Peer-Review 1: UML

Marco Scarpelli, Davide Trapletti, Martina Viganò
Gruppo 55

3 aprile 2022

Valutazione del diagramma UML delle classi del gruppo 65.

1 Lati positivi

1.1 School

La classe `School` permette di controllare in maniera più ordinata la gestione della plancia e la classe è facilmente serializzabile in caso andasse passata tra view e controller. Inoltre rende la classe `Player` più contenuta.

1.2 Divisione tra Team e Player

La divisione tra `Team` e `Player` consente di gestire le partite con 4 giocatori senza cambiamenti radicali alla classe `Player` e alla gestione della partita in generale.

1.3 Calcolo influenza

La classe `InfluenceCalculationPolicy` permette di decidere dinamicamente il calcolo tenendo conto dei `Character`s in uso. Diviene quindi teoricamente più facile gestire l'aggiunta di nuove carte al gioco o eventuali altri effetti.
Ci sono però secondo noi alcuni lati negativi: vedere 2.3.

1.4 Considerazioni

Abbiamo segnalato quelli che per noi erano gli aspetti salienti, ma in generale il modello ci sembra ben fatto e pensiamo che possa gestire le funzionalità aggiuntive desiderate in maniera abbastanza efficace.
La maggiore lunghezza della sezione Lati negativi è data dal fatto che ci siamo soffermati di più sulle motivazioni e abbiamo proposto delle soluzioni.

2 Lati negativi

2.1 Gestione isole

Le isole sono gestite con una `List<Island>` all'interno di `Match`, a cui si accede tramite indice. L'unione delle isole, si evince, ne elimina una dalla lista; la fusione delle isole però comporta la creazione di un'unica entità che contenga tutti gli `Student`s e le `Tower`s; ciò non consente più di distinguerle una dall'altra e potrebbe essere un problema quando, ad esempio, si volesse rappresentarle a video tenendo traccia degli `Student`s sui singoli isolotti e in generale del numero di isole in sè che compongano la macro-isola, anche se è un dettaglio più stilistico.
Potrebbe essere sensato aggiungere una classe che gestisca gruppi di `Island`s, spostando in essa la logica del conteggio delle pedine, oppure tener traccia nella singola `Island` delle altre isole che fanno parte del gruppo.

2.2 12 sottoclassi di carte

La soluzione non permette di gestire l'eventuale aggiunta di nuovi personaggi in modo ottimale.
L'approccio di dividere tra `Character` e `StudentCharacter` è sensato; sarebbe possibile unire ulteriormente altri personaggi suddividendoli semplicemente in base alla scelta richiesta al giocatore (o alla sua assenza, in quanto alcune carte sono "automatiche"), magari andando ad agire direttamente su `InfluenceCalculationPolicy` e ampliandola.

2.3 Calcolo influenza

`InfluenceCalculationPolicy` potrebbe essere ampliata per gestire tutti i possibili aumenti di influenza, che sono sparsi nelle altre classi.

Ad esempio, si potrebbe integrare in essa `Player.additionalInfluence` ed anche prevedere un attributo che conteggi la differenza fra il numero di professori posseduti e quelli del proprietario per poterne assumere il controllo, senza "nascondere" tale attributo direttamente nella funzione di controllo dell'ownership.
Per quest'ultimo, un altro approccio potrebbe essere quello di creare una classe ad hoc per gestire tale calcolo, ma sarebbe una classe poco utile; avrebbe quindi forse più senso integrare in `InfluenceCalculationPolicy` ed eventualmente rinominarla per riflettere il fatto che gestisca non solo il calcolo del punteggio ma anche dei professori.

2.4 Altro

- In `Tower` c'è il metodo `getColor`, ma `Player` ha un attributo `towerColor` salvato in locale - potrebbe essere direttamente dedotto da `Tower`.
- La classe `ThreePlayerMatch` potrebbe essere eliminata e previsti in `Match` una serie di attributi per il conteggio delle pedine assegnate in partenza.

3 Confronto tra le architetture

Sostanziale differenza fra la nostra architettura e quella presa in esame è l'utilizzo di classi per pedine, professori e torri al posto delle nostre liste di interi; le due scelte implementative ci paiono ugualmente valide in quanto hanno entrambe i propri pro e contro.
Altra differenza è l'assenza di una classe dedicata per il turno o la fase, ma anche questa può essere una scelta sensata perchè le operazioni preliminari, come ad esempio il riempimento delle nuvole, possono essere fatte svolgere al controller che si prenderà anche carico del controllo di flusso, potendo accedere a tutte le operazioni sugli oggetti a propria discrezione e senza un'impostazione "rigida" delle sequenze dettata dal modello in sè. Anche qui ci sentamo di dire che la scelta presenta pro e contro; non sentiamo di consigliare un approccio o l'altro.
La nostra architettura potrebbe invece beneficiare dall'aggiunta di una classe `School` all'interno del giocatore, costituita dall'aggregazione di due ulteriori sottoclassi `EntryHall` e `DiningHall` che implementerebbero l'interfaccia `ICanContainPawns` per standardizzare di più le chiamate a funzione.