Peer-Review 1: UML

Matteo Colombi, Andrea Colombo, Leonardo Conti, Lorenzo Demontis
Gruppo AM16

2 aprile 2024

Introduzione

In questa peer review abbiamo analizzato il diagramma UML del modello progettato dal gruppo AM25. Abbiamo richiesto chiarimenti riguardo alcuni dubbi su certi metodi di cui non avevamo compreso immediatamente la funzione, o su cui non eravamo sicuri che la nostra interpretazione fosse corretta. In particolare, le domande rivolte al gruppo revisionato riguardavano:

- quante e quali funzionalità avanzate sono state prese in considerazione per lo sviluppo, poiché non evidenti dal diagramma UML;
- il funzionamento dei `Deck`, e in particolare l'utilità dell'attributo `numberOfCardsDrawn`;
- il funzionamento del metodo `removePlayer` e `getStartingPlayer` di `Game`;
- il funzionamento di `placeablePositions`, e in particolare il suo uso.

A seguito delle nostre domande, il refrente di AM25 ci ha fatto notare che le funzionalità avanzate da loro scelte, ovvero chat, resilienza alle disconnessioni e persistenza, non sono ancora state modellate perché hanno considerato più efficiente definire prima un progetto per controller e comunicazione client-server sulla rete. Per quanto riguarda le altre domande, le risposte sono state pertinenti e comunque di carattere implementativo, dunque non vengono riportate qui in quanto non strettamente collegate a una critica del modello. Infine, ci è stata inviata la JavaDoc per migliorare la nostra comprensione dell'intero modello.

1 Lati positivi

i) Il modello mostra una buona divisione dei compiti tra i moduli applicativi. Questo dimostra una buona applicazione dei principi della programmazione ad oggetti, lasciando spazio alla possibilità di espandere il progetto senza particolari difficoltà. Un esempio di questa buona

modularizzazione è l'interfaccia `PointCalculator`: tale astrazione elimina possibili ridondanze nel modello tra il calcolo dei punti delle carte obiettivo e delle carte risorsa e oro che danno punti.

ii) Il modello progettato è essenziale. Questo ne rende complessivamente agevole la comprensione e, probabilmente, anche l'implementazione.

2 Lati negativi

i) Il modello manca di interfacce ben definite da esporre a Controller e View. Non è quindi chiaro come questi componenti dovrebbero leggere lo stato del modello senza avere anche accesso a tutti i metodi modificatori dello stesso.

ii) Mantenere lo stato dell'area di gioco con una matrice è molto inefficiente dal punto di vista computazionale: una chiamata a `placeCard` comporta la necessità di cercare all'interno della matrice la carta con il dato `baseCardId`.

iii) Nonostante l'essenzialità del modello sia generalmente apprezzabile, questo progetto manca di alcuni attributi e metodi per la gestione dello stato della partita. Immaginiamo che questi compiti siano delegati al Controller, di cui però non è fornito un accenno di progetto, dunque non possiamo valutare la bontà di questa scelta.

iv) `PlayArea` non contiene una struttura dati che fornisca l'ordine di piazzamento delle carte. Questa è un'informazione necessaria alla View per poter visualizzare le carte con le sovrapposizioni corrette.

v) Come detto, per comprendere alcuni aspetti del modello abbiamo dovuto fare domande ulteriori al referente del gruppo AM25. Questo è sintomo di una scarsa autoesplicatività di tali aspetti del modello.

3 Confronto tra le architetture

L'architettura del nostro modello è decisamente più complessa. Se da un lato questo è considerabile un lato positivo, ci fa considerare l'idea di "sfoltire" il modello per eliminare metodi o classi in esubero.

Come già accennato nella sezione 1, pensiamo che `PointCalculator` sia una buona astrazione che potremmmo implementare nel nostro modello. L'unico problema che vediamo con questa implementazione è la possibilità di associare a una carta obiettivo la tipologia di punteggio `AngleCalculator`, che ovviamente non ha senso.

La classe `PlayArea` fornisce un metodo (`getPlaceablePositions`) per ottenere tutte le posizioni su cui si può piazzare una carta. Questo è molto efficiente e semplifica notevolmente il controllo circa la legalità di una mossa. Il nostro metodo `checkLegalMove` (sempre nella classe `PlayArea`)

implementa invece una logica abbastanza complessa che viene eseguita ad ogni piazzamento di una carta.

Osservazioni

i) In `Game` manca un attributo al path del file JSON da cui leggere le informazioni per le carte obiettivo. Questo path invece c'è nei `Deck` per quanto riguarda carte risorsa, oro e carte iniziali. Immaginiamo quindi che sia una svista.

ii) `PlayCard` non è dichiarata nel diagramma UML come *abstract*. Al contrario, nella JavaDoc si legge che questa classe è astratta. Questa è probabilmente una svista nella stesura del Class Diagram.

iii) In `Colour`, `RED` è dichiarato come tipo `CornerPosition`. Questo è ovviamente un refuso.