen und die Rückgabe dieser Auswahlentscheidungen zu verarbeiten. Zusätzlich müssen die GUI-Elemente zur Anzeige der Namen, Karotten- und Salatbestände aktualisiert werden, basierend auf den von der Logik bereitgestellten Daten.

- **Fehlerbehandlung:** Die Fehlerbehandlung sollte so konzipiert sein, dass sie den Benutzer über Fehler informiert und ihm Lösungsvorschläge anbietet, ohne das Spielerlebnis unnötig zu beeinträchtigen. Fehlermeldungen sollten klar, hilfreich und nicht-invasiv sein, um eine störungsfreie Benutzererfahrung zu gewährleisten.
- **Spielablauf:** Das Spiel folgt einem rundenbasierten und sequenziellen Ablauf, typisch für viele Spiele dieser Art. Zu jedem Zeitpunkt wird nur eine Aktion ausgeführt. Diese Struktur vereinfacht die Handhabung von Spielzuständen und Benutzerinteraktionen.

## REALISATIONSANALYSE

Da das Spiel einen sequenziellen Ablauf hat, und nach fast jeder Ausführung einer Aktion eine GUI-Anweisung kommt, worauf die Logik auf eine Benutzereingabe warten muss, bietet sich ein Callback-System an. Hier bietet sich das Runnable-Interface an. Für jeden Befehl an die GUI könnte ein Runnable mitgegeben werden, das nach der Benutzereingabe ausgeführt wird. Für die Feldauswahl könnte ein Custom Interface genutzt werden, das das ausgewählte Feld beinhaltet. Entscheidungen könnten jeweils eigene Runnables erhalten, da je nach Entscheidung die Logik unterschiedlich fortfährt. Eine alternative Methode wäre, der GUI die Verantwortung zu überlassen, nach einer Benutzereingabe die passende Funktion in der Logik aufzurufen. In dem ersten Lösungsansatz wird der Programmablauf fast ausschließlich durch die Logik kontrolliert. Im zweiten Lösungsvorschlag wird ein großer Teil des Programmablaufes durch die GUI bestimmt, da sie die jeweilige Funktion in der Logik aufruft, welche nach einer Benutzereingabe ausgeführt wird. Der Unterschied ist also bei dem einen bestimmt die Logik, was nach einer Benutzereingabe ausgeführt wird, bei dem anderen bestimmt es die GUI. In beiden Fällen sollte die Kommunikation mit der GUI über ein Interface laufen, so können theoretisch unterschiedliche GUIs implementiert werden, welche beispielsweise verschiedene Sprachen unterstützen könnten.

Eine genauere Betrachtung des Benutzerhandbuchs und der Spielregeln zeigt, dass je nach Spielerzustand und dem Feld, auf dem er sich befindet, unterschiedliche Aktionen ausgeführt werden. Einige Aktionen haben teilweise ähnliches oder sogar identisches Verhalten, wie das Betreten eines Karottenfeldes oder das Ziehen der Hasenkarte, welche die Aktion eines Karottenfeldes, bis auf das Weiterziehen, nachahmt. Deshalb erscheint es sinnvoll, die Aktionen in eine separate Klasse auszulagern, um Teile wiederverwenden zu können und neue Aktionen einfach integrieren zu können. Auch die Feldtypen könnten in eine oder mehrere separate Klassen ausgelagert werden, da jeder Feldtyp eigene Bedingungen und mögliche Aktionen hat, wie: ob dieser Feldtyp betreten werden kann, eigene mögliche Aktionen, die beim Betreten des Feldes passieren können, wie beim Igel auf Karotten zu bekommen und eigene Aktionen, die beim Start auf einem Feld ausgeführt werden. Vererbung könnte hierbei hilfreich sein, allerdings besteht der Nachteil, dass Informationen übermittelt werden müssen, die nur für spezielle Feldtypen relevant sind.

Da es immer nur ein Spielbrett gibt, und dieses unabhängig vom Spiel immer die gleiche Reihenfolge an Feldern hat, sollte dieses ebenfalls eine eigene, wahrscheinlich statische Klasse sein. Der Spieler sollte auch eine eigene Klasse sein, da die Spieler sich keine Namen oder andere Attribute teilen. Eine allgemeine Spielklasse, die alle diese Systeme zusammenführt, wäre ebenfalls sinnvoll. In dieser können dann die Spieler in einem Array gespeichert werden. Den aktuellen Spieler kann man dann durch seinen Index in dem Array abbilden, so wird es auch in der vorgegebenen JSON-Datei gehandhabt. In der Speicher Datei wird auch die Liste, der im Ziel angekommenen Spieler, als Array der Indizes gespeichert. Hier macht es aber wahrscheinlich mehr Sinn auf eine Listenstruktur zurückzugreifen, da sich während des Spiels die Liste noch vergrößern kann.

Das Speichern des Spiels ist größtenteils durch die Aufgabenstellung vorgegeben und die Verwendung von Gson zur Umwandlung des Spiels in eine JSON-Datei wird empfohlen. Das Laden könnte über einen Konstruktor einer Spielklasse erfolgen, der ein de-serialisiertes Spiel-JSON erhält, wodurch ein neues Spiel mit den geladenen Daten erstellt wird. Das Neuerstellen des Spiels beim Laden hat den Vorteil, dass viele Attribute unveränderlich sein können, jedoch müssen einige Informationen, die bereits vorhanden sind, neu erstellt werden.

Da es immer nur ein Log gibt, der bei Bedarf überschrieben wird, wäre es sinnvoll, diesen ebenfalls als statisch zu behandeln. Der Spielablauf lässt sich gut abbilden, wenn alle Aktionen dem Log-Befehle geben können, Nachrichten einzutragen. Da immer nur eine Aktion zurzeit ausgeführt wird, sollten keine Probleme auftreten, dass mehrere Aktionen gleichzeitig versuchen, etwas in das Log zu schreiben. Da der Kartenstapel in jedem Spiel zufällig generiert wird, das Log aber dazu dient, den Spielablauf rekreieren zu können, könnte es sinnvoll sein, einen Game Seed im Log festzuhalten, der im Spiel gesetzt wird, um ein Spiel exakt gleich, mit demselben Kartenstapel, rekreieren zu können.

Bei der Fehlerbehandlung können Fehler effektiv im Log erfasst werden, was nützlich ist, um zu verstehen, warum ein Spiel möglicherweise abgestürzt ist. Zusätzlich sollte die GUI in der Lage sein, Fehlermeldungen an den Spieler zu übermitteln, die klar und verständlich sind, und möglicherweise Lösungsansätze bieten. Das Ziel ist es dann, das Spiel, wenn möglich, fortzuführen, auch wenn Fehler auftreten.

## REALISATIONSBESCHREIBUNG

Siehe für Klassenstrukturen und Zuständigkeiten Programmorganisationsplan. Als Schnittstelle zwischen Logik und GUI wird in der Logik ein GUI-Connector Interface verwendet. Dieses beinhaltet alle möglichen Befehle, welche an die GUI gesendet werden können. Diese bekommen dann alle ein Runnable, welches nach einer Benutzereingabe ausgeführt wird. Die Kommunikation mit der GUI läuft also über ein Callback-System, dieses hat den Vorteil, dass der komplette Spielfluss so durch die Logik gesteuert wird. So wird verhindert das Bugs in der Logik durch die GUI verursacht werden, wie beispielsweise das Aufrufen einer falschen Funktion in der Logik nach einer Benutzereingabe verursacht durch die GUI. Findet man also ein Problem in der Spiellogik ist sichergestellt, dass der Bug innerhalb des Logik-Paketes zu finden ist.

Alle Methoden im GUI-Connector, welche dem Spieler nur Informationen zeigen und auf eine eindeutige Benutzereingabe warten, bekommen ein Runnable. Die Methode, welche die Feldauswahl aktiviert, bekommt ein OnFieldSelected Interface, welches wie das Runnable Interface funktioniert und auch noch als Eingabe ein Feld bekommt. Dadurch kann dann die Feldauswahl des Benutzers an die Logik kommuniziert werden. Für die Wahlen aus drei Optionen, die ein Spieler treffen kann, werden einfach drei verschiedene `Runnables` in die Funktion gegeben, jedes Runnable steht dabei für eine bestimmte Option.

Die verschiedenen Feldtypen erben alle von der abstrakten Klasse Tile, welche Methoden implementiert, um zu überprüfen, ob das Feld betreten werden kann, welche Aktionen beim Betreten und beim Start auf diesen Feldtypen ausgeführt werden sollen. Die Feldklassen werden dann in einem enum-Register noch weiter unterteilt. Die Zahlenfelder funktionieren nämlich prinzipiell alle gleich, sie sind also eine Klasse, jedoch je nach Zahl in ihrer Ausgabe unterschiedlich, in dem Register werden dann die benötigten Zahlenfelder alle gespeichert. Die enums lassen sich nun einfach in die Board Klasse in einem Array aneinanderreihen und bilden so das Spielbrett. Da alle diese Felder von der Tile-Klasse abstammen, können wir nun für jedes beliebiges Feld abfragen, ob es betretbar ist, was beim Betreten und beim Start auf diesem Feld passieren soll. Dies macht die Implementierung und potenziellen Änderungen in der Zukunft in den Spielregeln einfach und dynamisch.

Ein potenzielles Problem was dadurch jedoch eingeführt wird, ist das die wir eine Menge Information in diese drei Methoden geben müssen, da die Feldtypen alle abhängig von unterschiedlichen Kontexten im Spiel sind, so kann es schnell passieren das wir versehentlich Seiteneffekte einbauen. Hier gibt es

zum Glück eine einfache Lösung, welche das Problem fast vollständig eliminiert. Schauen wir uns genauer an was diese Methoden für Informationen brauchen, stellt sich heraus, dass es immer nur Kontext vom Spiel ist, welcher auch nur gelesen werden muss. Die Spiel Klasse implementiert also so ein Spiel Kontext interface, dieses beinhaltet dann nur pure functions und das Problem der Seiteneffekte ist eliminiert (Wir haben eine Ausnahme in diesem Interface). Das Board hat also jetzt ein Array der Feldtypen. In der Spieler Klasse können wir die Information, auf welchem Feld der Spieler steht, speichern, indem wir den Index in dem Feldtypen Array speichern. Das Inventar (Karotten und Salate) speichern wir auch in der Spieler Klasse. Ebenso den Status, ob der Spieler nächste Runde aussetzen muss oder einen Salat essen wird.

Die Spielklasse fragt diesen Status nun ab: Muss der Spieler aussetzen wird der Status zurückgesetzt und der Spieler übersprungen, isst der Spieler einen Salat, wird der Status zurückgesetzt und die `Aktion` zum Essen eines Salates wird ausgeführt, ist keiner dieser beiden Status gesetzt, wird das Feld, auf welchem der Spieler steht nach seiner `Aktion` gefragt und diese mit dem aktuellen Spieler ausgeführt. Die Spiel Klasse prüft am Ende jedes Spielerzuges, ob der Spieler das Ziel erreicht, hat, wenn ja wird der Spieler einer ArrayList hinzugefügt. Die Liste muss spezifisch als ArrayList implementiert sein, um auch die Reihenfolge der fertigen Spieler zu speichern. Anders als in der Speicherdatei, wird hier gleich der Spieler in die Liste gesetzt, da der Spielerindex hier ein unnötiger Umweg wäre. Den aktuellen Spieler speichern wir jedoch als Index, wie in der Speicherdatei, ebenso die Spieler in ein Array, so können wir nämlich den nächsten Spieler leichter setzen. Ob ein Spiel endet, wird von der Spielklasse überprüft, indem die Differenz aus gesamten Spielern und Spielern, welche im Ziel sind, gebildet wird.

Der Stapel an Hasenkarten bekommt eine eigene Klasse, welche dann die Funktionalitäten, wie das Ziehen einer Karte oder das Mischen des Stapels beinhaltet, um die Spiel Klasse nicht zu überfüllen. Die Hasenkartenstapel Klasse muss dann nur in der Spiel Klasse erzeugt und gespeichert werden. Die verschiedenen Hasenkarten werden als enum gespeichert. Die Aktionen, die eine Hasenkarte bewirken und Aktionen, die durch ein Feld ausgelöst werden, werden alle durch ein ActionBuilder erzeugt, um einige Elemente mehrfachverwenden zu können. Die `Aktion` an sich ist nur ein Interface, welches den Spiel Kontext, die Verbindung zur GUI, ein Runnable, was nach der Aktion ausgeführt wird, und den Spieler, für den diese Aktion gilt, bekommt. Das Spiel Kontext Interface ist hier wieder besonders nützlich, denn Aktionen wie „Rücke eine Position vor“, müssen beispielsweise wissen, wo der Spieler mit dem besseren Rang steht. Durch Verwendung dieses Interfaces verhindern wir hier wieder mögliche Seiteneffekte. Wir geben also manchmal mehr Informationen als nötig ein, jedoch ohne einen Schaden damit anzurichten, erhalten im Gegensatz dafür aber eine einfache und dynamische Struktur, welche Zukünftige Änderungen gut unterstützt.

Der ActionBuilder erzeugt aus den gegebenen Informationen nun verschiedene `Aktionen`. So wird er in Klassen der Feldtypen und in der Spiel Klasse verwendet, um die nötigen Aktionen zu erzeugen. Die ganzen Informationen bekommt der ActionBuilder erst beim Ausführen, dann passiert auch erst die eigentliche Erstellung der Aktion. So können wir verhindern, dass die Aktionen mit alten Informationen erzeugt werden. Im Spiel Kontext Interface gibt es eine Ausnahme, welche keine pure function ist: die Kartenziehen Aktion. Ein Spieler kann potenziell mehrere Karten in einer Runde ziehen (siehe Benutzerhandbuch), damit wir in jeder Aktion jetzt nicht immer die gleiche Karte bekommen, müssen wir sie vom Stapel ziehen. Es wäre möglich, dass das Spiel Kontext Interface nur pure functions hat und das Kartenziehen Problem außerhalb zu lösen, aber der Einfachheit wegen, machen wir hier fürs erste eine Ausnahme, denn das Ziehen von Karten passiert momentan nur auf Hasenfeldern.

In `Aktionen` werden auch die meisten Befehle zum Loggen von Nachrichten versendet. Der Logger ist statisch, aufgeteilt in zwei Klassen: Der LoggerCore, welcher die Core Funktionen zum Erstellen und Schreiben des Logs bereitstellt, der LogMessenger, welcher eine genaue Implementation der Nachrichten, welche geloggt werden können, beinhaltet. Der LoggerCore ist nur für den LogMessenger zugänglich. Der LogMessenger soll so in Zukunft durch ein Interface ersetzt werden können, um beispielsweise unterschiedliche Sprachen oder ausführlichere Log Nachrichten zu ermöglichen, ohne

dass die Basis neugeschrieben werden muss. Das Speichern eines Spiels funktioniert über die Gson-Bibliothek, da diese in der Aufgabenstellung empfohlen wurde. Da die JSON-Dateien, die daraus erzeugt werden, einem vorgegebenen Format folgen müssen, verwenden wir zwei (De-)Serialisier Klassen: GameData und PlayerData, dadurch sind wir flexibler in der Benennung und Verwendung von Datenstrukturen im Code, ohne dass wir die JSON-Daten verändern.

In der Spieler- und Spielklasse stößt man manchmal auf das Keyword transient in Attributen, dies wird verwendet, um spezielle Attribute nicht zu serialisieren, dies ist jedoch nur notwendig, falls die beiden (De-)Serialisier Klassen nicht verwendet werden können, also ist dies eher zum Debuggen gedacht. Das Laden funktioniert, indem wir die GameData in den Konstruktor der Spielklasse geben. So wird für jedes geladene Spiel immer ein neues erzeugt. Da wir keinen Game Manager oder ähnliches haben, muss dies in der GUI passieren. Durch das Neuerzeugen, anstelle des Veränderns eines Spieles, können wir mehr Felder, wie das Spieler-Array immutable machen, so ist der Code weniger Fehleranfällig.

Als letztes noch ein Hinweis zum Callback-System, es wurde geschaffen für die Kommunikation zwischen der Logik und der GUI unter Verwendung des GUIConnectors, man trifft aber auch an anderen Stellen, welche nicht direkt mit der GUI kommunizieren auf das System. Dies wurde gemacht, um Konsistenz im Code zu gewährleisten und die Lesbarkeit zu verbessern. Man sollte unbedingt mit Lambda Ausdrücken und funktionalen Interfaces vertraut sein, um den Code richtig verstehen zu können. Außerdem schadet es nicht mit Konzepten der funktionellen Programmierung vertraut zu sein, da der Code aus Konzepten der funktionellen Programmierung (Bsp. Callback-System) und der Objekt Orientierten Programmierung (Bsp. Vererbung bei den Feldtypen) zusammengestellt ist.

## 2.2.2 GUI-DARSTELLUNG DES SPIELMENÜS

### PROBLEMANALYSE

Das Spielmenü muss, als Menüleiste am Spielbrett angezeigt werden. Es müssen vor dem Start eines Spiels, die Anzahl der Spieler eingestellt werden. Jeder Spieler muss benannt werden können, aber es muss auch möglich sein das Spiel mit Default Name zu starten. Im Menü soll es auch möglich sein ein laufendes Spiel zu speichern oder ein gespeichertes Spiel zu laden. Wird ein Spiel gestartet muss es möglich sein, während des Spiels ein neues zu erstellen. In einem laufenden Spiel muss dem Benutzer auch angezeigt werden, welcher Spieler mit Namen gerade dran ist und wie viele Karotten und Salate dieser hat.

### REALISATIONSANALYSE

Wo die Menüleiste angezeigt wird, ob nun rechts, links oder unter dem Spielbrett ist nur eine Frage der Preference. Das Menü ist ein idealer Fall für eine FXML-Datei, da es nicht wirklich dynamische Inhalte mit einer Ausnahme beinhaltet. Ein Spinner kann verwendet werden, um die Spielerzahl einzustellen, alternativ könnte man auch ein normales Eingabefeld verwenden, müsste dann aber die Eingabe prüfen. Die Spielernamen sollte man als Text-Felder anzeigen. Dies ist das einzig dynamische Verhalten, welches abhängig von der Spieleranzahl ist. Man könnte zwar 6 Textfelder in die FXML-Datei einbauen und sie bei Bedarf verstecken, aber dann ist man auch in Zukunft auf 6 Spieler begrenzt. Erzeugt man diese Felder prozedural hat man dieses Problem nicht. Als Default Namen ist es am einfachsten die Prompt-Texte zu überschreiben und wenn der normale Text leer ist, diesen zu verwenden. Das Starten und Neustarten eines Spiels, ist ein Anwendungsfall für ein Button. Da Starten und Neustarten nie gleichzeitig existiert können wir hier denselben Button verwenden und ändern das Verhalten nach Bedarf. Das Speichern und Laden lässt sich auch durch Buttons implementieren. Das Dateiauswahl Fenster, wurde durch die Aufgabenstellung schon fast komplett gegeben. Die Informationen über einen bestimmten Spieler, während eines laufenden Spiels, lassen



sich durch Labels darstellen, das ist nicht visuell schönste Variante, aber für die Aufgabenstellung erfüllt es den Zweck.

## REALISATIONSBESCHREIBUNG

Das Menü wird als FXML-Datei implementiert. Das Design lässt sich so einfach verändern, ohne in den Code einzugreifen. Für die Anzeige des aktuellen Spielers mit seiner Karotten- und Salatanzahl verwenden wir Label, für die Einstellung der Spieleranzahl einen Spinner, für das Erstellen, Speichern und Laden eines Spiels verwenden wir Buttons. Die Spielernamen erstellen wir prozedural als Textfelder direkt unter der Spieleranzahl gereiht. Der Prompt-Text stellt dabei die Default-Namen da. Beim Starten eines Spiels erstellen wir eine neue Schnittstelle mit der Logik und ein neues Spiel. Beim Laden und Speichern lassen wir den Benutzer frei auswählen, wo er die Datei speichern kann und welche Dateien geladen werden. Dies ermöglicht es den Benutzer eigene Spielstände zu schreiben oder bestehende zu verändern. Geladene Dateien prüfen wir erst auf Syntax und Logik, ob die Daten auch ein funktionierendes Spiel darstellen können, bevor wir sie als neues Spiel verwenden. Erfüllt eine Datei die Bedingungen nicht wird der Benutzer darüber benachrichtigt und der Zustand vor dem Laden bleibt erhalten. Beim Neuerstellen oder Beenden eines laufenden Spiels, wird geprüft, ob der Spieler gespeichert hat, wenn nicht, bekommt er die Option nochmal vorher zu speichern. Bricht der Spieler das Speichern hierbei ab, bleibt der alte Zustand bestehen und es wird nicht beendet oder ein neues Spiel erstellt, damit der Spieler nicht versehentlich seinen Spielstand verliert. Dies prüfen wir in dem wir die zuletzt gespeicherte Datei speichern und überprüfen, ob diese noch den aktuellen Spielstand repräsentiert. Beim Laden verzichten wir auf diesen zusätzlichen Dialog, da sich beim Testen gezeigt hat, dass es an dieser Stelle mehr stört als hilft. Das Menü wird stilisiert über eine CSS-Datei, welche wir in die FXML-Datei einbinden, wodurch wir den Stil jederzeit einfach austauschen können.

### 2.2.3 GUI-DARSTELLUNG DES SPIELBRETTES

#### PROBLEMANALYSE

Das Design des Spielbretts präsentiert sich in Form eines einzelnen statischen Bildes, welches 63 klickbare Felder umfassen soll. Die Felder müssen so gestaltet sein, dass sie die Form und Anordnung der abgebildeten Felder präzise nachbilden, um eine genaue Interaktion mit der Maus zu ermöglichen. Wenn der Mauszeiger über ein Feld schwebt, sollte eine Umrandung angezeigt werden: grün, wenn das Betreten des Feldes durch die Spiellogik erlaubt ist und somit eine Interaktion möglich sein soll, und rot, wenn das Betreten nicht gestattet ist, wobei das Feld in diesem Fall nicht anklickbar sein darf. Zusätzlich muss ein separates, klickbares Zielfeld vorhanden sein, welches diese Umrandung nicht hat.

Für das Spiel müssen zudem zwischen zwei und sechs Spielsteine generiert werden können, die über das Brett bewegt werden. Es ist wichtig, dass diese Bewegungen durch Animationen dargestellt werden können, wofür die genauen Feld-, Start- und Endpunkte bekannt sein müssen. Außerdem muss der momentan ausgewählte Spielstein bestimmt werden, dieser bekommt dann eine farbige Umrandung. Zudem muss das Spielbrett so gestaltet werden, dass es bei einer Vergrößerung des Fensters entsprechend skaliert wird. Die Proportionen müssen dabei aber nicht unbedingt einbehalten werden.

Darüber hinaus muss der Spieler über bestimmte Ereignisse im Spiel informiert werden, was manchmal die Auswahl zwischen drei Optionen beinhaltet. Außerdem muss die GUI flexibel genug sein, um während eines laufenden Spiels, das Spiel neu zu starten oder ein neues laden zu können, wobei es erforderlich ist, dass alle Spielsteine gelöscht oder ihre Positionen verändert werden können. Sollte das Spiel mit einer veränderten Anzahl von Spielern gestartet werden, müssen zusätzliche Spielsteine hinzugefügt oder bestehende verschwinden. Es ist auch essenziell, dass keine Nachrichten oder Feldumrandungen aus vorherigen Spielen mehr angezeigt werden, um keine inkorrekten Zustände anzuzeigen.



## REALISATIONSANALYSE

Die Darstellung des Spielbretts kann am effektivsten durch ein ImageView mit einem vordefinierten Bild erreicht werden. Die Feldmittelpunkte sind bereits in einer JSON-Datei gegeben. Die Start- und Endpunkte lassen sich einfach zu dieser hinzufügen. Der Point2D-Datentyp in JavaFX, der häufig für grafische Elemente verwendet wird, ermöglicht es uns, die Punkte aus der JSON-Datei einfach als Point2D-Objekte zu laden. Änderungen an den Feldpunkten erfordert somit nur Modifikationen in der JSON-Datei, nicht im Code, was das Laden unterschiedlicher Spielbrett-Layouts in Zukunft ermöglichen würde.

Die klickbaren Spielfelder könnten entweder durch Buttons oder Rectangles umgesetzt werden, denen dann ein MouseEvent zugewiesen wird. In beiden Fällen ist es notwendig, MouseEvents hinzuzufügen, damit beim Überfahren eines Feldes mit der Maus eine rote oder grüne Umrandung erscheint. Diese lässt sich einfach durch das Färben des Stroke-Property realisieren. Bei Verwendung von Buttons müsste deren Funktionalität deaktiviert werden, wenn ein Feld nicht anklickbar sein soll. Die Skalierbarkeit sowohl für Buttons als auch für Rectangles, wird in beiden Fällen durch das Gleiche Verfahren erreicht werden.

Die Spielsteine, dargestellt durch JPEG-Bilder, könnten ebenfalls als ImageViews realisiert werden oder alternativ als Circles oder Rectangles, mit hohem Arc-Wert (wirkt wie ein Kreis), mit einem ImagePattern-Fill. Die Umrandung dieser Steine kann erneut durch die Färbung des Stroke-Property erzeugt werden. Für die Animation der Spielsteine wäre die Verwendung einer TranslateTransition die einfachste Option. Allerdings könnte dies Probleme verursachen, wenn viele Properties der Spielsteine an das Spielbrett gebunden sind, dann funktioniert die Funktion der TranslateTransition, welche die Objekte bewegt, nicht mehr. Das Binden verschiedener Properties an das Spielbrett kann jedoch erforderlich sein, um Skalierbarkeit zu ermöglichen. Eine andere Möglichkeit wäre die Erstellung eines eigenen Animators, der eine Timeline und die Zwischenpunkte, die wir durch die Feldmittelpunkte kennen, verwendet. Dies würde eine genauere Anpassung des Animationsverhaltens erlauben und sicherstellen, dass die Animation auch während des Veränderns der Fenstergröße reibungslos funktioniert.

Informationen über Spielereignisse könnten entweder direkt auf dem Spielfeld oder über ein weiteres Fenster wie ein Alert angezeigt werden. Ein Alert hätte den Vorteil, dass sich während seiner Anzeige das Fenster nicht verändern lässt und somit keine Skalierbarkeit erforderlich ist. Zudem wird sichergestellt, dass der Spieler die Nachricht tatsächlich wahrnimmt. Bei der Auswahl zwischen drei Optionen könnte der Alert so modifiziert werden, dass er drei Buttons anzeigt, wobei jeder Button ein anderes Event auslöst. Aus einer Perspektive des Game Designs wäre jedoch das Öffnen eines zweiten Fensters für das Spiel eine unpraktische und benutzerunfreundliche Lösung.

Das generelle Design des Spielbretts, einschließlich seines ImageViews, könnte in einer FXML-Datei gespeichert werden, zusammen mit anderen Elementen wie dem Spielmenü. Die Spielfelder sollten jedoch dynamisch erzeugt werden, um den Vorteil der JSON-basierten Anpassungsfähigkeit für unterschiedliche Spielbrett-Layouts zu erhalten. Die Spielsteine könnten in der FXML erzeugt und dann dynamisch bewegt werden, jedoch erscheint dieser Ansatz auf den ersten Blick nicht intuitiv. Da für gewöhnlich nur die Grundlegende Struktur der Benutzeroberfläche, nicht aber dynamische Objekte in FXML-Dateien gespeichert werden. Da während eines Spiels ein neues gestartet oder ein bestehendes geladen werden kann, muss sichergestellt werden, dass die Anzahl der Spielsteine korrekt ist und keine Nachrichten aus vergangenen Spielen mehr angezeigt werden. Der letzte Punkt ist besonders wichtig, da wegen des Callback-Systems in der Logik, die Nachrichten noch angezeigt werden könnten, obwohl in der GUI ein neues Spiel und ein neuer GUIConnector verwendet wird, da das alte Spiel nicht gestoppt wird. *(Anmerkung: Warum wird das Spiel nicht schon in der Logik gestoppt? Es wird einfacher sein das Callback-System in der GUI zu stoppen, da der Logik Ablauf sequenziell ist, wird dies keine Probleme verursachen.)*

## REALISATIONSBESCHREIBUNG

Das Spielbrett ist als ImageView in einer FXML-Datei gespeichert, welche bei Programmstart geladen wird. Alle Feldmittelpunkte sind in einer einzigen JSON-Datei gespeichert. Durch eine DataProvider-Klasse können alle nötigen Ressourcen, welche die GUI betreffen, also Spielbrett, Spielsteine und Feldpunkte, geladen werden. Es ist dadurch sehr einfach andere Layouts oder Designs zu verwenden. Die Feldpunkte werden als Point2D-Objekte in drei verschiedene Arrays geladen: Anzahl Spieler viele für den Startpunkt, Anzahl Felder viele für die Felder zwischen Start und Ziel, Anzahl Spieler viele für den Zielpunkt. Da wir für die klickbaren Felder Rectangle verwenden, welche auch mit Point2D arbeiten, können wir die Feldpunkte als Mittelpunkte der Rectangle setzen.

Die Rectangle Füllfarbe ist transparent, die Ecken sind so weit abgerundet, dass die Form perfekt zu den Feldern auf dem Spielbrett-Bild passen. Die Umrandungsfarbe lassen wir beim Erstellen erstmal auf transparent. Die Positions-Attribute und alle anderen Attribute, die wir verändert haben, binden wir zum Spielbrett ImageView. So stellen wir sicher das die Felder immer denen auf dem Spielbrett-Bild nachbilden, auch wenn das Bild skaliert wird. Zum Erstellen dieser speziellen Rectangle verwenden wir eine RectangleFactory als Helfer, um diese Schritte möglichst einfach für jedes beliebige Rectangle durchzuführen.

Die erzeugten Felder speichern wir in einem Array in dem UserInterfaceController. Wir geben den Array dann in die JavaFXGUI. So können wir für jede neue JavaFXGUI die gleichen Felder verwenden, da sich ihre Anzahl nie ändern wird. Damit diese Felder anklickbar werden, fügen wir MouseEvents zu diesen. Das wird in der JavaFXGUI immer dann gemacht, wenn die Logik der GUI befiehlt, Felder auswählbar zu machen. Wird ein Feld schließlich ausgewählt, entfernen wir die Mouseevents wieder. In den Mouseevents können wir auch die jeweilige Farbe für die Feldumrandungen setzen. Ein Quality of Life Feature welches hier noch eingefügt wurde, ein Preview für die Karotten eines Spielers anzuzeigen. Dies wird realisiert, indem wir neben den betretbaren Feldern auch die jeweiligen Karottenkosten, welche von der Logik berechnet wurden in die Methode hinzugeben, Das anzeigen des Previews wird dann wieder über das Mouse Event gemacht.

Für die Spielsteine verwenden wir Rectangle, da wir eine RectangleFactory haben, welche sich bereits um Sachen wie Skalierbarkeit kümmert. Die abgerundeten Ecken werden so weit erhöht, dass sich ein Kreis bildet. Wir verwenden ein Imagefill um die Bilder der Spielsteine da drin anzuzeigen. Für die Spielsteine verwenden wir noch eine extra Klasse PlayerVisual, in welcher wir die Funktionen zum Auswählen und Bewegen eines Spielers implementieren. Da wir die Spielsteine durchs Binden ihrer Attribute ans Spielbrett nur noch bewegen können, indem wir die Attribute neubinden, stellen wir durch die PlayerVisual-Klasse sicher, dass wir die Spielsteine ordnungsgemäß bewegen.

Da TranslateTransition die Spielsteine leider nicht so bewegt und dementsprechend nicht mehr funktioniert, müssen wir einen eigenen Animator verwenden. Dieser ist auch in der PlayerVisual-Klasse implementiert. Wir müssen nur die Feldmittelpunkte als Array, wie eine Art Strecke, übergeben und können mit einer Timeline und Key Frames die Spielsteine über diese Strecke bewegen in dem wir zwischen ihnen interpolieren, unter Verwendung der diesmal richtigen Bewegungsmethode, verursachen wir auch so keine Probleme.

Ein weiterer Vorteil ist, dass wir flexibler in der Animation sind, erstens funktioniert die Animation so auch problemlos beim Skalieren des Fensters und zweitens können wir die Geschwindigkeit der Animation genauer bestimmen. Je länger die Strecke ist, die ein Spielstein zurückzulegen hat, desto schneller bewegt sich der Spielstein mit der Zeit. Dies dient der Benutzererfahrung, damit der Benutzer nicht ewig auf die Animationen warten muss, da es im Spiel möglich ist sehr große Strecken zurückzulegen.

Die PlayerVisual werden in einem Array in der JavaFXGUI gespeichert, da wir diese zum gleichen Zeitpunkt erzeugen und löschen müssen. Der ausgewählte Spieler wird nochmal einzeln als Attribut gespeichert. Die Logik kann über den Spielerindex dann befehlen den jeweiligen Spieler auszuwählen. Die Logik befiehlt der GUI die Spielsteine zu bewegen, dafür übergeben wir das Start Feld, von wo der Spieler bewegt werden soll und das End Feld wohin der Spieler, welcher gerade als ausgewählter

Spieler gespeichert ist, bewegt werden soll, als Index des Feldes. Das Startfeld geben wir rein, um erkennen zu können, ob der Spielstein, an der gleichen Position ist wie, der Spieler in der Logik, ist dies nicht der Fall wird man den Spieler in der Animation springen sehen können und weiß das GUI und Logik nicht mehr synchronisiert sind, es dient also hauptsächlich zum Debuggen.

Die übergebenen Feld Indices setzen wir in den vom DataProvider gestellten Arrays der Feldmittelpunkte ein, um die Felder in eine Strecke von Point2D umwandeln zu können, ein. Das Startfeld funktioniert ähnlich wie eine Warteschlange. Der Spieler ganz rechts auf dem Startfeld bewegt zuerst, der Rest der Spieler rückt auf und wird ein Spieler hinzu gefügt wird er ans Ende angefügt. Mehr dazu in Algorithmen. Im Zielfeld werden die Spieler jedoch von links nach rechts aufgefüllt, also der am weitesten linke Spieler war der erste, welcher das Ziel erreicht hat.

Die Spielerbenachrichtigungen werden wie in der Aufgabenstellung empfohlen über Alerts realisiert, da dies die einfachste Lösung ist, welche die wenigsten Probleme verursacht. Dies ist zwar etwas benutzerunfreundlich, aber ist für die Aufgabenstellung völlig ausreichend. Da viele der Alerts sehr ähnlich sind und sich meistens nur die angezeigte Nachricht ändert, verwenden wir hierfür eine Klasse GameAlert. Diese kümmert sich darum die Alert alle an der richtigen Position, nämlich in der Mitte des Spielbretts anzuzeigen und die drei unterschiedlichen Arten von Spielerbenachrichtigungen bereitzustellen: der normale Alert, welche für Spielereignisinformationen zuständig ist, der Hasenkarten Alert, welche anzeigt welche Karte ein Spieler gezogen hat und der Entscheidungsalert, bei welchen wir drei verschiedene Optionen bereitstellen, welche alle etwas anderes bewirken können. Diese GameAlert-Klasse wird im UserInterfaceController erstellt, damit wir sie in neuerzeugten JavaFXGUI wiederverwenden können.

Die JavaFXGUI implementiert dann die durch den GUIConnector benötigten spezifischen Benachrichtigungen. Um Artefakte, wie plötzlich auftauchende Spieler oder Alerts aus einer alten JavaFXGUI, aufgetreten wegen des Callback-Systems zu vermeiden, zerstören wir die JavaFXGUI durch eine Methode, bevor wir ein neues Erzeugen. In dieser Methode werden alle Felder, welche noch mögliche Maus Events haben, überschrieben und alle Spielsteine gelöscht. Außerdem wird nachdem bewegen des Spielers gecheckt, ob das Objekt noch aktiv ist, über ein boolean Attribut, welches beim Zerstören auf false gesetzt wird, so das nach einer Animation kein Alert mehr ausgegeben wird, wenn das Objekt nicht mehr aktiv ist. Ohne diesen Check würden wir in neuen Spielen möglicherweise plötzlich alte Alerts angezeigt bekommen.

## 2.3 ALGORITHMEN

### CALLBACK-SYSTEM

Ein Callback-System ermöglicht es einem Programm, eine Methode (den Callback) zu übergeben, die zu einem späteren Zeitpunkt oder bei einem bestimmten Ereignis aufgerufen werden soll. In „Hase und Igel“ wird dies durch das Übergeben eines Runnable oder OnFieldSelected-Interfaces erreicht. Callbacks sind nützlich für asynchrone Operationen, wie das Warten auf Benutzereingaben, da sie es einem Programm erlauben, fortzufahren, während auf das Ereignis gewartet wird, und dann die spezifizierte Callback-Methode auszuführen, wenn das Ereignis eintritt. Da das Programm sequenziell aufgebaut ist, wird dieser Vorteil nicht genutzt. Das Ausführen einer vorher spezifizierten Methode nach einer Benutzereingabe ist ein Vorteil, welcher jedoch ausgenutzt wird. Dies führt zu einer stärkeren Trennung der Logik und der GUI, da die Logik die Methoden im Voraus festlegt.

### MOVE-METHODE (ANIMATIONSERSTELLUNG)

1. **Parameterprüfung:** Wenn der eingegeben Pfad nur ein Punkt beinhaltet, wird der Spieler direkt auf diesen Punkt gesetzt und die Methode endet mit dem Ausführen eines Callbacks.
2. **Initialisierung:** Die Geschwindigkeit der Bewegung wird gesetzt und eine Sequenz wird erstellt, welche die Timeline-Objekte speichern wird.
3. **Berechnung und Erstellung der Animationen:** Eine Schleife durchläuft jedes Paar aufeinanderfolgender Punkte im Pfad. Für jedes Paar berechnet sie die Distanz und die Dauer, die verwendet wird, um diese Distanz mit der gegebenen Geschwindigkeit zurückzulegen. Die Dauer ist abhängig vom Index des Ziels, wodurch spätere Bewegungen schneller werden. Für die Berechnung wird hier die Quadratwurzel verwendet, so werden Bewegungen zwischen den Paaren immer schneller, nehmen aber immer weniger an Geschwindigkeit zu, damit sie nicht zu schnell werden.
4. **Erstellung der Keyframes für die Timeline:** Für jedes Paar von Punkten wird eine Timeline erstellt. Die beiden Punkte eines Paares sind jeweils ein Keyframe. Für sanftere Übergänge interpolieren wir zwischen den beiden Keyframes, mehrmals und erstellen aus den berechneten Punkten dann Zwischenframes. Die Zwischenframes werden mit den Keyframes sequenziell in die Timeline eingefügt. In den Frames wird die Position des Spielers geändert.
5. **Hinzufügen der Timeline zur Sequenz:** Jede Timeline wird zur Sequenz hinzugefügt. Diese Sequenz wird dann abgespielt, um eine fließende Bewegung zu erreichen.

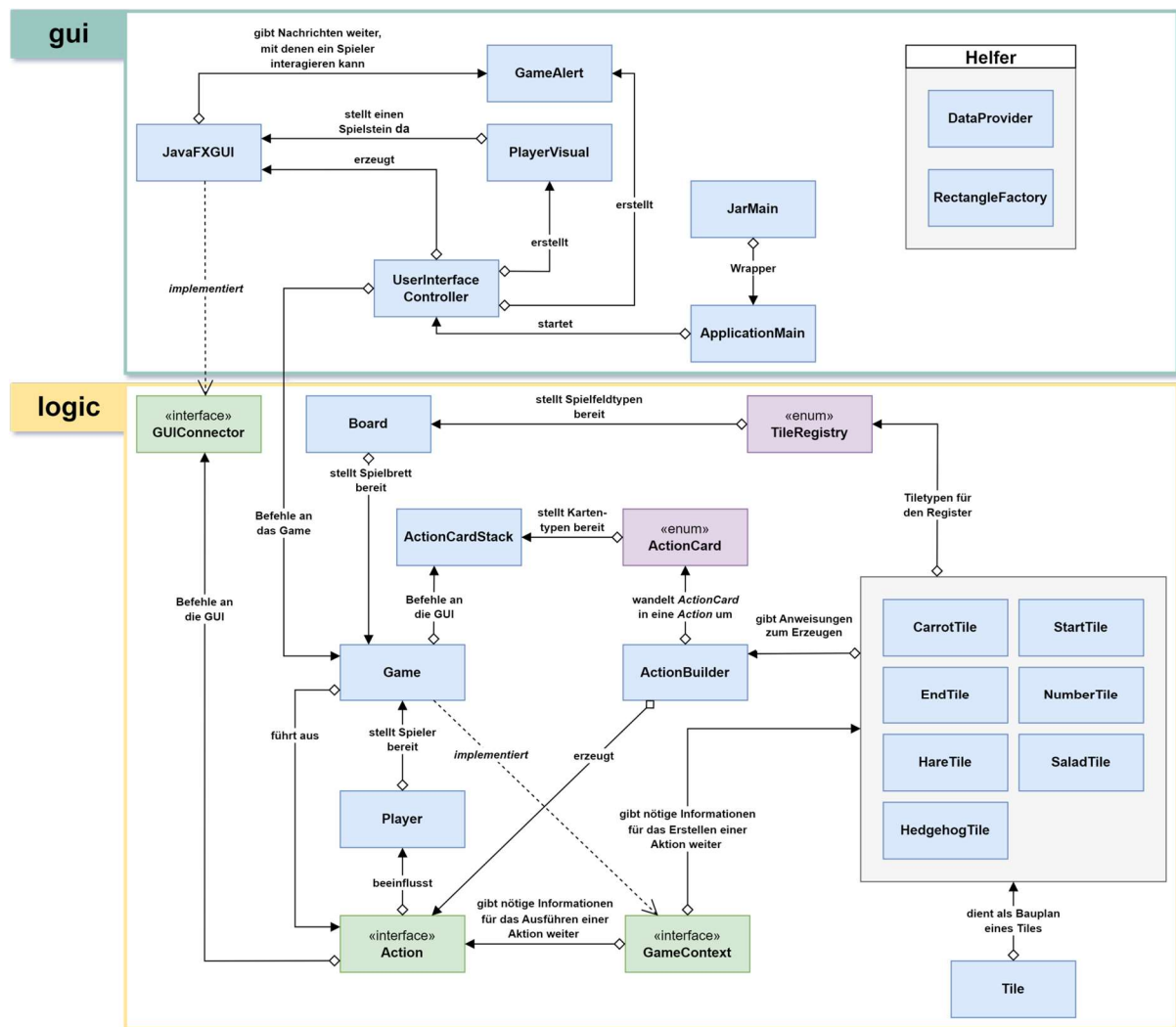
### KAROTTENKOSTEN FÜR EINE DISTANZ BERECHNEN

Die Kosten für die Bewegung auf dem Spielbrett steigen linear mit jedem Schritt an. Die Gesamtkosten können also über die Formel  $\frac{n \times (n+1)}{2}$  berechnet werden, wobei `n` hier der Distanz entspricht. Zum Beispiel, wenn die Distanz 3 ist (d.h., der Spieler bewegt sich 3 Felder vorwärts), dann sind die Gesamtkosten  $\frac{3 \times (3+1)}{2} = \frac{3 \times 4}{2} = \frac{12}{2} = 6$  Karotten.

### MAXIMALE DISTANZ MIT ANZAHL VON KAROTTEN BERECHNEN:

Die Kostenformel ist eine quadratische Gleichung:  $\frac{n \times (n+1)}{2} = k$ . Um `n` zu finden, multiplizieren wir beide Seiten mit 2 und erhalten:  $n^2 + n = 2 \times k$ . Das ist eine quadratische Gleichung der Form:  $n^2 + n - 2k = 0$ . Nun kann die quadratische Lösungsformel angewendet werden, um `n` zu berechnen:  $n = \frac{-1 \pm \sqrt{1+8k}}{2}$ . Da `n` nicht negativ sein kann (es repräsentiert eine Distanz), verwenden wir nur die positive Wurzel:  $n = \frac{\sqrt{1+8k}-1}{2}$ . Zum Beispiel, wenn der Spieler 10 Karotten hat, dann ist die maximale Distanz, die er zurücklegen kann:  $n = \frac{\sqrt{1+8 \times 10}-1}{2} = \frac{\sqrt{81}-1}{2} = \frac{9-1}{2} = \frac{8}{2} = 4$  Felder.

## 2.4 PROGRAMMORGANISATIONSPLAN



Die Grundlegenden Funktionen der einzelnen Klassen sollte in dem Kapitel Problemanalyse und Realisation klargeworden sein (Siehe Problemanalyse und Realisation). Hier werden jetzt die Beziehungen zwischen den Klassen genauer erläutert. Das Interface `OnFieldSelected` und die beiden Serializer-Klassen `GameData` und `PlayerData` sind in der Grafik nicht zu finden, da diese unwichtig in Hinblick auf die Programmorganisation sind und die Grafik nur unnötig füllen würden. Außerdem wird hier nicht jede Verbindung zwischen den Klassen gezeigt, da einige Verbindungen nicht von Relevanz sind, um den Programmaufbau zu verstehen.

Die `Game`-Klasse bildet hier den Kern der Logik und stellt alle nötigen Informationen für die anderen Logik-Klassen durch das `GameContext`-Interface bereit. Es führt außerdem alle Aktionen aus und gibt so an die Aktionen auch die nötigen Informationen weiter, wie beispielsweise auf welcher GUI die Aktionen angezeigt werden sollen. Dies passiert erst beim Ausführen, um die Informationen immer aktuell zu halten. Es werden also nirgendwo in der Logik Informationen immer weitermitgezogen, sondern die Aktionen können immer aktuell über den `GameContext` abgerufen werden. Die `Tile`-Klasse ist eine abstrakte Klasse und stellt den Aufbau für alle Feldtypen-Klassen bereit. Diese Feldtypen-Klassen verwenden dann den `ActionBuilder` um die jeweiligen Aktionen beim Betreten und Starten auf dem Feld für ihren Feldtypen zu erstellen.

Die `Game` Klasse fragt dann die `Board`-Klasse, welche Aktion auf einem bestimmten Feld ausgeführt werden soll, dabei fragt die `Board` Klasse dann immer den jeweiligen Feldtypen nach der Aktion. Die `Game` Klasse bekommt so die passende Aktion und füllt diese beim Ausführen dann mit den konkreten Spieldaten. Die Aktionen geben über den `GUIConnector` dann Befehle was angezeigt werden soll. Das

Game kann jedoch auch ohne eine Aktion auszuführen dem GUIConnector Befehle erteilen. Der GUIConnector wird konkret von der JavaFXGUI implementiert, welche genau bestimmt, wie diese Befehle ausgeführt werden sollen und wurde der Befehl zu Ende ausgeführt wird ein `Callback` aufgerufen. Der UserInterfaceController ist eine Art übergeordneter Spielemanager, er erstellt also neu Spiele und verwaltet das aktuelle Spiel. Er stellt also die Funktionen des Menüs dar. Die DataProvider und RectangleFactory Klassen sind zwar nur Helferklassen, sind aber essenziell für das Laden der Spielressourcen und das Erstellen dynamischer Grafikelemente, wie die Spielsteine und werden primär von dem UserInterfaceController genutzt, wenn beispielsweise neu Spielsteine erstellt werden müssen.

## 2.5 DATEIEN

### Log-Datei:

Im Log wird der Spielverlauf festgehalten. Gestartet wird ein Log mit einem Zeitstempel und einem Seed, welcher die zufälligen Elemente im Spiel bestimmt. Dadurch kann jedes Spiel manuell mithilfe der Log-Datei exakt repliziert werden und Fehler können reproduziert werden. Aktionen, die ein Spieler ausführt, sind mit einem Stern gekennzeichnet. Neben diesen Informationen werden jedoch auch einige Zustandswechsel und andere relevante Informationen festgehalten, welche zum Debuggen helfen können. Mit einem Minus wird ein Zustandswechsel verzeichnet, beispielsweise ob der Spieler einen Salat isst. Ein Ausrufezeichen wird verwendet, um aufzuzeigen, dass es eine andere wichtige Information ist, wie beispielsweise das Ende des.

### Test-Spielstände:

Für einige Programmtests in 2.6 ist das Laden von Spielständen nötig. Diese sind im Projekt in einem Unterordner der Tests zu finden, welcher Spielstand zu welchen Programmtest gehört, wird in den Programmtests selbst erläutert.

## 2.6 PROGRAMMTTESTS

Testfall	Ergebnis
Klicken auf größer/kleiner Pfeile unter der Spieleranzahl. Versuchen die Spieleranzahl über das Maximum/unter das Minimum zu stellen.	Die Pfeile sollten die Zahl in dem Spieleranzahlfeld verändern und neue Felder unter Spielernamen erstellen, welche bereits Default-Namen beinhalten sollten. Es sollte nicht möglich sein die Spieleranzahl über sechs zu erhöhen oder unter zwei zu verringern.
Klicken auf den Speichern-Button, bevor das Spiel gestartet wurde.	Der Speichern-Button sollte ausgegraut sein und beim Anklicken sollte nichts passieren.
Klicken auf den Laden-Button bevor das Spiel gestartet wurde und Abbrechen im folgenden Fenster drücken.	Beim Klicken des Laden-Buttons sollte sich ein Datei-Auswahl-Fenster öffnen. Beim Abbrechen sollte sich das Fenster wieder schließen und keine Fehlermeldung angezeigt werden.
Klicken auf den Laden-Button, bevor das Spiel gestartet wurde und öffnen einer validen und einer fehlerhaften Spielstands Datei. Die valide Datei kann erzeugt werden durch das Speichern eines Spiels über den Speichern-Button und die fehlerhafte kann erzeugt werden durch Löschen des Inhalts der Datei. Dabei können auch folgende Dateien getestet werden: „Invalid_Test“, „Invalid_No_Json_Test“, „Invalid_Game_Test“	Beim Öffnen der validen Datei, sollte ein Spiel starten und beim Öffnen der fehlerhaften Datei sollte ein Fenster sich öffnen mit dem Text: „Laden fehlgeschlagen! ...“. Beim Drücken von „OK“ in diesem Fenster, sollte die Fehlermeldung verschwinden.
Klicken auf den Start-Button mit ausgewählter Spieleranzahl von zwei Spielern und dem ersten Spielernamen mit „Spieler 1“ benannt im Spielernamenfeld.	Das Spielbrett sollte in Farbe erscheinen mit zwei Spielsteinen und dem rechten von Beiden mit einer Umrandung. Die Spieleranzahl und Spielernamen Einstellung sollte verschwunden sein, stattdessen sollte oben der Spielername des ersten Spielers „Spieler 1“ und seine Karotten und Salate angezeigt werden. Der Speichern-Button sollte nicht mehr ausgegraut sein und der Start-Button sollte jetzt den Namen „Neu“ tragen.
Folgend nachdem letzten Test den Spieler auf das erste Feld bewegen, dann über den Speichern-Button den Zustand speichern. Danach den zweiten Spieler auf das zweite Feld bewegen und den gespeicherten Zustand über den Laden-Button laden.	Das Spiel sollte nun in den Zustand zurückkehren, in welchem sich nur der erste Spieler auf Feld 1 bewegt hat.
Folgend nachdem letzten Test den zweiten Spieler wieder auf das zweite Feld bewegen und danach den Neu-Button drücken. Darauf dann Abbrechen auswählen.	Es sollte ein Fenster auftauchen mit dem Text „Das Spiel wurde seit dem letzten Speichern verändert!“. Beim Abbrechen sollte das alte Spiel normal weiterlaufen.
Das gleiche wie beim vorherigen Test nur auf „Spiel erstellen“ drücken.	Hier sollte man zurück im Menü landen, welches das Aussehen haben sollte wie in den ersten Tests.
Das gleiche wie beim vorherigen Test nur auf „Speichern und Spiel erstellen“ drücken und dann bei der Dateiauswahl abbrechen.	Das alte Spiel sollte normal weiterlaufen, wenn das Speichern abgebrochen wird, um zu verhindern, dass der Benutzer aus Versehen nicht speichert.



Testfall	Ergebnis
Ein Spiel starten über den Starten-Button und dann auf den Neu-Button drücken.	Es sollte ein Fenster auftauchen mit dem Text „Das Spiel wurde noch nicht gespeichert!“.
Das gleiche wie beim vorherigen Test nur die Datei speichern anstelle bei der Dateiauswahl abzubrechen.	Das alte Spiel sollte gespeichert werden und man sollte zurück im Menü landen, welches das Aussehen haben sollte wie in den ersten Tests.
Ein Spiel starten, dann speichern und dann auf den Neu-Button drücken.	Man sollte sofort im Menü landen, welches das Aussehen haben sollte wie in den ersten Tests.
Die Tests für den Neu-Button replizieren für den Beenden-Button	Das Ergebnis sollte dem vom Neu-Button entsprechend, der Text und das Verhalten danach können jedoch variieren.
Bewegen eines Spielers auf ein Feld.	Von der Position des Spielers sollte eine Animation zum ausgewählten Feld angezeigt werden. Der Spielstein bewegt sich dabei von einem Feld zum Nächsten. Bewegt der Spieler sich über eine größere Distanz sollte die Animation über den Weg schneller werden.
Bewegen eines Spielers vom Startfeld weg auf ein Hasenfeld. Dabei sollen jedoch noch andere Spieler auf dem Startfeld stehen.	Der Spieler sollte sich von ganz rechts auf dem Startfeld auf die ausgewählte Position bewegen. Die anderen Spieler auf dem Startfeld sollten dabei nacheinander aufrücken. Danach erst soll das Fenster für die Hasenkarte sichtbar werden.
Bewegen eines Spielers auf ein Feld. Das Fenster soll dabei während der Bewegung vergrößert/verkleinert und verschoben werden.	Die Bewegungsanimation sollte sich dabei entsprechend das Fensters anpassen und währenddessen auch weiterlaufen.
Bewegen eines Spielers auf ein Hasenfeld. Noch bevor die Animation abgeschlossen ist, soll ein neues Spiel gestartet werden.	Im neugestarteten Spiel darf kein Spielstein aus dem alten Spiel erscheinen. Außerdem sollte nicht plötzlich eine Hasenkarte oder ein anderes Fenster auftauchen.
Ein Spieler soll auf ein Karottenfeld bewegt werden und bis zu Beginn seines Zuges gespielt werden. Im Auswahl-Fenster soll versucht werden das Fenster durch die Schließen Schaltfläche zu schließen, ohne vorher etwas ausgewählt zu haben.	Beim Drücken auf die Schließen-Schaltfläche sollte nichts passieren und das Fenster sollte geöffnet bleiben, bis eine der drei Wahlen getroffen wurde
Folgend dem vorherigen Test sollte nun jedoch jede Auswahl einmal ausprobiert werden.	„10 nehmen“: Die Karottenanzeige sollte in der nächsten Runde um zehn Karotten grösser sein. „10 geben“: Die Karottenanzeige sollte in der nächsten Runde um zehn Karotten weniger sein. „weiterziehen“: Ein Feld sollte wieder ausgewählt werden können.
Laden der Datei „CarrotField_ZeroCarrots_Test“ und bewegen des Spielers auf das Karottenfeld. Danach Bewegen des zweiten Spielers auf das Igel-Feld vor diesem. Im Auswahl-Fenster auf Weiterziehen drücken.	Im Auswahl-Fenster sollte die Wahl „10 geben“ wegfallen. Beim Weiterziehen sollte der Spieler benachrichtigt werden, dass er zum Start zurückziehen muss, weil er keine Karotten mehr hat und auf kein Feld ziehen kann. Dies ist das gewünschte Verhalten, da es die Situation repräsentiert, dass der Spieler keine Karotten und freie Felder mehr hat. Diese Situation ist in den Spielregeln definiert.
Laden der Datei „End_Test“ und bewegen beider Spieler ins Ziel.	Es sollte ein Spiel geladen werden, in welchen beide Spieler sofort ins Ziel ziehen können. Nachdem beide Spieler im Ziel angekommen sind, sollte eine Nachricht erscheinen. Diese sollte den ersten Spieler als Gewinner nennen.