Peer-Review 1: UML

Leonardo Bianconi, Alexandru Gabriel Bradatan, Mattia Busso
Gruppo 36

4 aprile 2022

In questo documento verrà svolta una valutazione del diagramma UML delle classi del gruppo 09. L'analisi sarà suddivisa in 3 parti: individuazione dei lati positivi, di quelli negativi e un confronto con l'architettura del progetto del nostro gruppo.

1 Lati positivi

1.1 Uso di `Enum` per gli assistenti

Gli assistenti sono un'entità che contiene solo dati senza funzionalità. Perciò l'utilizzo di `Enum` è un approccio migliore rispetto a creare una classe.

1.2 Astrazione per il movimento degli studenti

L'utilizzo di una classe astratta `Place` che rappresenta un'entità capace di ricevere o mandare studenti ad altre entità dello stesso tipo è un approccio che permette di caratterizzare in modo efficace entità simili tra loro come ad esempio `Cloud`, `Entrance` e `Table`. Tuttavia, l'implementazione ha anche un difetto, vedasi 2.1.

1.3 La classe `Board`

L'utilizzo di una classe apposita che contiene tutti le entità che modellano oggetti fisicamente presenti sul tavolo di gioco permette di ridurre la dimensione della classe `Game` e di regolare l'accesso tramite i metodi esposti.

1.4 Uso del pattern *Strategy* per le carte personaggio

L'uso del pattern *Strategy* permette di astrarre il comportamento delle singole carte personaggio fornendo un'interfaccia unificata.

1.5 Uso del pattern *Factory* per la creazione di `Game`

L'uso de pattern *Factory* permette di nascondere il processo di creazione di `Game` all'esterno.

2 Lati negativi

2.1 Uso scorretto dell'ereditarietà

`TwoPlayersGame` e `ThreePlayersGame` non aggiungono comportamento alla superclasse e sarebbero potute essere implementate parametrizzando la classe `Game`.

La classe `Place` definisce anche comportamento per la gestione dei professori, funzione che non appartiene a tutte le sue sottoclassi (per esempio `Cloud`). Sarebbe stato oppurtuno definire due interfacce: una che caratterizza le entità che trattano studenti e un'altra che caratterizza quelle che trattano professori.

2.2 Uso errato o mancato del pattern *Decorator*

La decorazione della classe `Game` per aggiungere la modalità esperti risulta eccessiva: come anche per `TwoPlayersGame` e `ThreePlayersGame` sarebbe più semplice parametrizzare direttamente la classe `Game`. Il metodo `drawCharacter()` sarebbe più appropriato nella classe `Board` e chiamato durante `StartGame()`.

Un caso invece in cui l'utilizzo del pattern sarebbe adeguato è per modificare a runtime il comportamento dell'influenza.

2.3 `Coin` è superflua

La classe `Coin` è vuota e potrebbe essere rimpiazzata da un intero.

2.4 Carte personaggio definiscono il proprio effetto a runtime

Le carte personaggio hanno un comportamento fissato per tutta la durata del gioco. Il metodo `ChooseStrategy()` permetterebbe di cambiare a runtime gli effetti delle carte, comportamento che è contro le regole del gioco. Unificare `Character` e `StrategyFX` implementando una classe per ogni carta permetterebbe di ovviare a questo problema.

2.5 L'astrazione introdotta da `GamePhase` è insufficiente

La classe `GamePhase` è un'opportunità mancata per implementare un pattern *State* che avrebbe potuto esprimere in modo più efficace e chiaro la macchina a stati del gioco.

3 Confronto tra le architetture

Le due architetture differiscono su un punto fondamentale: il gruppo 09 implementa il pattern MVC utilizzando un controller "`thick`" che esegue, oltre alla validazione sintattica dei messaggi del client, anche la validazione semantica. La nostra architettura, invece, esegue il controllo semantico all'interno del model, assottigliando il controller.

Due punti di forza dell'architettura del gruppo 09 rispetto alla nostra sono i seguenti:

1. L'uso della classe `Board`
2. L'uso di `Enum` per gli assistenti

Nella nostra architettura le responsabilità della classe `Board` sono delegate alla classe `Game`, che però si occupa già della progressione della macchina a stati. Separare le due permetterebbe una migliore "separation of concerns" della classe `Game`. Inoltre la nostra classe `Assistant` teoricamente permetterebbe la creazione di carte con combinazioni di valori non valide. Un `Enum` permetterebbe di definire un insieme di valori legittimi.