# Peer-Review 1: UML

Simone Callegarin, Giacomo Carugati, Filippo Buda
Gruppo AM07

3 aprile 2022

Valutazione del diagramma UML delle classi del gruppo AM37.

## 1 Lati positivi

Nella classe `GameController` al fine di gestire l'ordine di gioco dei giocatori nel turno è stata utilizzata la struttura dati TreeSet nell'attributo `assistantPlayed`.
Questo permette di tenere traccia automaticamente dell'ordine del prossimo turno e impedire di utilizzare la medesima carta.
Le enumerazioni sono state sfruttate opportunamente, in particolare in `FactionColor` è stato associato ad ogni colore un indice, facilitando così la ricerca nel vettore di interi che rappresenta gli studenti.
Per il design delle Character Cards è stato sfruttato un "database", ovvero una classe che contiene al suo interno la logica per costruire ciascuna carta e un'ArrayList di effetti base che andranno a comporre gli effetti completi.

## 2 Lati negativi

I nomi delle classi, attibuti e metodi risultano essere poco autoesplicativi, rendendone così complicata la comprensione del Design (Ad esempio `hasNoEntryTile` in `Island`, attributo che rappresenta le carte divieto oppure `intPar` in `Option` di cui non è chiaro il significato).
La classe `GameController` riporta una quantità ridotta di metodi per la gestione del gioco, inoltre non viene fatta una distinzione tra le diverse fasi e modalità di gioco.

La gestione della modalità esperto sembrerebbe usare un approccio procedurale, dato che non viene sfruttato l'approccio a oggetti per fare distinzione fra le due modalità, difatti si utilizzano booleani che comportano l'uso di numerosi if.
Alcuni metodi implementati in `Island` si potrebbero gestire con una classe `IslandManager`, come ad esempio `uniteIfPossible`, che gestisce il processo di unificazione delle isole dato che come attualmente implementato risulta essere costretto a ricorrere ad una eccessiva notazione puntata per la lettura di `islands:ArrayList<Island>`.
L'idea di utilizzare delle classi come container di studenti ci sembra un'eccessiva astrazione del concetto di insiemi di studenti.
A tal proposito risulterebbe più appropiato l'uso di interfacce che espongano per ogni classe i metodi per l'aggiunta e la rimozione di singoli studenti o di gruppi di studenti.
In particolare la distinzione tra la classe `UnlimitedStudentsContainer` e `LimitedStudentsContainer` risulta essere eccessiva, dato che non esiste alcun vero e proprio contenitore illimitato di studenti nel gioco.
All'interno del package `character` l'UML non rende chiaro le modalità di utilizzo e la natura di Option e State.
La distinzione fra classe `CharacterEffect` e `Character` al momento non presenta alcun vantaggio dato che potrebbe essere riconducibile ad una singola classe.
L'utilizzo di una struttura dati `coinsTaken` per tenere traccia delle monete guadagnate nella `diningRoom` è scorretto dato che da regolamento è possibile ottenere più volte una moneta dalla stessa casella.
Per le tessere divieto `noEntryTiles` non vi è alcuna classe che ne gestisce il numero, rendendo così possibile l'utilizzo di più di 4 tessere per partita.
La classe `Wizard` non è essenziale alle funzionalità dell'applicazione.

# 3 Confronto tra le architetture

Nel nostro caso l'approccio utilizzato comprende l'uso di interfacce, che non sono presenti nell'UML del gruppo revisionato.
Abbiamo inoltre sfruttato l'ereditarietà per differenziare tra modalità esperto e normale.
In riferimento alla gestione degli studenti abbiamo optato per l'utilizzo di

strutture dati, quali HashMap, mentre il gruppo revisionato ha deciso di implementare classi apposite unicamente per la loro gestione.
Per l'implementazione delle `CharacterCards` entrambe le architetture incorporano una logica per la creazione di singoli effetti base, ma presentano delle modalità di combinazione di tali effetti differenti.
A tal proposito nella nostra architettura è stato utilizzato un pattern Factory per l'assemblaggio degli effetti completi, mentre l'altro gruppo ha optato per una combinazione direttamente in un database.