Peer-Review 1: UML

Andrea Alari, Giovanni Barbiero, Chiara Bordegari
Gruppo AM33

1 aprile 2022

Valutazione del diagramma UML delle classi del gruppo AM06.

1 Lati positivi

In generale la struttura del gioco è ben suddivisa in più classi che riescono a semplificare il controllo della partita. La gestione delle isole e della board è ben fatta, mentre, anche se l'idea di separare il round in una classe a se stante è corretta, il modo in cui è implementata è confusionario (vedi sezione 2).
I pattern utilizzati a nostro parere sono adeguati, sia il `Singleton` per il `Game` sia il `Factory Method` per le carte personaggio. Quest'ultimo, pur essendo concettualmente corretto, non rende facilmente comprensibile in che modo viene gestita l'effettiva creazione delle carte.
Inoltre risulta interessante la gestione delle scelte dei giocatori all'interno dei `Player` anche se mancano gli attributi dove salvare queste scelte.

2 Lati negativi

Nel complesso non risulta possibile simulare il gioco con le classi e i metodi forniti.
Sono stati rilevati i seguenti problemi:

- I metodi di tutte le classi risultano con visibilità public o private assegnata in modo errato; ciò rende alcuni metodi inaccessibili dalle altre classi e altri accessibili anche se non utilizzati da classi esterne.

- `MotherNature` non necessita di essere singleton dato che ci si salva il riferimento nel `Game`. Inoltre il metodo `move` non può ritornare l'island group dato che non c'è alcun modo di accedere all'array di island group nel `Game` o metodi per ottenere l'island group dato l'index.

- Il `Round` è confusionario nella gestione della fase d'azione e concettualmente le `Cloud` non appartengono al round ma al `Game`. Inoltre consigliamo di non frammentare la gestione dei giocatori e di metterla tutta o nel `Game` o nel `Round` oppure in una classe apposita. Complessivamente, la struttura del `Round`, per come ci appare, manca di alcuni metodi che a nostro parere sarebbero fondamentali per il suo funzionamento. Detto questo è probabile che la logica della classe venga gestita senza utilizzare metodi più specifici come, per esempio, calculatePlayerOrder. Consigliamo, quindi, di spezzettare il più possibile i metodi per migliorare la leggibilità e facilitare il debug.

- La gestione degli studenti e dei professori può essere semplificata. Mantenendo la stessa struttura `Creature` dovrebbe essere astratta poiché non viene mai creata una `Creature` che non sia uno studente o un professore. Questa struttura potrebbe essere resa più semplice tenendo in considerazione che studenti e professori possono essere rappresentati dalla stessa classe e possono essere gestiti da degli array o set di Creature o CreatureColor dove cambia solo il nome (Es. professors: Set<CreatureColor> e students: ArrayList<CreatureColor>).

- La gestione delle carte personaggio è approssimativa: gli effetti delle carte sono impossibili da implementare dato che, non avendo alcun riferimento ad una classe, non può accedere ai metodi effettivi per applicare gli effetti. Questo funzionerebbe se il `Game` esponesse questi metodi e le carte ci accedono grazie al singleton. Vogliamo fare notare inoltre che è necessario gestire le scelte dei giocatori che possono variare in base agli effetti delle carte.

- Nell'UML mancano i costruttori, però crediamo che siano stati volutamente omessi.

- Il progetto potrebbe essere suddiviso in più package migliorando l'information hiding. Inoltre si potrebbe separare il `Game` per la modalità esperti creando una classe separata ed esporre all'esterno un'unica interfaccia.

3 Confronto tra le architetture

Abbiamo rilevato varie similitudini tra le architetture dei due progetti. Inoltre troviamo interessante la gestione delle scelte degli utenti all'interno del `Player`; nel nostro progetto è il Controller che riceve le scelte dall'utente e, dopo averle ricevute tutte, chiama i metodi del Game.