Peer-Review 1: UML

Griffanti, Masiero, Ferrario

Gruppo 25

Valutazione del diagramma UML delle classi del gruppo 24.

Lati positivi

- Dal diagramma UML del gruppo 24 si può notare come abbiano utilizzato un numero contenuto di classi, che se ben definite e legate tra loro, possono facilitare la gestione del Model.

- All'interno della classe Game, tra gli altri attributi, abbiamo notato expertsVariant, booleano che consente di indicare se una specifica partita è giocata con la variante per esperti oppure no. Pensiamo che possa essere utile mantenere nella classe Game (corrispondente alla nostra classe Match) questa informazione, in modo da poter differenziare metodi e costruttori e attivare determinati controlli che risultano necessari per implementare correttamente la variante per esperti.

- Il gruppo 24 ha gestito lo stato di avanzamento del gioco con un enum, chiamato "GameState", dove ha elencato i possibili stati in cui può trovarsi una partita, diversamente dal nostro UML nel quale abbiamo pensato di lasciare al controller la gestione dello stato del gioco.
  Riteniamo importante in particolare il fatto di mantenere salvato o comunque avere un avviso nel momento in cui avviene una interruzione del server (il gruppo 24 ha infatti salvato nell'enumerazione prima citata anche la costante "DISCONNECTED"); i dati della partita vengono conservati e non modificati in attesa della riattivazione dello stesso.

Lati negativi

- Il primo aspetto negativo di cui ci siamo accorti è come, in alcuni casi, il gruppo 24 abbia definito classi, che a nostro parere risultano eccessivamente "grandi", ovvero contenenti troppi metodi e/o troppi attributi; il che è in contrasto con il principio secondo il quale è preferibile mantenere, senza portare all'eccesso, una suddivisione delle responsabilità, tramite la creazione di più classi, evitando il "sovraccarico" di una sola.
  È nelle classi Island e SchoolBoard, che possiamo trovare due esempi di questa pratica, ma su queste classi ci soffermeremo, per un'analisi più approfondita, nel paragrafo riguardante i confronti tra le nostre architetture.

- Gestione Coin: Per quanto riguarda la gestione delle monete, il gruppo 24 ha inserito la variabile contenente le monete non utilizzate all'inizio del gioco, una riserva di monete rimanenti disponibili allo scambio (all'inizio del gioco ci sono 20 monete, ogni giocatore ne riceve una, le altre rimangono sul tavolo come riserva generale).
  Nella classe Player è presente la variabile "numCoins: int" e i metodi "getNumCoins(): int" e "addCoin(): void"; tali metodi agiscono sulla variabile al fine di tenere aggiornato il

numero di Coins che un giocatore possiede. Risulta però mancante l'interazione tra classi al momento dell''utilizzo di una carta personaggio: la classe "Player", infatti, non si relaziona alla classe "CharacterCard", nella quale è presente il costo della carta personaggio.
È inoltre assente il controllo delle caselle della Dining Room: nelle regole viene infatti citato che se si sposta uno studente su una casella di quest'area contenente l'icona di una moneta, il giocatore ne riceve una. Non essendo presente la verifica di quali caselle vengono occupate in SchoolBoard, non avviene alcuna modifica (non c'è aumento di monete) in "numCoins" di "Player".

- Un ulteriore elemento, che ci sentiamo di suggerire di aggiungere, è un controllo sulle carte personaggio già utilizzate. Nello specifico, noi abbiamo gestito ciò attraverso il boolean dell'HashMap in CharactersManager mentre in DeckCharacterCard del gruppo 24 questo elemento è assente. In aggiunta a ciò, ci è sembrato ridondante l'attributo Deckofgame nella classe match il quale, ci sembra, svolga la stessa identica funzione dell'attributo cards in DeckCharacterCard.

I successivi, più che lati negativi, riteniamo siano note/promemoria riguardanti alcuni particolari, che crediamo possano essere d'aiuto per il gruppo 24:

- Facciamo notare quello che potrebbe semplicemente essere una dimenticanza dovuta ad una versione non definitiva dell'UML oppure ad una visione più astratta e meno rigorosa delle associazioni, ovvero la mancanza di riferimenti, nella classe Match, delle classi a cui questa è collegata. Nello specifico abbiamo notato ciò per quanto riguarda la classe Island, StudentBag, CloudTile e MotherNature.

- Pensiamo sia necessario modificare 'noEntryTiles' nella classe Island in un int, dato che vi possono essere più carte divieto su un'isola o gruppo di isole.

- Riguardo gli attributi della classe StudentBag: noi riteniamo poco utile l'utilizzo dell'attributo 'numStudents', il quale indica il numero di studenti rimasti nel sacchetto, poiché questo dato può essere facilmente derivato dall'altro attributo, ovvero l'array students, che contiene per ogni indice un intero che rappresenta il numero di studenti di un particolare colore, rimanenti nel sacchetto (basterebbe infatti andare a sommare questi interi per ottenere il numero totale di studenti rimasti).

- Il gruppo 24 ha gestito lo stato di avanzamento del giocatore con un enum, chiamato "PlayerStatus", dove ha elencato i possibili stati in cui può trovarsi un giocatore, diversamente dal nostro UML nel quale abbiamo pensato di lasciare al controller la gestione dei turni di gioco. Pensiamo possa essere utile anche il fatto di mantenere salvato lo stato "DISCONNECTED" di un giocatore, causato dalla sua disconnessione dal gioco, il quale continua saltando il turno del giocatore fino al momento della riconnessione.

Confronto tra le architetture

Abbiamo avuto un approccio differente nei seguenti punti:

- Schoolboard: il gruppo 24, a differenza nostra, non ha gestito separatamente le quattro sezioni che compongono la plancia del giocatore. Un vantaggio di questa implementazione potrebbe essere quello di avere concatenazioni di chiamate più brevi. Al contrario, nella nostra architettura abbiamo realizzato quattro classi aggregate a Schoolboard, e questo ci permette di mantenere una suddivisione degli attributi e dei metodi di ciascuna classe.

- Madre natura: per la gestione di madre natura il gruppo 24 ha deciso di utilizzare una classe dedicata. Questa scelta, mentre da un lato potrebbe favorire una visione più 'concreta' di madre natura stessa, dall'altro potrebbe portare a dei riferimenti superflui nel codice che vengono invece evitati nella nostra implementazione, nella quale madre natura viene gestita attraverso un flag e due attributi che lo modificano presenti nella classe Island.

- Isole: è stato utilizzato un approccio differente anche per quanto riguarda la gestione delle isole: il gruppo 24 ha utilizzato un'unica classe attraverso cui gestire la singola isola e l'unione delle stesse, a differenza di quanto fatto nella nostra implementazione in cui troviamo sia una classe arcipelago, che inizialmente rappresenta la singola isola e poi l'unione di queste ultime, sia la classe isola per la gestione di alcuni aspetti peculiari (come ad esempio il flag di madre natura discusso precedentemente).
  L'approccio del gruppo 24, seppur meno complesso ad una prima occhiata, ci è sembrato riduttivo e inadatto a rappresentare in modo chiaro ed esaustivo lo stato di tutte le tessere-isola, questo perché con una sola classe prevedono di gestire, come spiegato sopra, sia l'isola singola (intesa come tessera-isola), sia l'unione di più isole.
  In definitiva riteniamo che questa scelta architetturale possa causare alcune complicazioni in fase di implementazione.

- Classe Match: una caratteristica evidente dell'architettura del gruppo 24 è la centralità della classe Game la quale, di fatto, è collegata in associazione e aggregazione con tutte le parti fondamentali del gioco. Abbiamo trovato questa implementazione interessante soprattutto in relazione alla nostra classe 'Realm', la quale funge da intermediario tra la nostra classe Match e alcune parti fondamentali del gioco. Su quest'ultima abbiamo ragionato, prendendo spunto dal gruppo 24, che potrebbe essere vantaggioso eliminarla per favorire, eventualmente, un legame più diretto tra le varie parti del gioco e la classe principale, match per l'appunto.

- Assistantcard: Nell'osservare la classe AssistantCard abbiamo ragionato sullo stato 'expired' presente nell'enumerazione che si occupa dello stato della carte assistente. Nello specifico, crediamo che questo possa essere superfluo, infatti non serve nel gioco conservare carte assistente già utilizzate ma è necessario solo avere un qualche riferimento all'ultima carta usata che finisce nella pila degli scarti, azione che noi svolgiamo attraverso il metodo 'useCard' nella classe AssistantsDeck.

- PlayerStatus: per quanto concerne la gestione dello stato del giocatore, cioè se si trova in attesa del suo turno di gioco, se è attualmente in gioco o se ha raggiunto la fine del gioco con conseguimento della vittoria, il gruppo 24 ha utilizzato un'enumerazione, contenente

una costante per situazione (tre totali), con riferimento e relativi metodi nella classe Player. Noi, differentemente, abbiamo pensato che il controller possa essere il gestore di queste tre situazioni: sarà il controller a stabilire i turni di gioco, relazionandosi appositamente con la nostra classe Match.