Descrizione UML Modello

Gruppo CG61 (*Vento – Valentino – Verdicchio*)

1. Approccio Generale

Nell'approcciarci al pattern MVC si è deciso di realizzare un modello che contenesse sia lo Stato della Partita, e tutti i dati relativi ad esso e utili al suo aggiornamento, sia della *game logic* di basso livello.

Nella nostra visione, per l'appunto, il *model* contiene tutti i metodi di base che poi il *controller* chiamerà per aggiornare lo stato, con l'eccezione delle Carte Personaggio, che abbiamo preferito dettagliare solo nei loro connotati più generali, lasciando poi al *controller* il compito di implementarne gli effetti.

Questo ultimo punto rimanda anche a un altro principio che abbiamo tentato di tenere a mente durante la progettazione, ovvero la modularità e l'espandibilità. Numerosi metodi agiscono in modo standard su determinate collezioni e attributi: manipolando (spesso da *controller*) correttamente questi ultimi, è possibile implementare funzionalità avanzate (come quelle delle Carte Personaggio) utilizzando gli stessi metodi di base presenti nel model.

Illustrerò ora brevemente il funzionamento del modello per "macro aree", per evitare un'eccessiva verbosità nel descrivere ogni singola classe.

2. STATO PARTITA

Lo stato della partita è diviso in due classi: *CurrentGameState* e *CurrentTurnState*.

CurrentGameState è la classe “regina” del nostro *model*: contenendo riferimenti a tutte le classi utili è l’interfaccia ideale di comunicazione fra modello e controllore. In essa teniamo traccia di componenti fondamentali della partita, generalmente sotto forma di stato di oggetti di gioco (Professori, Carte Personaggio e così via) e lochiamo metodi di particolare importanza, e che spesso necessitano di dover accedere a molte delle classi meno importanti (*checkWinner* per esempio).

Il *CurrentTurnState* tiene invece semplicemente traccia del turno in cui siamo, se siamo in fase di preparazione o di azione, quale sia la squadra in testa e se il turno in cui ci troviamo sia l’ultimo della partita.

La divisione degli ambiti fra queste due importanti classi deriva dalla volontà di voler delineare due compiti funzionalmente ben precisi e distinti: il tracciamento dello stato attraverso gli oggetti della partita, e attraverso il concetto più astratto di turno.

2. GIOCATORI, SQUADRE E ASSISTENTI

Al *CurrentGameState* saranno associati due o tre *Team*, a seconda della modalità di gioco; al *Team*, a sua volta, vengono associati uno o due *Player*.

A ogni *Player* è associato un *AssistantDeck* composto di dieci Carte Assistente.

Team terrà traccia del numero di professori controllati, utile per la scalabilità (associando ciò al player ci sarebbero difficoltà per la partita a quattro giocatori, in cui i professori sono, al netto del calcolo dell’influenza, assegnati alla squadra), e delle Isole attualmente da esso controllate.

Player avrà invece informazioni sulla Scuola ad esso assegnata, sul suo mazzo di Assistenti e sulla carta volta per volta giocata e i suoi valori.

Da *Team* e *Player* vengono chiamati metodi fondamentali come *updateProfessors* che si riferisce ai professori nella plancia scuola del giocatore corrispondente, o i metodi di gestione Monete e Assistente.

3. PLANCE

Per modellizzare le varie plance in gioco abbiamo utilizzato il criterio secondo il quale ogni entità sulla quale è possibile piazzare una pedina può essere definita come plancia (con l'eccezione di due Carte Personaggio). Basandoci su questo, una classe astratta *Board* lega tematicamente tutte le plance al metodo *placeToken*, specializzato poi in ognuna delle classi plancia.

School viene associata univocamente a un giocatore e contiene strutture dati apposite per gestire *Entrance*, *DiningRoom* e il numero di Torri contenute. La variabile *Checkpoint*, inoltre, consente di tener traccia, in maniera sempre aggiornata tramite il metodo associato, degli spazi della *DiningRoom* relativi al guadagno di una Moneta.

Cloud contiene semplicemente una struttura adatta ad ospitare i vari Studenti.

Islands rappresenta "l'orologio" delle nostre isole; il motivo della sua esistenza è la funzione di *idManagement*, la quale si occupa, non con una certa complessità, di verificare la possibilità di formazione di Gruppi di Isole, raggruppare le stesse e riordinare gli *id* delle isole singole in modo opportuno.

Island invece si occupa, oltre che di contenere utili attributi che ne tracciano lo stato (per capire se l'Isola è in realtà un Gruppo di Isole, oppure se Madre Natura è presente o meno), del calcolo dell'influenza. Il metodo a ciò preposto, *calculateInfluence*, determina quale Squadra detiene maggiore influenza e dunque controlla l'Isola, appoggiandosi all'attributo *TeamsInfluence*, aggiornato dall'apposito metodo ogni volta che Madre Natura si "posa" sull'Isola stessa.

4. PERSONAGGI

Per quanto Riguarda le Carte Personaggio, si è deciso di implementarne (per ora) solamente otto. La loro scelta è tale da rendere possibile una loro suddivisione in “MacroClassi” : carte di modifica dell’influenza, carte che modificano attivamente il comportamento di Madre Natura e carte che ci permettono di piazzare Studenti extra.

Questo concetto è diviso però fra *model* e *controller*; l’implementazione “grezza”, la si può vedere qui nel *model*. L’implementazione di cui sopra, dove saranno racchiusi, e scelti tramite appositi *strategy*, gli effetti delle varie carte, verrà svolta dal *controller*.

5. CONCLUSIONI

Non rimane molto da dire su *Student*, *Pouch* e altri elementi il cui non possa essere molto facilmente estrapolato dall’UML stesso.

In conclusione, sicuramente la mancanza di alcuni oggetti come le Torri o i Professori può generare qualche perplessità; durante la fase progettuale abbiamo deciso di effettivamente modellare come oggetti gli elementi di particolare importanza funzionale e/o grafica. Per questo motivo le Torri e i professori, il cui unico attributo è di avere un colore, sono stati implementati come flag nelle classi che ne devono tenere traccia.

Diversamente invece per gli Studenti che, dovendosi trovare in numerose plance diverse, sono stati implementati come oggetti: supponiamo che questo ci semplificherà di molto il lavoro in fase grafica.

Stesso discorso per altre strutture dati: piuttosto che trattare una pesante matrice di oggetti, la *diningRoom* è diventata un semplice array di interi che, sfruttando la proprietà *ordinal* delle enumerazioni,

associa a ogni posizione un colore, e così tiene traccia degli Studenti posizionati.