Funktionale und objektorientierte Programmierkonzepte
Projekt

Prof. Karsten Weihe

Wintersemester 23/24	v1.2.0
Themen:	Alle Inhalte der FOP
Relevante Foliensätze:	*
Abgabe des Projekts:	15.03.2024 bis 23:50 Uhr

FOP Projekt **Gesamt: 100 Projektpunkte**

Die Siedler von Catan

> **Beachten Sie die Seite *Verbindliche Anforderungen für alle Abgaben* im Moodle-Kurs.**
>
> Verstöße gegen verbindliche Anforderungen führen zu Punktabzügen und können die korrekte Bewertung Ihrer Abgabe beeinflussen. Sofern vorhanden, müssen die in der Vorlage mit `TODO` markierten `crash`-Aufrufe entfernt werden. Andernfalls wird die jeweilige Aufgabe nicht bewertet.

Die für diese Hausübung relevanten Verzeichnisse sind `src/main/java/Projekt` sowie `src/main/ressources` und ggf. `src/test/java/Projekt`.

Organisation und Information

Grundlegende Informationen und Bonus

Das FOP-Projekt ist *nicht* verpflichtend. Sie benötigen es *nicht* um die formale Prüfungszulassung (Studienleistung) zu erreichen. Sie können jedoch mit dem FOP-Projekt zusätzliche Bonuspunkte erreichen. Wir empfehlen jedoch dringend am FOP-Projekt teilzunehmen, da Sie hier „richtig“ programmieren lernen. Veranstaltungen in weiterführenden Semestern setzen das Programmieren mehr oder weniger stillschweigend voraus.

Voraussetzung für den Bonus ist die bestandene Studienleistung. Sie können sich - neben den Bonuspunkten der Hausübungen - im Projekt weitere Bonuspunkte für die Klausur erarbeiten. Im Projekt können Sie zusätzlich 100 Hausübungspunkte erreichen. Diese erhalten Sie, wenn Sie 80 Projektpunkte erreichen.

Die Umrechnung von ihren erreichten Projektpunkten x zu den erreichten Hausübungspunkten y erfolgt nach folgender Formel: $y = \left\lfloor \frac{x}{80} \cdot 100 \right\rfloor$.

Projektpunkte	Hausübungspunkte
80-100	100
60	75
50	62
40	50
20	25
0	0

Tabelle 1: Beispiele für die Umrechnung von Projektpunkten zu Hausübungspunkten

> **Anmerkung:**
>
> Wenn Sie bereits 100% der Hausübungspunkte erreicht haben, haben Sie bereits den „vollen Notenbonus“ erreicht und können keine weiteren Bonuspunkte mehr sammeln. Wenn nicht können Sie diese jedoch mit dem Projekt „auffüllen“. Das Projekt eignet sich aber dennoch zum Üben und Vertiefen der Programmierkenntnisse.

Sie werden das Abschlussprojekt in Teams von **drei bis vier** Personen bearbeiten.

Zeitplan im Überblick

- bis 14.03.2024, 23:50 Uhr: Gruppenanmeldung
- 04.03.2024 - 15.03.2024: Projektsprechstunden (online und in Präsenz)
- bis 15.03.2024, 23:50 Uhr: Abgabe des Projekts

Projektsprechstunden

Die Projektutor*innen bieten für Fragen zum Projekt im Zeitraum **04.03.2024 - 15.03.2024** Sprechstunden an. Sie finden eine Übersicht über die Sprechstundentermine im Projektabschnitt des zugehörigen Moodle-Kurses.

Moodle - Forum

Sie finden drei Foren im Projektabschnitt des Moodle-Kurses. Im Forum für organisatorische Fragen können Sie Fragen hinsichtlich der Organisation und des Zeitplans stellen. Das Forum für inhaltliche Fragen ist für Fragen rund um die Projektaufgaben gedacht. Das dritte Forum ist für die Gruppenfindung, falls sie noch keine Gruppe haben oder Ihrer Gruppe noch Mitglieder fehlen. Fragen im Forum werden auch außerhalb der Projektsprechstunden beantwortet.

Gruppenanmeldung (bis 14.03.2024, 23:50 Uhr)

Um sich anzumelden, bilden Sie Gruppen aus **3-4** Studierenden. Sollten Sie noch keine Gruppe haben, so können Sie im Projektabschnitt des Moodle-Kurses ein Forum zur Gruppenfindung nutzen. Alle Gruppenmitglieder müssen sich manuell in die Gruppen eintragen. Kleinere Gruppen oder Einzelpersonen werden von uns eine Woche vor der Abgabe des Projekts automatisch zu Gruppen zusammengefasst. Wenn Sie Ihre Gruppe wechseln möchten, müssen Sie dies bis zum 14.03.2024, 23:50 Uhr, im Moodle-Kurs tun. Danach ist keine Änderung mehr möglich. Wir empfehlen jedoch dringend, sich spätestens zwei Wochen vor der Anmeldefrist zu einer Gruppe zusammenzufinden.

Abgabe des Projekts

Das Projekt ist bis zum **15.03.2024 um 23:50 Uhr Serverzeit** auf Moodle abzugeben. Sie exportieren das Projekt genauso wie die Hausübungsabgaben über den `mainBuildSubmission` Gradle-Task. Dabei gelten auch die gleichen Konventionen wie bei den Hausübungen. Hierbei muss nur eine Person aus der Gruppe das Projekt abgeben und ihre üblichen Daten eintragen.

Sie geben neben dem Code zusätzlich eine PDF-Datei ab, die sich ebenfalls in der komprimierten Abgabe befindet. Dazu speichern Sie diese bitte im Ordner `src/main/resources/documentation` und benennen Sie die Datei `documentation.pdf`. Diese PDF umfasst die Dokumentationen der weiterführenden Aufgaben. Die Dokumentationen sollen es uns als Veranstaltende ermöglichen, Ihre Gedanken zu den Aufgaben zu verstehen.

> **Verbindliche Anforderung (**Für die Abgabe des Projekts**):**
>
> Nachdem eines Ihrer Teammitglieder das Projekt auf Moodle hochgeladen hat, müssen alle anderen Teammitglieder diese Aufgabe im entsprechenden Modul bestätigen, und zwar jedes Mal wenn eine neue Version des Projekts hochgeladen wird. Andernfalls wird die Aufgabe nur im Entwurfsmodus gespeichert und nicht als Abgabe gewertet. **Es werden keine Abgaben im Entwurfsmodus akzeptiert. Diese werden nicht bewertet und unabhängig vom Inhalt der Abgabe mit 0 Punkte bewertet.** Geben Sie daher Ihr Projekt nicht kurz vor der Deadline ab. Ihre Teammitglieder müssen genügend Zeit haben, um die Abgabe zu bestätigen.

Plagiarismus und die Nutzung von KI

Selbstverständlich gelten die gleichen Regelungen zum Plagiarismus und zur Nutzung von KI, wie auch bei den Hausübungsabgaben. Daher hier nochmal der Hinweis, dass Ihr gemeinsames Repository für die Gruppenarbeit privat sein muss, falls Sie git verwenden. Ein Austausch mit anderen Gruppen ist untersagt. Beachten Sie: Tritt der Fall ein, dass Ihre Dokumentation der Lösungswege unvollständig ist oder uns den Anlass für einen Verdacht gibt, dass Sie nicht nur innerhalb der eigenen Gruppe gearbeitet haben, so müssen Sie damit rechnen, dass Sie Ihre Ergebnisse bei einem privaten Testat bei Prof. Weihe persönlich vorstellen und erläutern müssen.

Inhaltliche Information zum Projekt

Im Laufe des Projekts werden Sie eine Implementation des beliebten Brettspiels „Die Siedler von Catan“ erarbeiten und diese ergänzen. Sie erhalten eine Vorlage, in welcher Sie – je nach Aufgabenstellung – Code ergänzen oder erweitern müssen, damit eine lauffähige Version des Brettspiels entsteht. Sie werden viele bekannte Themen aus der Vorlesung im Projekt behandeln und anwenden müssen. Damit stellt das Projekt auch eine gute Vorbereitung auf die Klausur dar. Anders als in den Hausübungen, in denen Sie die Vorlage nicht verändern dürfen und nur die Methoden implementieren, die mit `TODO` markiert sind, dürfen (und sollen) Sie im Projekt die Vorlage an den entsprechenden Stellen verändern und erweitern. Außerdem sind sie hinsichtlich der verwendeten Klassen und Methoden frei in der Wahl, solange die Funktionalität des Spiels erhalten bleibt. Mehr dazu in den weiterführenden Aufgaben.

Wir wünschen Ihnen viel Spaß bei der Implementierung des Brettspiels und wir freuen uns auf die, hoffentlich gelungenen, Implementationen!

Spielekonzept Die Siedler von Catan

Im Brettspiel „Die Siedler von Catan“ besiedeln die Spieler gemeinsam eine Inselwelt. In der Inselwelt bauen sie Siedlungen, errichten Straßen und sammeln Rohstoffe. Das Ziel des Spiels ist es als Erster 10 oder mehr Siegpunkte zu erreichen, indem man Siedlungen zu Städten erweitert und Entwicklungskarten erwirbt.

Spielmaterial

Zum Spielmaterial gehören:

- Hexagonfelder (Tiles) mit unterschiedlichen Geländetypen: Wälder, Hügel, Felder, Berge und Wüsten
- Siedlungen und Städte in verschiedenen Farben für jeden Spieler
- Straßen für den Bau von Wegen zwischen den Siedlungen
- Rohstoffkarten für Ressourcen wie Holz, Lehm, Getreide, Erz und Wolle
- Zwei Würfel zur Generierung von Ressourcen
- Entwicklungskarten mit verschiedenen Funktionen
- Ein Räuber

Spielablauf

Start

Die Spieler platzieren der Reihe nach ihre Siedlungen und Straßen auf der Insel. Jeder Spieler beginnt mit zwei Siedlungen und zwei Straßen.

Aktionen

Die Spieler würfeln nacheinander zu Beginn ihres Zuges. Die gewürfelte Zahl bestimmt, welche Felder Rohstoffe produzieren. Spieler erhalten Ressourcen, wenn die Zahl auf den Feldern gewürfelt wird, die an ihre Siedlungen angrenzen. Ein Feld wirft keinen Ertrag ab, wenn sich der Räuber darauf befindet. Danach können die Spieler handeln, Straßen und Siedlungen bauen oder Entwicklungskarten erwerben. Ein Sonderfall tritt auf, wenn eine „7“ gewürfelt wird. Dann gibt jeder Spieler, der mehr als 7 Rohstoffe besitzt die Hälfte (abgerundet) seiner Rohstoffkarten ab. Zusätzlich setzt der Spieler, der gerade am Zug ist, den Räuber auf ein anderes Tile. Abschließend stiehlt dieser Spieler bei einem anderen Spieler, der eine Siedlung oder Stadt an diesem Tile besitzt, eine Rohstoffkarte aus der verdeckten Hand.

Straßen- und Siedlungsbau:
Sie müssen außerdem auf bestimmte Bedingungen achten, wenn Sie Strukturen wie Siedlungen oder Straßen bauen. Siedlungen müssen auf einer Kreuzung gebaut werden, zu der mindestens eine eigene Straße führt. Dabei muss die Abstandsregelung beachtet werden. Die Abstandsregelung besagt, dass eine Siedlung nur dann auf einer Kreuzung gebaut werden darf, wenn die angrenzenden Kreuzungen nicht von Siedlungen oder Städten besetzt sind, unabhängig davon wem sie gehören.

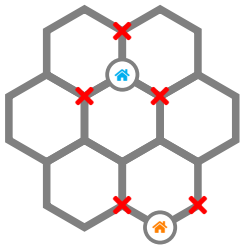


Abbildung 1: Die Abbildung zeigt die Abstandsregelung bei Siedlungen. Auf den mit „X“ markierten Kreuzungen dürfen keine Siedlungen gebaut werden.

Spieler dürfen Straßen nur auf Wegen bzw. Kanten bauen. Auf jedem Weg darf maximal nur eine Straße gebaut werden. Außerdem dürfen Straßen nur an einer Kreuzung anliegen, an der eine eigene Straße, Siedlung oder eine Stadt angrenzt und auf der keine fremde Siedlung oder Stadt steht.

Handel:
Nachdem ein Spieler gewürfelt und seine Rohstoffe bekommen hat, hat er die Möglichkeit mit seinen Mitspielern oder der Bank zu handeln. Handelt ein Spieler mit der Bank, ist das Standard-Verhältnis 4:1. Der Spieler bietet 4 Rohstoffe des selben Typs und bekommt ein Rohstoff seiner Wahl. Hat der Spieler Zugang zu einem Hafen, also eine Siedlung oder Stadt an einer Kante mit einem Hafen, verbessert sich das Verhältnis für den Spieler. Allgemeine Häfen erlauben den Handel mit Verhältnis 3:1. Spezialhäfen erlauben den Handel sogar mit einem Verhältnis von 2:1, allerdings akzeptieren sie nur den auf ihnen abgebildeten Rohstofftyp.
Beim Handel mit Mitspielern ist das Verhältnis frei wählbar. Der initiierende Spieler kann beliebig Rohstoffe anbieten, und ebenso frei wählen welche Rohstoffe er haben möchte. Anders als die Bank, müssen die anderen Spieler den Handel allerdings nicht annehmen.

Will oder kann der Spieler in seinem Zug keine Aktion mehr durchführen, erklärt der Spieler seinen Zug für beendet. Der nächste Spieler im Uhrzeigersinn ist dann an der Reihe.

Spielende

Das Spiel endet, wenn ein Spieler mindestens 10 Siegpunkte erreicht. Siegpunkte werden für Siedlungen und Städte, aber auch für einige Entwicklungskarten vergeben:

- **Siedlungen:** Jede Siedlung bringt 1 Siegpunkt.
- **Städte:** Verbesserte Versionen von Siedlungen, die 2 Siegpunkte bringen.
- **Längste Handelsstraße:** Bonuspunkte für den Spieler mit der längsten durchgehenden Straße; mindestens 5 Straßen-Segmente (2 Siegpunkte). Durchgehend heißt in diesem Fall, dass die Straße nicht durch Siedlungen oder Städte anderer Spieler unterbrochen wird. (Aktuell wegen hoher Komplexität nicht implementiert)
- **Größte Rittermacht:** Bonuspunkte für den Spieler mit den meisten aufgedeckten Ritterkarten; mindestens 3 Karten (2 Siegpunkte).
- **Siegpunktkarten:** Diese Entwicklungskarten bleiben das gesamte Spiel über verdeckt und werden erst am Ende aufgedeckt (1 Siegpunkt pro Karte). Sie zählen auch verdeckt zu den Siegpunkten eines Spielers.

Zusätzliche Informationen

Weitere Informationen sowie die offizielle Spielanleitung erhalten Sie unter den nachfolgenden Links:

- Offizielle Anleitung
- Almanach, erweiterte Anleitung
- Wikipedia Artikel
- Youtube Video, welches die Spielregeln erläutert

Wichtige Klassen aus der Java-Standardbibliothek und JavaFX

Das Projekt nutzt an vielen Stellen Datenstrukturen und andere Klassen aus der Java-Standardbibliothek. Wenn Sie sich unsicher sind, wofür und wie eine solche Klasse verwendet wird, kann sich ein Blick in die offizielle Dokumentation lohnen:

- `java.util.Set`, `java.util.List` und `java.util.Map` sowie `java.util.Map.Entry`
- `java.util.stream.Stream`, `java.util.stream.Collectors` und `java.util.Optional`
- `java.util.Collections`, vor allem die `unmodifiable*`-Methoden
- `java.util.function.Supplier`, `java.util.function.Predicate`, `java.util.function.Function` und `java.util.function.Consumer`
- `java.util.Comparator`
- Enums (`java.lang.Enum`) und Records (`java.lang.Record`)
- JavaFX

Lokale Dokumentation

Mit Hilfe des Gradle-Tasks „javadoc“ können Sie sich lokal die Dokumentation des Projekts als statische Webseiten erzeugen lassen. Während dem Ausführen des Tasks aufkommende Warnungen können Sie ignorieren. Die fertige Dokumentation finden sie im Ordner `build/docs/javadoc`. Zum Anzeigen öffnen Sie einfach die Datei `index.html` in einem Browser ihrer Wahl. Das Suchfeld oben rechts erlaubt es nach Dokumentation zu Klassen, Methoden und Attributen des Projekts zu suchen.

> **Hinweis:**
>
> Da es den Rahmen dieser Aufgabenstellung noch mehr Sprengen würde, werden hier nicht alle Klassen im Detail erklärt. Es ist daher wichtig, dass Sie sich mit der Dokumentation vertraut machen, sollte die Funktion einer Klasse oder Methode nicht direkt ersichtlich sein.

Design Patterns

Die Struktur und die Implementation des Projekts orientieren sich am Entwurfsmuster Model-View-Controller. Dieses Design Pattern wird im Abschnitt Model-View-Controller des Foliensatzes *Fehlersuche und fehlervermeidender Entwurf* behandelt. Wie Sie anhand des Aufbaus des Projekts sehen können, unterteilt sich das Projekt in drei wichtige Komponenten. Die drei Komponenten entsprechen drei Packages, welche heißen:

- `projekt.model`
- `projekt.view`
- `projekt.controller`

Das Package `projekt.model` implementiert im Wesentlichen die grundlegenden Daten wie den Aufbau eines Spielers oder des Spielfeldes. Das zweite Package `projekt.view` stellt die Daten der Komponente Model dar und interagiert mit dem Benutzer. Bei Änderungen der View wird die Komponente Controller benachrichtigt. Es werden keine Daten in der Komponente View verarbeitet, sondern nur entgegengenommen. Das Package `projekt.controller` lenkt und verwaltet die beiden anderen Komponenten, implementiert also die Spiellogik.

Das MVC-Pattern lässt sich sehr gut auf die echte Welt bzw. das Originalspiel übertragen:

- Das Model stellt die einzelnen physischen Komponenten des Spiels dar; z.B. das Spielbrett, Ressourcen- und Entwicklungskarten, Straßen, Siedlungen, aber auch die Spieler selbst. Es beschreibt diese Dinge allerdings nicht nur, sondern speichert auch deren Zustände.
- Die View ist was die Spieler sehen und womit sie interagieren. Analoge Beispiele wären die Spielfiguren oder der Aufdruck des Spielfelds und der Karten. Views können das Model und die Zustände der Komponenten nutzen bzw. anzeigen, aber nicht direkt manipulieren / updaten.
- Der Controller verknüpft Model und View und stellt gewissermaßen die Spielregeln dar. Er entscheidet, welche Aktionen der Spieler ausführen darf und unter welchen Umständen. Außerdem führt er diese Aktionen aus und manipuliert dafür das Model, z.B. Bau einer Straße.

Implementationsstruktur

HexGrid - Das Spielfeld

Das HexGrid repräsentiert das Spielfeld als hexagonale Struktur. Es besteht aus einzelnen Tiles, welche die Spielumgebung definieren.

Das `HexGrid` an sich ist ein Interface und wird in `HexGridImpl` implementiert.

Das Spielfeld verwendet ein **axiales Koordinatensystem** mit den Achsen q, r und s. Die Koordinaten müssen die Bedingung $q + r + s = 0$ erfüllen. Durch dieses System können Tiles auf dem Spielfeld eindeutig lokalisiert werden. Die s-Koordinate wird berechnet als $s = -q - r$.

Die Verwendung von **Spiralringen** ist eine effektive Methode, um systematisch durch die Hexagon-Kacheln zu navigieren. Unser Algorithmus teilt das Spielfeld in Ringe auf und füllt dann die Hexagon-Kacheln in jedem Ring nacheinander. Dies ermöglicht eine geordnete Durchquerung des HexGrids, was für die Implementierung des Setzens der einzelnen Tiles auf dem Spielfeld entscheidend ist. Spiralringe bieten auch die Möglichkeit, die Anzahl der Hexagon-Kacheln in einem größeren Hexagon zu berechnen und können für die Bestimmung von Bewegungsbereichen in Spielen verwendet werden.

Eine **informative Erklärung** mit anschaulichen Grafiken, die das axiale Koordinatensystem veranschaulichen, steht für eine vertiefte Einsicht unter dem folgenden Link zur Verfügung:

`https://www.redblobgames.com/grids/hexagons/#coordinates`

Bei der Umsetzung des Spielfeldes haben wir uns stark von dieser Erklärung inspirieren lassen, um die grundlegenden Konzepte klar und verständlich zu integrieren.

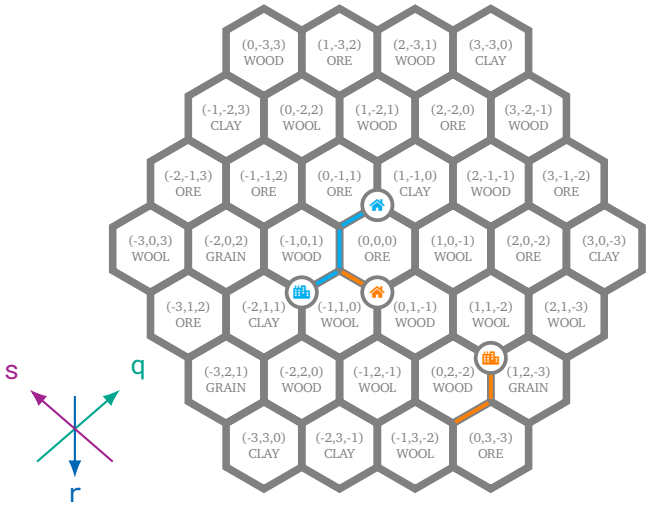


Abbildung 2: Die Abbildung zeigt eine vereinfachte Darstellung des Spielfelds. Koordinaten werden im Format (q, r, s) angegeben. Hierbei steht WOOD für Holz, CLAY für Lehm, GRAIN für Getreide, ORE für Erz und WOOL für Wolle.

TilePosition

Die Klasse `TilePosition` ist zentral für die Positionierung auf dem Spielbrett im axialen Koordinatensystem. Sie repräsentiert eine Position durch die axialen Koordinaten q und r, wobei die s-Koordinate berechnet wird, um die Bedingung $q + r + s = 0$ zu erfüllen.

Mit Funktionen wie `add`, `subtract` und `scale` ermöglicht es die Klasse grundlegende Rechenoperationen zwischen Positionen durchzuführen. Eine wichtige Funktion ist `neighbours`, die alle benachbarten Positionen einer gegebenen Position zurückgibt.

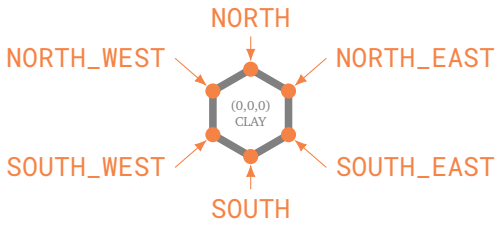
Die Richtungen, um eine Position werden durch `EdgeDirection` definiert, wobei Funktionen wie `left` und `right` die Navigation zwischen benachbarten Richtungen erleichtern. Diese Richtungen sind relevant für den Spielaufbau, insbesondere für den Bau von Straßen und Siedlungen. Zusätzlich werden Richtungen in Bezug auf Kreuzungen `IntersectionDirection` definiert. Jede Richtung hat „linke“ und „rechte“ Nachbarrichtungen.

Tile

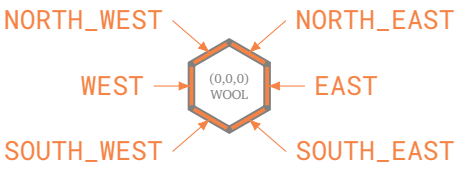
`Tile` ist ein Interface, welches in der Klasse `TileImpl` implementiert wird. Diese fungiert als einer der zentralen Elemente in der Implementierung des Spiels „Die Siedler von Catan“. Jedes Tile repräsentiert einen Baustein des Spielfelds und bietet Zugriff auf verschiedene Eigenschaften und Operationen. Diese umfassen Höhe und Breite als `ObservableDoubleValues`, den Tile-Typ (`WOODLAND`, `MEADOW`, `FARMLAND`, `HILL`, `MOUNTAIN` und `DESERT`), eine mit dem Tile verbundene Würfelzahl und die Position im axialen Koordinatensystem.

Das Tile ist eng mit dem HexGrid verbunden und ermöglicht es, die Nachbarn eines Tiles abzurufen. Außerdem gibt es Hilfsmethoden, um die verbundenen Kreuzungen `Intersections` und die darauf platzierten Siedlungen sowie den Kanten `Edges` und den darauf platzierten Straßen zu erhalten. Es gibt auch Funktionen zum Platzieren und Abrufen von Straßen und Siedlungen. Mit der Methode `hasRobber()` kann auch überprüft werden, ob sich der Räuber derzeit auf dem Tile befindet. Den verschiedenen Tile-Typen sind Farben und Ressourcenarten zugeordnet. Insgesamt bildet die Tile-Klasse die Grundlage für die Implementierung der Spielmechanik von „Die Siedler von Catan“ auf dem HexGrid. Die Tile-Klasse definiert nämlich die Struktur und Beziehungen zwischen den Bausteinen des Spielfelds.

Abbildung 3: Die Abbildung zeigt ein Tile auf dem Spielfeld mit den Koordinaten (0, 0, 0) und dem Ressourcentyp „WOOD“.



(a) `IntersectionDirections` eines `Tiles`



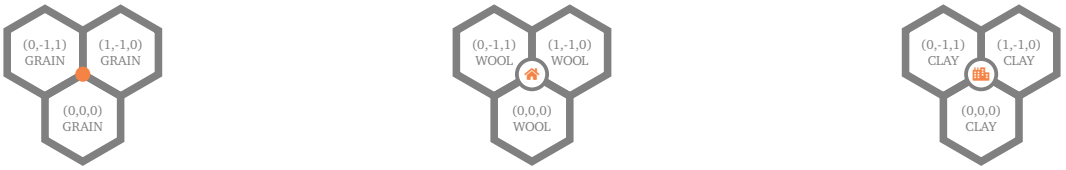
(b) `EdgeDirections` eines `Tiles`

Abbildung 4: Enums in `TilePosition` beschreiben die Richtungen von Kreuzungen und Kanten eines `Tiles` (hier jeweils orange markiert).

Intersection

`Intersection` ist ein Interface, welches die Schnittpunkte auf dem Spielfeld repräsentiert und von der Klasse `IntersectionImpl` implementiert wird. Eine `Intersection` hat drei Eigenschaften:

1. Sie befindet sich immer zwischen drei – ggf. gedachten – Tiles (siehe `TilePosition`).
2. Sie kann eine Siedlung oder eine Stadt besitzen.
3. Sie hat mindestens zwei verbundene Kanten (siehe `Edge`).



(a) ohne Siedlung (orange markiert) (b) mit Dorf (c) mit Stadt

Abbildung 5: Die Abbildung zeigt eine Intersection und die drei angrenzenden Tiles. Eine Intersection kann auch weniger als drei angrenzende Tiles haben, wenn sie sich am Spielfeldrand befindet. Um genau zu sein: An jeder Ecke jedes `Tiles` befindet sich eine Intersection.

Die Klasse bietet verschiedene Funktionen zur Interaktion mit den Spielmechaniken. Durch die Klasse können Spieler Siedlungen auf den Kreuzungen platzieren, vorhandene Siedlungen zu Städten verbessern und Häfen (Ports) zum Handeln nutzen. Es bietet auch den Zugriff auf verbundene Kanten, verbundene Straßen, angrenzende Kreuzungen und benachbarte Tiles. Die Funktionen umfassen das Abrufen von Informationen, wie platzierten Siedlungen, Upgrades zu Städten, zugehörigen Häfen und verbundenen Straßen. Das Klasse erleichtert auch den Zugriff auf benachbarte Kreuzungen und Tiles sowie das Prüfen der Verbindung zu bestimmten Positionen.

Edge

`Edge` ist ein Interface, das die Kanten auf dem Spielfeld repräsentiert und in `EdgeImpl` implementiert wird. Eine `Edge` hat vier Eigenschaften:

1. Sie befindet sich immer zwischen zwei – ggf. gedachten – Tiles (siehe `TilePosition`).
2. Sie beginnt bei einer Kreuzung (siehe `Intersection`) und endet bei einer anderen Kreuzung.
3. Sie kann einen Besitzer haben, in diesem Fall liegt eine Straße auf der Kante.
4. Sie kann an einen Hafen angrenzen.

Abbildung 6: Die Abbildung zeigt eine Kante (orange markiert) zwischen zwei Tiles.

In unserem Model speichern wir nur die beiden anliegenden `TilePositions` und den Besitzer der Kante. Die Reihenfolge der `TilePositions` spielt keine Rolle, da wir die Kanten als ungerichtet betrachten. Auch hier gibt es verschiedene Hilfsfunktionen, um z.B. die anliegenden Kreuzungen oder die anliegenden Straßen des gleichen Spielers zu finden.

Port

`Port` ist ein Record, welches einen Integer-Wert `ratio` und einen `ResourceType` `resourceType` speichert. Die Ports werden in den `Edge`s gespeichert und ermöglichen es den Spielern, Ressourcen zu einem besseren Tauschverhältnis zu tauschen, wenn sie eine Siedlung an einer Kante mit einem Hafen bauen.

Player

Die Klasse `PlayerImpl`, welches das Interface `Player` implementiert, bietet einen umfassenden Zugriff auf die verschiedenen Aspekte eines Spielers. Hier erhält man Einblicke in die Ressourcenverwaltung, den Status von Siedlungen und Straßen sowie die Entwicklungskarten des Spielers.

Über diese Klasse können Spieler ihre aktuellen Ressourcen und deren Mengen abfragen, sowie Ressourcen hinzufügen oder entfernen. Die Klasse speichert auch Informationen zu den Straßen und Siedlungen eines Spielers sowie die Möglichkeit, die Anzahl der verbleibenden Bauwerke zu ermitteln. Die Entwicklungskarten des Spielers (Anzahl und Typen) und zusätzliche Informationen wie die Anzahl der gespielten Ritter und die Spielerfarbe werden ebenfalls von dieser Klasse verwaltet.

GameState

Die Klasse `GameState` ist das Herzstück des Spiels. Sie speichert neben der `HexGrid` auch die Spieler und eine Liste der aktuellen Gewinner. Ist diese Liste leer, so läuft das Spiel noch. Ist sie nicht leer, so ist das Spiel beendet und die Liste enthält die Gewinner des Spiels.

Implementation des Models

Im Package `projekt.model` sind die oben genannten Klassen implementiert. Um die Klassenstruktur und die verschiedenen Beziehungen visuell zu veranschaulichen, haben wir ein Diagram mit Hilfe von **Unified Modeling Language (UML)** modelliert. Hilfe zum Lesen eines UML-Diagramms finden Sie im Wikipedia-Artikel. Abbildung 7 zeigt das UML-Diagram der verschiedenen Klassen zur Implementierung des Models vom Spiel.

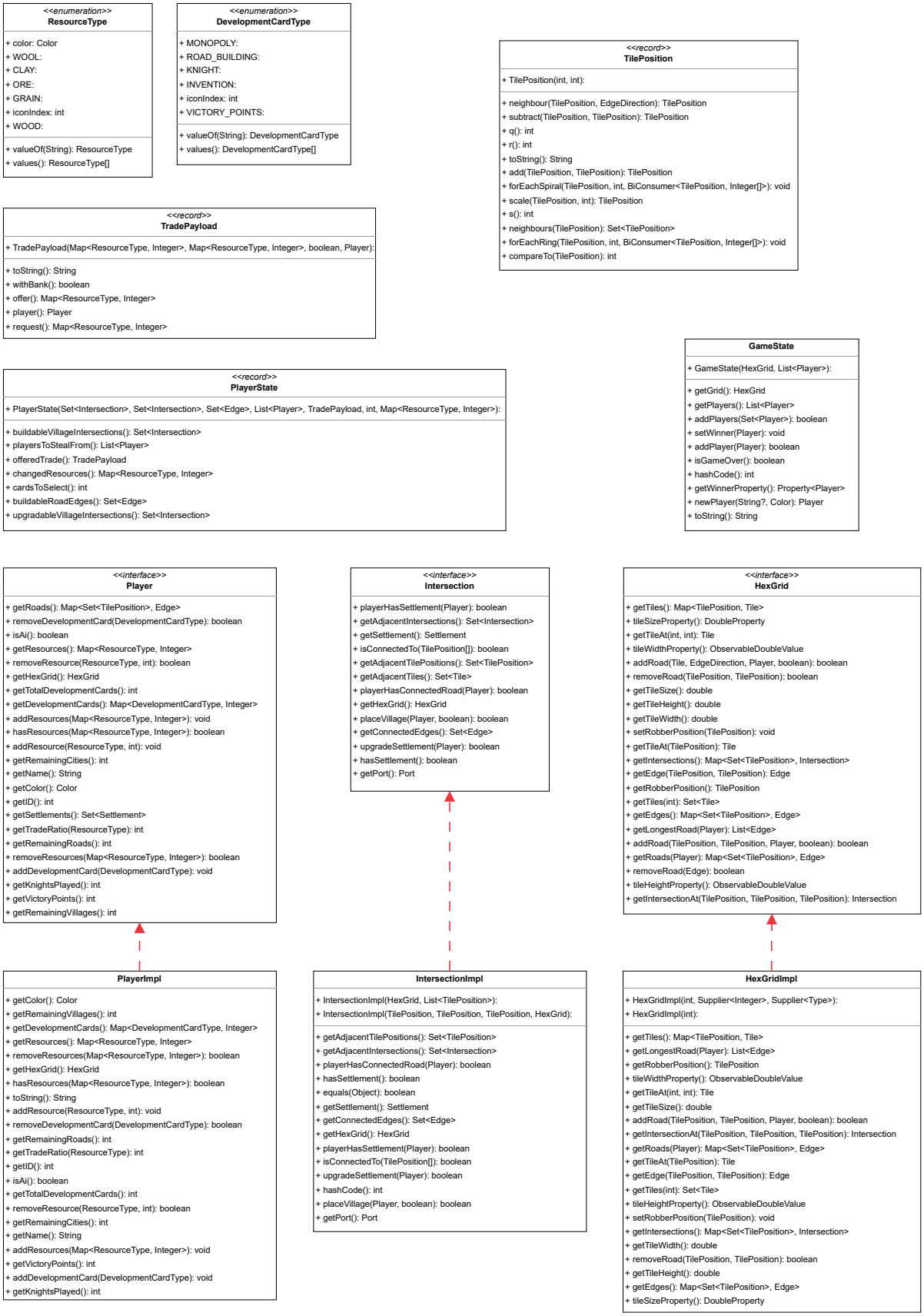


Abbildung 7: Die Abbildung zeigt die Beziehungen der einzelnen Klassen vom Model.

Graphical User Interface

In diesem Kapitel zeigen wir Ihnen die einzelnen Views der GUI und beschreiben diese.

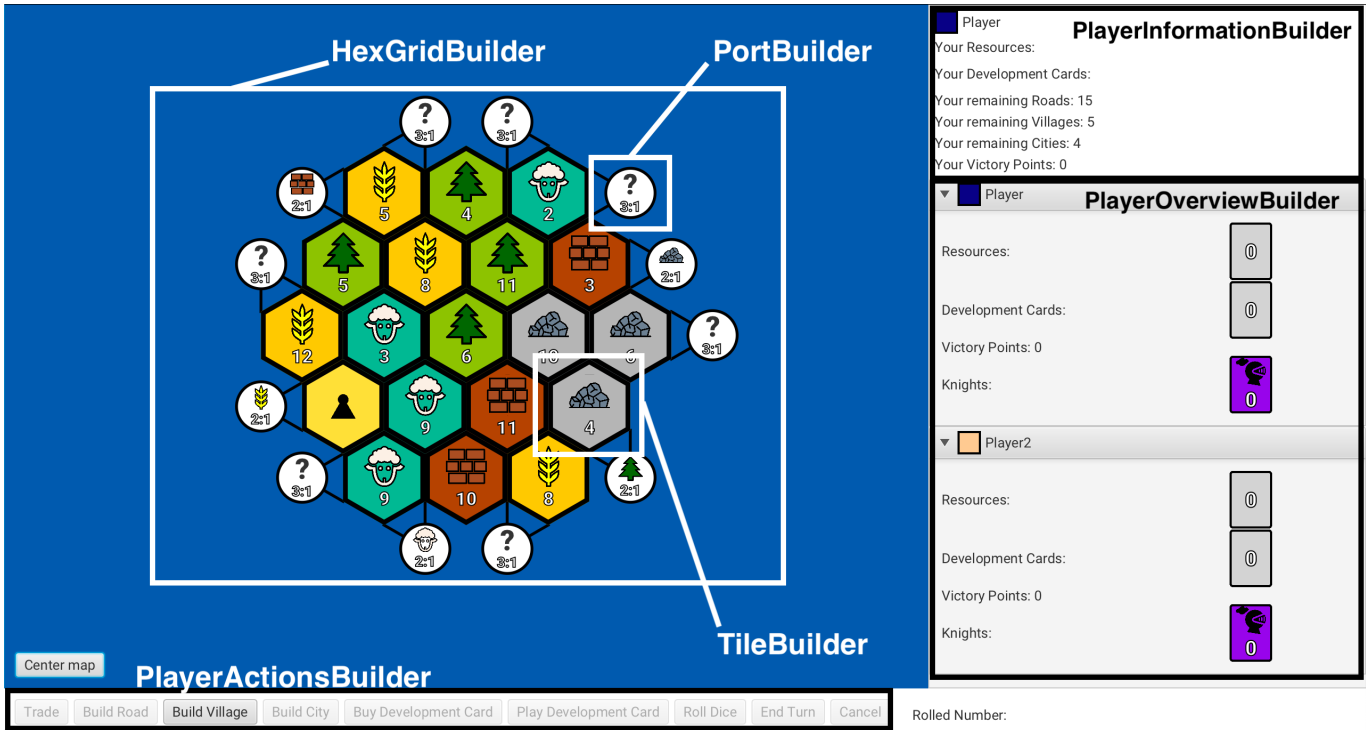


Abbildung 8: In dieser Abbildung sehen Sie das GUI. Auf der linken Seite befindet sich das Spielfeld auf dem Sie die einzelnen Tiles sehen können. Auf der rechten Seite sehen Sie die Informationen zu den einzelnen Spielern. Diese Ansicht ist wiederum aufgeteilt in einzelne Sektionen. In der obersten Sektion sehen Sie die Informationen zum Spieler, der aktuell am Zug ist. In den weiteren Sektionen sehen Sie die Informationen zu den einzelnen Spielern. Auf der unteren Seite der View können Sie die einzelnen Button sehen, welche die einzelnen Spieleraktionen darstellen.

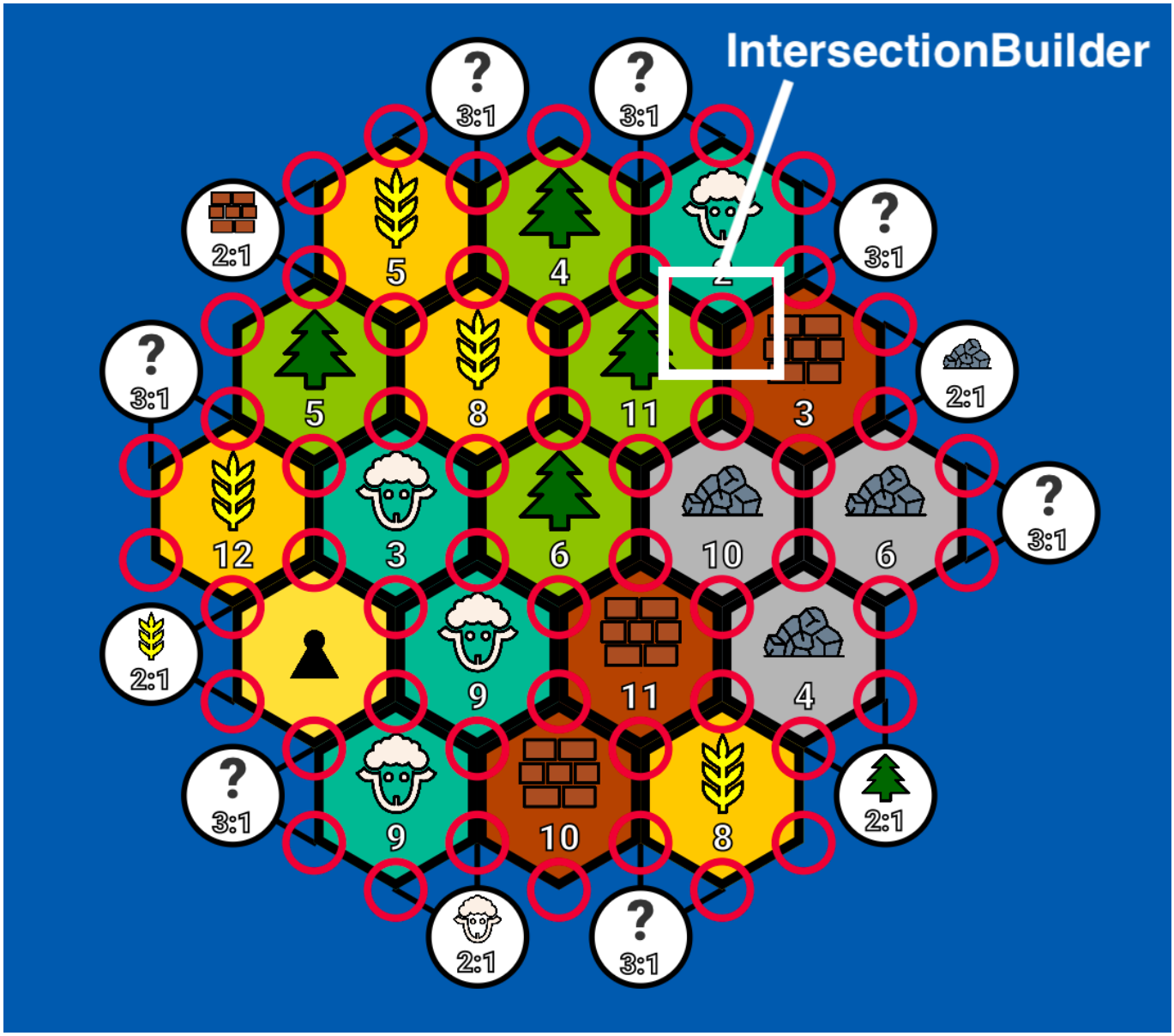


Abbildung 9: Diese View bekommen die Spieler gezeigt, wenn Sie eine Siedlung bauen möchten. Die Spieler können die Kreuzung, auf der die Siedlung platziert werden soll, mittels Mausklick aussuchen.

Abbildung 10: In dieser Abbildung sehen Sie die View, in der eine Straße gebaut werden kann.

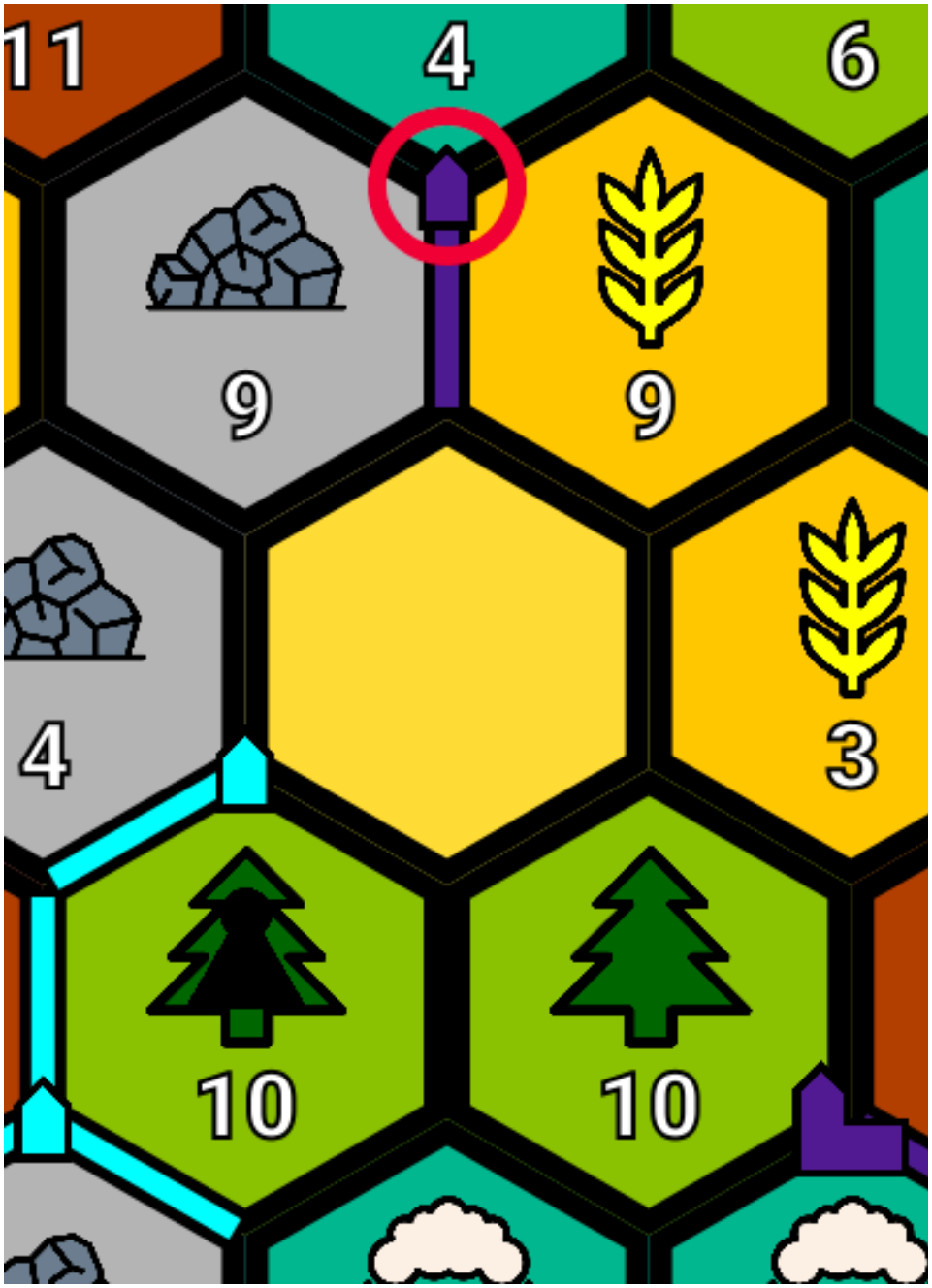


Abbildung 11: Auf dieser Abbildung sehen Sie die View, wenn Spieler eine Siedlung zu einer Stadt aufzuwerten möchten. Es werden nur die Kreuzungen angezeigt, an denen bereits Siedlungen vom jeweiligen Spieler errichtet worden sind.

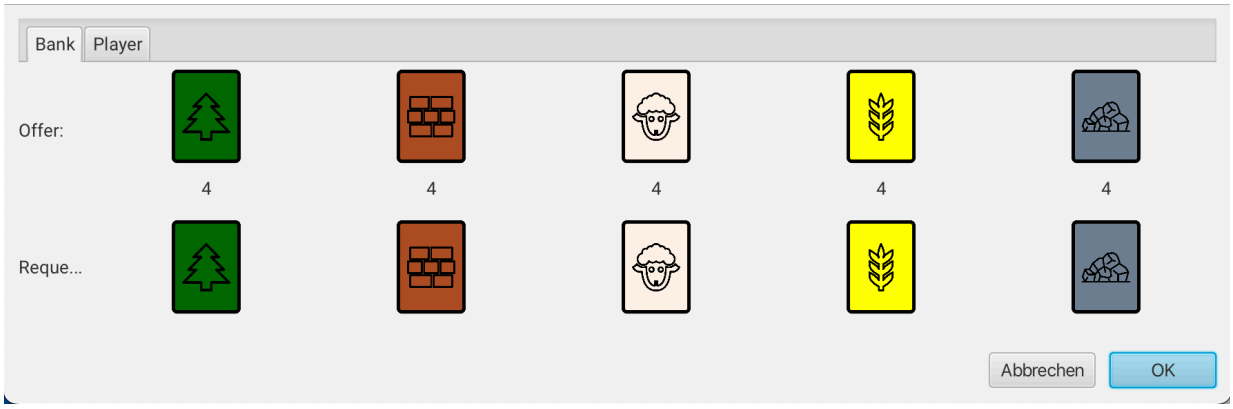


Abbildung 12: In dieser Abbildung sehen Sie die View, die den Handel zwischen einem Spieler und der Bank darstellt.

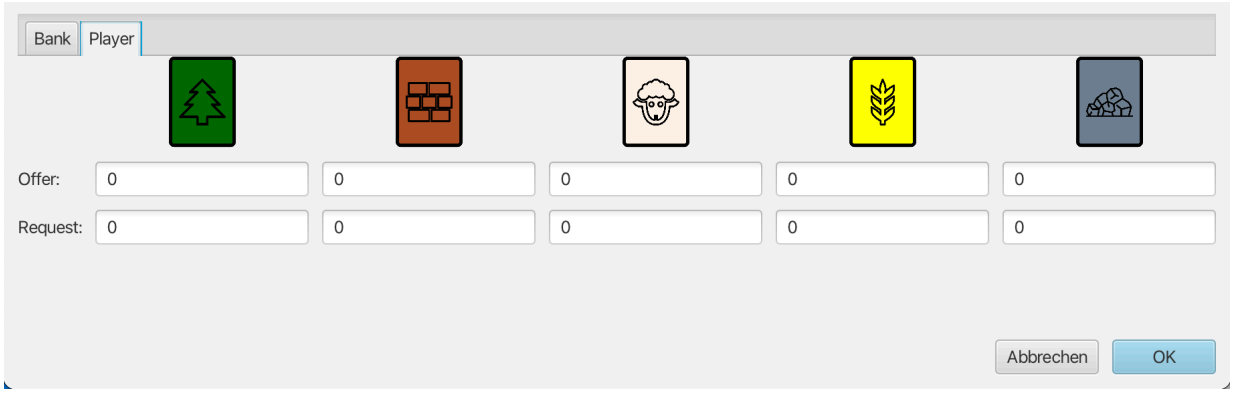


Abbildung 13: In dieser Abbildung sehen Sie die View, die den Handel zwischen Spielern darstellt.

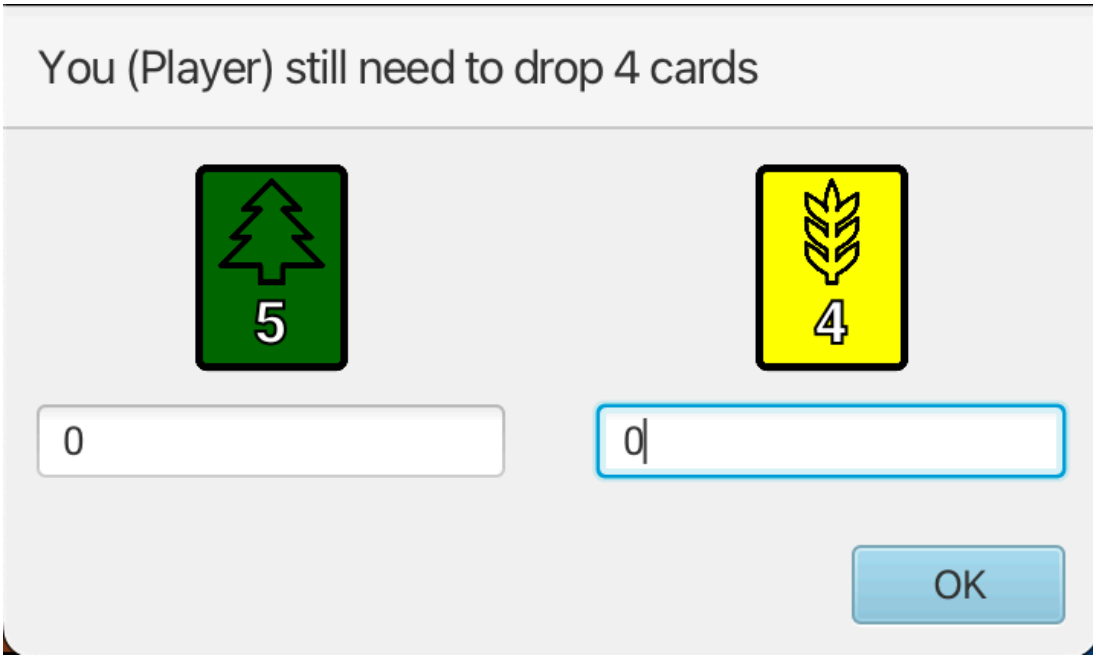


Abbildung 14: Dieser Dialog zum Ablegen von Karten wird angezeigt, wenn eine „7“ gewürfelt worden ist.

Abbildung 15: Dieser Abbildung zeigt die View, in der die Spieler die erworbenen Entwicklungskarten ausspielen können.

Basisaufgaben

Das Projekt besteht aus zwei verschiedenen Aufgabentypen:

- **Basisaufgaben:** In den Basisaufgaben geben wir Ihnen genau vor, welche Funktionalität Sie implementieren sollen. Außerdem stellen wir Ihnen Tests zur Verfügung, die Sie verwenden können, um Ihre Implementierung zu überprüfen. Die Tests decken alle Bewertungskriterien ab, die wir für die Basisaufgaben festgelegt haben. Wir behalten uns jedoch vor, für die Endbewertung zusätzliche Tests zu verwenden oder die Tests bei Fehlern zu überarbeiten.

- **Weiterführende Aufgaben:** In den weiterführenden Aufgaben erweitern Sie selbstständig das Projekt nach Ihren Vorstellungen, mehr dazu in H4.

Nun beginnen wir mit den Basisaufgaben. Die Aufgaben sind so konzipiert, dass diese Sie schrittweise an eine lauffähige Version des Spiels heranführen. Deshalb wird Ihnen empfohlen, die Aufgaben in der nachfolgenden Reihenfolge abzuarbeiten.

H1: Implementierung des Modells ?? Projektpunkte

In dieser Aufgabe implementieren Sie das Modell. Alle Klassen, die Sie in dieser Aufgabe implementieren, befinden sich im Package `projekt.model`. Dazu fokussieren wir uns zunächst auf den Spieler und anschließend auf die Bauwerke.

Wie Sie im Einleitungstext bereits gelesen haben, modelliert das Interface `Player` einen Spieler. In der Klasse `PlayerImpl` werden Sie nun dieses Interface implementieren. Damit können Sie die Ressourcen, Siegespunkte und andere Informationen eines Spielers speichern.

H1.1: Inventarsystem des Players ?? Projektpunkte

> **Hinweis:**
>
> Mit der Methode `Collections.unmodifiableMap` können Sie eine unveränderbare Sicht auf eine Map erhalten.

Die Klasse `PlayerImpl` verfügt über eine Objektkonstante `resources` vom Typ `Map<ResourceType, Integer>`. Die Schlüsselwerte dieser Map sind die jeweiligen Ressourcentypen (`WOOD`, `CLAY`, etc.). Einem Schlüsselwert ist die Anzahl (mindestens 0) der entsprechenden Ressource, die dieser Spieler besitzt, zugeordnet.

Das Ziel dieser Aufgabe ist es, dass Spieler Ressourcen sammeln, benutzen und handeln können. Implementieren Sie die folgenden Methoden der Klasse `PlayerImpl`:

- **Methode `getResources` (1 Punkt zusammen mit den nächsten beiden Methoden):**
  Diese Methode soll eine unveränderbare Sicht auf die Ressourcen des Spielers zurückgeben.

- **Methode `addResource`:**
  Diese Methode soll `amount` (2. Parameter) viele Ressourcen des übergebenen Typs hinzufügen. Eine Prüfung, ob der Spieler die Ressource überhaupt besitzen kann, ist nicht notwendig.

- **Methode `addResources`:**
  Diese Methode soll analog zu `addRessource` mehrere Ressourcen hinzufügen.

- **Methode `hasResources` (1 Punkt):**
  Diese Methode soll überprüfen, ob der Spieler die gegebenen Ressourcen besitzt.

- **Methode `removeResource` (1 Punkt zusammen mit nächster Methode):**
  Diese Methode soll eine Ressource des übergebenen Typs entfernen. Sollte der Spieler zu wenig von der zu entfernenden Ressource besitzen, so wird `false` zurückgegeben und der Zustand des Spielers nicht verändert. Ansonsten wird die Ressource entfernt und `true` zurückgegeben.

- **Methode `removeResources`:**
  Diese Methode soll analog zu `removeResource` mehrere Ressourcen entfernen. Auch hier darf der Zustand des Spielers nicht verändert werden, wenn eine oder mehrere der im Parameter übergebenen Ressourcen nicht in ausreichender Menge vorhanden ist.

- **Methode `getTradeRatio`: (3 Punkte)**
  Diese Methode soll das beste Verhältnis für die übergebene Ressource zurückgeben. Sollte der Spieler einen Spezial-Hafen für diese Ressource besitzen, wäre die Rückgabe 2, bei allgemeinen Häfen 3 und ansonsten 4. Eine Rückgabe von n bedeutet, dass ein Spieler n Einheiten dieser Ressource tauschen kann, um 1 Einheit einer beliebigen anderen Ressource zu erhalten.

H1.2: Entwicklungskarten des Players ?? Projektpunkte

> **Hinweis:**
>
> `getDevelopmentCards`, `addDevelopmentCard` und `removeDevelopmentCard` funktionieren analog zu ihren Gegenstücken in H1.1.

Ähnlich der Verwaltung von Ressourcen, braucht ein Spieler Möglichkeiten, um Entwicklungskarten zu sammeln und zu verwenden. Die Klasse `PlayerImpl` verfügt über eine Objektkonstante `developmentCards` vom Typ `Map<DevelopmentCardType, Integer>`. Implementieren Sie die folgenden Methoden in der Klasse `PlayerImpl`:

- **Methode `getDevelopmentCards` (1 Punkt zusammen mit den nächsten beiden Methoden):**
  Diese Methode soll eine unveränderbare Sicht auf die Entwicklungskarten des Spielers zurückgeben.

- **Methode `addDevelopmentCard`:**
  Diese Methode soll eine Karte des übergebenen Typs hinzufügen.

- **Methode `removeDevelopmentCard`:**
  Diese Methode soll eine Karte des übergebenen Typs entfernen. Sollte der Spieler zu wenig von der zu entfernenden Karte besitzen, so wird `false` zurückgegeben und der Zustand des Spielers nicht verändert. Ansonsten wird die Karte entfernt und `true` zurückgegeben.

  > **Hinweis:**
  >
  > Wenn eine Karte entfernt wird, wird sie offen ausgespielt. Um die Methode `getKnightsPlayed` sinnvoll zu implementieren, müssen sie also auch speichern, welche Karten ausgespielt wurden.

- **Methode `getTotalDevelopmentCards` (1 Punkt zusammen mit der nächsten Methode):**
  Diese Methode soll die Anzahl aller Entwicklungskarten im Besitz des Spielers zurückgeben.

- **Methode `getKnightsPlayed`:**
  Diese Methode soll die Anzahl aller *gespielten* Karten vom Typ `KNIGHT` zurückgeben.

H1.3: Alle Wege führen nach ... ?? Projektpunkte

Als Nächstes implementieren Sie die Funktionalität der Straßen.

Dazu betrachten wir zunächst die Klasse `HexGridImpl`. Diese besitzt ein Attribut `edges` vom Typ `Map<Set<TilePosition>, Edge>`, welches eine Menge von zwei `TilePosition` auf ein `Edge`-Objekt abbildet, welches zwischen diesen beiden Positionen liegt. Hier müssen die folgenden Methoden implementiert werden:

- **Methode `getRoads` (1 Punkte):**
  Diese Methode soll eine Sicht auf die Straßen des im Parameter spezifizierten Spielers zurückgeben.

- **Methode `addRoad` (3 Punkte):**
  Diese Methode soll eine neue Straße zwischen den beiden angegebenen `TilePosition`s (erster und zweiter Parameter) hinzufügen. Die Straße darf nur hinzugefügt werden, wenn an dieser Position noch keine andere Straße gebaut wurde und wenn der platzierende Spieler / Besitzer (dritter Parameter) durch eine andere Straße angrenzt. Wenn der vierte Parameter (`checkVillages`) `true` ist, reicht es zu prüfen, ob der platzierende Spieler eine an diese Kante angrenzende Siedlung besitzt. Zuletzt gibt die Methode `true` zurück, wenn die Straße erfolgreich hinzugefügt wurde und `false` in allen anderen Fällen.

Nun implementieren Sie die folgenden Methoden im Record `EdgeImpl`:

- **Methode `getIntersections` (2 Punkte):**
  Diese Methode soll die beiden Kreuzungen zurückgeben, die durch die Kante verbunden sind.

  > **Hinweis:**
  >
  > Suchen Sie in `TilePosition` und `HexGrid` nach sinnvollen Hilfsmethoden.

- **Methode `connectsTo` (1 Punkt):**
  Diese Methode erhält als Parameter eine andere Kante. Die Methode gibt genau dann `true` zurück, wenn die beiden Kanten über eine gemeinsame Kreuzung miteinander verbunden sind. Sie können dafür die Rückgabe der Methode `getIntersections` verwenden.

- **Methode `getConnectedRoads` (2 Punkte):**
  Diese Methode gibt die Menge von Straßen zurück, die an eine der beiden Kreuzungen anliegen und zu einem gegebenen Spieler gehören. Sie können dafür die Rückgabe der Methode `getConnectedEdges` verwenden.

H1.4: Ein Dach über dem Kopf ?? Projektpunkte

Jetzt, da Spieler existieren, brauchen diese auch ein Dach über dem Kopf, damit ihnen nicht kalt wird. Aus diesem Grund wollen wir die Methoden für Siedlungen, `placeVillage` und `upgradeSettlement`, in `IntersectionImpl` vervollständigen.

Siedlungen werden durch die Klasse `Settlement` modelliert. Sie werden unterschieden in Dörfer (`Settlement.Type.VILLAGE`) und Städte (`Settlement.Type.CITY`). Die Klasse `IntersectionImpl` besitzt ein Objektattribut `settlement`, welches ein Objekt vom Typ `Settlement` speichert, wenn auf dieser Kreuzung eine Siedlung gebaut wurde. Implementieren Sie die folgenden Methoden in der Klasse `IntersectionImpl`:

- **Methode `placeVillage` (2 Punkte):**
  Diese Methode platziert ein Dorf auf der aktuellen Kreuzung, wenn diese nicht belegt ist und wenn der übergebene Spieler die Kreuzung durch seine Straßen erreichen kann. Wenn der zweite Parameter (`ignoreRoadCheck`)

`true` ist, muss der aktive Spieler keine angrenzende Straße besitzen. Sie gibt `true` zurück, wenn eine Siedlung erfolgreich platziert wurde und `false` in allen anderen Fällen.

> **Hinweis:**
>
> Nutzen Sie die Methode `playerHasConnectedRoad` aus der Klasse `IntersectionImpl`.

- **Methode `upgradeSettlement` (1 Punkt):**
  Diese Methode wird verwendet, um existierende Siedlungen in Städte umzuwandeln. Sie ersetzt das gespeicherte `Settlement`-Objekt mit einem entsprechend initialisierten Objekt, wenn sich auf der aktuellen Kreuzung ein Dorf im Besitz des übergebenen Spielers befindet. Des Weiteren gibt sie `true` zurück, wenn das Dorf erfolgreich ausgebaut wurde und `false` wenn nicht.

H2: Implementierung des Controllers ?? Projektpunkte

In H1 haben wir uns mit der Implementation des Modells beschäftigt. Nun wenden wir uns der Implementierung des Controllers zu. Hierfür verwenden wir die von Ihnen vervollständigten Methoden aus dem Modell und nutzen diese für die Methoden aus dem Controller.

Im Endeffekt wollen wir im Controller diese Zugübersicht implementieren:

`https://www.catan.de/sites/default/files/2021-06/CATAN_DasSpiel_zuguebersicht.pdf`

Machen Sie sich mit den dort aufgeführten Regeln vertraut.

Da im Controller auf Aktionen der Spieler gewartet wird, läuft der Game Loop in einem anderen Thread als das GUI. Da Threading in der FOP nicht wirklich behandelt wird, müssen Sie die folgende technische Erklärung nicht vollständig verstehen, um die Aufgaben zu bearbeiten. Dennoch möchten wir Ihnen die Funktionsweise des Game Loops kurz erläutern.

Zunächst wird im `GameController` ein Zustand vorbereitet, in dem der Spieler eine Aktion ausführen kann. Bevor der Spieler nun seine Aktion ausführen kann, wird das Ziel des Spielers (im Folgenden Objective genannt) im zugehörigen `PlayerController` gesetzt. Im Enum `PlayerObjective` sind die möglichen Ziele des Spielers zusammen mit den dazugehörigen Aktionen definiert. Der `GameController` wartet nun mit der Methode `waitForNextAction` auf eine gültige Aktion des Spielers. Dieser Methode kann man auch direkt eine `PlayerObjective` übergeben, um beides zu kombinieren. Wenn die Aktion ausgeführt wurde, wird das Objective zurückgesetzt und der nächste Zustand wird vorbereitet.

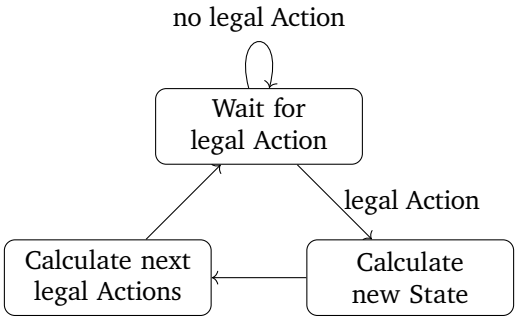
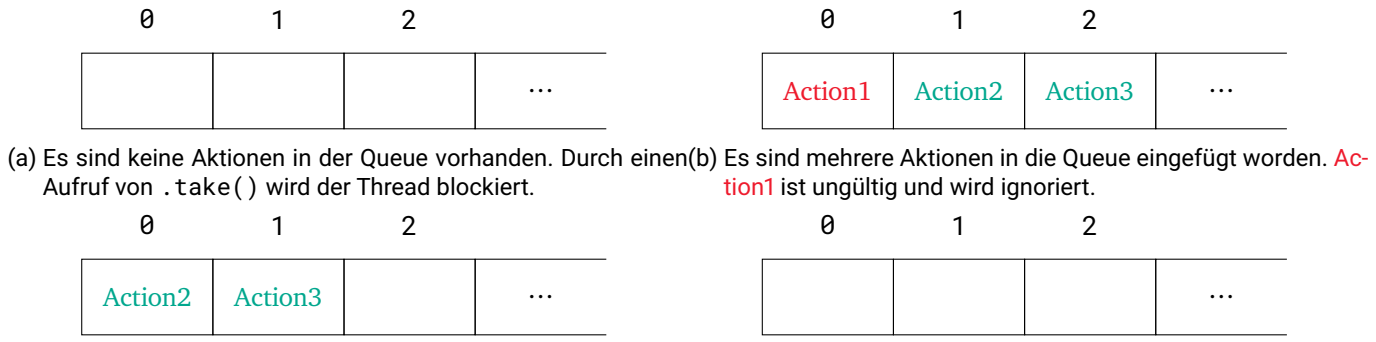


Abbildung 16: Idee des Game Loop

Nun zur Erklärung der Methode `waitForNextAction`: Diese Methode nutzt die im `PlayerController` vorhandene `BlockingDequeue`, um auf eine gültige Aktion des Spielers zu warten. Die `BlockingDequeue` hat eine Methode `take`, die den Thread blockiert, bis ein Element in der Queue vorhanden ist, und dieses dann zurückgibt.

Bei einem Aufruf von `waitForNextAction` wird also zunächst der Thread blockiert, bis eine gültige Aktion in die Queue eingefügt wird. Sobald eine Aktion in die Queue eingefügt wird, wird der Thread wieder freigegeben, und die Aktion von `waitForNextAction` geprüft. Falls die Aktion ungültig ist, wird wieder auf die nächste Aktion gewartet, solange bis eine gültige Aktion in der Queue vorhanden ist. Es können auch mehrere Aktionen in der Queue vorhanden sein, hierbei werden die Aktionen in der Reihenfolge abgearbeitet, in der sie in die Queue eingefügt wurden.



(a) Es sind keine Aktionen in der Queue vorhanden. Durch einen Aufruf von `.take()` wird der Thread blockiert.

(b) Es sind mehrere Aktionen in die Queue eingefügt worden. Action1 ist ungültig und wird ignoriert.

(c) Nun wird Action2 von `waitForNextAction` geprüft und als gültig erkannt. Die Aktion wird ausgeführt

(d) Alle verbleibenden Aktionen in der Queue werden verworfen, da die Aktion abgeschlossen ist.

Abbildung 17: BlockingDequeue eines `PlayerControllers` wird zum warten auf Aktionen verwendet (interne Logik der Methode `waitForNextAction`).

Falls ihnen die Logik des Game Loops nicht klar ist, können Sie sich an die gegebenen Tests halten, die die Funktionalität des Game Loops überprüfen. Wichtig ist, dass Sie verstehen, dass der `GameController` die Aufgabe hat, den Game Loop zu steuern und auf Aktionen der Spieler zu warten, der `PlayerController` hingegen die Aufgabe hat, die Aktionen der Spieler zu verarbeiten. Die Methoden im `PlayerController` werden indirekt durch die Actions aufgerufen, wodurch garantiert wird, dass nur gültige Aktionen ausgeführt werden können.

Anmerkung:

Die Logik zum Blockieren und Warten auf Aktionen der Spieler wird von Ihnen nicht implementiert. Sie müssen lediglich die Methoden implementieren, die die Aktionen der Spieler verarbeiten. Bitte blocken Sie den Thread nicht selbst und verwenden Sie die von uns bereitgestellten Methoden.

Verbindliche Anforderungen:

- Bei jeder Methode, die den aktiven Spieler für eine Aktion ändern soll, wird die Methode `withActivePlayer` verwendet, die den aktiven Spieler ändert und nach Beendigung der Methode den übergebenen `PlayerController` wieder auf den Zustand `PlayerObjective.IDLE` und den aktiven Spieler auf `null` setzt.
- Für das Warten auf eine Aktion des `PlayerControllers` wird die Methode `waitForNextAction` verwendet, die auf eine Aktion des übergebenen `PlayerControllers` wartet und diese ausführt, falls sie erlaubt ist. Die Rückgabe dieser Methode ist die Aktion, die der Spieler ausgeführt hat.

Hinweis:

`withActivePlayer` soll immer dann genutzt werden, wenn Sie den aktiven Spieler für die Dauer des übergebenen `Runnables` überschreiben möchten. Das `Runnable` wartet nahezu immer an irgendeiner Stelle auf eine Aktion des Spielers. Die Methode garantiert, dass der aktive Spieler immer korrekt gesetzt ist, wenn ein Spieler eine Aktion ausführen soll und stellt sicher, dass der Spieler danach keine Aktionen mehr ausführen kann und kein Spieler mehr aktiv ist.

H2.1: Ready. Set. Go. ?? Projektpunkte

Nun gehen Sie die Implementierung der Logik des Game Loops an. Hierfür verwalten wir wie in der Einleitung beschrieben, die Objectives der `PlayerController` und warten jeweils auf die Aktionen der Spieler. Implementieren Sie die folgenden Methoden in der Klasse `GameController`:

- **Methode `firstRound` (2 Punkte):**
  Beim Beginn des Spiels platziert jeder Spieler 2 Siedlungen und Straßen auf dem Spielfeld. Dafür implementieren Sie die Methode `firstRound()`. Diese Methode soll den Spielstart modellieren. Hierbei platzieren die Spieler ihre ersten Siedlungen und Straßen auf dem Spielfeld. Die Methode soll über die einzelnen Spieler iterieren und für jeden Spieler der am Zug ist, diesen nacheinander auf Aktionen der Zustände `PlayerObjective.PLACE_VILLAGE`, `PlayerObjective.PLACE_ROAD` warten lassen.

- **Methode `regularTurn` (2 Punkte):**
  Diese Methode soll einen regulären Zug modellieren. Hierbei kann davon ausgegangen werden, dass der aktive Spieler bereits korrekt gesetzt wurde. Dazu wird so oft auf eine Aktion des `PlayerControler`s des aktiven Spielers im Zustand `PlayerObjective.REGULAR_TURN` gewartet, bis eine Aktion vom Typ `EndTurnAction` ausgeführt wird. Das heißt nicht, dass Sie direkt eine Aktion ausführen müssen, sondern Sie warten bis die ausgeführte Aktion der gewünschten Aktion entspricht. Das Ausführen einer Aktion wird bereits von der Methode `waitForNextAction` übernommen.

- **Methode `diceRollSeven` (2 Punkte):**
  Diese Methode soll das Ereignis modellieren, wenn beim Würfeln eine „7“ fällt. Zunächst wird durch alle Spieler iteriert. Wenn ein Spieler insgesamt mehr als 7 Rohstoffe besitzt, so muss dieser die Hälfte seiner Karten abgeben. Dazu wird auf die nächste Aktion des Typs `PlayerObjective.DROP_CARDS` gewartet. Nachdem alle Spieler, die es mussten, ihre Karten abgegeben haben, wird der ursprüngliche aktive Spieler wiederhergestellt. Dieser soll nun abschließend nacheinander Aktionen der Zustände `PlayerObjective.SELECT_ROBBER_TILE` und `PlayerObjective.SELECT_CARD_TO_STEAL` ausführen.

H2.2: Die Würfel sind gefallen…, wer bekommt nun Rohstoffe? ?? Projektpunkte

Wie im Spielekonzept beschrieben, werden Ressourcen verteilt, wenn der zu würfelnde Spieler keine „7“ geworfen hat. Dann werden im Spiel die Ressourcen an diejenigen Spieler verteilt, die an dem jeweiligen Feld ein Dorf oder eine Stadt besitzen.

Implementieren Sie die Methode `distributeResources` der Klasse `GameController` für die Verteilung von Ressourcen basierend auf der geworfenen Zahl. Ermitteln Sie die entsprechenden `Tile`s des Spielfelds basierend auf dem übergebenen `int diceRoll`. Dann überprüfen Sie, ob auf angrenzenden `Intersection`s dieses Felds Dörfer oder Städte gebaut sind. Falls dies zutrifft, fügen Sie den Besitzern die entsprechenden Ressourcen zu.

H2.3: Jetzt wird gehandelt… ?? Projektpunkte

Nachdem wir den Game Loop im `GameController` implementiert haben, wenden wir uns nun dem `PlayerController` zu. In dieser Aufgabe soll der Handel zwischen Spielern sowie der Handel mit der Bank ermöglicht werden. Implementieren Sie dafür die folgenden Methoden in der Klasse `PlayerController`:

- **Methode `acceptTradeOffer` (3 Punkte):**
  Die Methode soll ein Handelsangebot eines anderen Spielers annehmen, sofern dieses existiert. Dazu hat `PlayerController` die Attribute `Player tradingPlayer`, `Map<ResourceType, Integer> playerTradingOffer` und `Map<ResourceType, Integer> playerTradingRequest`. Zunächst muss überprüft werden, ob ein Handelsangebot existiert. Falls nicht, so wird eine `IllegalActionException` mit der Nachricht `"No trade offer to accept"` geworfen. Wenn der Übergebene `boolean accepted` `false` ist, so wird der Handel abgelehnt. Dazu wird die Objective auf `PlayerObjective.IDLE` gesetzt und die Methode wird abgebrochen. Ansonsten wird der Handel durchgeführt und die entsprechenden Ressourcen zwischen den Spielern übertragen. Dazu muss zunächst überprüft werden, ob beide Spieler genügend Ressourcen für den Handel haben. Falls nicht, wird eine `IllegalActionException` mit der Nachricht `"Player does not have the requested resources"`, bzw. `"Other player does not have the offered resources"` geworfen, ohne den Zustand der Spieler zu verändern. Andernfalls wird der Handel durchgeführt und die entspre-

chenden Ressourcen zwischen den Spielern übertragen. Abschließend wird die Objective des aktuellen Spielers auf `PlayerObjective.IDLE` gesetzt.

- **Methode `tradeWithBank` (3 Punkte):**
  Die Methode erwartet die Parameter `ResourceType offeredType`, `int offeredAmount` und `ResourceType requestedType`. Überprüfen Sie zunächst das Handelsverhältnis des Spielers für den angeforderten Ressourcentyp. Falls dieses nicht der angebotenen Menge entspricht oder der Spieler nicht über genügend Ressourcen für den Handel verfügt, werfen Sie eine `IllegalActionException` mit einer aussagekräftigen Fehlermeldung, ohne den Zustand des Spielers zu verändern. Falls genügend Ressourcen zur Verfügung stehen, werden die angebotenen Ressourcen vom Spielerkonto entfernt und *eine* Ressource des angefragten Typs dem Spielerkonto hinzugefügt. Ein Aufruf dieser Methode soll nur einen Handel abbilden. Falls ein Spieler mehrere Ressourcen handeln möchte, muss der jeweilige Spieler mehrfach mit der Bank handeln.

Nun müssen wir noch die folgende Methode in der Klasse `GameController` implementieren:

- **Methode `offerTrade` (3 Punkte):**
  Diese Methode erwartet die Parameter `Player offeringPlayer`, `Map<ResourceType, Integer> offer` und `Map<ResourceType, Integer> request`. Die Methode soll der Reihe nach das Handelsangebot des `offeringPlayer`s an alle anderen Spieler `player` senden. Dazu wird der Zustand des Spielers jeweils auf `PlayerObjective.ACCEPT_TRADE`, und die Attribute `tradingPlayer`, `playerTradingOffer` sowie `playerTradingRequest` entsprechend gesetzt, und nach der Aktion wieder zurückgesetzt. Sobald der Handel von einem Spieler angenommen wurde, wird der Handel durchgeführt und die entsprechenden Ressourcen zwischen den Spielern übertragen. Andernfalls wurde der Handel abgelehnt und die Methode wird fortgesetzt. Abschließend wird der aktive Spieler auf den `offeringPlayer` gesetzt.

> **Hinweis:**
>
> Um zu überprüfen, ob ein Spieler einen Handel angenommen hat, können Sie (ähnlich zu `regularTurn` in der H2.1) die `AcceptTradeAction` abfangen und daraus auslesen, ob der Handel angenommen wurde oder nicht.

H2.4: Baumeisterische Eskapaden: Errichte Dörfer und Straßen ?? Projektpunkte

Das Ziel dieser Aufgabe ist es, das Bauen der jeweiligen Strukturen zu ermöglichen. Implementieren Sie dafür die folgenden Methoden in der Klasse `PlayerController`:

- **Methode `canBuildVillage` (1 Punkte):**
  Diese Methode prüft, ob ein Spieler ein Dorf bauen kann. Sie gibt `true` zurück, wenn das aktuelle Ziel des Spielers ist ein Dorf zu platzieren (siehe `PlayerObjective`) oder wenn der Spieler die benötigten Ressourcen für den Bau eines Dorfes besitzt. Außerdem muss der Spieler ausreichende Siedlungen besitzen, die er bauen kann. Andernfalls gibt die Methode `false` zurück.

- **Methode `canBuildRoad` (1 Punkte):**
  Diese Methode prüft, ob ein Spieler eine Straße bauen kann. Sie gibt analog zu `canBuildVillage` `true` zurück, wenn das aktuelle Ziel des Spielers ist eine Straße zu platzieren oder wenn der Spieler die benötigten Ressourcen für den Bau einer Straße besitzt. Außerdem muss der Spieler ausreichende Straßen besitzen, die er bauen kann. Andernfalls gibt die Methode `false` zurück.

- **Methode `buildVillage` (2 Punkte):**
  Diese Methode akzeptiert eine Kreuzung (`Intersection`) als Parameter. Zuerst wird geprüft, ob der Spieler berechtigt ist, ein Dorf zu errichten. Falls nicht, behandelt die Methode diesen Fall mit einer `IllegalActionException`. Andernfalls platziert sie ein Dorf an der angegebenen Kreuzung. Die Bedingung, dass mindestens eine Straße zur Kreuzung führen muss, kann ignoriert werden, wenn der Spieler sich in der ersten Runde befindet. Wenn kein Dorf gebaut werden kann, wird auch dieser

Fall mit einer `IllegalActionException` behandelt. Abschließend überprüft die Methode, ähnlich wie `canBuildVillage`, ob der Spieler aktuell das Ziel hat ein Dorf zu platzieren. Falls dies nicht der Fall ist zieht die Methode die benötigten Ressourcen für den Bau des Dorfes vom Spieler ab.

- **Methode `buildRoad` (2 Punkte):**
  Diese Methode nimmt als Parameter zwei `TilePosition`. Sie prüft zuerst, ob der Spieler eine Straße bauen darf. Falls nicht, wirft die Methode eine `IllegalActionException`. Beim Setzen der Straße achten Sie darauf, dass die Prüfung auf eine angrenzende Siedlung nur in der ersten Runde gemacht werden soll. Wenn keine Straße gebaut werden kann wird eine `IllegalActionException` geworfen. Abschließend prüft die Methode, wie in `canBuildRoad`, ob der Spieler aktuell das Ziel hat eine Straße zu platzieren. Falls dies nicht der Fall ist, zieht die Methode die benötigten Ressourcen für den Bau einer Straße vom Spieler ab.

Hinweis:

Stellen Sie sicher, dass Ihre Fehlermeldungen klar und verständlich formuliert sind, um eine einfache Fehlerbehandlung und Fehlerbehebung zu ermöglichen.

Mit der Methode `isFirstRound` kann überprüft werden, ob sich der Spieler gerade in der ersten Runde befindet.

Die für den Bau benötigten Ressourcen sind in `Config.SETTLEMENT_BUILDING_COST` und `Config.ROAD_BUILDING_COST` gespeichert.

H2.5: Urbanisierung: Vom beschaulichen Dorf zur aufstrebenden Stadt ?? Projektpunkte

Spieler haben die Möglichkeit, ihr Dorf in eine Stadt zu entwickeln.

Implementieren Sie dafür die Methode `upgradeVillage` in der Klasse `PlayerController`. Die Methode nimmt als Parameter eine Kreuzung (`Intersection`) an. Zuerst wird überprüft, ob der Spieler berechtigt ist ein Dorf zu einer Stadt zu entwickeln, falls dies nicht der Fall ist werfen Sie eine `IllegalActionException`. Andernfalls entwickeln Sie das Dorf an der angegebenen Kreuzung zu einer Stadt, wenn dies nicht möglich ist werfen Sie eine `IllegalActionException`.

Falls alle Bedingungen erfüllt sind, ziehen Sie die benötigten Ressourcen vom Spieler ab.

Hinweis:

Die benötigten Ressourcen für die Entwicklung eines Dorfes zu einer Stadt sind in `Config.SETTLEMENT_BUILDING_COST` gespeichert.

H3: Implementierung der View ?? Projektpunkte

Nachdem ein funktionierender GameLoop implementiert wurde, konzentrieren Sie sich in dieser Aufgabenreihe auf das Graphical User Interface. Der `GameController` und `PlayerController` kommunizieren mit dem GUI über JavaFX Properties. Die dafür benötige Logik haben wir Ihnen bereitgestellt, um mögliche Thread Probleme zu vermeiden.

Die verschiedenen Views, welche das GUI erstellen, sind im Package `projekt.view` zu finden und als JavaFX `Builder` implementiert. Sie müssen nicht genauer verstehen, wie diese funktionieren, da Sie diese nicht implementieren müssen. Jeder Builder, mit dem interagiert werden kann, hat seinen eigenen Controller im Package `projekt.controller.gui`. Diese stellen die Kommunikation zwischen GUI und GameLoop sicher.

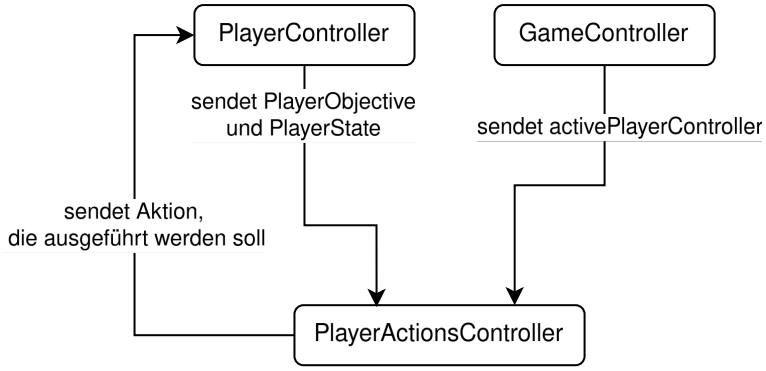


Abbildung 18: Eine Abstrakte Darstellung der Kommunikation zwischen den Controllern.

Damit das GUI weiß, welcher Spieler gerade am Zug ist gibt es das `activePlayerControllerProperty`, welches den aktuell am Zug befindlichen `PlayerController` beinhaltet. Außerdem werden, über den `PlayerState`, einige sonstige Informationen übermittelt, wie zum Beispiel, welche Kreuzungen und Kanten aktuell bebaut werden können oder wie viele Karten der Spieler auswählen soll.

Zusätzlich wird noch über das `playerObjectiveProperty` mitgeteilt, welches Ziel der Spieler gerade hat. Das `PlayerObjective` enthält auch Informationen darüber, welche Aktionen ausgeführt werden dürfen. Diese erhält man mit `PlayerObjective.getAllowedActions`. Die `allowedActions` sind Class Objekte, diese erhält man mit `{Klassenname}.class`.

Um beispielsweise die Aktion `SelectCardsAction` auszuführen, wenn es eine der erlaubten Aktionen ist, könnte man folgenden Code schreiben:

```
1 if (getPlayerObjective().getAllowedActions().contains(SelectCardsAction.class))
  ↪ {
2     getPlayerController().triggerAction(new
      ↪ SelectCardsAction(Map.of(ResourceType.WOOD, 1)));
3 }
```

Zunächst implementieren Sie einige Methoden im `PlayerActionsController`, danach geht es an die Views selbst.

Hinweis:

Um auf das aktuelle `PlayerObjective` und den aktuellen `PlayerState` zuzugreifen, nutzen Sie bitte die Methoden `getPlayerObjective()` und `getPlayerState()`.

H3.1: Was darf wo gebaut werden? ?? Projektpunkte

Als Erstes bestimmen wir, ob und was der aktuelle Spieler bauen darf.

Hierfür implementieren Sie die Methoden `updateBuildVillageButtonState`, `updateUpgradeVillageButtonState` und `updateBuildRoadButtonState`. Diese Methoden aktivieren bzw. deaktivieren die Buttons für die jeweiligen Aktionen entsprechend der in der Javadoc angegebenen Bedingungen. Zum aktivieren und deaktivieren von Buttons nutzen Sie die Methoden `builder.enable*` bzw. `builder.disable*` (* steht jeweils für das Element welches Sie ansteuern wollen).

Daraufhin müssen noch die Methoden `buildVillageButtonAction`, `upgradeVillageButtonAction` und `buildRoadButtonAction` implementiert werden. Diese Methoden werden aufgerufen, wenn der entsprechende Button gedrückt wird. Ziel ist es, die für die jeweilige Aktion relevanten Kreuzungen oder Kanten hervorzuheben und diesen eine Funktion zu geben, welche aufgerufen wird, wenn eine Kreuzung oder Kante angeklickt wird.

Mit der `HexGridController` Instanz können Sie die Controller für jede `Intersection` bzw. `Edge` erhalten. Welche Kreuzungen oder Kanten bebaut werden können, entnehmen Sie dem `PlayerState`. Die Controller haben eine Methode `highlight(Consumer<MouseEvent> handler)`, welche das jeweilige Element hervorhebt und den übergebenen Handler aufruft, wenn das Element angeklickt wird. Wenn die Kreuzung oder Kante angeklickt wird, sollte `PlayerController.triggerAction` verwendet werden, um die entsprechende `PlayerAction` aufzurufen, um die Straße/das Dorf/die Stadt zu bauen.

Nachdem eine Bauaktion ausgeführt wurde, müssen alle Hervorhebungen wieder entfernt werden, dazu kann `removeAllHighlights` verwendet werden und das UI aktualisiert werden.

Auch hier ist wieder alles genau in der Javadoc spezifiziert.

Es gibt für jede korrekt implementierte Methode einen Punkt.

Verbindliche Anforderung:

Um die Aktion auszuführen, müssen Sie `PlayerController.triggerAction` nutzen. Sie dürfen auf keine anderen Methoden des `PlayerController`s zugreifen.

Hinweis:

Schauen Sie sich an, welche Informationen dem Controller im `PlayerState` übermittelt werden.

Sie können die Methode `buildActionWrapper` nutzen, um zu garantieren, dass das UI aktualisiert wird.

Beispielweise kann `buildActionWrapper` folgendermaßen genutzt werden:

```
1 buildActionWrapper(event -> System.out.println("Hello World!"))
```

H3.2: Wann darf ich was machen? ?? Projektpunkte

Nun steht die Implementierung der Funktionalität an, die unsere GUI entsprechend der jeweiligen Spieleraktion aktualisiert. Dafür soll die Methode `updateUIBasedOnObjective` in der Klasse `PlayerActionsController` implementiert werden.

Die Methode `updateUIBasedOnObjective` aktualisiert die Benutzeroberfläche basierend auf dem übergebenen Spielerziel (`PlayerObjective`). Zuerst werden alle Hervorhebungen entfernt, dann werden die Kanten, Kreuzungen

und Tiles neu gezeichnet, und schließlich werden alle Buttons deaktiviert. Die Spielerinformationen werden ebenfalls mit `updatePlayerInformation()` aktualisiert.

Falls der aktuelle Spieler eine KI ist soll weiter nichts passieren.

Daraufhin werden je nach den erlaubten Aktionen bzw. des übergebenen Spielerziels, bestimmte Buttons aktiviert und Dialoge angezeigt. An dieser Stelle nutzen Sie die zuvor in H3.1 implementierten Methoden um zu entscheiden ob die Buttons zum bauen aktiviert werden. Für die Buttons zum kaufen und ausspielen von Entwicklungskarten gibt es solche Methoden bereits. Die Buttons zum Würfeln, Handeln und Runde beenden haben keine besonderen Bedingungen und können direkt über den `PlayerActionsBuilder` aktiviert werden.

Aktionen welche keine entsprechenden Buttons haben (zum Beispiel: `StealCardAction`), präsentieren dem Spieler einen Dialog oder heben bestimmte GUI Elemente hervor. Diese Funktionalität ist bereits implementiert. Sie müssen nur die folgenden Funktionen den passenden Aktionen zu ordnen und eventuelle Parameter entsprechend setzen:

- **`selectRobberTileAction`:**
  Diese Methode repräsentiert eine Action, welche ausgeführt wird, wenn ein Tile ausgewählt wird. Ähnlich zum hervorheben von Kreuzungen und Kanten in der H3.1 sollen Sie hier die Methode `getHexGridController().highlightTiles` nutzen.
- **`selectCardToStealAction`**
- **`selectResources`:**
  Diese Methode erwartet als Parameter einen `int` welcher angibt, wie viele Karten ausgewählt werden sollen.
- **`acceptTradeOffer`**

> **Hinweis:**
>
> Alle Aktionen können im Package `projekt.controller.actions` gefunden werden.

H3.3: Spiel erstellen — ?? Projektpunkte

In dieser Aufgabe sollen Sie den `CreateGameBuilder` im Package `projekt.view.menus` vervollständigen

Der `CreateGameBuilder` hat eine `ObservableList`, die für jeden Player einen `PlayerImpl.Builder` enthält.

Implementieren Sie die folgenden Methoden:

- **Methode `removePlayer`:**
  Diese Methode löscht den Player mit der übergebenen Id aus der Liste aller Player und aktualisiert die Ids aller verbleibenden Player, so dass alle Ids wieder aufeinander folgen.
- **Methode `createAddPlayerButton`:**
  Diese Methode gibt einen Button zurück, über den ein Player eingefügt werden kann. Nutzen Sie `nextPlayerBuilder`, um den `PlayerBuilder` für den nächsten Player zu erhalten.
- **Methode `createRemovePlayerButton`:**
  Diese Methode gibt einen Button zurück, mit dem der Player mit der übergebenen Id entfernt werden kann.
- **Methode `createPlayerColorPicker`:**
  Diese Methode erstellt einen ColorPicker[1], mit dem die Farbe des Players gewählt werden kann. Als Standardwert

[1] `https://openjfx.io/javadoc/21/javafx.controls/javafx/scene/control/ColorPicker.html`

soll `PlayerImpl.Builder.getColor()` verwendet werden. Um die Farbe des Players zu setzen nutzen Sie `playerBuilder.color`. Falls 2 Player die gleiche Farbe haben, soll ein Fehler ausgegeben werden und die Farbe nicht gesetzt werden.

- **Methode `createBotOrPlayerSelector`:**
  Diese Methode gibt eine Node zurück, die es ermöglicht auszuwählen, ob ein Player ein Bot ist oder nicht. Hierfür können Sie eine `CheckBox`[2] verwenden, dürfen aber auch gerne kreativ werden. Um zu prüfen ob ein Player aktuell als KI eingestellt ist oder nicht können Sie `playerBuilder.isAI` verwenden. Zum setzen des KI Statuses kann `playerBuilder.ai` verwendet werden. Alternativ können Sie `playerBuilder.aiProperty` nutzen um Änderungen am KI Status zu übernehmen.

Nun müssen noch die einzelnen Buttons, durch die Methode `initCenter()`, im Menü platziert werden. Der Button zum Hinzufügen von Spielern wird in die `mainBox` platziert und die restlichen Schaltflächen in die `playerListVBox`.

Jede implementierte Methode gibt einen Punkt und das Anzeigen der Schaltflächen im Menü einen weiteren.

Hinweis:

Sie können `Alert`[a] verwenden, um Fehler anzuzeigen.

Wenn Sie möchten, können Sie den Buttons Icons in Unicode hinzufügen. Es stehen Ihnen fast alle Icons der MaterialDesignIcons-Bibliothek[b] zur Verfügung. Mit Hilfe von `String.format` können Sie Unicode-Zeichen in einen String einfügen.

Beispiel: `String.format("%c ein Hexagon!", 0xF02D8)` fügt ein Hexagon[c] in den String ein.

[a] `https://openjfx.io/javadoc/21/javafx.controls/javafx/scene/control/Alert.html`
[b] `https://pictogrammers.com/library/mdi/`
[c] `https://pictogrammers.com/library/mdi/icon/hexagon/`

H3.4: Karten auswählen ?? Projektpunkte

Zum Abschluss implementieren Sie den Dialog, der angezeigt wird, wenn ein Spieler Karten auswählen soll. Dieser befindet sich in der Klasse `SelectResourcesDialog` im Package `projekt.view.gameControls`.

Der Dialog soll dem Spieler jede auswählbare Ressource anzeigen und eine Eingabe, über welche bestimmt werden kann, wie viel der jeweiligen Ressource der Spieler wählen möchte. Ist der Parameter `resourcesToSelectFrom` `null`, kann aus allen Ressourcen ausgewählt werden. Falls der Parameter `dropCards` `true` ist, soll dem Spieler mitgeteilt werden, dass die ausgewählten Karten abgeworfen werden.

Es soll nicht möglich sein den Dialog zu schließen, bevor der Spieler ausreichend Ressourcen gewählt hat. Außerdem soll angezeigt werden, wie viele Ressourcen noch ausgewählt werden müssen.

In JavaFX haben Dialoge einen Rückgabetyp, dieser Dialog soll eine `Map` von `ResourceType` nach `Integer` zurückgeben, die bestimmt, welche Ressource und wie viel von jeder Ressource der Spieler ausgewählt hat.

Damit der Dialog beim Schließen den richtigen Rückgabewert liefert, muss die Methode `setResultConverter` aufgerufen werden. Diese erwartet als Parameter ein Lambda. Dieses erhält den buttonType, mit dem der Dialog geschlossen wurde. Das Lambda gibt dann die `Map` zurück, die den Rückgabewert des Dialogs darstellt.

Die genaue Implementierung und Gestaltung ist Ihnen überlassen.

Punkteverteilung:

[2] `https://openjfx.io/javadoc/21/javafx.controls/javafx/scene/control/CheckBox.html`

- Für jede wählbare Ressource gibt es eine Eingabemöglichkeit: 4 Punkte
- Jede wählbare Ressource wird dargestellt: 4 Punkte
- Der Dialog kann erst geschlossen werden, wenn die korrekte Menge an Ressourcen ausgewählt wurde: 1 Punkt
- Der Dialog gibt die ausgewählten Ressourcen korrekt zurück, wenn er geschlossen wird: 1 Punkt

Hinweis:

Mit der Klasse `ResourceCardPane` kann eine Ressource als farbige Karte mit Icon dargestellt werden.

Um die einzelnen Elemente schön anzuordnen, können Sie beispielsweise die Klassen `GridPane`[a], `VBox`[b] oder `HBox`[c] aus der JavaFX-Bibliothek verwenden.

Mehr Informationen über JavaFX Dialoge finden Sie in der Dokumentation[d].

[a] `https://openjfx.io/javadoc/21/javafx.graphics/javafx/scene/layout/GridPane.html`
[b] `https://openjfx.io/javadoc/21/javafx.graphics/javafx/scene/layout/VBox.html`
[c] `https://openjfx.io/javadoc/21/javafx.graphics/javafx/scene/layout/HBox.html`
[d] `https://openjfx.io/javadoc/21/javafx.controls/javafx/scene/control/Dialog.html`

H4: Weiterführende Aufgaben ?? Projektpunkte

In den folgenden Aufgaben entwickeln Sie das Spiel weiter und gestalten es nach Ihren Vorstellungen. Dabei dürfen Sie den Quellcode des Spiels beliebig verändern und erweitern. Bitte beachten Sie dabei, dass Sie Klassen und Methoden mit der `@DoNotTouch`-Annotation nicht verändern dürfen, da diese für die Tests benötigt werden. Sie dürfen allerdings auch in solchen Klassen neue Methoden, Attribute, Enum-Konstanten, etc. hinzufügen, sofern die bestehende Funktionalität erhalten bleibt (testen Sie dazu am Besten ihre Implementierung mit den Public-Tests, sobald diese verfügbar sind). Lesen Sie sich die Javadoc-Dokumentation der Klassen und Methoden sorgfältig durch, um zu verstehen, wie Sie die Funktionalität des Spiels erweitern können. Begründen und dokumentieren Sie ihre Änderungen und Erweiterungen in einer PDF-Datei, die sie in `src/main/resources/documentation` unter dem Namen `documentation.pdf` ablegen. Die Dokumentation sollte für die Tutoren, also externe Leser, leicht nachvollziehbar und verständlich sein. Kennzeichnen Sie in der PDF eindeutig die Aufgabennummer. Sie können die Dokumentation auf Deutsch oder Englisch schreiben.

LaTeX-Vorlage

Falls Sie weitere Dateien zur Dokumentation benötigen, legen Sie diese bitte ebenfalls in `src/main/resources/documentation` ab. Beachten Sie die maximale Dateigröße von 20 MB. Falls Sie ihre Dokumentation mit LaTeX schreiben wollen, haben wir ihnen eine Vorlage in Moodle und eine Vorlage in Sharelatex bereitgestellt. Bitte Beachten Sie, dass Sie von unseren Tutoren keinerlei Unterstützung bei der Verwendung oder installation von LaTeX erhalten werden.

> **Hinweis:**
>
> Die Aufgaben der H4 sind sehr frei gehalten und bieten Ihnen die Möglichkeit, Ihre eigenen Ideen und Kreativität einzubringen. Je nach Komplexität ihrer Änderungen und Erweiterungen kann der Aufwand stark variieren. Sie müssen hier keine eigene Game Engine schreiben, solange die Änderungen und Erweiterungen sinnvoll und gut dokumentiert sind. Fehlende Dokumentation oder unverständliche Implementierungen können zu Punktabzügen führen.

H4.1: Weiterentwicklung des GUIs ?? Projektpunkte

In dieser Aufgabe werden Sie das Graphical User Interface (GUI) weiterentwickeln. Dies soll fundiert auf aktuellen wissenschaftlichen Ergebnissen aus Teilbereichen der Informatik, Psychologie und Wirtschaft geschehen (Stichworte: Usability, Design). Sie finden dazu in Moodle einen kleinen Leitfaden zum Thema Benutzerfreundlichkeit. Gehen Sie die Punkte durch und beachten Sie diese bei der Gestaltung der Oberfläche. Dokumentieren Sie in Ihrer PDF am Ende alle Erweiterungen und Änderungen an der Benutzeroberfläche und begründen Sie diese Änderungen hinsichtlich der Benutzerfreundlichkeit. Die Dokumentation sollte für die Tutoren, also externe Leser, leicht nachvollziehbar und verständlich sein. Kennzeichnen Sie in der PDF eindeutig die Aufgabennummer.

Sie können zusätzlich dazu auch eigene Literatur und Best-Practice-Beispiele suchen und einbringen, dokumentieren Sie diese Recherche und Ihre Ergebnisse ebenfalls. Bringen Sie das Wissen und Ihre Expertise aus Ihrem persönlichen Studiengang ein, um in Ihrer Gruppe ein möglichst stimmiges Gesamtergebnis zu erzeugen.

Nun zur eigentlichen Aufgabe: Implementieren Sie mindestens **fünf** Verbesserungen oder Erweiterungen des GUIs (z.B. ein View, der anzeigt, wie viel Gebäudestrukturen kosten, einen Dark Mode, bessere Buttons, etc.). Für jede Verbesserung oder Erweiterung gibt es einen Punkt.

H4.2: Neuer Rohstoff ?? Projektpunkte

Rohstoffe sind ein Schlüsselbestandteil dieses Spiels. Diese ermöglichen das Bauen von Strukturen und den Kauf von Entwicklungskarten. Im aktuellen Zustand des Spiels stehen 5 verschiedene Rohstofftypen zur Verfügung. Diese Rohstoffe sind Holz, Lehm, Wolle, Getreide und Eisen.

Fügen Sie nun mindestens einen neuen Rohstoff hinzu. Stellen Sie sicher, dass die Spieler die Möglichkeit haben über eine Landschaft und über den Handel mit der Bank an diesen Rohstoff zu kommen. Überlegen Sie sich ein sinnvolles Handelsverhältnis und dokumentieren und begründen Sie Ihre Entscheidungen.

Punkteverteilung:

- Der Neue Rohstoff wurde in das `RessourceType` Enum hinzugefügt: 1 Punkt
- Es wurde ein neuer `Tile.Type` hinzugefügt, welches den neuen Rohstoff produziert. Tiles dieses Typen taucht dann auch auf der `HexGrid` auf (passen Sie dazu die Datei `tile_ratios.properties` in `src/main/resources` an).: 1 Punkt
- Es gibt einen Hafen, der einen günstigen Handel mit dem neuen Rohstoff ermöglicht: 1 Punkt
- Der neue Rohstoff hat ein Icon und eine Farbe: 1 Punkt

H4.3: Neue Gebäudestruktur ?? Projektpunkte

Im Spiel existieren 2 Gebäudestrukturen, die Siedlung und die Stadt. Für den Bau jede dieser Strukturen werden bestimmte Rohstoffe benötigt. Diese Strukturen geben auch jeweils eine unterschiedliche Anzahl an Rohstoffen und an Siegpunkten.

Führen Sie nun mindestens eine neue Gebäudestruktur ein. Denken Sie sich vernünftige Kosten für den Bau der neuen Struktur aus. (Hier bietet sich auch der neue Rohstoff aus H4.2 an.)

Der Besitz einer neuen Struktur muss sich für die Spieler auch rentieren. Überlegen Sie sich auch hier einen Mehrwert für diese Struktur. Diese sollte die Spieler motivieren, diese zu bauen, um das Spiel zu gewinnen.

Beachten Sie jedoch, dass dies nicht der einzige Weg zum Sieg sein sollte. Dokumentieren Sie Ihre Entscheidungen und begründen Sie diese.

Punkteverteilung:

- Die neue Gebäudestruktur wurde dem `Settlement.Type` Enum hinzugefügt: 1 Punkt
- Die Kosten wurden in `Config.SETTLEMENT_BUILDING_COST` hinzugefügt: 1 Punkt
- Im `PlayerController` wurden mindestens die Methoden zum Bau, und zum überprüfen ob gebaut werden kann, implementiert: 1 Punkt
- Die neue Gebäudestruktur hat ein Icon: 1 Punkt
- Die neue Gebäudestruktur bereichert das Spiel und ist nicht zu stark oder zu schwach (wenn es z.B. nur einen Smaragd o.Ä. kostet, und gleich 20 Siegpunkte gibt ist das nicht sinnvoll): 1 Punkt

H4.4: Neue Entwicklungskarte ?? Projektpunkte

Ihre Aufgabe ist es, eine neue Entwicklungskarte zu implementieren.

Sie können frei entscheiden, was für eine Aktion diese Karte ausführen soll. Achten Sie allerdings darauf, dass es den Spielfluss nicht stört, sondern eher bereichert.

Integrieren Sie dann die neue Entwicklungskarte in den Spielablauf. Stellen Sie sicher, dass Spieler diese Karte für einen angemessenen Preis erwerben können.

Punkteverteilung:

- Die neue Entwicklungskarte wurde dem `DevelopmentCardType` Enum hinzugefügt: 1 Punkt
- Die Kosten für den Kauf der neuen Entwicklungskarte wurden in `Config.DEVELOPMENT_CARD_COST` hinzugefügt: 1 Punkt
- Die neue Entwicklungskarte hat ein Icon: 1 Punkt
- Die neue Entwicklungskarte bereichert das Spiel und ist nicht zu stark oder zu schwach: 1 Punkt
- Je nach Komplexität der Entwicklungskarte kann hier noch ein weiterer Punkt vergeben werden: 1 Punkt

H4.5: Neue Spielmechanik ?? Projektpunkte

In dieser Aufgabe sollen Sie eine neue Spielmechanik entwerfen und implementieren. Die Idee ist es, eine innovative Erweiterung zu schaffen, die das Spielerlebnis bereichert und neue strategische Elemente einführt.

Beginnen Sie damit, sich Gedanken über eine interessante Spielmechanik zu machen. Zum Beispiel könnten Sie eine Mechanik entwickeln, bei der nach dem Wurf eines Sechser-Pasches dreimal hintereinander ein besonderes Ereignis ausgelöst wird. Dieses Ereignis könnte eine Herausforderung oder Belohnung für die Spieler darstellen, die das Spiel auf unerwartete Weise beeinflusst.

Punkteverteilung:

- Die neue Spielmechanik wurde in das Spiel integriert und ist verständlich dokumentiert: 1 Punkt
- Die neue Spielmechanik bereichert das Spiel und ist nicht zu stark oder zu schwach: 1 Punkt
- Je nach Komplexität der Mechanik können hier noch bis zu drei weitere Punkte vergeben werden.: 1-3 Punkte

H4.6: AI als Gegner? ?? Projektpunkte

Nun soll das Spiel auch noch um einen Einzelspielermodus erweitert werden. Ein Computergegner wird erstellt, indem man in „Abbildung CreateGameBuilder“ auf „AI“ umstellt. Ihre KI muss die abstrakte Klasse `projekt.controller.AiController` erweitern.

In der Methode `initPlayerControllers` des `GameController` können Sie den `BasicAiController` durch Ihre KI ersetzen.

Punkteverteilung:

- Die Strategie des Computergegners ist gut dokumentiert und sinnvoll: 2 Punkt
- Der Computergegner ist implementiert und kann über das Menü ausgewählt werden: 1 Punkt
- Der Computergegner führt nur erlaubte Aktionen aus: 1 Punkt
- Wenn man zwei Computergegner gegeneinander spielen lässt, gewinnt einer der beiden: 1 Punkt

Hinweis:

Schauen Sie sich an, wie der `BasicAiController` implementiert wurde, um eine Idee zu bekommen, wie Sie Ihre eigene KI erstellen können.