Peer-Review 1: UML

Viola Renne, Sara Resta, Laura Puccioni
Gruppo GC16

Valutazione del diagramma UML delle classi del gruppo GC26.

1 Descrizione fornita dal gruppo CG26

Premesse:

- Funzionalità aggiuntive: 12 Personaggi, Persistenza, Resistenza alle disconnessioni (No 4 giocatori)
- Tutti i getter e setter delle classi non sono specificati, ma implementati nel codice

L'idea di base è quella di utilizzare un Decorator Pattern: l'interfaccia Game viene implementata dalla classe BasicGame, che salva lo stato del turno corrente se viene scelta la modalità del gioco per principianti. Se, invece, viene scelta la modalità per esperti, l'interfaccia Game verrà implementata da ConcreteExpertGame, che aggiunge un array di personaggi scelti casualmente e, all'interno dei suoi metodi, non fa altro che chiamare i metodi di un suo attributo istanza di BasicGame (come prevede il pattern). Quando viene chiamato il metodo useAbility di uno dei Character, si può procedere in due modi: se il character ha un effetto istantaneo sullo stato della partita (ex. aggiungi studenti là) useAbility modifica lo stato e ritorna void; se, invece, l'abilità modifica un metodo fondamentale della partita (ex. il calcolo dell'influenza) useAbility trasforma l'istanza del gioco in una GameMode specifica per il character, cosicché il Controller continui a chiamare i metodi principali dell'istanza del gioco senza modifiche nel codice di quest'ultimo. Alla fine del turno il gioco viene ritrasformato in un normale ConcreteExpertGame, copiando i valori dello stato. In BasicGame, il metodo Setup si occupa di

preparare il gioco a seconda del numero di giocatori. L'attributo islands è di tipo ArrayList<ArrayList<Island>>: inizialmente un ArrayList grande di 12 ArrayList piccoli con un'isola ciascuno. Se si fondono delle isole, si concatenano gli Array delle isole fuse.

2 Lati positivi

Nelle classi principali (BasicGame, ConcreteExpertGame) ci sono tutti i metodi necessari per svolgere la partita, ovvero la scelta delle AssistantCard, il movimento degli studenti sia nella DiningRoom che sulle Isole, il movimento di madre natura e la scelta della nuvola (ci aspettiamo che nel Controller, che non ci è stato fornito, ci sia anche un metodo per la scelta della carta personaggio che setterà di conseguenza il GameMode, se necessario, eventualmente questo metodo potrebbe essere aggiunto anche in ConcreteExpertGame). Nelle classi principali troviamo anche i metodi che verranno chiamati dai metodi elencati precedentemente, come l'assegnamento dei professori (chiamato nel momento in cui si sposterà uno studente nella DiningRoom) e il calcolo dell'influenza (chiamato nel metodo che muove madre natura) che ci aspettiamo contenga anche la possibile unificazione di isole. Inoltre, ci sembra che tutti gli aspetti del gioco siano stati affrontanti nel progetto, ovvero che siano presenti tutti i concetti necessari per un corretto svolgimento del gioco. Oltre a ciò, abbiamo apprezzato come si sia cercato di semplificare le classi utilizzando la modularizzazione. In questo modo tutti i concetti sono stati separati nelle loro apposite classi, le quali risultano ben compatte e facilmente leggibili. Infine, si è cercato di riutilizzare il codice il più possibile utilizzando l'ereditarietà e il Decorator.
Di seguito vorremmo elencare alcuni punti specifici:

Pattern Decorator L'uso del pattern decorator permette una gestione efficiente della scelta della modalità di gioco, inoltre consente di effettuare il cambio di alcune regole dovuto all'effetto di alcune carte personaggio (che si concretizza nelle quattro sottoclassi di ConcreteExpertExpertGame). In questo modo si sfrutta anche il riutilizzo del codice perché in ConcreteExpertGame i metodi dell'interfaccia Game vengono implementati richiamando i gli stessi metodi dell'attributo game (che è di tipo BasicGame).

Character L'uso di una classe per ciascun Character permette di estrarre e istanziare le tre carte in modo più compatto senza utilizzare costrutti decisionali. In questo modo, inoltre, è necessario istanziare solo tre classi per ogni partita avente modalità esperto.

Modularizzazione Riteniamo che la separazione dei concetti di Player e di SchoolBoard renda più semplice la gestione delle due classi, in quanto il Player rappresenta il giocatore, mentre la SchoolBoard rappresenta la sua plancia e dunque nasconde parte della sua complessità. Lo stesso vale anche per la creazione di Bag, che viene separata dal concetto di Game, e di Deck, che viene separato dal concetto di Player, in modo da mantenere più compatte le classi Player e SchoolBoard.

3 Lati negativi

Nella parte di modellazione delle regole base riteniamo che ci siano alcuni accorgimenti, alcuni di minore importanza e altri più rilevanti, che vorremmo elencare di seguito. Seppur, come scritto precedentemente, riteniamo che siano presenti tutti i concetti base del gioco, alcune classi e relazioni potrebbero essere gestite in maniera differente, soprattutto per permettere un più semplice implementazione del codice.

Island La classe Island e l'attributo islands in BasicGame potevano essere implementati diversamente. Ad esempio un problema potrebbe essere la gestione della NoEntry: essendo associata ad una singola isola, prima del calcolo dell'influenza sarà necessario scorrere tutta l'ArrayList<Island> per verificare che su quell'agglomerato sia possibile calcolare l'influenza.

Student, Tower Non riteniamo necessario avere una classe Student, in quanto l'unico attributo è colour, il cui concetto è già implementato nell'enum Colour. Allo stesso modo per la classe Tower, che contiene solamente il colore delle torri.

Wizard In Player manca il riferimento al mago (wizard) scelto

AssistantCard Le AssistantCard vengono istanziate 10 volte per il numero di giocatori, invece si potrebbero istanziare ciascuna una sola volta (per un totale di 10 istanze) per poi passarle ai vari giocatori che si uniscono al gioco. Abbiamo interpretato maxStep in Player come il numero di studenti che ogni giocatore deve muovere durante il suo turno oppure come gli spostamenti di madre natura. Nel primo caso riteniamo che si possa inserire l'attributo in Game, per evitare questo intero sia ripetuto in ogni giocatore e nel secondo caso si potrebbero usare delle get per ottenerlo da AssistantCard. Dall'UML non siamo riuscire a comprendere come e dove venisse mantenuta la AssistantCard corrente (ovvero quella giocata da ciascun giocatore all'inizio del turno corrente). Mantenere questo riferimento potrebbe essere utile per ottenere il numero massimo di step che può effettuare madre natura e anche per capire se quella carta è utilizzabile in quel turno, ovvero se nessun giocatore ha lanciato prima una carta assistente con lo stesso valore.
Aggiornamento: il gruppo CG26 ha confermato che era corretta la seconda ipotesi, e dunque vale il nostro consiglio di eliminare questo valore e di tener in Player una AssistantCard in modo da gestire meglio la scelta dei turni e la scelta delle AssistantCard

Cloud Per quanto riguarda le Cloud, queste sono un concetto che appartiene al tavolo, ovvero al gioco, e non al singolo giocatore. Di conseguenza, secondo noi potrebbe essere più lineare e di più facile implementazione (ad esempio all'interno del metodo chooseCloud) mantenere la lista di Cloud in BasicGame e non in Player. Inoltre, in questo modo ogni giocatore sceglierà una sola Cloud per ogni turno e i suoi studenti verranno immediatamente aggiunti alla entrance.

Gestione studenti e professori in SchoolBoard In SchoolBoard gli studenti e i professori vengono gestiti diversamente, ovvero la entrance e la professorTable vengono gestiste con una ArrayList, rispettivamente di Student e di Professor, mentre la diningRoom con un array di interi. Questo aspetto può rendere difficile l'implementazione del codice in quanto ci si deve ricordare che i tre concetti sono stati pensati diversamente e dunque devono essere trattati in maniera differente. Riteniamo che si potesse utilizzare lo stesso metodo per tutti e tre.
Aggiornamento: il gruppo CG26 ci ha comunicato che si trattava di un refuso nel loro UML

Attributi final Alcuni attributi potrebbero essere resi final, ad esempio players di tipo ArrayList<Player> in BasicGame o entrance di tipo ArrayList<Student> in SchoolBoard.

4 Confronto tra le architetture

Molti concetti sono stati trattati nello stesso modo, ad esempio la divisione tra Player e SchoolBoard, la classe Bag, la classe Island come isola singola, e le 12 classi per le carte personaggio che estendono una classe principale Character. Le modifiche che possiamo fare alla nostra architettura per migliorarla sono le seguenti:

Towers Ci siamo rese conto di dover aggiungere il colore delle torri ai giocatori.

Deck Non abbiamo pensato all'implementazione di un Deck all'interno di Player che potesse alleggerire quest'ultima classe.

Pattern Decorator Anche se nella nostra soluzione abbiamo implementato diversamente i Character considereremo la possibilità di sfruttare il pattern Decorator per rendere la nostra implementazione più estendibile. In particolare potremmo utilizzare i Character come decorator di una classe contenente tutti i metodi di Game che possono essere modificati tramite il metodo useAbility.