

Documento con commenti UML Gruppo GC28

Classe **Game**

Questa è la classe principale che gestisce una partita. I metodi più importanti, che permettono di giocare una partita e che andranno a costituire il controller, sono:

- **playGameCard**: permette di giocare una carta.
- **drawGameCard** : permette di pescare una carta.
- **chooseObjective** : permette di scegliere la carta obiettivo personale a inizio partita.

il metodo **drawGameCard** è implementato con due prototipi diversi poiché c'è la necessità di distinguere quando l'utente pesca una carta da un mazzo o dalle 4 carte visibili.

il metodo **setUpNextMove** viene chiamato dopo che un giocatore fa una mossa, vale a dire alla fine di **playGameCard**, **drawGameCard** e **chooseObjective**. Si occupa di chiamare **checkEndgame** oppure **endGame** (vedi sotto) e successivamente chiama il metodo **nextMove()** sull'**actionManager**, il quale aggiorna il giocatore che dovrà giocare il prossimo turno e il tipo di azione prevista.

Il metodo **checkEndGame** si occupa di controllare se un giocatore ha raggiunto un punteggio ≥ 20 , e in tal caso imposta la variabile **roundsLeft** al numero di round rimasti da giocare in base alla specifica del gioco.

Il metodo **endGame** viene chiamato al posto di **checkEndGame** quando **roundsLeft** non è empty, verifica se **roundsLeft** è 0 e in tal caso chiama **calculateObjectivePoints** e **calculateWinner** per eseguire i conteggi dei punti obiettivo, il calcolo del vincitore e quindi fa finire la partita.

Classe **Player**

abbiamo inserito un attributo **objectivePoints** poiché in caso di pareggio, a fine partita, vince il giocatore che ha fatto più punti tramite carte obiettivo, come da specifica.

Ogni giocatore ha un attributo corrispondente a un errore opzionale, che viene valorizzato dalla classe **ErrorManager** nel caso in cui, ad esempio, un giocatore provi a piazzare una carta nel turno sbagliato, fare una mossa illegale, etc...

Classe **Table**

Questa classe gestisce le informazioni su un tavolo da gioco, infatti ogni giocatore ha un attributo di tipo **Table**. le informazioni sul posizionamento delle carte piazzate è dentro **mapPositions**, la quale è una mappa che ha come chiavi delle coordinate e come valori degli oggetti di tipo **Cell** (un wrapper della carta piazzata con informazioni aggiuntive, ad esempio se è stata piazzata sul fronte o retro). Abbiamo fatto questa scelta poiché abbiamo notato che le carte possono essere piazzate solo in posizioni predeterminate, che identifichiamo con coordinate "cartesiane".

La prima carta (la carta iniziale) sarà piazzata nella coordinata (x=0, y=0); piazzare nell'angolo in alto a destra della carta iniziale significa piazzare una carta nella coordinata

(x=1, y=1), piazzare nell'angolo in alto a sinistra corrisponde ad aggiungere una cella alla coordinata (x=-1, y=1), etc...

Grazie a questa struttura dati, a fine partita riusciamo a calcolare agilmente i punti derivanti dalle carte obiettivo, poiché l'informazione sulla disposizione spaziale delle carte posizionate è facilmente ottenibile.

L'attributo **resourceCounters** memorizza quante sono le risorse visibili per ogni tipo di risorsa e viene aggiornato ogni qualvolta il giocatore piazza una carta.

playableCoords e **unplayableCoords** sono degli attributi che semplificano il processo di determinare se una coordinata è giocabile o meno quando il giocatore decide di piazzare una carta.

Classe **Objective** e il calcolo dei punti obiettivo

Abbiamo suddiviso i diversi tipi di obiettivo in due categorie:

- **ObjectivePosition**, vale a dire obiettivi posizionali
- **ObjectiveResources**, cioè obiettivi che calcolano il numero di certe risorse visibili.

entrambe queste classi estendono la classe astratta **Objective** e quindi implementano il metodo **calculatePoints**, che nel caso dell'**ObjectivePositions** calcola il punteggio a partire dalla **mapPositions**, mentre nel caso dell'**ObjectiveResources**, calcola il punteggio a partire dal **resourceCounters**.

Classe **ObjectivePositions**

La funzione **getNeighborsCoordinate** dell'attributo **positionType** è progettata per restituire un insieme di coordinate basate su un pattern posizionale specifico a partire da una coordinata iniziale passata come parametro alla funzione. Ad esempio, se il pattern è una diagonale principale e la coordinata di partenza è (x=2, y=2), la funzione restituirà un array ordinato che include le coordinate (x=1, y=1), (x=2, y=2) e (x=3, y=3).

Queste coordinate sono ordinate in modo specifico: in ordine crescente di x e, in caso di parità di x, in ordine crescente di y. In altre parole, il pattern viene costruito partendo dal punto in basso a sinistra fino a raggiungere quello in alto a destra della diagonale principale, e lo stesso vale per tutti gli altri pattern posizionali.

Questo è utile poiché l'attributo **patternPositionResources** determina per ogni posizione all'interno del pattern, quale risorsa deve appartenere alla carta in tale posizione per soddisfare il pattern, e usiamo la stessa convenzione di ordinamento della funzione **getNeighborsCoordinate**, vale a dire il primo elemento dell'array **patternPositionResources** è riferito alla prima posizione dell'array restituito da **getNeighborsCoordinate** e così via...

Questo approccio permetterebbe di creare nuovi tipi di obiettivo con molta facilità, ad esempio se volessimo aggiungere un obiettivo posizionale che riconosce diagonali principali di 4 elementi le cui prime due carte (a partire dal basso) siano di risorsa Foglia e le altre due carte di risorsa Farfalla, basterebbe creare un obiettivo nel modo seguente:

- **getNeighborsCoordinate** : (x, y) => [(x-2,y-2),(x-1,y-1), (x,y), (x+1,y+1)]
- **patternPositionResources** : [Foglia, Foglia, Farfalla, Farfalla]

Date le seguenti informazioni, è possibile calcolare il punteggio di un obiettivo iterando sulla **mapPositions**: per ogni coordinata considerare l'array di coordinate tramite la funzione **getNeighborsCoordinates**, verificare che le carte nelle posizioni ci siano e rispettino le risorse dentro **patternPositionsResources**, salvarsi tutti gli array di coordinate che soddisfano il pattern e alla fine prendere la combinazione di array di coordinate non sovrapponibili più numerosa (utilizzando **areCombinationsOverlapping**).

Classe **ObjectiveResources**

l'implementazione di **calculatePoints** in questo caso è più semplice e si basa sull'utilizzo del parametro **resourceCounters** descritto precedentemente e dell'attributo **resourcesNeeded**. In sostanza, **calculatePoints** si occupa di contare i punti in base al numero di risorse disponibili nel contatore (**resourceCounters**) e al numero minimo richiesto di ciascuna risorsa (**resourcesNeeded**).

Gerarchia di **Card**

- **Card**: classe astratta, superclasse di tutte le altre
 - **CardGame**: carte da gioco, ovvero piazzabili nella **Table**
 - **CardResource** : carte risorsa, possono dare punti quando vengono giocate
 - **CardGold** : le carte oro sono un caso particolare di **CardResource**
 - **CardInitial**: le carte iniziali hanno anche un attributo per memorizzare come sono i vertici sul retro, per tutte le altre carte da gioco i vertici sul retro sono sempre gli stessi, ovvero 4 vertici visibili senza risorse.
 - **CardObjective**: carte obiettivo

Classe **Challenge** per le carte oro

Questa classe si occupa del calcolo dei punti quando una carta oro viene giocata. Abbiamo suddiviso le challenge in:

- **PositionChallenge** , per le challenge di posizione (angoli coperti)
- **ResourceChallenge**, per le challenge che assegnano punti in base alle risorse

Entrambe le sottoclassi implementano il metodo **challengePoints**, che in base al tavolo e alla posizione dove viene giocata la carta, calcola i punti guadagnati.