P1.1: Daten des Players (in public class PlayerImpl)

- **getHexGrid()**, **getName()**, **getID()**, **getColor()**: dies sind einfache Getter-Methode für private Attribute, somit ist die Rückgabe das entsprechende Attribut
- **isAi()**: die Methode soll true zurückgeben, wenn der Player von KI gesteuert ist. Dafür überprüfen wir, ob das Attribut aiController null ist. Ist dieses Attribut null, dann ist der Player logischerweise nicht von KI gesteuert und soll false zurückgegeben werden und umgekehrt

P1.2: Bankkonto des Players

- PlayerImpl.**getCredits()**: wieder einfache Getter-Methode fürs Attribut credits, was einfach zurückgegeben wird
- PlayerImpl.**addCredits(int amount)**: die zu addierende Anzahl der Credits ist in Parameter amount übergeben und in der Methode wird einfache Addition durchgeführt
- PlayerImpl.**removeCredits(int amount)**: die Methode hat Rückgabetyp boolean, was darauf hinweisen kann, dass true zurückgegeben wird, wenn Subtraktion erfolgreich war. Damit müssen wir die übergebene Parameter auf mögliche Probleme überprüfen: amount darf keine negative Zahl sein (da dann werden die Credits des Players eigentlich erhöht und nicht reduziert als erwartet) und amount darf nicht größer als jetzige Anzahl von Credits des PLayers sein (da es bedeutet, dass der Player nicht genug Credits hat, um eine Aktion zu machen, abhängig davon, wo diese Methode in der Zukunft noch benutzt wird). Wenn einer davon der Fall ist, geben wir sofort false zurück. Ist aber dies nicht der Fall, dann führen wir die Subtraktion durch und geben dann true zurück, da Subtraktion erfolgreich war

P1.3: Alle Schienen führen nach ...

- HexGridImpl.**getRails(Player player)**: hier sollen wir alle Schienen vom als Parameter übergebenen Player in Form von Map<Set<TilePosition>, Edge> zurückgeben. Dafür können wir schon existierendes Attribut von HexGridImpl edges benutzen, das alle existierenden Rails auf dem HexGrid beinhaltet und das auch schon den gleichen Typ hat. Unser Vorgehensweise besteht darin, dass wir edges "ausfiltrieren" und zwar alle passende Edges in eine neue Variable (hier: Map<Set<TilePosition>, Edge> result) speichern. Zuerst hatten wir es gemacht, indem wir einfach prüften, ob edge.getRailOwners.contains(player) (wobei edge jedes einzelnes Element aus edges für jeden Schleifenrumpf repräsentiert), aber wir waren hier auf Problemen (failed public tests) gestoßen, deshalb haben lassen wir jetzt 2 geschachtelte Schleifen durchlaufen: die äußere Schleife iteriert durch jede Edge auf dem Spielgrid und innere Schleife iteriert durch owners von jede einzelne Edge und wird geprüft, ob jeweilige owner.toString() gleich zu player.toString() ist und, ist es der Fall, fügen wir jeweilige Edge zu result hinzu. Am Ende geben wir Collections.unmodifiableMap(result); zurück, da von uns unveränderbare Sicht verlangt wird
- EdgeImpl.**connectsTo(Edge other)**: es soll geprüft werden, ob zwei Edges (this und als Parameter übergebene other) ein gemeinsames Tile besitzen. Das ist der Fall, wenn entweder this.pos1 oder this.pos2 gleich zu einem von Positions von other ist. Genau das wird auch in unserer Implementation dieser Methode geprüft und entsprechende boolean-Wert wird zurückgegeben
- EdgeImpl.**getConnectedEdges()**: wir müssen alle Edges an den beiden Tiles (pos1 und pos2) "summieren" und als Set von Edges zurückgeben. Hier haben wir eine sehr sehr nützliche Methode TileImpl::getEdges gefunden. Um diese Methode nutzen zu können, brauchen wir auch Methode getTileAt aus class HexGridImpl. Mit getTileAt(TilePosition position) bekommen wir zu TilePosition position1 bzw. position2 entsprechende Objekte vom

Typ Tile und somit können auch o.g.Methode dieser Klasse nutzen. Wir deklarieren eine Variable edges, in der wir die Rückgabe von getEdges speichern und später zurückgeben

- EdgeImpl.**getConnectedRails(Player player)**: die Idee ist sehr ähnlich zu getRails früher in dieser Aufgabe, aber statt Attribut edges von HexGridImpl, das alle Edges auf dem SpielGrid beinhaltet, benutzen wir die Rückgabe von getConnectedEdges(), die nur die Edges zurückgibt, die an pos1 oder pos2 anliegen
- EdgeImpl.**addRail(Player player)**: wie auch in der Aufgabenstellung beschrieben, prüfen wir nach, ob auf diesem Edge ein Rail für gegebenen Player überhaupt gebaut werden darf.
    1. falls getRailOwners.contains(player) gilt, dann soll sofort false zurückgegeben werde, da player hier
    schon Schiene hat
    2. falls player hier noch keine Schiene hat:
    2.1: wenn es erste Schiene von player ist, dann soll diese Edge an mind eine von StartingCities anliegen. deshalb prüfen wir, ob startingCities von unserem grid entweder pos1 oder pos2 als key beinhaltet. Ist das nicht der Fall, wird false zurückgegeben
    2.2: wenn es schon mindestens 1 Schiene von diesem Player gibt, dann soll diese Edge an sein Schienennetzwerk angeschlossen sein. Das prüfen wir, indem wir die Rückgabe von grid.getRails(player) mittels connectsTo() filtrieren. Wenn es nach Filtration empty ist, dann ist diese Edge and Players Schienennetz nicht verbunden und soll false zurückgegeben werden.

    Wenn bisher kein false zurückgegeben wurde, dann darf unser player hier eine Schiene bauen: dafür fügen wir ihn zu railOwners hinzu und geben true zurück, da Schienenbau erfolgreich war

P1.4: ...Rom - Implementation der Städte

- TileImpl.**getNeighbour(EdgeDirection direction)**: wir müssen den Nachbarn von diesem Tile in gegebener Richtung zurückgeben. Dafür ist Methode getTileAt(TilePosition) von HexGrid sehr nützlich. Wir müssen dieser Methode ein Objekt vom Typ TilePosition als Parameter übergeben, und zwar die this.position+direction.position. Die Addition führen wir mit noch einer ganz nützlichen Methode durch und zwar TilePosition::add
- TileImpl.**getEdge(EdgeDirection direction)**: ähnlich zu o.g. getNeighbour, aber statt hexGrid.getTileAt hier nutzen wir hexGrid.getEdge(TilePosition, TilePosition)
- TileImpl.getConnectedNeighbours(Set<Edge> connectingEdges): wir nutzen wieder eine nützliche Methode von TilePosition und zwar TilePosition.heighbours(TilePosition). Die Rückgabe davon "mappen" wir zu entsprechende Tiles mithilfe hexGrid.getTileAt und filtrieren wir, sodass es keine null-Elemente gibt (für Sicherheit, sodass es später keine Exception ausgeworfen wird) und zweite - wichtigere - filter: sodass die this Tile und neighbour Tile verbindende Edge in connectingEdges (Parameter) gibt. Und das Ergebnis von Filtration geben wir zurück, nachdem wir das Stream in benötigte Rückgabetyp umwandeln
- HexGridImpl.**getConnectedCities()**: wir filtrieren das Attribut cities, die alle Städte beinhaltet, indem wir wie folgendes umgehen: für jede Stadt aus cities erhalten wir Tile mit entsprechender TilePosition. Dann erhalten wir alle anliegende Edges durch Tile::getEdges. Alle Edges, die jetzt an der jeweilige City anliegen, filtrieren wir, sodass sie nicht null sind (für Sicherheit) und eine Schiene schon auf sich haben (für Semantik)

Wenn die anliegende Edges nach Filtration Empty sind, dann soll diese City "rausgeschmissen" werden, da sie nicht connected ist (wir benutzen isEmpty() und ein NOT.Zeichen "!" vor dem Prädikat). Am Ende geben wir das Ergebnis zurück nach entsprechender Umwandlung von Stream to Map

- HexGridImpl.**getUnconnectedCities()**: genau das gleiche Code wir in getConnectedCities() sondern das Prädikat für Filterung hier mit isEmpty() ist nicht negiert
- HexGridimpl.getStartingCities(): wir filtrieren wieder das Attribut cities. Dieses Mal nutzen wir die Methode CityImpl::isStartingCity

- PlayerController.**canBuildRail(Edge edge)**: -
Die Methode hat einen Edge als Parameter, und gibt zurück, ob wir auf dieser Kante eine Schiene bauen dürfen.

Dafür sehen wir, ob eine Schiene schon auf der Kante liegt und ob es dem Spieler gehört. Wenn das nicht der Fall ist, prüfen wir, ob Spieler über genügend Guthaben verfügt, um die Schiene zu bauen,
wenn das Spiel sich in der Bauphase befindet, damit Spieler einige Kosten durch Würfelergebnisse bedeckt, sonst wird alles vom Credit abgezogen.

- PlayerController.**getBuildableRails()**: -
Die Methode gibt zurück alle Kanten, auf denen der Spieler bauen darf.

Falls der Spieler keine Schienen hat, darf er nur an Anfangsstädten bauen. Diese Kanten nehmen durch:

1) Sammeln aller Anfangsstädten
2) Umwandeln zu Stream von Cities
3) Sammeln aller Kanten an diesen Städten
4) Und Filtrieren durch canBuildRail();

Falls der Spieler Schienen gebaut hat, darf er nur an Netz liegen Kanten etwas bauen:

1) Wir sammeln alle Kanten auf dem Spielbrett, die Schienen haben, und filtrieren dadurch, ob sie Spieler als Besitzer haben
2) Wir sammeln alle dabei liegende Kanten
3) Schließlich filtrieren wir durch canBuild()

- PlayerController.**buildRail(final Edge edge)** -
die Methode prüft, ob wir auf die Kante eine Schiene bauen dürfen und wenn es möglich ist bauen wir. Sonst werfen eine Fehlermeldung.

Nach dem ersten canBuild-Test berechnen wir die Baukosten. Wenn der Spieler die Kosten bedecken kann:

1) Subtrahieren wir die Kosten vom Guthaben
2) Für alle parallele Schienen den anderen Spielern Credits gutschreiben
3) Dann schreiben wir dem Spieler Credit gut, für Verbinden unverbundenen Städten
4) Addieren wir den Spieler als Besitzer.

Sonst gibt es eine Fehlermeldung

- GameController**.executeBuildingPhase()** -

Die Methode führt die Bauphase aus, indem Spieler abwechselnd würfeln und Schienen bauen, bis bestimmte Anzahl von Städten unverbunden sind.
Mit jeder Aufruf der Schleife inkrementieren wir den RoundCounter. Dann dementsprechend den Spieler wechseln. Der gewählte Spieler würfelt und für jeden Spieler wird ausgeführt:

1) Baubudget wird aktualisiert
2) Spieler muss bauen.

- GameController.**chooseCities()** -
Die Methode wählt zwei Städte (Anfang und Zielstadt) für die Fahrphase.

Das machen wir in dem wir Stream von allen Städten filtrieren dadurch, ob sie schon gewählt waren. Dann wählen wir eine Zufällige und markieren es als gewählt.
Danach machen wir dasselbe, um die Zielstadt zu wählen.

- PlayerController.**canDrive()** -
Die Methode entscheidet, ob ein Spieler fahren darf. Dafür prüfen wir, ob die Phase jetzt Fahrphase ist, und ob der Spieler in der Liste von Fahrenden ist.

- PlayerController.**drive(final Tile targetTile)** -
Die Methode führt Fahren aus, indem die Position und Überschuss (beim Erreichen der Stadt) aktualisiert werden.
Als Erstes behandeln wir Fehlermeldungen für die Fälle:

1) Spieler darf nicht fahren
2) Der Zielkachel ist nicht erreichbar.

Dann itererieren wir über Tiles in path (Pfad bis Zielkachel):

1) Aktualisieren der Spielerposition
2) Beim Erreichen des Zielkachels überschuss aktualisieren.

- GameController.**letPlayersChoosePath()** -
Für jeden Spieler wird seine Position zurückgesetzt und der Spieler wird aufgefordert einen Pfad wählen und bestätigen.

- GameController.**handleDriving()** -
Die Methode behandelt die Fahrphase.
Wenn kein Spieler fährt, wird nichts gemacht.
Wenn nur ein Spieler fährt, wird er automatisch zum Gewinner.

Sonst sammeln wir alle Spieler, die die Zielstadt erreicht haben. Wir aktualisieren den Überschuss von Spielern, die den Zielkachel nicht erreicht haben. Dann sortieren wir nach Credits und aktualisieren die Liste von Fahrenden.
Danach machen wir eine Schleife bis entweder Config.WINNING_CREDITS.size() Spieler die Zielstadt erreicht haben, oder alle Spieler.
In dieser Schleife iterieren wir über jeden Spieler und fordern sie zu würfeln und fahren.
Beim Erreichen des Ziels fügen wir den Spieler zur Liste hinzu.
- GameController.**getWinners()** -
Die Methode gibt zurück am höchsten Config.WINNING_CREDITS.size() Spieler, die das Ziel erreicht haben.

Es sortiert die Spieler nach Würfelüberschuss und fügt die Gewinner zur Liste hinzu

- GameController.**executeDrivingPhase()** -
Die Methode behandelt die Fahrphase:
1) Wir sammeln alle nicht gewählte Städte und itererien durch eine Schleife, bis alle Städte gewählt sind.
2) Wir inkrementrieren den Rundenzähler und setzen alle relevanten Daten (Position, Überschuss und Spieler) zurück
3) Für jede 3. Runde wird eine Bauphase initialisiert
4) Der ((roundCounter.get() - 1) % state.getPlayers().size()). Spieler wählt die Städte
5) Die Spieler wählen Pfad, das Fahren fängt an und zuletz werden Gewinner ermittelt.