Funktionale und objektorientierte Programmierkonzepte Wintersemester 19/20

TECHNISCHE UNIVERSITÄT DARMSTADT

Abgabedatum: 27.03.2020
Gesamtpunktzahl: 36

Projekt - v1.0

Carcassonne

Abbildung 1: Das Vorbild Carcassonne in der Luftaufnahme [1]

Gesamtleitung: Prof. Karsten Weihe

Organisation und Aufgaben: Lukas Roehrig, Alexander Mainka

Implementation: Yannis Illies, Ayman El Ouariachi

Autor: Ayman El Ouariachi, Paul Hahn

Inhaltsverzeichnis

1 Organisation und Information

1.1 Grundlegende Informationen und Bonus

Das FOP-Projekt ist zwar nicht verpflichtend und auch für die formale Prüfungszulassung (Studienleistung) ausdrücklich **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.
Neben den Bonuspunkten der Hausübungen können Sie sich im Projekt weitere Bonuspunkte für die Klausur erarbeiten. Im Projekt können Sie zusätzlich 54 Punkte erreichen, welche am Ende durch 1.5 geteilt und auf die nächstkleinere, ganze Zahl abgerundet werden. Zusammenfassend können Sie also bis zu 36 Punkte zusätzlich für Ihr Punktekonto erarbeiten.
Sie werden das Abschlussprojekt in Teams von **genau** vier Personen bearbeiten.

1.2 Zeitplan im Überblick

- 14.02.20 - 18 Uhr: Deadline zur Anmeldung
- 17.02.20 - 28.02.20: verbindliche Teamtrainings der HDA
- 17.02.20 - 27.03.20 Poolsprechstunden
- 27.03.20 - 23.55 Uhr: Abgabe des Projekts

1.3 Poolsprechstunden

Für eventuell anfallende Fragen bieten die Projekttutoren vom 17.02. - 27.03 Sprechstunden an. Eine Übersicht über die Sprechstundentermine finden Sie im Projektabschnitt des zugehörigen **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. Diese werden natürlich auch außerhalb der Poolsprechstunden beantwortet.

1.5 Anmeldung (bis. 14.02.)

Um sich anzumelden, bilden Sie Gruppen aus **genau** 4 Studierenden. Sollten Sie noch keine Gruppe haben, so steht Ihnen im Projektabschnitt des **moodle**-Kurses ein Forum

zur Gruppenfindung zur Verfügung.
Bei der Eintragung in die Gruppen wählen Sie gleichzeitig Ihren Termin für das Teamtraining der HDA aus. Alle Gruppenmitglieder müssen sich **manuell** in die Gruppen eintragen.

1.6 HDA-Teamtraining

Die Teamtrainings liegen im Zeitraum vom 17. Februar bis zum 28. Februar und werden für jede Gruppe rund 2 Stunden dauern. Dort nehmen Sie mit Ihrer gesamten Gruppe und voraussichtlich 3 weiteren Gruppen teil. Bei Abwesenheit ist den Projekttutoren ein ärztliches Attest vorzulegen.
Fehlen Sie unentschuldigt bei Ihrem Teamtraining, erhalten Sie automatisch **keine** Punkte für das Projekt.

1.7 Abgabe des Projekts

Das Projekt ist bis zum **27.3.20 um 23.55 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 Ihrer Gruppe das gesamte Projekt ab.
Neben dem Code geben Sie zusätzlich eine PDF-Datei ab, diese soll sich ebenfalls im *ZIP*-Archiv befinden. In der PDF finden sich zum einen die, für unser Verständnis notwendigen, Dokumentationen der weiterführenden Aufgaben sowie die Lösungen der Theorieaufgaben.
Achtung:
Nachdem eines Ihrer Teammitglieder das Projekt auf **moodle** hochgeladen hat, müssen alle anderen Teammitglieder diese Aufgabe im entsprechenden Modul bestätigen. 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, da Ihre Teammitglieder genügend Zeit haben müssen, um die Abgabe zu bestätigen.

1.8 Plagiarismus

Selbstverständlich gelten die gleichen Regelungen zum Plagiarismus wie auch bei den Hausübungsabgaben!
Daher 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 den Anlass für einen Verdacht geben, 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.

1.9 Inhaltliche Informationen zum Projekt

Im Laufe des Projekts werden Sie eine Implementation des beliebten Legespiels **„Carcassonne“** erarbeiten und diese ergänzen. Dazu wird Ihnen eine Vorlage zur Verfügung gestellt, in welcher Sie, je nach Aufgabenstellung, Code ergänzen oder erweitern müssen, damit eine lauffähige Version des Legespiels 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 Legespiels und wir freuen uns auf die, hoffentlich gelungenen, Implementationen.

2 Spielkonzept Carcassonne

Im Legespiel „Carcassonne“ erstellen Sie zusammen mit Ihren Gegenspielern durch Zusammenfügen einzelner Landschaftsbilder eine Landkarte. Durch sogenannte „Meeple“ können Sie verschiedene Landschaften für sich beanspruchen und somit Punkte erwirtschaften. Am Ende des Spiels gewinnt der Spieler mit der höchsten Gesamtpunktzahl.

2.1 Spielmaterial

Das Legespiel „Carcassonne“ besteht aus 72 Legekarten, auf denen unterschiedliche, miteinander kombinierte Landschaften abgebildet sind. Diese können zeigen:

- Wiesenabschnitt
- Stadtabschnitt
- Kloster
- Straßenabschnitt
- Kreuzung

Abbildung 2: Beispielhafte Legekarte mit einem Stadtabschnitt, einem Straßenabschnitt und zwei Wiesenabschnitten [2]

An dem Spiel können 5 Spieler teilnehmen und jeder erhält 8 sogenannte „Meeple“, eine Mischung aus „my + people“, die auf eine Landschaft gestellt werden können. Näheres dazu in Abschnitt 2.2.
Zuletzt gibt es noch eine speziell konzipierte Startkarte, an die der erste Spieler zu Beginn anlegen wird.

2.2 Spielablauf

2.2.1 Start

Die Legekarten befinden sich umgedreht auf einem Stapel auf dem Tisch. Die Startkarte liegt offen auf dem Spielfeld. Es wird im Uhrzeigersinn gespielt und jeder Spieler erhält seine „Meeple“ in der gewählten Farbe.

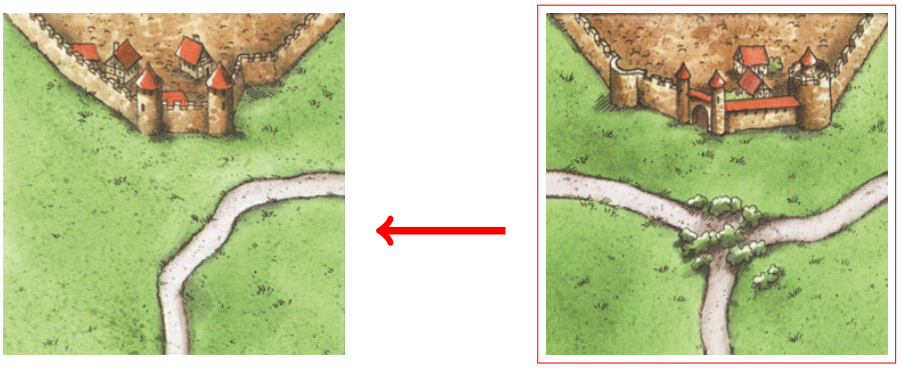


Abbildung 3: Anlegen der rotmarkierten Karte ist zulässig, da die Straße und die Wiese fortgesetzt werden [2]

2.2.2 Aktionen

Der Spieler zieht eine Karte vom Stapel und zeigt diese jedem anderen Spieler. Dann legt er diese an einer bereits gespielte Karte an. Es wird immer mindestens an eine Seite einer Karte angelegt. Dabei muss die Landschaft, mit der angelegt wird, die gleiche abbilden, wie die, an der angelegt wird. Dies bedeutet, dass Wiesenabschnitte, Stadtabschnitte und Straßenabschnitte fortgesetzt und eventuell fertiggestellt werden. Gleichaussehende Landschaften können auch wie in Abbildung 2 durch eine andere Landschaft getrennt sein. So gibt es auf der Abbildung 2 zwei Wiesenstücke, die durch eine Straße getrennt sind. Hat der Spieler eine Karte gelegt, so hat er die Möglichkeit einen seiner „Meeple“ auf eine der Landschaften der soeben gelegten Karte zu stellen. Ein anderer Spieler darf deshalb keinen seiner „Meeple“ auf diese Landschaft oder auf eine damit fortgesetzte Landschaft stellen. Landschaften, die „über eine Ecke “ verbunden sind, stellen also unabhängige Landschaften dar und dürfen besetzt werden. Somit stellen die Burgabschnitte in Abbildung 3 zwei unabhängige Landschaften dar und auf jede Karte darf ein „Meeple“ gestellt werden. Ist eine Landschaft abgeschlossen, bekommt der Spieler die Punkte, der mehr seiner „Meeple“ auf der Landschaft positioniert hat. Ist eine Landschaft fertiggestellt, werden die „Meeple “ von den zugehörigen Landschaftskarten genommen und die Punkte werden direkt ausgewertet und verrechnet.

2.3 Wertung während des Spiels

Der Spieler erhält während des Spiels die nachfolgend aufgelisteten Punkte nur, falls die Landschaft fertiggestellt ist.

- **Kloster**: Für die Karte und jede angrenzende Karte erhält der Spieler 1 Punkt. Ein Kloster gilt nur als fertiggestellt, falls es komplett von Landschaftskarten umgeben ist. Es bringt also 9 Punkte.
- **Stadt**: Für jeden Burgabschnitt, der zur abgeschlossenen Burg gehört, erhält der Spieler 2 Punkte. Für jedes *Wappen*, welches auf einem Burgabschnitt abgebildet ist, erhält der Spieler weitere 2 Punkte.
- **Straße**: Für jeden Straßenabschnitt erhält der Spieler 1 Punkt.

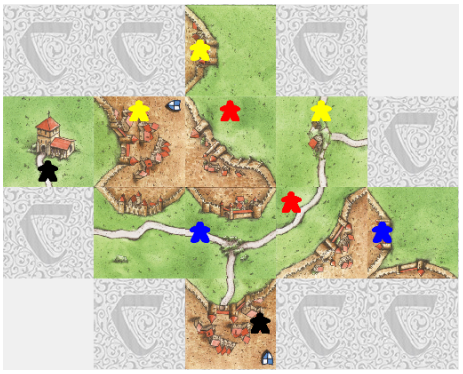


Abbildung 4: Beispielwertung nach Spielende

2.4 Wertung am Spielende

Unvollendete Landschaften, auf denen „Meeple“ stehen, bringen am Spielende dennoch Punkte:

- **Kloster**: Für die Karte und jede angrenzende Karte erhält der Spieler 1 Punkt.
- **Stadt**: Für jeden Burgabschnitt gibt es 1 Punkt. Jedes *Wappen* bringt 1 weiteren Punkt.
- **Straße**: Für jeden Straßenabschnitt erhält der Spieler 1 Punkt.
- **Wiese**: Für jeden Wiesenabschnitt erhält der Spieler 1 Punkt. Die Punktzahl für diese Wiesenlandschaft wird danach durch 4 geteilt. Dies ist ein wesentlicher Unterschied zur offiziellen Spielversion.

Demnach erhält der schwarze Spieler in Abbildung 4 für das Kloster 3 Punkte und nochmal 2 Punkte für seinen besetzten Burgabschnitt. Der gelbe Spieler 1 Punkt für die begonnene Straße und 6 Punkte für seine Burg. Der blaue Spieler erhält 2 Punkte für die Straße und nochmal 2 Punkte für seine besetzten Burgabschnitte. Zuletzt erhält der rote Spieler für seine Wiese 1 Punkt, da er insgesamt 6 Wiesenabschnitte besetzt hält.

2.5 Zusätzliche Informationen

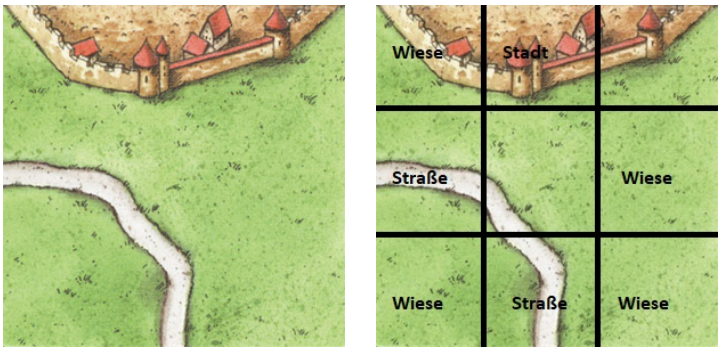
Weitere Informationen sowie die offizielle Spielanleitung erhalten Sie unter den nachfolgenden Links:

- `https://www.hans-im-glueck.de/_Resources/Persistent/3581b911bf5113e36646fe91e1f4b47ee03f67b0/CarcBB6_Rule_D_Final_2_Web.pdf`
- `https://de.wikipedia.org/wiki/Carcassonne_(Spiel)`
- `https://www.youtube.com/watch?v=JfnRm4wReJU`

3 Implementationsstruktur

3.1 Legekarte

Die Legekarten sind in der Datei *TileTypes.xml* beschrieben. Jede Karte wird, wie in Abbildung 5b zu erkennen, in ein 3×3 Gitter unterteilt, in welchem jede Teilfläche angibt, welche Landschaft zu sehen ist und ob ein „Meeple“ gesetzt werden kann.



(a) Legekarte mit einem Stadtabschnitt, einer Straße und zwei Wiesenabschnitten.

(b) Unterteilung der Legekarte in 9 gezeigte Landschaftsabschnitte.

Abbildung 5: Beispielhafte Legekarte und zugehöriges 3x3 Gitter

Dieses Gitter wird in der Implementation als Graph beschrieben, in welchem jeder Knoten einen Landschaftsteilabschnitt darstellt, der auf der Karte vorkommt, während jede Kante angibt, ob das Landschaftsstück mit einem anderen Landschaftsstück verbunden ist. Zu beachten ist jedoch, dass nicht jede Gitterfläche, wie in Abbildung 5b zu erkennen ist, einen Landschaftsteilabschnitt anzeigen muss, da manche Teilflächen keinen Mehrwert an Informationen bringen.

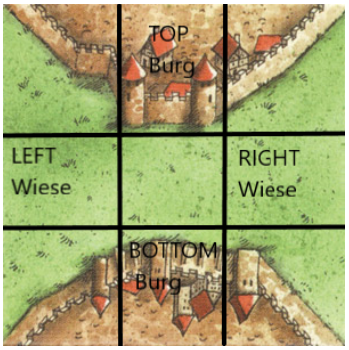


Abbildung 6: Notwendige Knoten der Legekarte *H*

Beim Durchlesen der Datei *TileTypes.xml* wird jedoch klar, dass die Knoten *TOP*, *LEFT*, *RIGHT* und *BOTTOM* immer einen Landschaftsteilabschnitt zeigen und somit für jede Legekarte existieren. Ein Beispiel dafür zeigt Abbildung 6. Zeigt eine Legekarte einen

Straßenabschnitt, so bedeutet dies, dass mindestens einer der vier notwendigen Knoten einen Straßenabschnitt repräsentiert. In diesem Fall werden auch die benachbarten Knoten wie in Abbildung 5b mit dem entsprechenden Landschaftsteilabschnitt initialisiert.

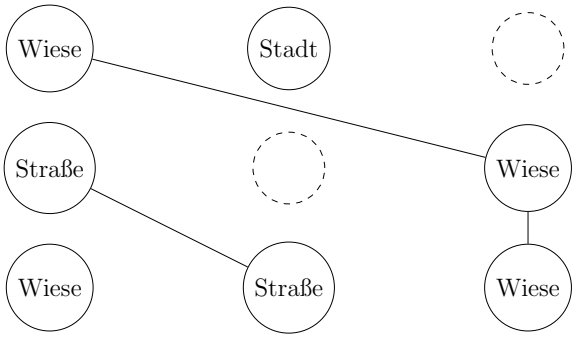


Abbildung 7: Graph der Legekarte aus Abbildung 5b

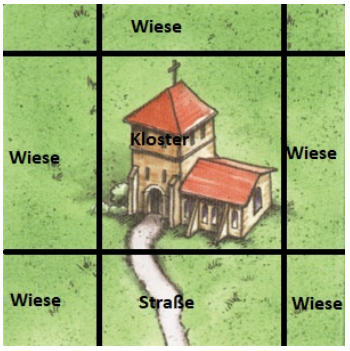
Wie man anhand Abbildung 5b und Abbildung 7 erkennt, muss zum Beispiel der Knoten mit der Position *LEFT* mit dem Knoten mit der Position *BOTTOM* verbunden sein, da der gleiche Straßenabschnitt zu sehen ist. Anhand dieser Abbildungen erkennt man zudem, dass die Knoten, welche einen Wiesenabschnitt oberhalb der Straße zeigen (also an den Positionen *TOPLEFT*, *RIGHT* und *BOTTOMRIGHT*), miteinander verbunden sein müssen. Der Graph aus Abbildung 7 kann auch mit der Hilfe von der XML-Darstellung der Karte aus Abbildung 8 nachvollzogen werden.

```
<tile type="K" ammount="3">
        <node feature="FIELDS" position="TOPLEFT"/>
        <node feature="CASTLE" position="TOP" meeplespot="true"/>
        <node feature="ROAD" position="LEFT" meeplespot="true"/>
        <node feature="FIELDS" position="RIGHT" meeplespot="true"/>
        <node feature="FIELDS" position="BOTTOMLEFT" meeplespot="true"/>
        <node feature="ROAD" position="BOTTOM"/>
        <node feature="FIELDS" position="BOTTOMRIGHT"/>
        <edge a="LEFT" b="BOTTOM"/>
        <edge a="TOPLEFT" b="RIGHT"/>
        <edge a="RIGHT" b="BOTTOMRIGHT"/>
</tile>
```

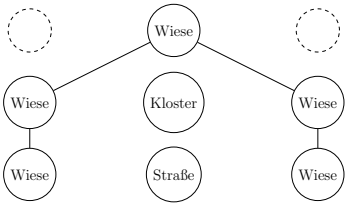
Abbildung 8: XML-Darstellung der Legekarte aus Abbildung 5b

Anhand der Abbildung 8 erkennt man zudem, auf welchen Knoten ein „Meeple“ gesetzt werden kann. So ist es zum Beispiel möglich, auf den Knoten mit der Position *TOP* eine seiner Figuren zu setzen.

(a) Legekarte mit Kloster



(b) Unterteilung der Legekarte in 9 gezeigte Landschaftsabschnitte



(c) Zugrunderliegender Graph der Legekarte aus Abbildung 9b

Abbildung 9: Beispielhafte Legekarte mit Kloster und zugehöriges 3x3 Gitter

Eine Ausnahme bildet der potenzielle Knoten *CENTER*. Dieser existiert nur, falls die entsprechende Legekarte ein Kloster zeigt. Ein entsprechendes Beispiel zeigt Abbildung 9.

3.2 Zulässige Legepositionen

Eine Karte darf an eine andere Karte angelegt werden, falls die zentralen Knoten der jeweiligen Anlegeseiten die gleiche Landschaft zeigen.
Beim Legen einer Spielkarte müssen aber alle Knoten der Seite, mit welcher angelegt wird, mit denen der bereits gespielten Karte durch Kanten verbunden werden. Dadurch entsteht im Laufe des Spiels ein sehr großer Graph mit vielen Zusammenhangskomponenten. Abbildung 10 zeigt eine beispielhafte Spielsituation und den dazugehörigen Graph in Abbildung 11.

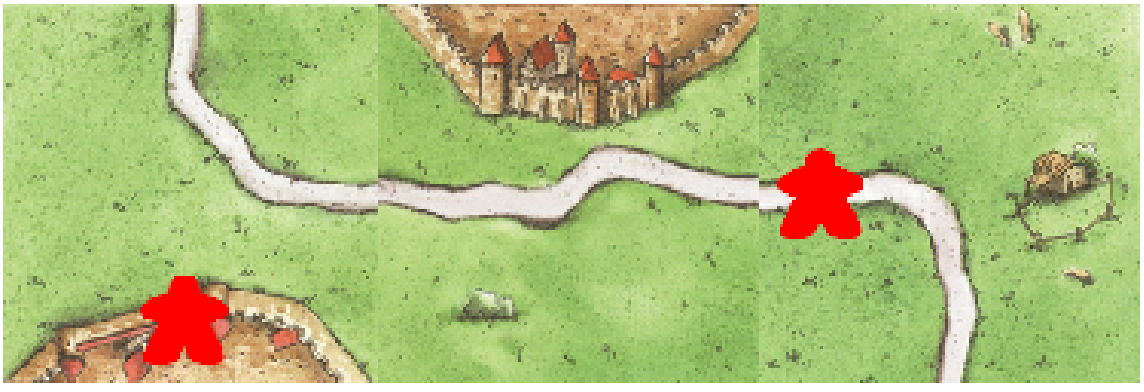


Abbildung 10: Spielsituation

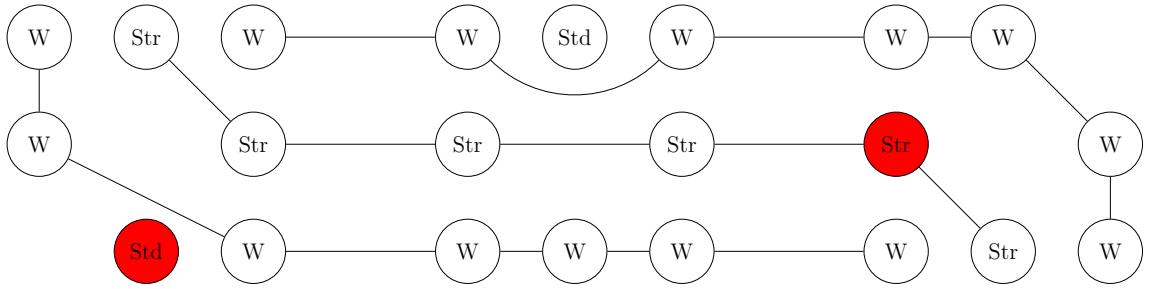


Abbildung 11: Graph der Spielsituation aus Abbildung 10

Jeder Knoten hat, wie in Abbildung 11 durch die Farbe gekennzeichnet, einen inneren Zustand in Form einer Objektvariable, welche anzeigt, ob ein „Meeple“ gesetzt werden kann und, falls ja, ob einer gesetzt ist. In Abbildung 11 kann man zudem gut die Straße aus Abbildung 10 wiedererkennen. Alle Straßenabschnitte sind miteinander verbunden und bilden eine Zusammenhangskomponente im Graph. Mit dieser Darstellung lassen sich die Spielpunkte leicht ausrechnen, da somit nur die Größe der Zusammenhangskomponete berechnet werden muss. Weitere Informationen dazu erhalten Sie in Abschnitt 3.4.

3.3 Implementation des Spielfeldes

Das Spielfeld ist eine 144×144-große Matrix, in dem die einzelnen Legekarten gespeichert werden. Wird also eine Karte an Position (i, j) mit $i, j < 144$ gelegt, so wird in der Matrix `Board` der Eintrag (i, j) gesetzt, und die Knoten der Karte werden mit den adjazenten Knoten in den Feldern $(i + 1, j), (i - 1, j), (i, j + 1), (i, j - 1)$ verbunden. Das Spielbrett ist als Objektvariable `Board` in der Klasse `Gameboard.java` implementiert. Die Klasse `Gameboard` enthält neben dem Spielfeld auch eine Objektvariable vom Typ `FeatureGraph`. Diese Variable speichert den Graph aus Abbildung 11. Näheres zur Implementation des Graphen erhalten Sie in Abschnitt 3.4.

3.4 Implementation des zugrundeliegenden Graphen

Im Package `fop.base` sind die Klassen `Graph`, `Node` und `Edge` zusammengefasst. Diese bilden das Grundgerüst für den Graphen aus Abbildung 11. Die Klasse `Graph` implementiert einen allgemeinen Graphen, indem ein Objekt der Klasse jeweils eine Liste an Objekten der Klasse `Node` und eine Liste an Objekten der Klasse `Edge` speichert. Die Klasse `Node` speichert im Allgemeinen einen inneren Zustand. In unserem Projekt speichert es den Landschaftsabschnitt, welchen er, wie in Abschnitt 3, repräsentiert. Die Klasse `Edge` speichert eine Kante zwischen zwei Knoten, indem sie den Startknoten und den Endknoten als Objektvariable speichert.
Von diesen Klassen leiten sich die folgenden Klassen ab:

- FeatureGraph
- FeatureNode
- FeatureEdge

Diese Klassen implementieren den eigentlichen Graphen.
Die Klassenstruktur und die verschiedenen Beziehungen zwischen Klassen werden mithilfe der **Unified Modeling Language** modelliert. Wie ein UML-Diagramm zu lesen ist, können Sie auf `https://de.wikipedia.org/wiki/Unified_Modeling_Language` nachlesen. Abbildung 12 zeigt das UML-Diagramm der verschiedenen Klassen zur Implementierung des Grundgerüsts des Spiels.

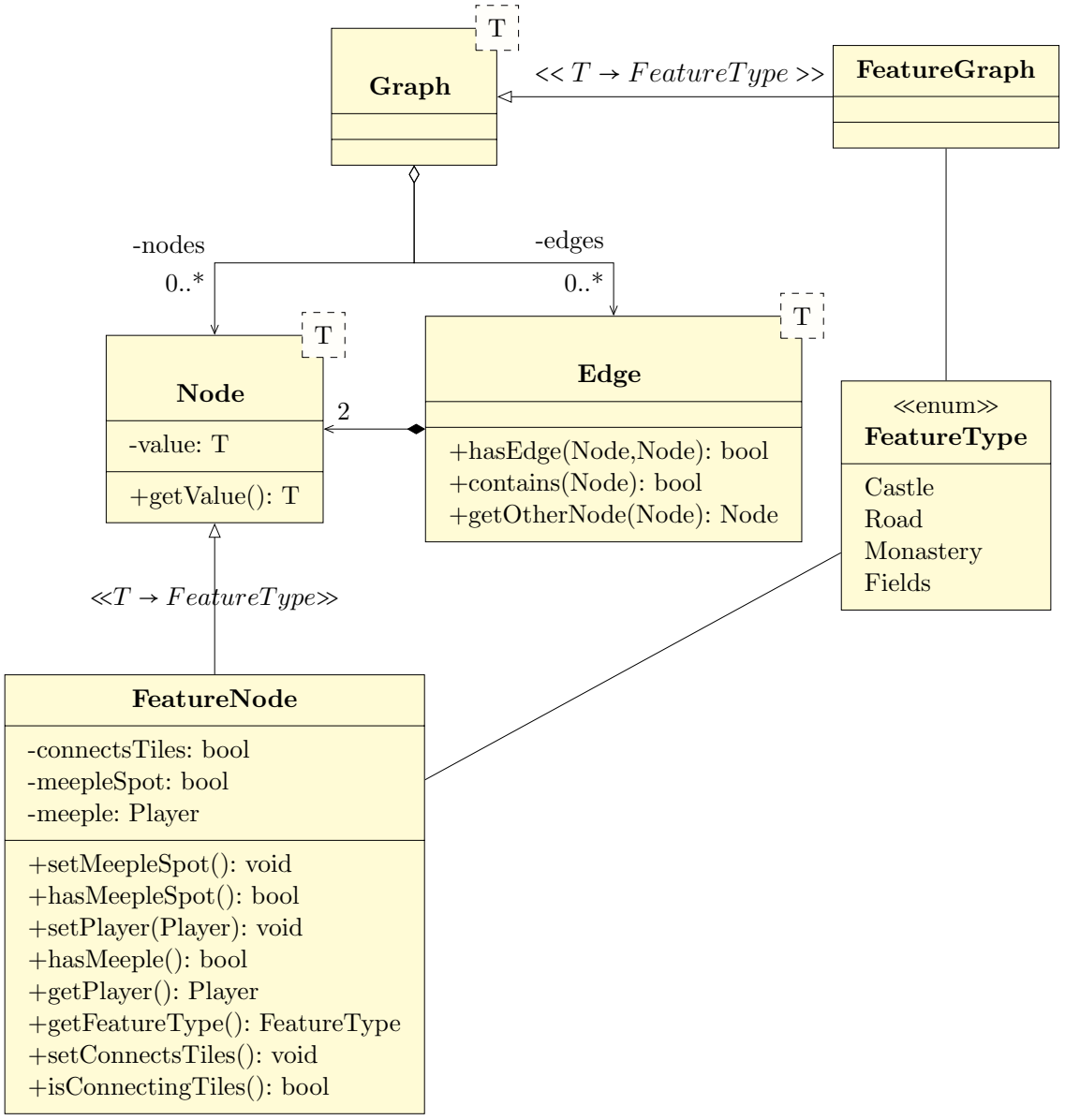


Abbildung 12: UML-Diagramm des Graphen

4 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 nachvollziehen können, unterteilt sich das Projekt in vier wichtige Komponenten. Eines davon ist das Package `fop.base` welches in Abschnitt 3 besprochen wurde. Die drei anderen `packages` sind:

- fop.model
- fop.view
- fop.controller

Das Package **fop.model** implementiert im Wesentlichen die grundlegenden Daten wie den Aufbau eines Spielers, einer Legekarte oder des Spielfeldes.
Das zweite Package **fop.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 **fop.controller** lenkt und verwaltet die beiden anderen Komponenten.

5 Graphical User Interface

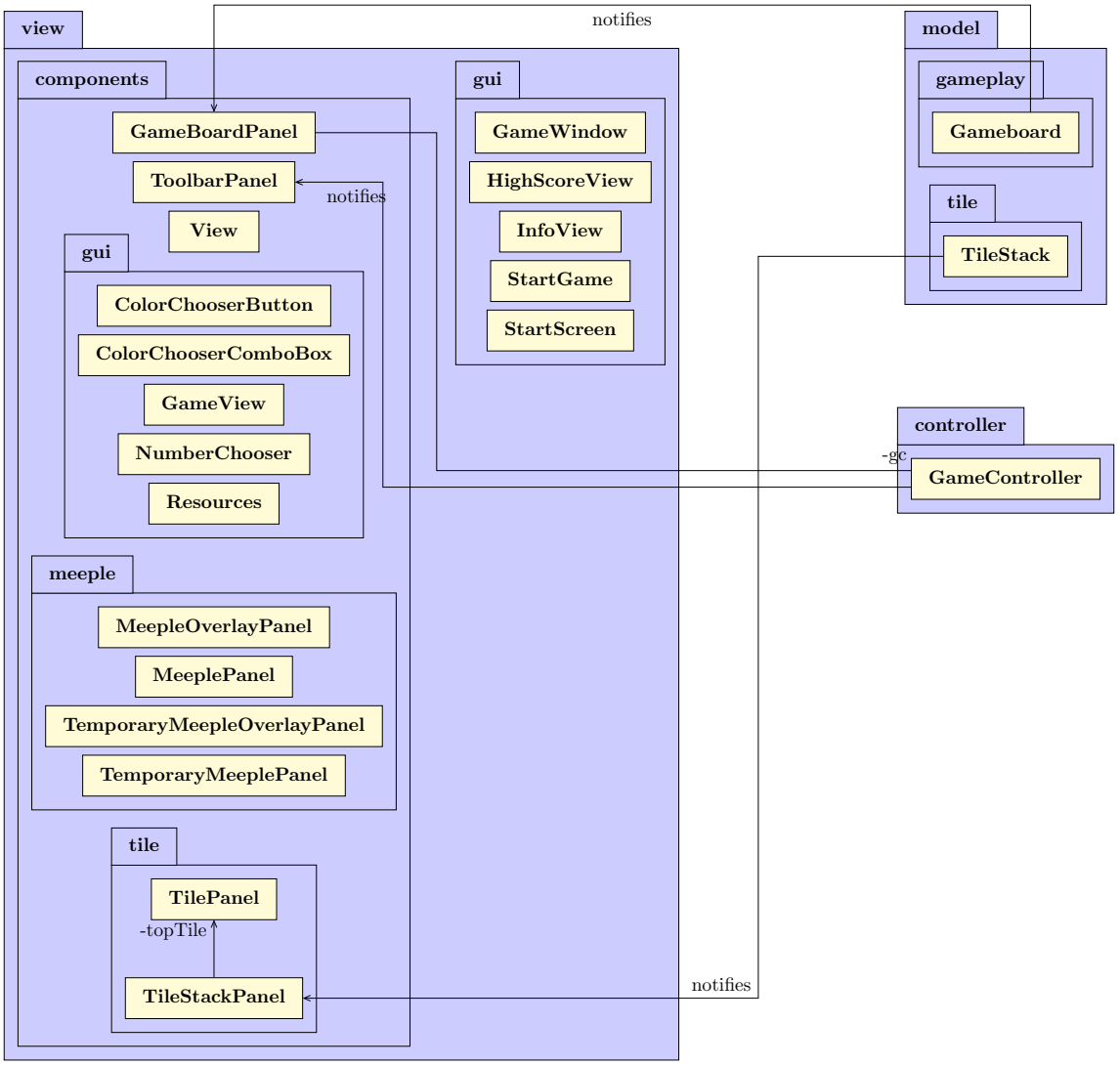


Abbildung 13: Vereinfachtes UML-Diagramm des Package **fop.view** und wichtige Beziehungen zwischen den Packages

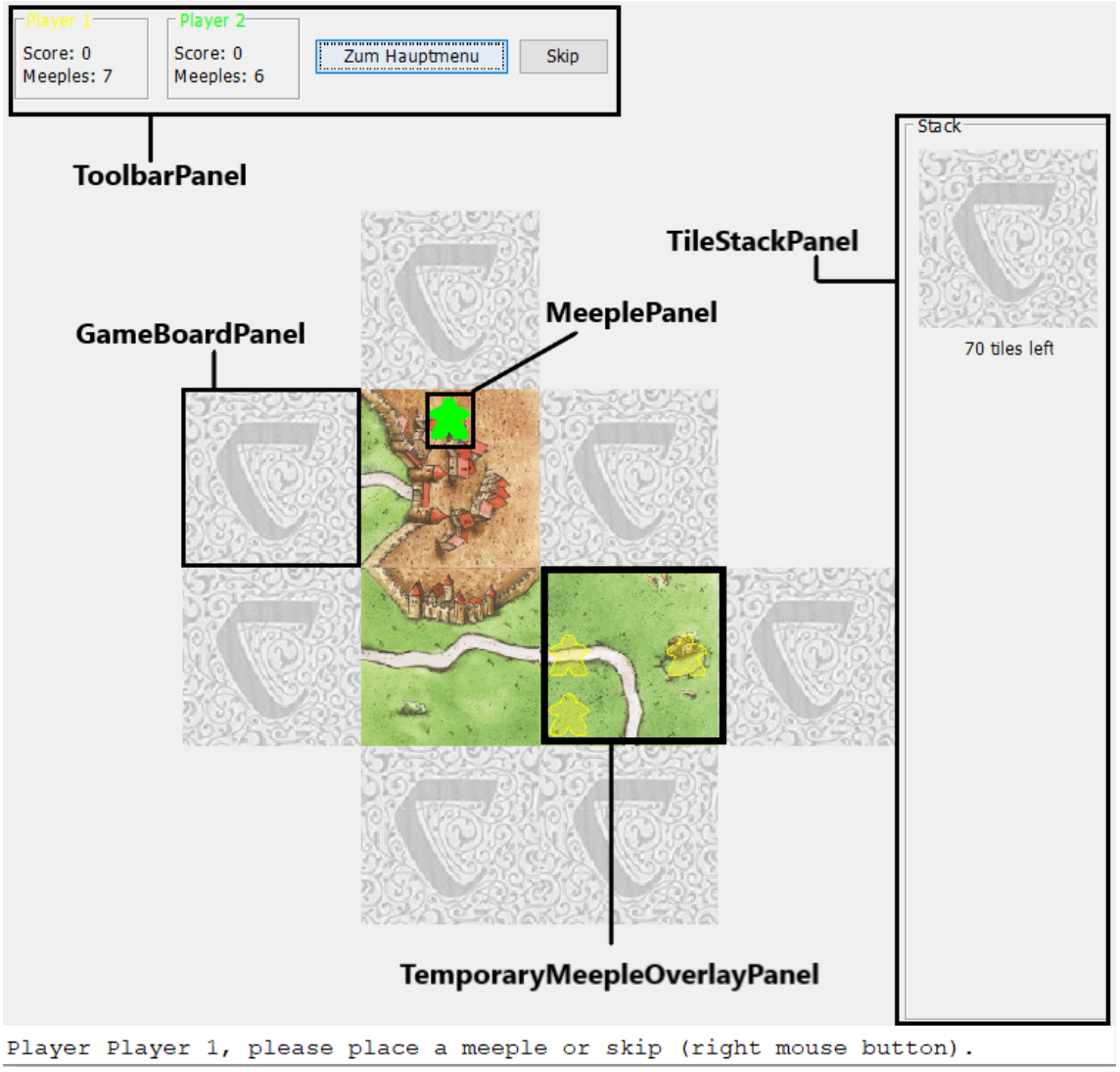


Abbildung 14: Korrespondenz zwischen GUI-Komponenten und Java-Klassen

6 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. Vergessen Sie nicht, diese gut zu dokumentieren, damit die Projekttutoren Ihre Implementationen nachvollziehen können. 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.

6.1 Der Graph (31 Punkte)

Wie bereits in Abschnitt 3 beschrieben, beschreibt der Graph die zugrundeliegende Struktur über die vorhandenen Landschaftsabschnitte und ihre Verbindungen untereinander. Die folgenden Aufgaben beziehen sich auf die Klasse `Gameboard` im Paket `fop.model.gameplay`.

6.1.1 Zulässige Legeposition — 3 Punkte

Für jede neu gezogene Legekarte t muss überprüft werden, ob man diese an eine bestimmte Stelle (x, y) im Spielfeld legen kann. Dazu benötigt man das Spielfeld aus Abschnitt 3.3 und den Graphen aus Abschnitt 3.4. Ergänzen Sie dazu die Methode `isTileAllowed(Tile t, int x, int y)`, welche `TRUE` zurückgibt, falls (x, y) eine valide Anlegestelle und `FALSE` zurückgibt, falls (x, y) eine ungültige Anlegestelle darstellt. Wie bereits in Abschnitt 3.4 beschrieben, kann eine Karte A an eine andere Karte B angelegt werden, falls die zentralen Knoten der jeweiligen Anlegeseiten den gleichen Landschaftsabschnitt zeigen.

6.1.2 Finden einer Legeposition — 5 Punkte

Nachdem Sie nun wissen, ob Sie an einer gegebenen Stelle eine Legekarte legen können, sollen Sie nun überprüfen, ob es **überhaupt** eine valide Anlegestelle auf dem gesamten Spielfeld gibt. Dazu wird die Methode `isTileAllowedAnywhere(Tile t)` in der Klasse `Gameboard` benötigt, welche jedoch von der bereits implementierten Methode `isTileAllowed` und der zu implementierenden Methode `rotateRight()` in der Klasse `Tile` aus dem Paket `fop.model.tile` abhängt.

- Die Legekarte kann natürlich auch gedreht werden, bevor sie angelegt werden kann. Implementieren Sie dazu die Methode `rotateRight`. Diese greift auf die `Map nodes` zu und rotiert die Knoten der Legekarte um 90° im Uhrzeigersinn. Dazu werden also die jeweiligen Knoten an die nächste Position im Uhrzeigersinn geschrieben. Speichern Sie zudem den Rotationsgrad der Legekarte ab. Beachten Sie dabei, dass eine Drehung um 360° gleichbedeutend mit einer Drehung um 0° ist.

- Nachdem Sie die Methode `rotateRight` implementiert haben, können Sie nun die Methode `isTileAllowedAnywhere(Tile t)` fertigstellen. Dazu müssen sie einmal über alle bereits gelegten Spielkarten iterieren und die vier möglichen freien Anlegestellen an die Karte auf dem Spielfeld überprüfen. Beachten Sie, dass Sie vielleicht die Legekarte t rotieren könnten und danach anlegen könnten.

6.1.3 Anlegen einer neuen Legekarte **5 Punkte**

Nun haben Sie die Voraussetzungen geschaffen, eine neue Legekarte an das Spielfeld anzulegen.
In Abschnitt 3.4 haben Sie bereits gelesen, dass Sie die Knoten der Spielkarten nach dem Legen auf das Spielfeld mit Kanten verbinden müssen, sonst können zum Beispiel Punkte nicht richtig berechnet werden. Implementieren Sie dazu die Methode `connectNodes(int x, int y)`.

1. Fügen Sie alle Knoten und Kanten der gelegten Spielkarte den entsprechenden Objektvariablen in der Klasse `Graph` hinzu.
2. Fügen Sie dem Graphen zusätzlich neue Kanten zwischen den Legekarten hinzu. Betrachten Sie dazu alle möglichen Nachbarn der gelegten Spielkarte und verbinden Sie, falls der jeweilige Nachbar existent ist, die zentralen Knoten der jeweiligen Anlegeseiten. Danach müssen zudem die beiden möglichen anderen Knoten der Anlegeseite mit den entsprechenden Knoten der anderen Anlegeseite der Legekarte verbunden werden. Beachten Sie dabei, dass die anderen Knoten der Anlegeseite nicht notwendigerweise vorhanden sind. Mehr dazu finden Sie in Abschnitt 3.1.

6.1.4 Punkteverteilung

Wie die Punkte berechnet werden, können Sie in Abschnitt 2.3 nachlesen.

Teil (a) **3 Punkte**

Zum Einstieg implementieren Sie die Berechnung der Punkte für die Landschaftskarten mit einem dargestellten Kloster.
Implementieren Sie dazu die Methode `calculateMonasteries(State state)`. Iterieren Sie dazu über das gesamte Spielfeld und suchen Sie nach Spielkarten, die einen Knoten mit dem `FeatureType` Kloster haben. Falls solch ein gefundener Knoten eine Spielfigur besitzt, dann werden die Punkte wie in 2.3 zusammengerechnet. Zum Schluss müssen noch die Punkte dem Spieler, der das Kloster „besetzt“, gutgeschrieben werden. Suchen Sie dafür nach geeigneten Methoden in den Klassen `FeatureNode` und `Player`. Da Sie über alle gespielten Legekarten iterieren, setzen Sie die Variable `score` nach jeder Berechnung zurück. Beachten Sie hierbei:

1. Während des Spiels werden die Punkte dem Spieler nur gutgeschrieben, wenn die Landschaft abgeschlossen ist. Nach dem Spielende werden auch die Punkte von un-

vollständigen Landschaften dem jeweiligen Spieler gutgeschrieben. Der Spielstatus kann mithilfe von der Abfrage `state == State.GAME_OVER` überprüft werden.

2. Falls der `state` nicht das Spielende bedeutet, also `state != State.GAME_OVER`, so werden die Spielfiguren wieder an den jeweiligen Spieler übergeben.

Teil (b) **10 Punkte**

Erweitern Sie nun auch die Methode `calculatePoints(FeatureType type, State state)`, sodass auch die Punkte für Burgen, Wiesen und Straßen berechnet werden. Beachten Sie, dass wir die Punkte für Wiesen anders berechnen (siehe dazu Abschnitt 2.3).
Dazu übergeben wir, wie in der Signatur angegeben, der Funktion den jeweiligen Landschaftstyp, für welchen wir die Punkte berechnen wollen, und den Status des Spiels, da die Punkte während und nach dem Spiel berechnet werden können.

1. Speichern Sie sich in einer Liste `nodeList` alle Knoten des Graphen, welche den Landschaftstyp `type` darstellen.
2. Sie müssen für jeden Knoten die entsprechende Legekarte finden, um die Kanten und damit die nächsten Knoten zu bekommen.
3. Danach traversieren Sie für jeden Knoten aus dieser Liste die daraus resultierende Zusammenhangskomponente. Dafür ist es notwendig, sich bereits besuchte Knoten in einer Liste zu speichern, damit Knoten nicht mehrfach gezählt werden.
4. Die Anzahl der Spielfiguren auf der Zusammenhangskomponente muss für jeden Spieler gespeichert werden, sodass die Punkte dem Spieler mit den meisten Spielfiguren gutgeschrieben werden können. Zudem müssen in einer Liste die Knoten, auf denen eine Spielfigur steht, gespeichert werden. Diese soll genutzt werden, um die Spielfiguren an die Spieler zurückzugeben und diese von den Knoten zu entfernen.
5. Ist jeder Knoten in der Zusammenhangskomponente besucht, addieren Sie die Punkte auf den Punktestand des entsprechenden Spielers. Unterscheiden Sie dabei die Punkteberechnung während des Spiels und nach Ablauf des Spiels. Betrachten Sie danach den nächsten Knoten aus der Liste `nodeList`
6. Erst wenn alle Knoten der Liste `nodeList` abgearbeitet sind, springt die Methode zurück.

6.1.5 Theorie **5 Punkte**

Beweisen oder widerlegen Sie durch ein Gegenbeispiel folgende Behauptungen über den Graphen aus Abschnitt 3.

1.) Der Graph ist ein azyklischer Graph.

2.) Gegeben seien k gespielte Legekarten $\{\ell_1, ..., \ell_k\}$. Die Anzahl n der Zusammenhangskomponenten im dazugehörigen Graphen ist durch die Ungleichung $n \leq k \cdot 9$ beschränkt.

3.) Sei der Graph $G = (V, E)$ mit der Menge der Knoten V und der Menge der Kanten E gegeben. Die Komplexität einer Suche nach einem Element in einer Liste legen wir der Einfachheit halber mit $\mathcal{O}(1)$ fest. Die Laufzeit der Methode `calculatePoints(FeatureType type, State state)` ist durch $\mathcal{O}(|E| \cdot |V|)$ beschränkt.

6.2 Highscore und Dateien (5 Punkte)

In dieser Aufgabe behandeln Sie den Umgang mit Dateien. Betrachten sie dazu die Klassen `fop.model.player.ScoreEntry` und `fop.view.components.gui.Resources`. Die Klasse `Resources` verwaltet dabei unter anderem Bilder, Icons und Schriftarten. Wir wollen nun, dass unsere letzten Spielergebnisse in einer Datei gespeichert werden und wieder ausgelesen werden können. Diese sollen dann im GUI (siehe dazu Abschnitt 6.3) als Highscore-Seite sortiert angezeigt werden.

Strings und PrintWriter — 2 Punkte

Ergänzen Sie in der Klasse `ScoreEntry` die Methoden

- `read(String line)`: Liest eine gegebene Zeile ein und wandelt dies in ein `ScoreEntry`-Objekt um. Ist das Format der Zeile ungültig oder enthält es ungültige Daten, wird `null` zurückgegeben. Eine gültige Zeile enthält in der Reihenfolge durch Semikolon getrennt: den Namen des Spielers, das Datum des Spielendes als UNIX-Timestamp (in Millisekunden) und die erreichte Punktezahl. Gültig wäre beispielsweise:
  Player 1;1575471508171;93
- `write(Printwriter printWriter)`: Schreibt den Eintrag als neue Zeile mit dem gegebenen `PrintWriter`. Der Eintrag sollte im richtigen Format (siehe vorherige Aufgabe) gespeichert werden.

File Streams — 2 Punkte

Ergänzen Sie in der Klasse `Resources` die Methoden

- `loadScoreEntries()`: lädt den Highscore-Table aus der Datei highscores.txt. Dabei wird die Liste `scoreEntries` neu erzeugt und befüllt. Beachten Sie dabei, dass die Liste nach dem Einlesen absteigend nach den Punktzahlen sortiert sein muss. Sollte eine Exception auftreten, kann diese ausgegeben, aber nicht weitergegeben werden, da sonst das Laden der restlichen Ressourcen abgebrochen wird.
- `saveScoreEntries()`: speichert alle Objekte des Typs `ScoreEntry` in der Textdatei highscores.txt. Jede Zeile stellt dabei einen `ScoreEntry` dar. Sollten Probleme auftreten, muss eine `IOException` geworfen werden. Die Einträge sind in der List `scoreEntries` zu finden.

6.2.1 Sortiertes Einfügen 1 Punkt

Ergänzen Sie in der Klasse `Resources` die Methode

- `addScoreEntry(ScoreEntry scoreEntry)`: fügt ein `ScoreEntry`-Objekt der List von Einträgen hinzu. Beachten Sie, dass nach dem Einfügen die Liste nach den Punktzahlen absteigend sortiert bleiben muss.

6.3 Weitergestaltung des Spiels (18 Punkte)

In den folgenden Aufgaben entwickeln Sie das Spiel weiter und gestalten es nach Ihren Vorstellungen. 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.

6.3.1 Highscore View 1 + 2 Punkte

In der Aufgabe 6.2 haben Sie bereits gelernt, wie man den Highscore und andere Informationen in eine Datei schreibt und wie man diese auch wieder ausliest. Erstellen Sie dazu nun ein Graphical User Interface, welches die Informationen aus der Aufgabe 6.2 lädt und in einer geeigneten Form anzeigt. Die beiden Tasten *Back* und *Delete* müssen nur auf der Oberfläche angeordnet werden und **nicht** implementiert werden (siehe dazu die Codevorlage). Je nach Komplexität der View können 2 weitere Punkte erarbeitet werden.

Abbildung 15: Beispielhafte Repräsentation der Ergebnisse in einer Tabelle

6.3.2 Offizielle Bewertung einer Wiese 5 Punkte

In unseren Spielregeln haben wir die Bewertung der Wiesenflächen deutlich vereinfacht. Jetzt ist es an der Zeit, dass Sie die offizielle Art der Bewertung einer Wiesenlandschaft implementieren. Folgendes bleibt wie bei der alten Bewertung:

Spielfiguren, welche auf einer Wiese stehen, werden nicht zurückgegeben. Die Bewertung einer Wiesenlandschaft findet zudem erst nach Spielende statt.
Ändern Sie das alte Bewertungsschema für Wiesen wie folgt ab:
Jede abgeschlossene Stadt, welche sich in der Wiese befindet oder an die Wiese grenzt, bringt 3 Punkte.

6.3.3 Mission **5 Punkte**

Erweitern Sie das Spiel um 2 Missionen, mit welchen ein Spieler das Spiel vorzeitig gewinnen kann.

1. Ändern Sie das Spiel ab, sodass ein Spieler sofort gewinnt, sobald er 3 Burgen mehr besetzt hält als jeder seiner Gegner. Trifft dies zu, soll das Spiel sofort abbrechen und den Gewinner bekanntgeben, indem sich ein neues Fenster öffnet, in welchem die Punktzahl und die Anzahl der besetzten Burgen jedes Spielers angezeigt wird.

2. Denken Sie sich zudem noch eine weitere Mission aus und implementieren Sie diese.

6.3.4 Computergegner **5 Punkte**

Nun soll das Spiel auch noch um einen Einzelspielermodus erweitert werden. Ein Computergegner wird erstellt, indem man in den Platzhalter für den Spielernamen den Wert „AI“ eingibt. Erweitern Sie die Klasse `Player` um die Methoden `draw(Gameplay gp, Tile tile)` und `placeMeeple(Gameplay gp)`. Dabei können und sollen Sie natürlich auf die bereits implementierten Methoden der Klasse `Gameboard` zurückgreifen.
Als Minimalanforderung sollte der Computergegner eine Spielkarte an die erste gültige Position legen, die er findet. Eine Spielfigur sollte an eine zufällige aber natürlich gültige Position gestellt werden.
Beachten Sie auch die Methoden `placing_Tile_Mode` und `placing_Meeple_Mode` der Klasse `GamePlay`. Dort werden die zu implementierenden Methoden aufgerufen.

Quellenverzeichnis

[1] Wikimedia Commons. *carcassonne aerial 2016.jpg — Wikimedia Commons, the free media repository*. 2016. URL: https://commons.wikimedia.org/w/index.php?title=File:1_carcassonne_aerial_2016.jpg&oldid=213570163.

[2] Doris Matthäus, Klaus-Jürgen Wrede. *Carcassonne*.