Peer-Review 1: UML

Riccardo Polelli Filippo Pozzi Federico Quartieri Giacomo Tessera

Valutazione del diagramma UML delle classi del gruppo GC57 da parte del gruppo GC10.

Lati positivi

Iniziamo con le decisioni che ci hanno colpito positivamente:

- L'introduzione dell'utilizzo di una **coda** rappresenta un'ottima strategia implementativa, in quanto semplifica notevolmente la scrittura dei metodi e contribuisce alla robustezza complessiva del codice. Questa scelta architetturale non solo aumenta l'efficienza nella gestione delle operazioni, ma migliora anche l'autoesplicabilità delle funzioni implementate, rendendo più chiare le logiche sottostanti.
- L'adozione di `PlayerTable` come struttura per memorizzare le posizioni delle carte costituisce un'aggiunta significativa al design complessivo del sistema. Questa scelta facilita la comprensione del codice e aumenta la trasparenza del funzionamento del gioco, rendendo più semplice tracciare il flusso delle interazioni tra le entità e migliorando la manutenibilità e la scalabilità del software.
- L'inclusione di una `Bag` per agevolare il conteggio delle risorse rappresenta un ulteriore passo avanti nell'ottimizzazione della struttura del gioco. Questa soluzione non solo semplifica le operazioni di gestione delle risorse all'interno del sistema, ma ne aumenta anche la chiarezza concettuale, consentendo a voi di comprendere rapidamente il flusso delle risorse nel contesto del gioco. Tale implementazione contribuisce quindi alla riduzione della complessità del codice e alla facilità di manutenzione nel lungo termine.
- L'implementazione del **timeout**, sebbene **non** richiesta dalle *specifiche*, rappresenta un'aggiunta ragionevole alla logica di gioco che può migliorare l'esperienza utente, seppur complicando l'implementazione.

Lati negativi

Abbiamo identificato alcune **avvertenze**:

- `cardID` non è univoco per carte differenti (puoi avere due carte con lo stesso id ma side diverso).
- Solitamente conviene instanziare **tutte** le carte durante l'inizializzazione dal file json (non è chiaro il vostro approccio, cioè quando accedete al json se solo all'inizio o anche durante l'esecuzione).
- Sembra dall'UML che voi usiate il json come database, quando in realtà è scomodo accedere dal json durante l'esecuzione.
- `Game` non è connesso a `Table` nell'UML.
- Cercate di essere aderenti con i nomi degli oggetti a quelli delle specifiche (soprattutto per la `ENUM` di `Resource`, meglio non italianizzare la prima parte del nome delle risorse e mischiarla con la parte inglese).
- Attenzione a usare delle classi `*DB` (è possibile arrivare allo stesso risultato salvando lo stato di gioco direttamente nella classe principale, o implementando un insieme metodo che salva lo stato). Pur essendo un bello spunto, ha bisogno di essere spiegato meglio e trattato con attenzione. In particolare è una bella idea se si vuole implementare la possibilità di fare più partite contemporaneamente con uno stesso server, sfruttando uno stesso database (in cui, per esempio, quello delle carte ha l'unico scopo che ci sembra essere quello di associare un id a una istanza di `Card`) per più partite.
- In `Player` non ci è chiaro perchè tenete l'array di `Objective Cards`, le carte obbiettivo le si ricava dalla table o dalla mano del player.

Inoltre abbiamo identificato alcuni **punti critici** a cui consigliamo di fare particolare attenzione:

- Non vediamo la possibilità di gestire le *challenge* richieste dalle `goldCard` di tipo *angolo coperto* oppure in cui **non** ci siamo *challenge* nella `goldCard`, ma solo la possibilità di gestire le *challenge* in cui si possono guadagnare punti nel caso in cui si abbiamo degli *oggetti* grazie all'uso di `multiplierResource`.
- Non ci è chiaro come riuscite a verificare:
  - Le differenti tipologie di *challenge* nelle `objectiveCard`, cioè come verificate che le composizioni siano soddisfatte. Non come si verifica che una composizione sia soddisfatta, ma come si identifica una composizione di una *challenge*, cioè quel'è l'obiettivo della *challenge*. Avete usato solo due tipi di challenge questo non permette per esempio di differenziare una ⌊ (*knight*) o una ⌉ (sempre *knight* per voi).
  - Che si abbiano abbastanza *risorse* per soddisfare una *challenge* (si dovrebbe sapere il tipo di *challenge* una volta che la carta viene instanziata, dove tenete questa informazione?). Questo problema potrebbe essere risolto scrivendo nel codice le condizioni ma questo limita di molto l'espandibilità del gioco.
- `Position` è un `ENUM`, quindi dovrebbe essere implementata come una `ENUM` (e dunque tutti gli elementi dovrebbero essere `final`).

Confronto tra le architetture

- Il *deck* usato come coda, mentre noi non abbiamo definito bene la modalità di accesso al deck di carte.
- Enumerazione per `Side` (`BACK`, `FRONT`), noi non abbiamo reso esplicito con una `ENUM` il side ma abbiamo usato un boolean.
- Aggiunta del metodo `skipRound()`, anche se non esplicitamente richiesto dalla specifica.
- Loro usano dei *database* per accedere ai dati, mentre noi teniamo traccia di queste informazioni con delle `HashTable`.