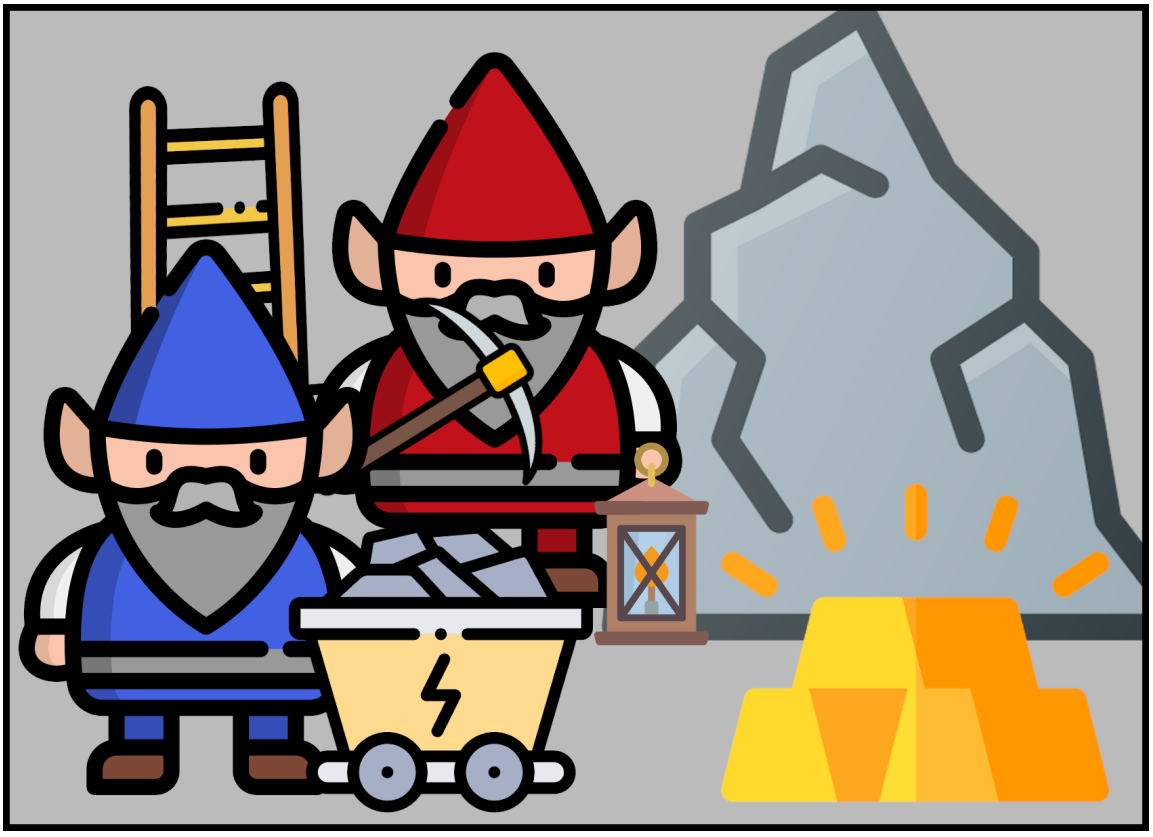
Funktionale und objektorientierte Programmierkonzepte
Wintersemester 20/21

Abgabedatum: 19.03.2021
Gesamtpunktzahl: 60

Projekt - v1.0

Saboteur



Hauptautor: Dominik Beese

Gesamtleitung: Prof. Karsten Weihe

Inhaltsverzeichnis

1 Organisatorische Informationen

1.1 Grundlegende Informationen und Bonus

Das FOP-Projekt ist zwar nicht verpflichtend und auch für die formale Prüfungszulassung (Studienleistung) **nicht** notwendig, aber mit dem FOP-Projekt können zusätzliche Bonuspunkte erreicht werden. Es wird jedoch dringend empfohlen am FOP-Projekt teilzunehmen, da Sie hier richtig programmieren lernen. In den weiterführenden Semestern wird dies mehr oder weniger stillschweigend vorausgesetzt. Voraussetzung für den Bonus ist die bestandene Studienleistung.

Das Projekt soll in Gruppen von vier Personen bearbeitet werden. In Ausnahmefällen ist auch eine Bearbeitung mit weniger Personen möglich.

1.2 Zeitplan im Überblick

- 05.03.2021 23:50 Uhr: Deadline zur Anmeldung
- 01.03.2021 - 19.03.2021: Sprechstunden
- 19.03.2021 23:50 Uhr: Abgabe des Projekts

1.3 Sprechstunden

Für eventuell anfallende Fragen bieten die Projekttutoren vom 1. bis zum 19. März Sprechstunden an. Eine Übersicht über die Sprechstundentermine finden Sie im Projektabschnitt des **moodle**-Kurses.

1.4 moodle-Forum

Es werden für Sie zwei Foren im Projektabschnitt des **moodle**-Kurses bereitgestellt. In einem Forum können Sie Fragen hinsichtlich der Organisation und des Zeitplans stellen. Das andere Forum ist für Fragen rund um die Projektaufgaben gedacht. Alle Fragen in den Foren werden natürlich auch außerhalb der Sprechstunden beantwortet.

1.5 Anmeldung (bis 05.03.)

Um sich anzumelden, bilden Sie Gruppen von vier Studierenden. In Ausnahmefällen ist auch eine Bearbeitung mit weniger Personen möglich. Sollten Sie noch keine Gruppe haben, steht Ihnen im Projektabschnitt des **moodle**-Kurses ein Forum zur Gruppenfindung zur Verfügung.

Alle Gruppenmitglieder müssen sich **manuell** in die Gruppen eintragen.

1.6 Abgabe des Projekts

Das Projekt ist bis zum 19. März um 23:50 Uhr Serverzeit auf **moodle** abzugeben. Das Projekt exportieren Sie genauso wie die Hausübungsabgaben als ZIP-Archiv, hier gelten die gleichen Konventionen. Die Benennung erfolgt folgendermaßen: `Projektgruppe_XXX`,

wobei der Suffix `XXX` durch Ihre Gruppennummer zu ersetzen ist. Dabei gibt eine Person aus Ihrem Team das komplette Projekt ab.

Neben dem Code geben Sie zusätzlich eine PDF-Datei ab, diese soll sich ebenfalls in dem abzugebenden ZIP-Archiv befinden. In der PDF-Datei finden sich die für unser Verständnis notwendigen Dokumentationen von Aufgabe 4.3.2 und Aufgabe 4.3.3.

1.7 Plagiarismus

Selbstverständlich gelten die gleichen Regelungen zum Plagiarismus wie auch bei den Hausübungsabgaben!

Insbesondere hier nochmal der Hinweis, dass Ihr gemeinsames Repository für die Gruppenarbeit privat sein sollte.

Beachten Sie: Sollte Ihre Dokumentation der Lösungswege unvollständig sein oder uns Anlass für den Verdacht geben, dass Sie nicht nur innerhalb der eigenen Gruppe gearbeitet haben, müssen Sie damit rechnen, dass Sie Ihre Ergebnisse bei einem privaten Testat bei Prof. Weihe persönlich vorstellen und erläutern müssen.

1.8 Inhaltlicher Überblick

Im Laufe des Projekts werden Sie eine Implementation des Kartenspiels „Saboteur“ erarbeiten und diese ergänzen. Dazu wird Ihnen eine Vorlage zur Verfügung gestellt, in der Sie, je nach Aufgabenstellung, Code ergänzen oder erweitern müssen, damit eine lauffähige Version des Spiels entsteht. Unter anderem werden Sie viele bekannte Themen aus der Vorlesung behandeln und anwenden müssen, sodass das Projekt eine gute Vorbereitung auf die Klausur darstellt.

Selbstverständlich wünschen wir Ihnen viel Spaß bei der Implementierung des Spiels und freuen uns auf gelungene Implementationen.

2 Spielkonzept

Im Kartenspiel „Saboteur“ schlüpfen die Spieler in die Rolle von Goldsuchern und Saboteuren. Dabei versuchen die Goldsucher auf schnellstem Wege zum Goldschatz zu gelangen während die Saboteure das Ziel haben, genau das zu verhindern. Den Spielern ist dabei nicht bekannt, welche Rolle die Mitspieler innehaben, sodass die Saboteure versuchen, unentdeckt zu bleiben und im Verborgenen zu sabotieren. Der Weg zu den Zielkarten, wovon nur eine den Goldschatz darstellt, wird mithilfe von sogenannten Wegekarten gelegt. Durch Aktionskarten können Karten aus dem Wegelabyrinth entfernt, Spieler gesperrt oder entsperrt werden. Mit der Schatzkarte ist sogar ein Blick unter eine Zielkarte erlaubt. Wer am Ende des Spiels sein Ziel erreicht hat, wird mit Punkten in Form von Goldstücken belohnt.

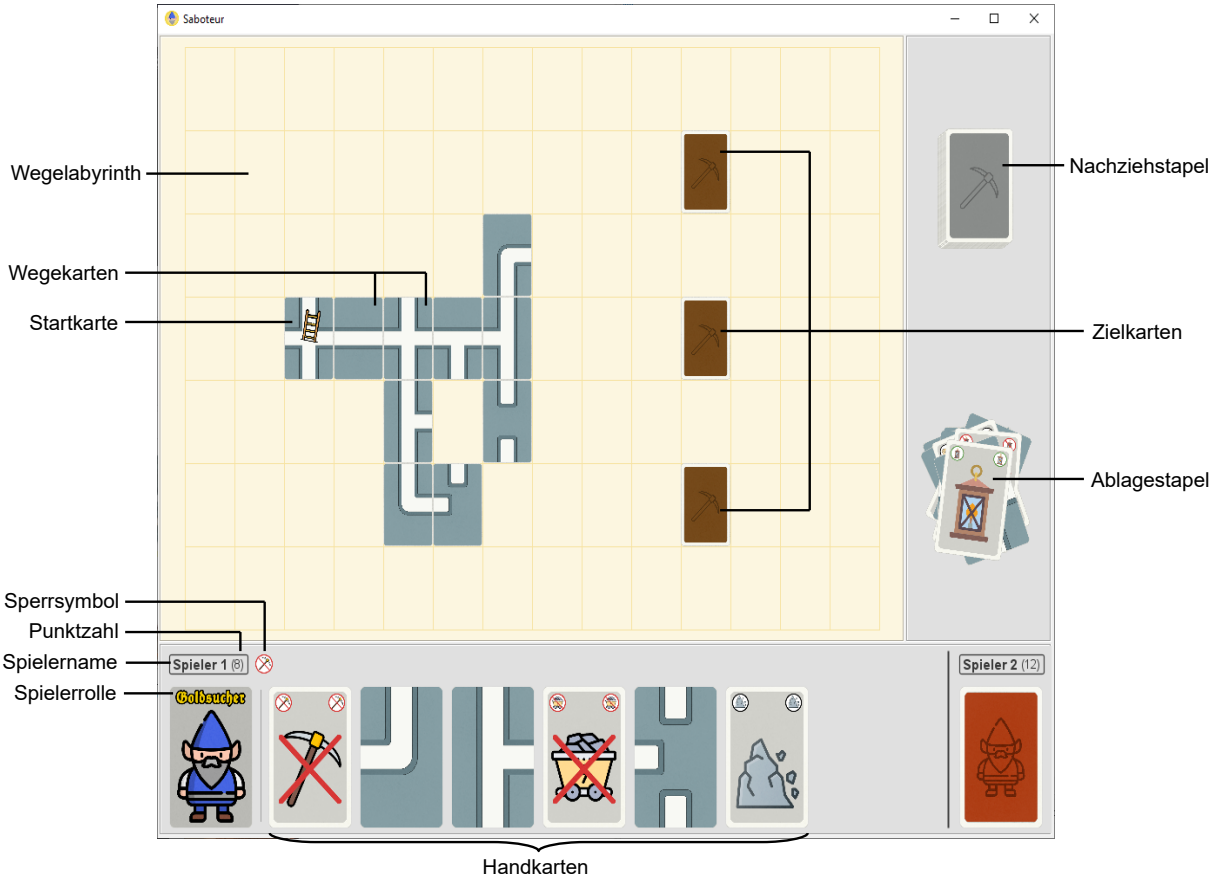


Abbildung 1: Übersicht über die Benutzeroberfläche des Spiels

2.1 Spielvorbereitung

Zu Beginn des Spiels wird jedem Spieler eine zufällige Charakterkarte - Goldsucher oder Saboteur - zugeteilt. Die Anzahl von Goldsuchern und Saboteuren richtet sich nach der Spieleranzahl und ist so gewählt, dass es meist zwei bis vier Saboteure weniger gibt als Goldsucher. Bei zwei Spielern gibt es genau einen Goldsucher und einen Saboteur. Im Einspielermodus versucht der Spieler als Goldsucher eine möglichst hohe Punktzahl zu erreichen.

Neben den Charakterkarten besteht das Spiel aus zwei Arten von Karten: Aktionskarten und Wegekarten. Diese werden gemischt und als Nachziehstapel bereitgestellt. Jeder Spieler erhält zu Beginn je nach Spieleranzahl vier bis sechs Handkarten. Neben dem Nachziehstapel entsteht der Ablagestapel, auf den benutzte Karten gelegt werden.

In das Wegelabyrinth werden die Startkarte sowie die drei Zielkarten gelegt. Dabei sind genau sieben Karten zwischen Start- und Zielkarten sowie jeweils eine Karte zwischen den Zielkarten.

2.2 Spielablauf

Ein Zug eines Spielers besteht aus einer Aktion und dem Nachziehen einer Karte. Im Folgenden werden die einzelnen Aktionen näher erläutert.

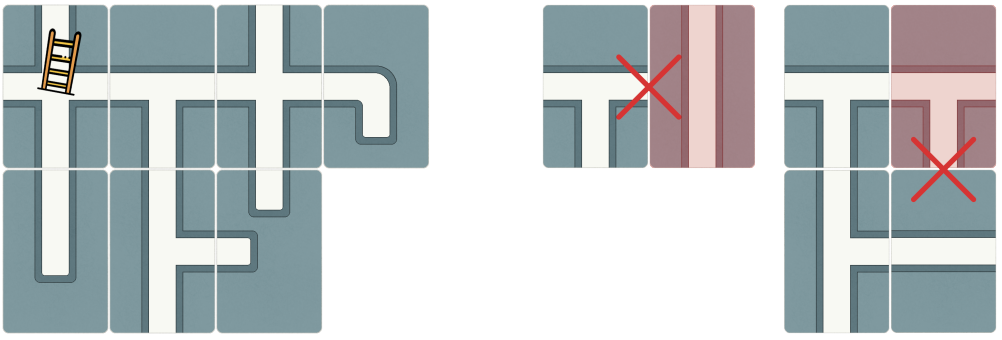
2.2.1 Eine Wegekarte ausspielen

Mithilfe von Wegekarten versuchen die Goldsucher einen Weg zu der Zielkarte mit dem Goldschatz zu bauen. Die verschiedenen Wegekarten sind in Abbildung 2 zu sehen.

(a) (b) (c) (d)

Abbildung 2: Arten von Wegekarten: (a) vier gewöhnliche Wegekarten, (b) eine verdeckte Zielkarte, (c) die Zielkarte mit dem Goldschatz und (d) eine Zielkarte mit einem Stein.

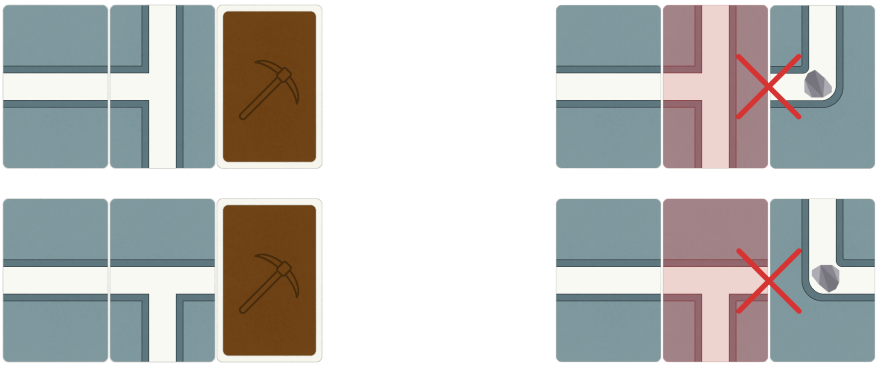
Eine Wegekarte darf immer nur so an bereits im Wegelabyrinth liegende Wegekarten angelegt werden, dass ein ununterbrochener Weg von der Startkarte zu der neu angelegten Karte besteht. Dabei müssen die Wege an allen Seiten der Karten genau zueinander passen. Beispiele hierfür sind in Abbildung 3 zu finden. Die rot hinterlegten Karten dürfen nicht an die gezeigten Positionen gelegt werden, da sie zur jeweils benachbarten Karte nicht passen. Die problematischen Stellen sind mit einem roten X markiert.



(a) Erlaubte Legepositionen (b) Falsche Legepositionen

Abbildung 3: Beispiele für normale Legepositionen

Wegekarten können solange beliebig neben eine Zielkarte gelegt werden, wie die Zielkarte verdeckt liegt. Wenn die Zielkarte umgedreht wurde, wird sie wie eine gewöhnliche Wegekarte behandelt und es gelten dieselben Regeln für Legepositionen. Einige Beispiele hierfür zeigt Abbildung 4. Wenn dadurch ein unterbrochener Weg von der Start- zur Zielkarte entstanden ist, wird die Zielkarte umgedreht. Falls es sich um den Goldschatz handelt, ist das Spiel beendet.



(a) Erlaubte Legepositionen (b) Falsche Legepositionen

Abbildung 4: Beispiele für Legepositionen an einer Zielkarte

2.2.2 Eine Wegekarte zerstören

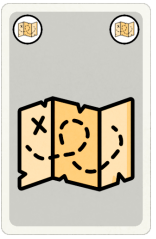
Die Karte mit dem Steinschlag kann benutzt werden, um eine beliebige Wegekarte im Wegelabyrinth mit Ausnahme der Start- und Zielkarten zu zerstören. Danach liegt die zerstörte Karte nicht mehr im Wegelabyrinth, sondern wird zusammen mit der Steinschlagkarte auf den Ablagestapel gelegt.

2.2.3 Werkzeug zerstören oder reparieren

Eine Sperrkarte, die ein zerbrochenes Werkzeug zeigt, kann vor einen Mitspieler gelegt werden, um den Spieler zu sperren. Gesperrte Spieler dürfen keine Wegekarten mehr in das Wegelabyrinth legen. Alle restlichen Aktionen sind weiterhin erlaubt. Ein Spieler darf nicht gleichzeitig mehrfach mit dem gleichen Werkzeug gesperrt werden.

Spieler, die ein Sperrkarte vor sich liegen haben, können mit einer Reparaturkarte, die ein oder mehrere reparierte Werkzeuge zeigt, wieder entsperrt werden. Auch Karten, die verschiedene Werkzeuge reparieren können, reparieren immer nur ein Werkzeug gleichzeitig. Welches Werkzeug repariert wird, entscheidet der Spieler, der diese Karte ausspielt. Nach dem Ausspielen der Karte werden das zerbrochene Werkzeug und die Reparaturkarte auf den Ablagestapel gelegt.

2.2.4 Die Schatzkarte benutzen



Der Spieler, der die Schatzkarte ausspielt, darf sich eine der noch verdeckten Zielkarten aussuchen und darunter schauen. Anschließend legt er die Zielkarte wieder verdeckt an ihren ursprünglichen Platz und die Schatzkarte auf den Ablagestapel.

2.2.5 Eine Karte abwerfen

Wenn ein Spieler keine der vorherigen Aktionen ausführen kann oder möchte, muss er eine seiner Handkarten auswählen und auf den Ablagestapel legen.

2.3 Spielende

Das Spiel ist beendet, wenn eines der folgenden Szenarien eintritt:

- Es wurde ein ununterbrochener Weg von der Startkarte zur Zielkarte mit dem Goldschatz gelegt. In diesem Fall haben die Goldsucher ihr Ziel erreicht und gewonnen.
- Der Nachziehstapel ist leer und kein Spieler hat noch Karten auf der Hand. Wenn das passiert, haben die Saboteure gewonnen.

2.4 Punkte

Während des Spiels können die Spieler durch folgende Aktionen Punkte erwerben:

- `n`+1 Punkte für jede ins Wegelabyrinth gelegte Wegekarte (Abschnitt 2.2.1) mit `n` benachbarten Karten. Es sind also jeweils 2 bis 5 Punkte möglich.
- 2 Punkte für jede eingesetzte Steinschlagkarte (Abschnitt 2.2.2)
- 2 Punkte für jede eingesetzte Sperrkarte (Abschnitt 2.2.3)
- 2 Punkte für jede eingesetzte Reparaturkarte (Abschnitt 2.2.3)
- 20 Punkte für jeden Sieger (Abschnitt 2.3)

3 Codevorlage

Die Packagestruktur und Implementation des Projekts orientiert sich am **Model-View-Controller** (MVC) Entwurfsmuster, das in Kapitel 14 behandelt wird. Dabei werden im Package `fop.model` die grundlegenden Elemente des Spiels wie Wegekarten, Aktionskarten und das Spielfeld implementiert. Im Package `fop.view` werden diese Komponenten benutzt und in Form einer grafischen Benutzeroberfläche dargestellt. Die Interaktion zwischen Model und View geschieht über den Controller im Package `fop.controller`. Dieser verwaltet alle Komponenten, informiert die View über eventuelle Änderungen und gibt Daten des Models an die View weiter.

Die Ordnerstrutktur der Vorlage ist so aufgebaut, dass der Ordner `src`, aufgeteilt in `src/main` für das Programm selbst und `src/test` für die Unittests, alle Quellcodedateien beinhaltet. Die Ressourcen wie Bilder und Konfigurationsdateien sind im Ordner `res` enthalten, der ebenfalls in `res/main` und `res/test` unterteilt ist.

3.1 View

Die View besteht aus zwei Abschnitten: dem Menü und dem Spiel. Abbildung 1 zeigt die Spieloberfläche. Die dazugehörigen Java-Klassen sind im Package `fop.view.game` zu finden. Das Package `fop.view.menu` beinhaltet die Java-Klassen für das Menü. Eine Übersicht über die Struktur und die Klassen der View zeigt Abbildung 5.

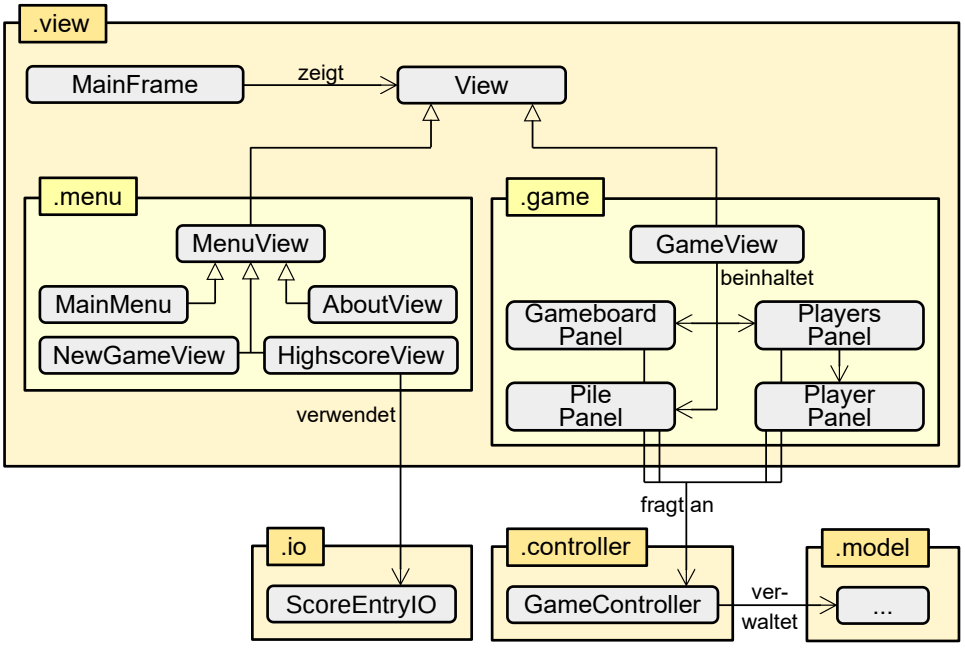


Abbildung 5: Vereinfachtes UML-Diagramm des Package `fop.view` und wichtige Beziehungen zwischen den Packages

3.2 Model

Das Model besteht aus mehreren Komponenten. Dazu gehören der Graph (Abschnitt 3.2.1), mit dem das Wegelabyrinth realisiert wird, die Spieler (Abschnitt 3.2.2), Wegekarten (Abschnitt 3.2.3), das Wegelabyrinth (Abschnitt 3.2.4) und Aktionskarten (Abschnitt 3.2.5). Die wichtigsten Bestandteile dieser Komponenten werden im Folgenden erläutert.

3.2.1 Graph

Der für dieses Spiel verwendete Graph ist ein ungerichteter und ungewichteter Graph. Realisiert wird dieser mithilfe einer Adjazenzliste[1] durch die generische Klasse `Graph<V>` im Package `model.graph`. Um sich den Graphen während der Entwicklung anzeigen zu lassen, können Sie die Methode `printGraph()` verwenden. Der auf der Konsole ausgegebenen Code kann mithilfe von Graphviz[2] visualisiert werden. Dazu können Sie die Website `http://webgraphviz.com/` verwenden.

3.2.2 Spieler und Computerspieler

Die Klasse `Player` im Package `model` stellt einen Spieler dar. Über die folgenden drei Methoden kann auf die Handkarten des Spielers zugegriffen werden:

```
1 public List<Card> getAllHandCards()
2 public void drawCard(Card card)
3 public void playCard(Card card) throws IllegalArgumentException
```

Diese Methoden werden für die in Abschnitt 2.2 erläuterten Aktionen vom Controller (Abschnitt 3.3) verwendet.

Die Aktionskarten, die ein Spieler vor sich liegen hat, wenn er gesperrt ist, werden mithilfe der folgenden Methoden verwaltet:

```
1 public boolean hasBrokenTool()
2 public boolean hasBrokenTool(ToolType type)
3 public BrokenToolCard getBrokenTool(ToolType type
4 public boolean canToolBeBroken(BrokenToolCard brokenToolCard)
5 public void breakTool(BrokenToolCard brokenToolCard)
6 public boolean canBrokenToolBeFixed(BrokenToolCard brokenToolCard,
     FixedToolCard fixedToolCard)
7 public void fixBrokenTool(BrokenToolCard brokenToolCard, FixedToolCard
     fixedToolCard)
```

Die erste Methode gibt `true` zurück, sobald mindestens ein Werkzeug zerbrochen ist, und klärt damit die grundlegende Frage, ob ein Spieler eine Wegekarte spielen darf oder nicht. Die Methoden in Zeile 2 und 3 fragen nach Sperrkarten eines bestimmten Werkzeugtyps - Laterne, Lore oder Spitzhacke (`ToolType`). Mit `canToolBeBroken` kann abgefragt werden, ob eine Sperrkarte (`BrokenToolCard`) auf den Spieler angewandt werden kann. Mit `canBrokenToolBeFixed` wird abgefragt, ob eine Reparaturkarte (`FixedToolCard`) die

[1] `https://de.wikipedia.org/wiki/Adjazenzliste`
[2] `https://graphviz.org/`

übergebene Sperrkarte des Spielers reparieren kann. Die Methoden in Zeile 5 und 7 werden genutzt, um die Sperr- und Reparaturaktionen auszuführen. Eine genaue Beschreibung der einzelnen Methoden ist in der Codevorlage als Javadoc gegeben.

Ein Computerspieler wird durch die Klasse `ComputerPlayer` realisiert. Dieser erbt von `Player` und ergänzt zusätzlich die Methoden `selectCard(Card card)` und `doAction()`. Innerhalb der zweiten Methode wird die Aktion, die der Computerspieler macht, durchgeführt. Dazu muss zuerst mithilfe der ersten Methode die zu spielende Karte ausgewählt werden und anschließend eine der in Abschnitt 3.3 gezeigten Methoden des Controllers aufgerufen werden.

3.2.3 Wegekarten

Eine Wegekarte wird durch ein Objekt der Klasse `PathCard` im Package `model.cards` modelliert. Diese beinhaltet einen Graphen (Abschnitt 3.2.1) mit Knotentyp `CardAnchor`. Die Enumeration `CardAnchor` definiert alle möglichen Ankerpunkte einer Wegekarte, an der ein Weg die Karte verlassen und somit eine Verbindung zu anderen Karten herstellen kann. Das sind *left*, *bottom*, *right* und *top*. Der Graph einer Karte wird durch die folgenden beiden Regeln erstellt:

1. Wenn eine Karte einen Weg an einer Seite hat, besitzt der Graph der Karte einen Knoten mit dem entsprechenden Ankerpunkt.
2. Wenn eine Karte einen Weg zwischen zwei Seiten hat, besitzt der Graph der Karte eine Kante zwischen den beiden Ankerpunkten.

Zwei Beispiele hierfür zeigt Abbildung 6.

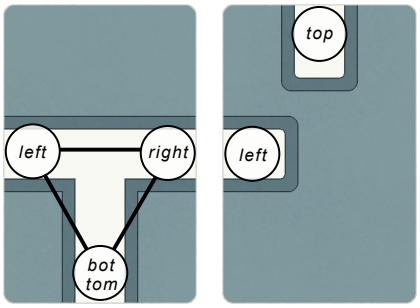


Abbildung 6: Graphen zweier Wegekarten

Zusätzlich implementiert `CardAnchor` zwei sehr wichtige Methoden zum Umgang mit Wegekarten, durch die Pfadberechnungen des Spielfelds vereinfacht werden und es leicht wird, neue Karten hinzuzufügen.

- `public CardAnchor getOppositeAnchor()`
  Diese Methode liefert den gegenüberliegenden Ankerpunkt des aktuellen `CardAnchor`. An eine Karte, die einen Ankerpunkt an der rechten Seite besitzt, kann nur eine Karte mit einem Ankerpunkt an der linken Seite angelegt werden.
  Die Relation von `right` zu `left` wird durch diese Methode modelliert.

```
1 CardAnchor anchor = CardAnchor.right;
2 CardAnchor opposite = anchor.getOppositeAnchor();
3 // opposite == CardAnchor.left
```

- `public Position getAdjacentPosition(Position pos)`
  Diese Methode liefert für eine übergebene Position `(x, y)` die Position der Karte, zu der der `CardAnchor` zeigt. An eine Karte an Position `(4, 0)` mit einem Ankerpunkt an der rechten Seite kann nur eine Karte rechts an Position `(5, 0)` mit einem Ankerpunkt an der linken Seite angelegt werden.
  Die Relation von `(4, 0)` zu `(5, 0)` wird durch diese Methode modelliert.

```
1 CardAnchor anchor = CardAnchor.right;
2 Position position = Position.of(4, 0);
3 Position adjacent = anchor.getAdjacentPosition(position);
4 // adjacent == Position.of(5, 0)
```

Ein Spezialfall der Wegekarten bildet die Zielkarte, die durch die Klasse `GoalCard` im Package `model.cards` dargestellt wird. Diese erbt von `PathCard` und enthält zusätzlich den Typ der Zielkarte - Goldschatz oder Stein - sowie eine Variable, die angibt, ob die Karte verdeckt ist oder nicht. Das kann durch die folgenden Methoden abgefragt werden:

```
1 public GoalCard.Type getType()
2 public boolean isCovered()
```

Definiert werden die Wegekarten in der XML-Datei[3] `pathcards.xml` im Ordner `res/main`, wobei jede Wegekarte durch das folgende Schema beschrieben wird:

```
1 <card name="{string}" count="{int}">
2     <node value="{card_anchor}"/>
3     ...
4     <edge start="{card_anchor}" end="{card_anchor}"/>
5     ...
6 </card>
```

Dabei gibt der Wert von `name` den Namen der Karte an, der für das Laden des korrekten Bildes benötigt wird. Mithilfe von `count` wird festgelegt, wieviele Karten dieser Art im Spiel vorhanden sind. Dadurch können auch unterschiedliche Bilder für Karten mit demselben Graphen benutzt werden: `name_1`, `name_2` etc. Mit den `node`- und `edge`-Tags wird der Graph der Karte beschrieben, wobei die Knoten, die Teil einer Kante sind, nicht noch einmal explizit genannt werden müssen. Die Definitionen für die Wegekarten aus Abbildung 6 sehen wie folgt aus:

```
1 <card name="three_horiz" count="5">
2     <edge start="left" end="bottom"/>
3     <edge start="left" end="right"/>
4     <edge start="bottom" end="right"/>
5 </card>
6 <card name="dead_curve_up">
7     <node value="top"/>
8     <node value="left"/>
9 </card>
```

[3] `https://praxistipps.chip.de/xml-was-ist-das-einfach-erklaert_47836`

3.2.4 Wegelabyrinth

Die Klasse `Gameboard` im Package `model.board` modelliert das Wegelabyrinth und beinhaltet die Attribute `Map<Position, PathCard> board` und `Graph<BoardAnchor> graph`. Dabei speichert `board` alle Wegekarten, die im Wegelabyrinth liegen, und ihre Position. Der `graph` ist ähnlich aufgebaut wie der Graph einer Wegekarte (Abschnitt 3.2.3) und wird zum Bestimmen der Positionen, an die eine Wegekarte angelegt werden kann, benötigt. Auch hier kann der Graph mithilfe der Methode `printGraph()` und `http://webgraphviz.com/` visualisiert werden.

Der Knotentyp des Graphen `BoardAnchor` besteht aus einem `CardAnchor` (Abschnitt 3.2.3) und einer `Position`, die wiederum aus einer `x`- und einer `y`-Koordinate besteht und die Position einer Karte im Wegelabyrinth darstellt. Ein beispielhafter Graph ist in Abbildung 7 zu sehen.

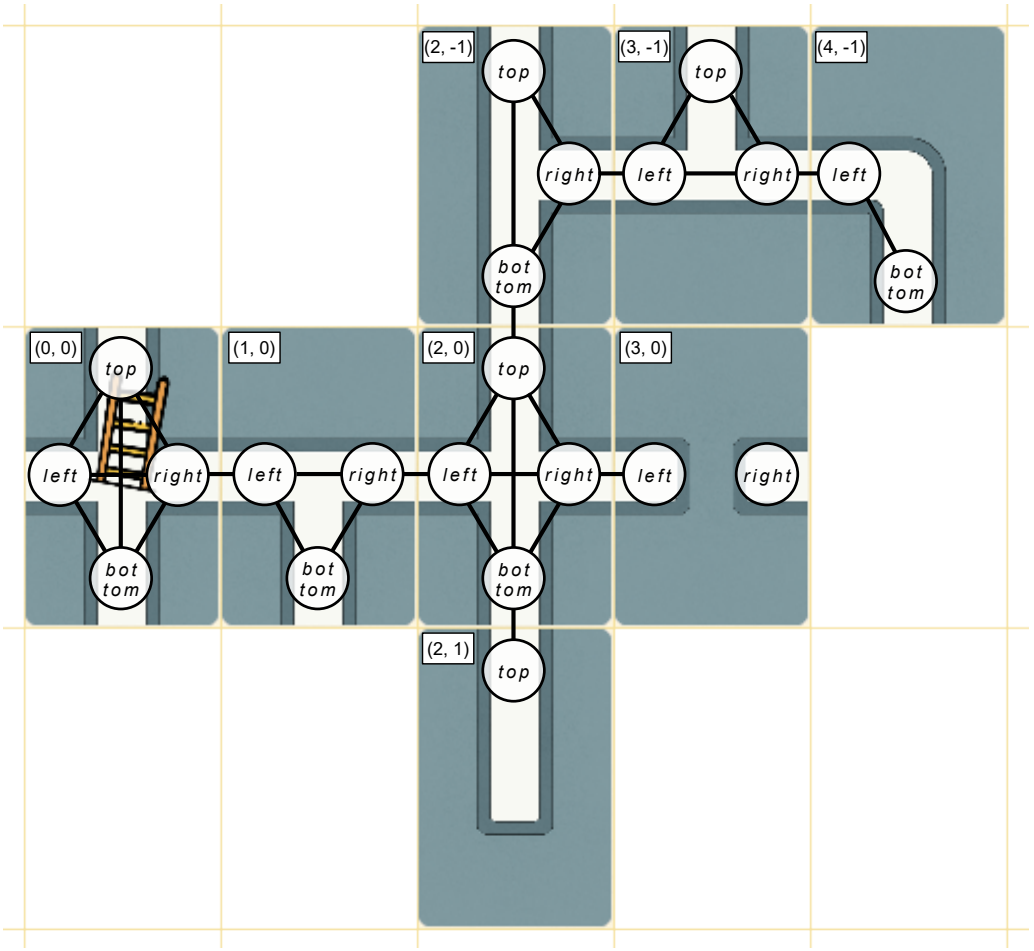


Abbildung 7: Graph des Wegelabyrinths

Die folgenden beiden Methoden werden vom Controller (Abschnitt 3.3) verwendet, um Karten ins Wegelabyrinth zu setzen und zu entfernen:

```
1 public void placeCard(int x, int y, PathCard card)
2 public PathCard removeCard(int x, int y)
```

Die Methode `public boolean canCardBePlacedAt(int x, int y, PathCard pc)` stellt die zentrale Methode der Klasse dar und gibt genau dann `true` zurück, wenn die übergebene Karte an der übergebenen Position platziert werden kann. Dazu müssen die folgenden drei Bedingungen gelten:

1. `private boolean isPositionEmpty(int x, int y)`
   Die Position im Wegelabyrinth ist noch nicht besetzt.
2. `private boolean existsPathFromStartCard(int x, int y)`
   Es gibt mindestens eine Startkarte, von der aus ein ununterbrochener Pfad zur übergebenen Position existiert.
3. `private boolean doesCardMatchItsNeighbors(int x, int y, PathCard pc)`
   Die übergebene Karte passt an der übergebenen Position zu allen ihren Nachbarn. Das heißt, dass alle Ankerpositionen der übergebenen Karte mit den Ankerpositionen der entsprechend benachbarten Karten kompatibel sind.

3.2.5 Aktionskarten

Alle Aktionskarten sind im Package `model.cards` zu finden und erben von der Basisklasse `Card`. Die Vererbungshierarchie ist in Abbildung 8 zu sehen.

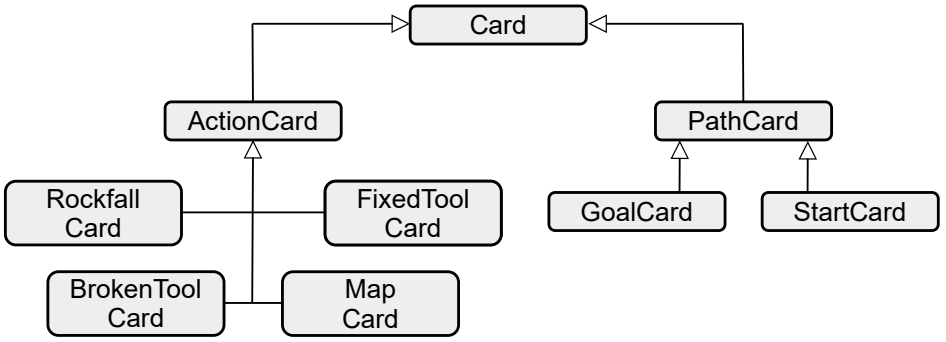


Abbildung 8: Vererbungshierarchie aller Karten des Package `model.cards`

Um zur Laufzeit des Programms entscheiden zu können, um welche Art von Karte es sich handelt, ohne `instanceof` verwenden zu müssen, deklariert und implementiert die Klasse `Card` die folgenden Methoden:

```
1 public boolean isBrokenTool()
2 public boolean isFixedTool()
3 public boolean isMap()
4 public boolean isRockfall()
5 public boolean isPathCard()
6 public boolean isStartCard()
7 public boolean isGoalCard()
```

Diese werden von den erbenden Klassen entsprechend überschrieben, sodass das gewünschte Verhalten erzielt wird.

Im Folgenden wird kurz auf jede Klasse der verschiedenen Aktionskarten eingegangen und das Schema für die `actioncards.xml` Datei erläutert, die analog zu den Wegekarten (Abschnitt 3.2.3) alle Aktionskarten des Spiels beschreibt.

- `BrokenToolCard`
  Eine Sperrkarte erhält in ihrem Konstruktor einen `ToolType`, der angibt, um welches Werkzeug es sich handelt. Abgefragt werden kann dieser über die Methode `public ToolType getToolType()`.
  Das XML-Schema zur Definition einer solchen Karte ist im Folgenden gezeigt:

```
1 <card type="broken_tool" name="{string}" count="{int}">
2     <tool type="{tool_type}"/>
3 </card>
```

- `FixedToolCard`
  Eine Reparaturkarte bekommt in ihrem Konstruktor einen oder mehrere `ToolType` übergeben, die angeben, welche Werkzeuge repariert werden können. Abgefragt werden können diese über die Methoden `public Set<ToolType> getToolTypes()` und `public boolean canFix(ToolType toolType)`.
  Das Schema für die XML-Datei lautet wie folgt:

```
1 <card type="fixed_tool" name="{string}" count="{int}">
2     <tool type="{tool_type}"/>
3     ...
4 </card>
```

- `MapCard`
  Die Karte mit der Schatzkarte wird wie folgt definiert:

```
1 <card type="map" name="{string}" count="{int}"/>
```

- `RockfallCard`
  Zur Definition einer Karte mit dem Steinschlag wird die folgende Zeile benötigt:

```
1 <card type="rockfall" name="{string}" count="{int}"/>
```

3.3 Controller

In der Klasse `GameController` sind die folgenden für das Gameplay benötigten statischen Methoden gegeben:

```
1 public static void selectCard(Card card)
2 public static void placeSelectedCardAt(int x, int y)
3 public static void destroyCardWithSelectedCardAt(int x, int y)
4 public static void fixBrokenToolCardWithSelectedCard(Player player,
      BrokenToolCard brokenToolCard)
5 public static void breakToolWithSelectedCard(Player player)
6 public static void lookAtGoalCardWithSelectedCard(GoalCard goalCard)
7 public static void discardSelectedCard()
```

Dabei wird die erste Methode genutzt, um eine Handkarte des aktiven Spielers auszuwählen, sodass die möglichen Aktionen auf der Benutzeroberfläche markiert werden. Die anderen Methoden stellen die in Abschnitt 2.2 beschriebenen Aktionen dar, wofür die zuvor ausgewählte Karte verwendet wird. Eine genaue Beschreibung der einzelnen Funktionen ist in der Codevorlage als Javadoc zu finden.

3.4 Tipps

Um während der Entwicklung des Spiels einige Dialoge zu deaktivieren, können Sie in der Klasse `view.game.DialogHandler` die Variable `DISABLE_DIALOGS` auf `true` setzen. Dadurch werden die Dialoge *nextPlayer* und *drawCard* nicht mehr angezeigt. Nach der Fertigstellung des Spiels sollte die Variable wieder auf `false` gesetzt werden, um das vollständige Spielerlebnis zu bieten.

Mit dem in der Codevorlage vorhandenen Ant[4]-Skript können Sie ihr fertiges Programm als ausführbare `.jar`-Datei exportieren, sodass Sie das Spiel auch ohne Eclipse starten können. Dazu müssen Sie zuerst in der Menüleiste von Eclipse `Window > Show View > Ant` auswählen und anschließend das Skript in der neuen Ansicht über den Knopf mit dem grünen Plussymbol *Add Buildfiles* hinzufügen. Das Skript befindet sich direkt im Hauptordner des Projekts und heißt `ant.xml`. Nach dem Hinzufügen genügt ein Doppelklick auf das Skript *Saboteur* und die ausführbare Datei, die Sie mit einem Doppelklick starten können, befindet sich im Ordner `dist` des Projekts.

Allgemeiner Hinweis: Das ist **Ihr** Spiel. Solange die Anforderungen der Aufgabenteile eingehalten werden **und** das Grundgerüst des Spiels nicht verändert wird, gilt: *feel free to modify*. Sie können das Spiel beliebig erweitern, verändern und umschreiben. Überraschen Sie uns. Am sinnvollsten halten Sie größere Änderungen aber immer in Ihrer PDF-Datei fest.

[4] `https://ant.apache.org/`

4 Aufgaben

Das Projekt besteht aus zwei verschiedenen Aufgabentypen: Basisaufgaben und weiterführende Aufgaben. In den Basisaufgaben geben wir Ihnen genau vor, welche Funktionalität Sie zu implementieren haben. In den weiterführenden Aufgaben erweitern Sie das Projekt nach Ihren Vorstellungen.

4.1 Graphen und das Labyrinth (33 Punkte)

4.1.1 Knoten und Kanten — 4 Punkte

In dieser Aufgabe sind Methoden zum Hinzufügen und Entfernen von Knoten und Kanten des in Abschnitt 3.2.1 beschrieben Graphen zu ergänzen. Alle Methoden sind in der Klasse `Graph<V>` im Package `model.graph` zu finden.

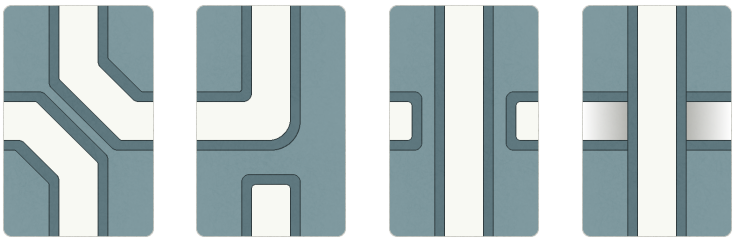
- `public void addVertex(V v)`
  Diese Methode bekommt einen Knoten des Typs `V` übergeben und fügt diesen der Adjazenzliste des Graphen hinzu. Alle Kanten, die mit einem bereits vorhandenen Knoten verbundenen sind, sollen dabei erhalten bleiben.
- `public boolean addEdge(V x, V y)`
  Mit dieser Methode wird eine Kante zwischen den beiden übergebenen Knoten hinzugefügt. Benötigte Knoten werden dabei automatisch hinzufügt. Die Methode gibt genau dann `true` zurück, wenn die Kante hinzugefügt wurde. Beachten Sie, dass es sich bei diesem Graphen um einen ungerichteten Graphen handelt und somit eine Kante von `x` zu `y` auch eine Kante von `y` zu `x` impliziert. Beide Kanten müssen der Adjazenzliste des Graphen hinzugefügt werden.
- `public boolean removeVertex(V v)`
  Beim Aufruf dieser Methode wird der übergebene Knoten und alle mit ihm verbundenen Kanten entfernt. Die Methode gibt genau dann `true` zurück, wenn der Knoten entfernt wurde.
- `public boolean removeEdge(V x, V y)`
  Diese Methode entfernt die Kante zwischen den beiden übergebenen Knoten und gibt genau dann `true` zurück, wenn die Kante entfernt wurde. Beachten Sie auch hier, dass es sich um einen ungerichteten Graphen handelt.

4.1.2 Pfadfinder — 4 Punkte

Die wichtigste Methode der Klasse `Graph<V>` im Package `model.graph` ist die Methode `public boolean hasPath(V x, V y)`. Diese gibt genau dann `true` zurück, wenn ein Pfad vom ersten übergebenen Knoten `x` zum zweiten übergebenen Knoten `y` existiert, also wenn `y` von `x` erreichbar ist. Sind Start- und Zielknoten gleich, soll die Methode ebenfalls `true` zurückgeben. Ihre Aufgabe ist es, diese Methode zu implementieren. Dafür dürfen Sie sowohl einen eigenen Ansatz entwickeln als auch ein etabliertes Verfahren implementieren. Beachten Sie dabei mögliche Sonderfälle.

4.1.3 Neue Wege **4 Punkte**

Dem Spiel können durch Ändern der `pathcards.xml` Datei im Ordner `res/main` neue Wegekarten hinzugefügt werden. Fügen Sie dieser Datei die benötigten Zeilen hinzu, um die in Abbildung 9 gezeigten Wegekarten mit der angegebenen Anzahl zu definieren. Die benötigten Bilder zur grafischen Darstellung sind bereits vorhanden. Damit die Bilder korrekt geladen werden können, müssen die Karten genau wie in der Abbildung beschrieben benannt werden. Alle benötigten Informationen können Sie Abschnitt 3.2.3 entnehmen. Vergessen Sie nicht, das Projekt zu aktualisieren (`F5`), falls Sie die Datei außerhalb von Eclipse bearbeitet haben.



(a) (b) (c) (d)

Abbildung 9: (a) 1x `curve_double_down`, (b) 2x `curve_up_dead_bottom`, (c) 1x `line_vert_dead_two` und (d) 2x `tunnel`

4.1.4 Wegekarte anlegen **5 Punkte**

In den folgenden Aufgaben ergänzen Sie Methoden der Klasse `Gameboard` im Package `model.board`. In Abschnitt 3.2.4 finden Sie Informationen zum Wegelabyrinth sowie den Knotentypen `BoardAnchor` und `CardAnchor`.

Implementieren Sie die Methode `public void placeCard(int x, int y, PathCard card)`. Mit dieser Methode kann die übergebene Wegekarte `card` an der übergebenen Position `(x, y)` in das Wegelabyrinth gelegt werden. Dazu muss die Karte zuerst dem `board` hinzugefügt werden. Danach müssen alle Knoten und Kanten des Graphen der Karte dem Graphen des Wegelabyrinths `graph` hinzugefügt werden. Anschließend müssen noch die Knoten der angelegten Karte mit den Knoten benachbarter Karten verbunden werden.

Verbindliche Anforderung: Greifen Sie zur Bestimmung benachbarter Karten **nicht** direkt auf Konstanten von `CardAnchor` zu, sondern verwenden Sie hierfür die Methoden `getOppositeAnchor()` und `getAdjacentPosition(Position pos)` eines Ankerpunkts.

Hinweis: Lassen Sie den Aufruf der Methode `checkGoalCards()` am Ende stehen. Diese Methode kümmert sich um das Umdrehen der Zielkarten, was in der Codevorlage bereits implementiert ist.

4.1.5 Wegekarte zerstören **1 Punkt**

Implementieren Sie die Methode `public PathCard removeCard(int x, int y)` der Klasse `Gameboard`. Mit dieser Methode wird die Wegekarte, die sich an der übergebenen Position im Wegelabyrinth befindet, aus dem Wegelabyrinth entfernt. Dazu muss die Karte

zuerst aus der `Map board` entfernt werden. Anschließend müssen alle Knoten und Kanten, die zur entfernten Karte gehören, aus dem Graphen des Wegelabyrinths `graph` entfernt werden. Ebenso müssen alle mit der entfernten Karte verbundenen Kanten entfernt werden. Anschließend soll der Graph keine Informationen mehr über die entfernte Karte beinhalten. Der Rückgabewert der Methode ist die Karte, die zuvor an der übergebenen Position lag.

Verbindliche Anforderung: Benutzen Sie auch hier **nicht** explizit die Konstanten von `CardAnchor`. Stattdessen können Sie mittels `CardAnchor.values()` ein Array mit allen möglichen Ankerpunkten erhalten.

4.1.6 Erlaubte Position? Teil 1/3 — 1 Punkt

Eine Karte darf ins Wegelabyrinth gelegt werden, wenn die drei in Abschnitt 3.2.4 beschriebenen Bedingungen erfüllt sind.

Die erste Bedingung lautet, dass auf der übergebenen Position noch keine Wegekarte liegen darf. Implementieren Sie zur Überprüfung dieser Bedingung die Methode `private boolean isPositionEmpty(int x, int y)` in der Klasse `Gameboard`, die genau dann `true` zurückgibt, wenn die Bedingung erfüllt ist.

4.1.7 Erlaubte Position? Teil 2/3 — 8 Punkte

Die zweite Bedingung ist, dass es mindestens eine Startkarte gibt, von der aus ein ununterbrochener Pfad zur übergebenen Position existiert. Implementieren Sie die Methode `private boolean existsPathFromStartCard(int x, int y)`, die das überprüft und genau dann `true` zurückgibt, wenn die Bedingung erfüllt ist. Überlegen Sie, wie Sie sich die Methode `hasPath(V x, V y)`, die Sie in Aufgabe 4.1.2 implementiert haben, zunutze machen können.

Verbindliche Anforderung: Benutzen Sie auch hier falls nötig `getOppositeAnchor()`, `getAdjacentPosition(Position pos)` und `CardAnchor.values()` statt direkt auf Konstanten von `CardAnchor` zuzugreifen.

Hinweis: Sie können weder davon ausgehen, dass sich die Startkarte an einer festgelegten Position befindet, noch, dass die Startkarte Ankerpositionen an allen vier Seiten besitzt.

4.1.8 Erlaubte Position? Teil 3/3 — 6 Punkte

Die dritte Bedingung lautet, dass die übergebene Karte an der übergebenen Position zu allen ihren Nachbarn passt. Das heißt, dass alle Ankerpositionen der übergebenen Karte mit den Ankerpositionen der entsprechend benachbarten Karten kompatibel sein müssen. Beispiele hierfür sind in Abbildung 3, Abbildung 4 und Abbildung 7 zu finden. Implementieren Sie die Methode `private boolean doesCardMatchItsNeighbors(int x, int y, PathCard card)`, die genau dann `true` zurückgibt, wenn diese Bedingung erfüllt ist.

Verbindliche Anforderung: Berechnen Sie **keine** Positionen durch direktes Ändern der x- oder y-Koordinate, wie beispielsweise durch `Position.of(x+1, y)`. Benutzen Sie stattdessen die Methode `getAdjacentPosition(Position pos)`. Greifen Sie ebenso

nicht auf Konstanten von `CardAnchor` zu, sondern verwenden Sie `getOppositeAnchor()` und `CardAnchor.values()`.

Hinweis: Sie können die Position einer verdeckten Zielkarte genauso behandeln wie eine Position ohne Karte. Aber auch hier können Sie **nicht** annehmen, dass sich die Zielkarten an festgelegten Positionen befinden.

4.2 Highscores und Dateien (7 Punkte)

4.2.1 Strings und PrintWriter — 3 Punkte

Die Klasse `ScoreEntry` im Package `model` stellt einen Highscore-Eintrag dar, der aus dem Namen des Spielers, dem Zeitstempel des Spiels und der erzielten Punktzahl besteht. Ihre Aufgabe ist es, die folgenden beiden Methoden zu implementieren, die zum Laden und Speichern eines Eintrags genutzt werden.

- `public static ScoreEntry read(String line)`
  Diese Methode bekommt einen einzeiligen `String` übergeben, der die Bestandteile durch je ein Semikolon getrennt in der folgenden Reihenfolge beinhaltet: den Namen, den Zeitstempel bestehend aus Datum und Uhrzeit sowie die Punktzahl. Ein Beispiel hierfür ist `"Dominik;2020-06-15T19:23:27.818535900;61"`. Die Rückgabe ist ein neues `ScoreEntry` Objekt mit den entsprechenden Werten oder `null` wenn der `String` nicht das korrekte Format besitzt. Zum Einlesen des Zeitstempels können Sie `LocalDateTime.parse(timestamp)` verwenden.
- `public void write(PrintWriter printWriter)`
  Diese Methode schreibt das `ScoreEntry` Objekt als neue Zeile mit dem übergebenen `PrintWriter`. Dabei soll das oben genannte Format benutzt werden. Zum Schreiben einer neuen Zeile können Sie die Methode `println(String x)` des `PrintWriter` verwenden. Zum Umwandeln des Zeitstempels in einen String im korrekten Format können Sie `dateTime.toString()` verwenden.

4.2.2 File Streams — 3 Punkte

Die in der vorherigen Methode erstellen `ScoreEntry` Objekte können mithilfe der Klasse `ScoreEntryIO` im Package `io` in Dateien geschrieben und aus Dateien gelesen werden. Implementieren Sie dazu die folgenden beiden Methoden.

- `public static List<ScoreEntry> loadScoreEntries()`
  Mit dieser Methode werden alle `ScoreEntry` Objekte aus der Datei, deren Pfad in der Variable `PATH` gespeichert ist, gelesen. Der Rückgabewert ist eine Liste aller eingelesenen und gültigen Highscore-Einträge in der Reihenfolge, in der sie in der Datei vorkommen. Wenn die Datei nicht vorhanden ist oder es Probleme beim Einlesen der Datei gibt, soll eine leere Liste zurückgegeben werden. Verwenden Sie zum Umwandeln der einzelnen Zeilen in `ScoreEntry` Objekte die in Aufgabe 4.2.1 implementierte Methode `read(String line)`.
- `public static void writeScoreEntries(List<ScoreEntry> scoreEntries)`
  Diese Methode schreibt die übergebenen `ScoreEntry` Objekte in die Datei, deren Pfad in der Variable `PATH` gespeichert ist. Die Einträge sollen in der Reihenfolge in die Datei geschrieben werden, in der sie in der Liste vorkommen. Verwenden Sie zum

Schreiben eines einzelnen Highscore-Eintrags die in Aufgabe 4.2.1 implementierte Methode `write(PrintWriter printWriter)`.

4.2.3 Sortiertes Einfügen — 1 Punkt

Mithilfe der Methode `public static void addScoreEntry(ScoreEntry scoreEntry)` können der Highscore-Datei neue Einträge hinzugefügt werden. Implementieren Sie diese Methode der Klasse `ScoreEntryIO`. Laden Sie dafür zuerst die aktuellen Einträge durch Aufruf der in Aufgabe 4.2.2 implementierten Methode `loadScoreEntries()`. Fügen Sie der Liste anschließend das übergebene `ScoreEntry` Objekt an der passenden Stelle hinzu, sodass die Liste weiterhin absteigend nach der Punktzahl sortiert ist. Wenn der übergebene Eintrag dieselbe Punktzahl wie ein Eintrag der Datei hat, soll der übergebene Eintrag danach eingefügt werden. Zum Vergleich der Punktzahl zweier Einträge können Sie die Methode `compareTo(ScoreEntry other)` verwenden, die durch das Interface `Comparable<T>` deklariert und durch die Klasse `ScoreEntry` implementiert wird. Speichern Sie zuletzt die geänderte Liste durch Aufrufen der zweiten in Aufgabe 4.2.2 implementierten Methode `writeScoreEntries(List<ScoreEntry> scoreEntries)`.

4.3 Weitergestaltung des Spiels (20 Punkte)

4.3.1 Highscore View — 4 Punkte

In der vorherigen Aufgabe haben Sie sich bereits mit Highscores und wie man diese in einer Datei speichern kann beschäftigt. Damit die Highscores auch im Spiel einsehbar sind, sollen Sie nun eine Highscore View ähnlich zu Abbildung 10 umsetzen.

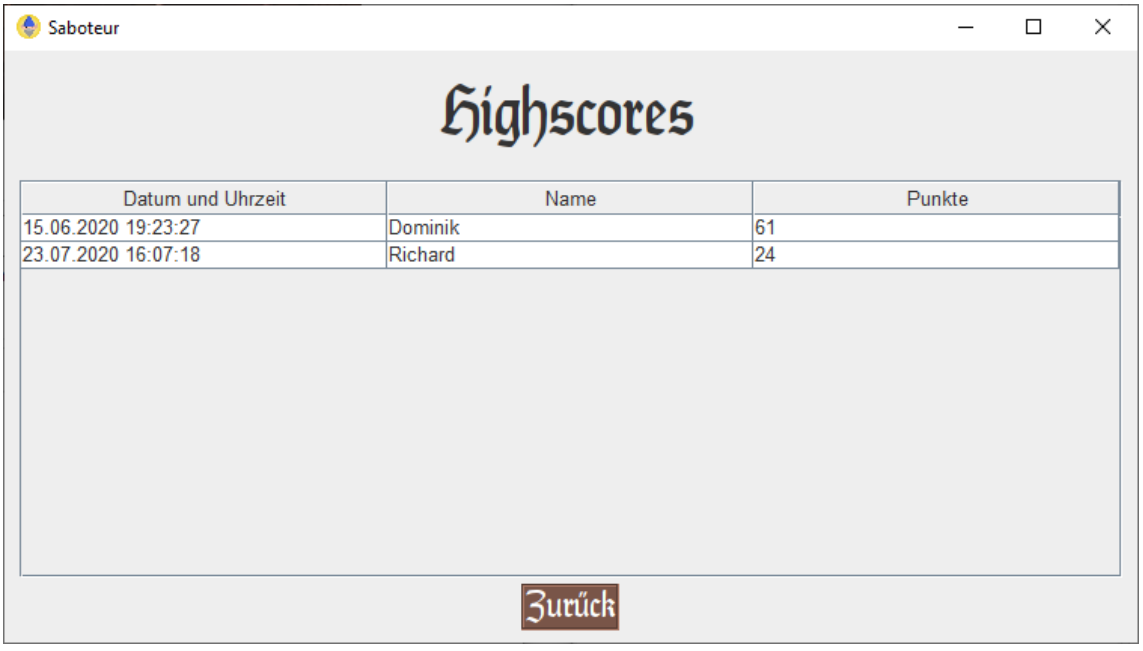


Abbildung 10: Beispiel für eine Highscore View mit einer Tabelle zur Darstellung der Ergebnisse und einem Zurück-Button.

Erstellen Sie dazu eine Klasse `HighscoreView`, die direkt oder indirekt von `View` erbt. Empfehlenswert ist es, sich hierfür näher mit der Klasse `MenuView` im Package `view.menu`

zu beschäftigen. Rufen Sie die Methode `ScoreEntryIO.loadScoreEntries()` auf, um die Highscores aus der Datei zu lesen und stellen Sie die Ergebnisse entsprechend dar. Fügen Sie ebenso einen Zurück-Button hinzu, der dem Benutzer die Möglichkeit gibt, wieder ins Hauptmenü zurückzukehren. Um einem `JButton` diese Funktionalität zu geben, können Sie die folgende Codezeile verwenden.

```
1 backButton.addActionListener(evt -> getWindow().setView(new
     MainMenu(getWindow())));
```

Kommentieren Sie ebenso Zeile 67 in der Klasse `MainMenu` ein, damit die Highscore View über den Punkte-Button im Hauptmenü aufgerufen werden kann.

```
1 highscoreButton.addActionListener(evt -> getWindow().setView(new
     HighscoreView(getWindow())));
```

Hinweis: Zum Umwandeln des `LocalDateTime` Objekts einer `ScoreEntry` in einen String können Sie einen `DateTimeFormatter`[5] verwenden.

4.3.2 Ein neuer Spieler — 7 Punkte

Überlegen Sie sich neben den bereits bestehenden Rollen, Goldsucher und Saboteur, eine dritte Rolle. Ergänzen Sie die Enumeration `Role` der Klasse `Player` im Package `model` entsprechend und erstellen Sie für die neue Rolle ein passendes Bild. Benennen Sie das Bild `role_xxx.png`, wobei `xxx` durch den Namen Ihrer neuen Rolle wie in der Enumeration benannt, aber alles klein geschrieben, zu ersetzen ist. Platzieren Sie dieses Bild in den Ordner `res/main/image` und aktualisieren Sie ihr Eclipse-Projekt (`F5`).

Damit die neue Rolle auch das Spiel gewinnen kann, muss die Methode `getWinners()` der Klasse `GameController` im Package `controller` angepasst werden. Die Methode gibt eine Liste aller Spieler zurück, die gewonnen haben, oder `null`, falls zum aktuellen Zeitpunkt noch kein Gewinner feststeht. Überlegen Sie sich, wann Ihre Rolle gewinnt und passen Sie die Methode entsprechend an. Sie dürfen dabei auch die Gewinnbedingungen der anderen Rollen ändern und es ist auch möglich, dass Spieler unterschiedlicher Rollen gleichzeitig gewinnen.

Damit die Punktevergabe abwechslungsreicher wird, ist es nun Ihre Aufgabe, das Punktesystem (Abschnitt 2.4) zu ändern. Mit dem Aufruf `scorePoints(points)` können dem aktuellen Spieler für seine Aktion Punkte vergeben werden. Diese Punktevergabe findet in den in Abschnitt 3.3 beschriebenen Methoden des `GameController` vor dem Aufruf von `nextPlayer()` statt. Werden Sie kreativ und vergeben Punkte abhängig von der Rolle des Spielers oder der Rolle des Spielers, der gesperrt wird, oder der Reihenfolge, in der Aktionen ausgeführt werden.

Verbindliche Anforderung: Halten Sie Ihre Ideen und Änderungen in der PDF-Datei fest, die Sie zusammen mit dem Code abgeben (Abschnitt 1.6). Achten Sie beim Programmieren darauf, dass die Funktionalität der Methoden und damit des Spiels erhalten bleibt und lediglich die Ermittlung der Gewinner bzw. die Punktevergabe geändert wird.

[5] `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/time/format/DateTimeFormatter.html`

Hinweis: Wenn Sie die dritte Rolle hinzugefügt haben, ändert sich die Anzahl von Goldsuchern und Saboteuren, da Platz für die neue Rolle geschaffen wird. Diese neue Rolle ist ab einer Spielerzahl von zwei Spielern verfügbar und etwas häufiger als die Rolle des Saboteurs.

4.3.3 Computerspieler **9 Punkte**

Der Computerspieler der Codevorlage kann nur eine einzige Aktion durchführen: eine zufällige Handkarte abwerfen. Das sollen Sie in dieser Aufgabe ändern. Die Aktion, die der Computerspieler macht, wird in der Methode `doAction()` der Klasse `ComputerPlayer` im Package `model` durchgeführt. Details hierzu finden Sie in Abschnitt 3.2.2.

Ändern Sie die Methode so ab, dass der Computerspieler in der Lage ist, den Regeln entsprechend Karten ins Wegelabyrinth zu legen. Ebenso soll er mindestens drei weitere Aktionen wie Wegekarten zerstören, Spieler sperren oder entsperren, Karten abwerfen oder die Schatzkarte benutzen können. Machen Sie den Computerspieler so clever, dass er seiner Rolle entsprechend handelt. Das kann sich äußern, indem er als Goldsucher verstärkt in Richtung Zielkarte baut oder als Saboteur bevorzugt Mitspieler sperrt. Denken Sie hierbei auch an die in der vorherigen Aufgabe hinzugefügte dritte Rolle.

Verbindliche Anforderung: Halten Sie Ihre Ideen in der PDF-Datei fest. Erklären Sie, wann der Computerspieler welche Aktionen ausführt und wie er seiner Rolle entsprechend handelt. Beachten Sie auch, dass der Computerspieler nur den Regeln entsprechend, wie ein normaler Mitspieler, spielen darf. Das bedeutet auch, dass er die Rollen der Mitspieler **nicht** kennt.