Peer-Review 1: UML

Alberto Schiaffino, Federico Spoletini, Alessandro Zenati

Gruppo 26

Valutazione del diagramma UML delle classi del gruppo 36.

Descrizione fornita dal gruppo 36

La classe Game contiene i metodi principali per le varie azioni che avvengono durante la partita: muovere le varie pedine (madre natura, le torri, gli studenti), calcolare l'influenza in un'isola (computeKing), controllare chi possiede i professori (checkProfIsMine), assegnare le monete ai giocatori, inserire gli studenti a caso nelle tessere nuvola, giocare una carta assistente, ecc.

Ci sono delle associazioni tra Game e tutte le componenti del gioco:

- le tessere nuvola (che hanno due sottoclassi ThreePlayersCloud e EvenPlayersCloud in base al numero di giocatori): a loro volta le nuvole hanno una associazione con gli studenti che contengono
- il sacchetto (Bag, che ha un attributo di tipo ArrayList <Student>)
- madre natura (Goddess): ha una associazione con Island, ha infatti un attributo di tipo private Island currentHost per dire quale isola in quel momento la sta ospitando
- le isole: associazione con Player, hanno un attributo di tipo private Player king che tiene conto di quale giocatore ha piu influenza su quell'isola precisa in un dato momento
- i players: ogni giocatore possiede una plancia (Board, che a sua volta è una composizione di Entrance, ToweBoard, Hall e ProfBoard) e un Deck
- il mazzo Deck, che è composto di AssistantCard
- i professori: c'è una associazione con Player perche ogni professore ha un attributo private Player currentOwner

Il Game ha inoltre un attributo di tipo GameState, classe che tiene conto: dell'ordine in cui si gioca (ad esempio c'è un metodo per resettare l'ordine dal momento che cambia ogni turno in base alla carta assistente giocata dai Players), delle carte giocate, si occupa dello stato del gioco.

Abbiamo inoltre deciso di usare delle interfacce StudentHost (implementata da Cloud, Island, RowHall, Entrance che sono tutti i punti in cui si possono spostare gli

studenti) e TowerHost (implementata da Island e ToweBoard che sono i punti in cui si possono trovare le torri). RowHall è la singola riga (sono cinque in totale, una per ogni colore) di 10 spazi che fa parte della Hall.

Abbiamo utilizzato il pattern Decorator per trattare alcuni effetti delle carte Personaggio: il Game (che ha un attributo boolean isExpertGame) implementa l'interfaccia AbstractGame, che rappresenta il gioco "non decorato", e che si trova nel package GameDecorators. In GameDecorators troviamo anche altre classi (che sono i vari decorators, ovvero gli effetti, ad esempio TwoAdditionalJumpsDecorator), e anche la classe GameDecorator che rappresenta appunto il Game "decorato" dei vari effetti delle carte.

Lati positivi

Il design, in generale, ci sembra ben svolto e molto accurato, riportiamo i principali aspetti positivi:

- **Modularizzazione e Scalabilità**: Pensiamo che la scelta di dividere il design in molto sottoclassi piuttosto snelle, come avviene ad esempio per RowHall dentro Hall oppure per Board dentro Player, possa rendere più veloci e facilmente realizzabili eventuali future modifiche e aggiunte al gioco.
- **Decorator Pattern per Abilità Personaggi**: L'uso di un Decorator Pattern per implementare l'effetto delle abilità personaggi sul gioco risulta particolarmente efficace; tuttavia, suggeriamo anche la creazione di una nuova classe Character (vd. Lati Negativi).
- **Game e GameState**: Riteniamo particolarmente valida la divisione tra Game e GameState: la classe Game implementa tutti i metodi utili allo sviluppo del gioco, che verranno chiamati dal Controller, che presumibilmente si occuperà della conversione dei parametri da interi a classi del modello; mentre la classe GameState mantiene l'ordine dei giocatori e delle rispettive carte giocate nel turno.
- **ThreePlayersCloud** e **EvenPlayersCloud**: Consideriamo particolarmente intelligente la creazione di due sottoclassi di Cloud per distinguire il numero massimo di studenti presenti sull'isola nella modalità a 3 giocatori o in quelle a 2/4 giocatori.

Lati negativi

- **DumbGame e ExpertGame:** Riteniamo piuttosto ridondante l'utilizzo di DumbGame, che può essere inglobata nella classe Game senza particolari effetti collaterali.
- **Goddess:** Salvare l'isola su cui si trova Madre Natura dentro Goddess potrebbe rendere piuttosto complicata l'implementazione di alcuni metodi del Game (come il calcolo dell'influenza sull'isola). Consigliamo di salvare semplicemente l'indice dell'isola su cui si trova Madre Natura in Game.
- **Implementazione delle abilità dei personaggi**: L'utilizzo di un Decorator Pattern per implementare l'effetto delle abilità dei personaggi non rende possibile mantenere l'informazione sui divieti o sugli studenti presenti sulla carta personaggio (ad esempio, il Personaggio 1 dovrebbe avere un insieme di studenti sulla carta, che possono essere spostati su un'isola scelta dal giocatore che usa l'abilità). Questa informazione non viene salvata nello stato del gioco nell'UML presentato.

Confronto tra le architetture

In generale l'architettura del gruppo 36 e quella del nostro gruppo si somigliano molto per alcune scelte progettuali come l'utilizzo di un Decorator Pattern per l'implementazione degli effetti sul gioco delle carte Personaggio e, in generale, sulla selezione delle entità che possono essere rappresentate da classi e sui rispettivi attributi e metodi. Abbiamo notato una maggiore propensione alla scalabilità e alla modularizzazione dell'architettura in esame.
Riteniamo particolarmente interessante sotto l'aspetto di astrazione del modello, forse meno da un punto di vista pratico e di sviluppo del codice, le interfacce StudentHost e TowerHost, che vengono implementate dalle classi che possono contenere rispettivamente studenti e torri al loro interno.
Inoltre, abbiamo notato, analizzando l'UML in questione, che possiamo creare una classe separata da Game che salvi l'ordine dei giocatori e le rispettive carte giocate, come la classe GameState.