Relazione UML gruppo GC19

Aryan Sood, Federico Villa, Matteo Vitali, Marco Zanzottera

1 Struttura

IL *Model* del nostro progetto è strutturato in cinque package:

- Card
- Station
- Deck
- Game
- Chat

Il package `Enum`, invece, è globale e non fa parte del modello.

2 Card

Il package `Card` contiene tutte le classi che rappresentano le carte del gioco.

2.1 Card

Ogni carta è identificata mediante un codice alfanumerico univoco; ad esempio, per le carte risorsa: `resource_01`, ..., `resource_40`. La classe `Card` è astratta e fornisce i metodi comuni a tutte le carte (obbiettivo e giocabili): `countPoints`, `getCardDescription` e `getCardCode`.

2.2 PlayableCard

Le carte giocabili sono dotate di due matrci 2×2 di `Corner` che ne rappresentano gli angoli nella faccia superiore (`frontGridConfigguration`) e in quella inferiore (`backGridConfiguration`).
`Corner` è un'interfaccia che viene implementata dalle *enum* `Symbol`, `EmptyCorner` e `NotAvailableCorner`. In questo modo, possiamo gestire in modo semplice lo stato di un angolo (notare, a tale proposito, il metodo `Optional<Symbol> getSymbol()` di `Corner`). Inoltre, il codice è più leggibile e possiamo rappresentare le risorse disponibili ad un giocatore con una `HashMap<Symbol, Integer>` senza preoccuparci di chiavi che sono `Corner` ma non simboli.

In Java, una *enum* non può estendere un'altra *enum*, quindi `Symbol` fornisce `getResources()` e `getObjects()` che ritornano i sottoinsiemi (`EnumSet<Symbol>`) rispettivamente delle risorse enumerate e degli oggetti.

`PlayableCardType` rappresenta il tipo di una carta giocabile: alla fine tutte le carte risorsa, oro e iniziali hanno la stessa struttura (angoli, risorse permanenti, precondizioni al piazzamento, al più vuote e punteggio) quindi si può evitare di creare 3 classi separate.

Lo stato di una carta viene gestito mediante il pattern *state*. Privatamente alla classe `PlayableCard`, infatti, abbiamo definito l'interfaccia `CardState` che viene implementata da `CardUp` e `CardDown`. Chiamando un qualsiasi metodo di `PlayableCard`, questo verrà eseguito in una delle sue due versioni, quella per `CardOrientation.UP` o quella per `CardOrientation.DOWN` in base a `cardState`. Il metodo `swapCard` permette di girare una carta, cambiando il suo stato interno.

Carte risorse e carte oro hanno anche un seme, mentre quelle iniziali no. Quindi, la `getSeed()` deve ritornare un `Optional<Symbol>`.

L'enum `CornerPosition` associa un nome ad ogni posizione della matrice 2×2, ad esempio $(0, 0) \rightarrow$ `CornerPosition.UP_LEFT`. Questo, insieme al metodo `getOtherCornerPosition()` di `Direction` permette di scrivere del codice più semplice e comprensibile:

```
currentX = this.cardPosition.get(anchor).x() + direction.getX();
currentY = this.cardPosition.get(anchor).y() + direction.getY();
for(Direction dir : Direction.values()){
    neighborCard = this.cardSchema[currentX + dir.getX()]
                                  [currentY + dir.getY()];
    if(neighborCard != null &&
       !neighborCard.canPlaceOver(dir.getOtherCornerPosition())){
        return false;
    }
}
```

2.3 GoalCard

Le carte obbiettivo sono dotate solamente di un `GoalEffect`, cioè dell'effetto che rappresentano.

2.4 PlayableEffect e GoalEffect

L'effetto collegato a `PlayableCard` e `GoalCard` è stato implementato mediante il pattern *strategy*. Abbiamo definito due interfacce: `PlayableEffect` per le carte giocabili e `GoalEffect` per quelle obbiettivo. Ciò garantisce maggiore leggibilità e *type-safety*: associato ad una `GoalCard`, infatti, non ci può essere un

effetto di `PlayableCard` e viceversa. `PatternEffect` rappresenta un pattern di carte (notare la generalità di `PatternEffect` grazie alla `ArrayList <Tuple<Integer, Integer>>` di spostamenti nella griglia da seguire per realizzare il pattern). `SymbolEffect` implementa `GoalEffect` e `PlayableEffect` perché sia le carte giocabili che quelle obbiettivo possono dare punti per ogni insieme di simboli visibili (ad esempio 2 per ogni coppia di piume). `CornerEffect` serve per le carte che danno punti in base al numero di angoli che coprono. Infine, `NoConstraintEffect` rappresenta quelle carte (come le risorse) che o non hanno associato alcun punteggio (`cardValue` = 0) oppure danno punti per il solo fatto di averle posizionate.

3 Station

Il package `Station` rappresenta la postazione di gioco di un giocatore.

3.1 Station

La classe `Station` definisce il campo di gioco di un giocatore e contiene i metodi per gestire le carte e il loro posizionamento. Ogni stazione tiene traccia delle carte visibili nel campo, delle carte obiettivo private e delle carte che un giocatore ha in mano. La classe fornisce i metodi per modificare le carte che un giocatore ha in mano `updateCardsInHand()`, per posizionare le carte `placeInitialCard()` e `placeCard()`, per aggiornare i punti `updatePoints()` e per verificare che una carta sia piazzabile sopra un'altra (detta ancora) in una certa direzione e una direzione `cardIsPleaceable()`.

3.2 CardSchema

La classe `Station` nasconde al giocatore la struttura dati utilizzata per gestire le carte giocabili presenti nel campo di gioco: ogni qual volta che è necessario interagire con il campo da gioco, i metodi di `Station` chiamano i metodi di `CardSchema`. Così facendo l'implementazione del gioco risulta più modulare e, in caso di aggiornamenti futuri, permette di modificare facilmente la struttura dati e i metodi che lavorano sul campo di gioco.

Lo schema è definito tramite una matrice di `PlayableCard`, scelta efficace dato che il numero massimo di carte che un giocatore può avere nel proprio campo è modesto e molti metodi richiedono di controllare, data una carta, quella presente in una data direzione.
`CardSchema` contiene vari metodi tra cui: `cardOverAnchor()` che verifica se esiste una carta nella posizione definita da una carta ancora e una direzione; `isPlaceable()` che controlla è possibile posizionare una carta in una posizione specificata da una carta ancora e da una direzione; `getLastPlaced()` che ritorna l'ultima carta posizionata; `cardIsInSchema()` che controlla se una data carta è

presente nel campo di gioco; `getCardOverlap()` che ritorna per ogni posizione della matrice di gioco un valore che indica l'ordine sequenziale di posizionamento delle carte oppure 0 se nessuna carta è presente in quella posizione.

Con i metodi `getCardSchema()` e `getCardOrientation()` è possibile ottenere una matrice contenente i codici alfanumerici univoci delle carte posizionate e la loro orientazione. Questi metodi saranno molto utili in fase di salvataggio dello stato del gioco su disco.

4 Deck

Questo package contiene solo la classe `Deck`.

`Deck` è una *generic class* poichè dipende dal paramentro `cardType` (estende `Card`) che rappresenta il tipo di carte (`PlayableCard` o `GoalCard`) contenute. Mediante `pickACard()` è possibile pescare dal mazzo (se è vuoto viene sollevata `EmptyDeckException`). `insertCard()` è utile per ripopolare il mazzo dopo il *crash* del server. Infine, `shuffleDeck()` mischia il mazzo.

5 Game

Il package `Game` gestisce una partita di gioco, permettendo di creare e modificare i giocatori, le postazioni di gioco e i mazzi.

5.1 Game

La classe game gestisce una partita di gioco, andando a creare i quattro deck, definendo la lista di giocatori della partita e l'ordine di questi nel gioco.
Sono stati definiti i metodi per gestire il pescaggio delle carte dai mazzi `pickCard FromDeck(PlayableCardType)` e per pescare le carte dal tavolo `pickCard FromTable(PlayableCardType)`.
I giocatori sono aggiunti e rimossi dal gioco con i metodi `createNewPlayer()` e `removePlayer()` e l'ordine dei giocatori viene definito dal metodo `setFirstPlayer()`, che viene chiamato quando il gioco inizia, cioè alla chiamata di `startGame()`. Il giocatore corrente, cioè quello che nel turno corrente deve pescare e posizionare la carta nella sua stazione, è gestito con l'attributo `activePlayer`. Il metodo `getNextPlayer()` ritorna, in base all'`activePlayer` corrente, il giocatore successivo.

Lo stato del gioco viene gestito dal controllore con i metodi `setGameState()`, `getGameState()`, `getTurnState()` e `setTurnState()`. Il gioco può essere in 4 stati: *SETUP*, *PLAYING*, *PAUSE* e *END*. Quando il gioco è nello stato *PLAYING* ci sono due modalità: *PLACE* e *DRAW*.

Come supporto ai metodi per ottenere, dato il codice alfanumerico univoco di una carta, il riferimento alla carta e le informazioni sulla carta sono definite due HashMap che collegano il codice di una carta al riferimento della `PlayableCard` e della `GoalCard`. I metodi per ottenere la descrizione di una carta sono `getInfoCard` e `getInfoAllCards`; invece per ottenere i riferimenti alle carte dato il loro codice univoco di sono stati definiti `getPlayableCardFromCode()` e `getGoalCardFromCode(cardCode)`.

5.2 Player

La classe `Player` rappresenta un giocatore connesso ad una partita.

Ogni giocatore è identificato da un nickname univoco ed è associato al colore delle sue due pedine ed alla sua postazione di gioco, rappresentata dalla classe `Station`. I metodi `getName()`, `getStation()` e `getColor()` ritornano rispettivamente il nome del giocatore, la postazione del giocatore e il colore associato. Il metodo `setColor()` imposta il colore delle pedine di un giocatore.

6 Chat

Questo package rappresenta la chat di gioco.

6.1 Message

La classe astratta `Message` riproduce un generico messaggio che viene mandato nella chat.
Le sue due concretizzazioni sono `OneToOneMessage` e `OneToMoreMessage` che rappresentano rispettivamente un messaggio per un singolo giocatore e per un gruppo di giocatori (anche non tutti). Notare che di mittenti e destinatari non si memorizza l'oggetto `Player`, ma solo il nome.

6.2 Chat

Implementa la logica della chat che è rappresentata come una `ArrayList<Message>`. Oltre al metodo `pushMessage`, abbiamo `toString` che servirà alla *view* per visualizzare l'elenco dei messaggi.