Peer-Review 1: UML

Alessio Buda, Dario Mazzola, Gabriele Munafo'

Gruppo 21

Valutazione del diagramma UML delle classi del **gruppo 20**.

Note

Manca la gestione dei colori delle torri e la scelta del mago (retro della carta) del giocatore. Benché deducibili dal giocatore stesso potrebbe essere utile tenerne traccia per semplificare alcuni controlli. Inoltre, alcuni errori minori di sintassi, dovuti probabilmente alla poca dimestichezza con il tool grafico, non sono stati considerati nella review. Infine, ci scusiamo per eventuali errori dovuti ad un'interpretazione errata del diagramma.

Lati positivi

- La presenza nelle classi Bag e Game dei costruttori fa pensare che in realtà l'intenzione fosse di realizzare un pattern Singleton (nonostante non sia implementato correttamente). Se così fosse, l'idea potrebbe essere interessante, in quanto assicura l'esistenza di un'unica istanza delle relative classi.
- Nella classe SchoolDashboard è molto utile passare un parametro intero ai metodi per la gestione delle torri. Questo perchè, avanzando nel gioco, sarà sempre più comune che vengano conquistate isole unite e che quindi vadano sottratte o aggiunte più torri in un'unica soluzione

Lati negativi

- Gli studenti, con unico attributo `color`, potrebbero essere più semplicemente rappresentati, dove necessario, tramite una struttura dati che metta in relazione il numero di studenti con il colore corrispondente. Inoltre, questo eviterebbe di dover creare liste di oggetti che, osservando le cardinalità delle relazioni, arriverebbero a lunghezze particolarmente elevate (ad esempio nella classe Bag si avrebbero inizialmente 130 studenti). Questo chiaramente comporterebbe una occupazione di memoria eccessiva, oltre che un peggioramento delle prestazioni: per diverse operazioni sarebbe necessario scorrere l'intera lista (ad esempio in SchoolDashboard per contare il numero di studenti di un determinato colore per il calcolo dell'influenza) . La considerazione precedente è valida per ogni classe in cui è presente un riferimento agli studenti. Considerazioni analoghe valgono anche per la classe Professor.

- Nella classe Game è presente un metodo `checkInfluence()`, che riteniamo debba essere implementato nel controller e non nel model, dato che l'operazione è relativa alla logica di gioco e va dunque oltre le competenze del model. Inoltre, sarebbe necessario implementare un pattern *Strategy* per poter tenere conto dell'effetto di alcune carte personaggio.

- Sarebbe necessario implementare un metodo `insertStudent()` nella classe Bag per reinserire studenti nel sacchetto, operazione consentita dalla carta personaggio AllRemoveColor.

- Per prelevare gli studenti dalle CloudCard, sarebbe necessario implementare un metodo `removeStudents()` che modifichi l'attributo `full` (ponendolo a `false`) e che rimuova gli studenti dalla struttura dati relativa alla CloudCard. Quest'ultima operazione potrebbe anche essere omessa, dato che

la presenza di `full` a `false` potrebbe implicitamente indicare che i dati contenuti non siano significativi, dunque potranno essere successivamente sovrascritti. Inoltre, dato che il numero di studenti su una nuvola varia in base al numero di giocatori, sarebbe opportuno avere un attributo (`final`, dato che non varierà durante la partita) per indicare il numero di studenti che ogni nuvola può contenere.

- Nella classe IslandGroup, l'utilizzo dei puntatori all'isola precedente e all'isola successiva non rappresenta la soluzione più efficace per la gestione della lista di isole. Potrebbe essere più semplice delegare la sua gestione a Game, aggiungendo i metodi necessari per la gestione della lista. Ad esempio, il metodo `merge()` sarebbe più semplice da implementare nel Game, dato che non richiederebbe l'utilizzo dei riferimenti all'elemento precedente e all'elemento successivo. Sarebbe inoltre necessario inserire un attributo booleano che controlli la presenza della tessera divieto introdotta dalla carta personaggio BlockIslandOnce.

- La classe MotherNature risulta superflua in quanto potrebbe essere rappresentata in Game semplicemente da un attributo che ne indica la posizione della lista di isole.

- Nella classe Player, sarebbe necessario porre un attributo che tenga in memoria l'ultima carta assistente utilizzata, dato che nel regolamento è specificato che quest'ultima deve essere mostrata. Inoltre, il metodo `moveStudents()`, non avendo parametri in ingresso, non ha modo di capire se richiamare il metodo `moveToIsland()` o il metodo `moveToLounge()` della SchoolDashboard. Infine, sarebbe opportuno che i metodi moveMotherNature, chooseCloudCard e activateCharacterCard vengano spostati nel Game, poiché non strettamente relativi al Player. Inoltre, `moveMotherNature()` non riceve parametri in ingresso; quindi, non ha modo di cambiare la posizione della MotherNature.

- La relazione di associazione tra AssistantCard e Player, così com'è scritta, comporta che sia presente un attributo Player per ogni Assistant e un attributo di tipo AssistantCard per ogni Player. Di conseguenza, esisteranno 40 diverse carte assistente che si differenziano solo per l'appartenenza ad uno specifico Player. Sarebbe più semplice avere solo 10 carte assistente (senza alcun riferimento a Player) e avere in Player un attributo che rappresenti le carte (o meglio, l'intero mazzo di carte).

- Nell'implementazione delle CharacterCard, spesso il semplice utilizzo del metodo `doEffect()` non sembra sufficiente. Sarebbe necessario pensare a specifici attributi per realizzare certe funzioni. Ad esempio, nella carta Choose3ToLobby bisognerebbe indicare in qualche modo quali studenti sono presenti sulla carta. Inoltre, l'utilizzo dell'attributo `int usageCount` potrebbe essere sostituito con un `boolean` per indicare se la carta sia stata utilizzata almeno una volta. Infatti, come anche specificato sul canale Slack del corso, il costo della carta personaggio aumenta solamente dopo il primo utilizzo, rimanendo costante (incrementato di 1 moneta) per tutti i successivi utilizzi.

Confronto tra le architetture

Confrontando le due architetture UML, emergono diverse differenze. L'analisi di queste ci ha permesso di ragionare più a fondo sul nostro diagramma UML e di apportare alcune utili modifiche. In particolare:

- è interessante la scelta di implementare le classi Bag e Game mediante il pattern *Singleton*
- nella Cloud, può essere utile inserire l'attributo booleano `full`, in modo da sapere facilmente quali Cloud sono disponibili
- abbiamo notato la mancanza nel nostro diagramma degli attributi (e dei relativi metodi) necessari alla gestione delle monete

- abbiamo adottato metodi che implementavano l'aggiunta e rimozione di studenti, professori e torri operando su un singolo elemento. In alcune circostanze può invece essere utile avere metodi analoghi ma che operino con più elementi alla volta (nel diagramma UML del gruppo 20, un esempio è il metodo `addTowers(int numTowers)` di SchoolDashboard)