Peer-Review 1 - UML

Restelli Matilde, Salvi Niccolò, Scianna Marco, Villa Alessio

Gruppo 38

POLITECNICO
MILANO 1863

A.S. 2023-2024

Valutazione del diagramma UML delle classi del gruppo 28

Lati Positivi

- Action Manager: è stata riscontrata molto utile e flessibile la gestione delle azioni che può compiere un giocatore, durante il suo turno, con un manager apposito e un'enumerazione per esse.
  Ciò facilita l'interazione tra i giocatori, la gestione sia delle turnazioni che degli errori che possono essere commessi dai player (come provare a compiere un'azione quando non è il loro turno).

- Error Manager: la gestione degli errori mirata ad ogni giocatore è molto intuitiva e utile anche in ottica di dover mostrare a video questi messaggi di errori in una futura view del progetto.
  E' sempre buona pratica mostrare all'utente perchè e dove ha sbagliato piuttosto che limitarsi a non far avvenire l'azione scorretta.

- Gestione dei pattern delle carte Obiettivo, metodo getNeughborsCoordinate e attributo positionType.
  L'implementazione utilizzata delle varie tipologie di pattern che risultano sulle carte obiettivo permette molto facilmente l'aggiunta futura di nuove tipologie di esse.
  Un gioco non è per forza statico nel corso degli anni, l'implementazione di esso deve quindi poter essere facilmente estensibile e adattabile.

- Metodo areCombinationsOverlapping: acuta osservazione è stata quella della possibilità che la scelta delle carte del "Table" del giocatore per realizzare un obiettivo potrebbe portare ad un diverso numero di combinazioni che lo completino.
  Risulta quindi fondamentale controllare tutte le possibili combinazioni e scegliere quelle che permettano al giocatore di acquisire più punti, completando per più volte l'obiettivo senza sovrapporsi.

- Attenzione alla gestione delle possibili dinamiche di fine di una partita:
  - Attributo "ObjectivePoints" nella classe Player: visto che in caso di pareggio, a fine partita, vince il giocatore che ha accumulato più punti con le carte obiettivo avere questo dato già pronto velocizza di molto l'eventuale check di questa condizione.
  - L'utilizzo dell'attributo "roundsLeft" permette agilmente di tener traccia del corso della partita e gestire, come da specifica, la fine di essa

Lati Negativi

- La classe astratta “Card”: per accomunare le carte da gioco (denominate “CardGame”) e le carte obiettivo è stata implementata una classe madre di entrambe, astratta, “Card”. Questa classe non viene mai utilizzata, non contiene nessun metodo e nessun attributo perchè le due tipologie di carte, sopra menzionate, non hanno nulla in comune.
  La classe Card viene utilizzata solo come tipo di return del metodo “get..Card” nella classe Table di un giocatore. Essa però in questa situazione può essere tranquillamente sostituita con l’utilizzo della classe CardGame, in quanto un giocatore non può aver utilizzato una carta obiettivo nel suo Table.
  Inoltre l’utilizzo del riferimento alla classe Card in questo caso genera anche ulteriori problemi. L’oggetto Card restituito dal metodo “get…Card” verrà utilizzato per chiamare metodi per conoscerne il tipo di risorsa o altri attributi che sono presenti solo nella classe CardGame (sottoclasse) e non in Card, questo richiederebbe l’utilizzo di operazioni ogni volta da effettuare con il costrutto “Instance Of” e con particolare attenzione al polimorfismo dinamico.

- La presenza di un solo attributo “Deck” nella classe Game: nella classe Game è dichiarato un solo attributo Deck, ma questa scelta non è abbastanza chiara.
  Nonostante la decisione di trattare le carte gold (“CardGold”) come una sottoclasse delle carte risorse (“CardResource”) non è corretta la creazione di un singolo mazzo per entrambe le tipologie di carte.
  Soprattutto questa è una scelta poco implementabile nella futura View del gioco dove si dovranno vedere chiaramente due mazzi differenti sul tavolo, uno per le carte gold e uno per quelle risorse, e anche le 4 carte scoperte di essi saranno rispettivamente 2 carte Gold e 2 carte Resource.

- Mancato controllo della “Playability” delle carte “CardGold”: in questo gioco le gold card hanno dei requirements che devono essere rispettati dal giocatore perchè esse siano utilizzate da lui.
  L’implementazione della mappa delle risorse necessarie al giocatore per utilizzare una carta gold è correttamente presente, ma essa non viene mai controllata. Infatti nella classe Table è presente un metodo corrispondente “checkPlayability”, ma non si riferisce a questa condizione in quanto come parametri possiede solo le coordinate della posizione in cui si vuole giocare la carta e non l’oggetto carta stesso, quindi non può risalire ai requirements necessari per utilizzare la CardGold.
  E’ quindi necessaria l’aggiunta di un metodo apposito che, quando il giocatore decide di giocare una carta Gold, controlli che egli possieda le risorse necessarie per farlo.

- Mancanza degli identificativi: nessuna delle classi principali che caratterizzano il gioco (come player, card o game) possiede un identificativo.
  E’ stato notato il fatto che ci sia un’interazione tra classi (come il Game ha come attributo i riferimenti agli oggetti Player che sono in gioco), ma sono necessari anche identificativi

univoci per ognuno di essi. Pensando ad esempio al fatto che si debba notificare a tutti i giocatori che il Player X ha vinto, oppure che se il Player X gioca la carta numero 12 essa non potrà essere presente nelle carte di un altro giocatore, ecc..
Viene quindi suggerita l'aggiunta di un attributo ID ad ognuna delle classi principali (Game, Player, GameCard, Objective),

- La gestione dell'azione "chooseObjective": nella descrizione dell'uml viene descritta come è stata adottata una scelta di gestione delle mosse fattibili da un giocatore e cronologicamente il flusso di esecuzione di esse.
  L'azione di un giocatore di pescare la carta obiettivo segreta personale viene considerata come una di esse, ma non costituisce una mossa che viene eseguita ogni turno o viene ripetuta. E' sicuramente parte di uno stato iniziale di SetUp del Game, della partita in corso, dove si registrano i giocatori, si distribuiscono le carte, posizionano i mazzi, peschino le carte obiettivo comune e, appunto, quelle personali per ogni giocatore.

Confronto tra le architetture

- Differenza tra strutture dati utilizzate: per gestire le carte posizionate da un giocatore il nostro gruppo ha utilizzato una matrice di interi (id delle carte) 81x81, invece dell'utilizzo di una mappa con delle coordinate associate a ciascuna carta come qui. Andremo ad analizzare meglio i pro e contro di queste due differenti strutture dati, poiché si tratta di bilanciare l'occupazione di spazio in memoria e la velocità con cui si caricano e ottengono informazioni. Con la matrice evitiamo anche l'utilizzo di un return null se in quella posizione non si trova nessuna carta, mentre nella mappa se quelle coordinate non sono state inizializzate non sono proprio presenti.
- Aggiunta nel nostro progetto di un Error Manager per gestire gli eventuali errori che si possono verificare.
- Valutare , per snellire le nostre classi di Game e Player, l'inserimento di una classe apposita che raggruppi tutte le azioni che un singolo giocatore può compiere (come qui Action Manager)
- Modificare la gestione del controllo di validazione delle carte obiettivo, aggiungendo la casistica specifica per considerare le varie possibili combinazioni di pattern in modo tale da scegliere quella che apporta più punti al giocatore (il corrispondente metodo presente areCombinationsOverlapping)