Peer-Review 1: UML

Davide Tonsi, Mattia Zambetti, Leonardo Vaia

Gruppo GC35

Valutazione del diagramma UML delle classi del gruppo GC45.

# Lati positivi

L’UML proposto dal gruppo GC45 ha diversi lati positivi:

* L’idea di utilizzare la classe astratta Game Component è una buona idea, ma può essere gestita meglio (vedere lati negativi); essa, infatti, rende il programma più robusto e riadattabile.
* Ottimo è l’utilizzo dei pattern: in primis la classe Game implementata attraverso un Decorator Pattern può essere un buon approccio alla gestione dell’expert mode.
* L’utilizzo di un pattern per le carte personaggio (classe Character) è una buona soluzione per gestire le varie funzionalità, tuttavia, bisogna studiare bene il collegamento con i metodi di ExpertGame.
* Buono il fatto che sia già stata inserita una classe Controller, la quale si interfaccia con la classe Game, che, a sua volta, si occupa di gestire l’aggiornamento dello stato del gioco.
* La gestione di LaunchHall ed Entrance come classi a sé, e collegate a player, secondo noi è un’ottima scelta: permette infatti di gestire lo spostamento degli studenti in modo più efficace.

# Lati negativi

Cercando di essere il più critici possibile abbiamo incontrato alcune lacune nell’UML revisionato:

* La classe astratta Game component sarebbe meglio implementata come interfaccia; al di là di questo comunque va contro le regole del gioco il fatto che si possano inserire studenti nella Bag, o rimuovere dalle classi Island e LaunchRoom. Potrebbe essere uno spunto per gestire future modifiche alle regole del gioco, ma vanno limitate in caso non si possano usare (potrebbe essere utile invece creare un metodo separato in Bag che la riempie per una nuova partita).
* A nostro parere il Factory Pattern non è quello più adatto per la gestione delle carte personaggio, ma sarebbe meglio utilizzare uno Strategy Pattern.
* La classe Controller contiene troppi metodi ed è confusionaria se confrontata con la classe Game, essi dovrebbero essere infatti raggruppati in modo più utile, come ad esempio in un metodo per ogni fase del gioco. Attualmente Controller assomiglia più ad una copia di Game.
* Un errore grave nell’implementazione dei metodi all’interno della classe Game è restituire direttamente l’oggetto istanziato anziché una copia, permettendo così a classi esterne di modificare degli attributi privati, e perdendo di conseguenza una delle caratteristiche fondamentali della programmazione ad oggetti.
* La gestione delle torri, delle carte e degli studenti è abbastanza confusionaria: sono infatti gestiti tutti attraverso contatori, rendendo possibile, in caso di errori o funzionamenti errati, la presenza di un “oggetto” in punti diversi.
* Le carte non riportano il valore del movimento di madre natura, e una volta giocate la gestione dei movimenti e delle azioni successive è poco comprensibile (Game come agisce dopo che è stata giocata una carta?). La loro struttura dati potrebbe esser stata implementata attraverso byte (con un funzionamento simile a dei flag) per facilitare lo scambio di dati con il server; tuttavia, la stessa cosa può essere gestita in modo più pulito attraverso un protocollo di comunicazione ad hoc, il quale si occupa di scambiare informazioni sullo stato del gioco grazie a delle stringhe, nelle quali vengono codificati dei particolari dati.

# Confronto tra le architetture

Per quanto riguarda il pattern da adottare per le carte personaggio, una differenza tra il nostro modello e quello del gruppo revisionato è l’utilizzo dello Strategy anziché del Factory: riteniamo infatti che il numero di carte da creare sia sufficientemente basso da non richiedere un Factory Method, mentre la Strategy permette di inserire nuovi tipi di carte nel momento in cui il gioco venisse aggiornato, oltre a rendere semplice la gestione dinamica dei loro effetti.

Dal punto di vista dell’esposizione delle strutture dati private, avevamo commesso lo stesso errore: in diversi metodi restituivamo infatti degli attributi privati rendendoli modificabili dall’esterno. Per riparare a questo errore avevamo semplicemente cambiato l’approccio e rivisto i metodi, i quali possono funzionare in modo simile restituendo copie per mostrare lo stato; per quanto riguarda la modifica dello stesso è stato invece sufficiente creare dei metodi specifici per variare gli attributi privati.

Un esempio è la classe Player, che restituisce direttamente la LaunchHall: sarebbe meglio creare un metodo del tipo showLaunchHallState() in LaunchHall, il quale restituisce una copia dello stato interno. Ci sarà poi un altro metodo che verrà richiamato dalla classe Game sullo specifico Player per gestire un’azione, come ad esempio il movimento degli studenti da Cloud a Entrance. Tutto questo può successivamente essere usato anche dalla view, ottenendo facilmente una copia dello stato interno da leggere e mostrare.

Possibile consiglio per le sottoclassi ereditate dall’astratta GameComponent potrebbe essere ridistribuire i metodi, inseriti per ora nella classe Game, nelle altre classi (per esempio calculateInfluence direttamente su Island, la quale verrà usata dalla classe Game per le varie fasi del gioco).

Per quanto riguarda l’utilizzo dei contatori anche noi avevamo pensato di gestire in questo modo le torri; tuttavia, ci siamo resi conto che un modello simile è rischioso e poco comodo, oltre ad essere poco adatto ad un approccio Object Oriented.

Un ultimo aspetto differente rispetto al nostro progetto è la gestione di diverse componenti di gioco all’interno della classe Player: il mazzo di carte, infatti, è gestito solo da questa classe, unificando diversi oggetti reali e la loro modifica (come la gestione delle carte e la gestione della plancia) che sarebbe meglio a nostro parere tenere separate, anche con classi diverse.