Peer-Review 1: UML

Gabriele Carminati, Gabriele Carrino, Matteo Cenzato, Alessandro Capellino

Gruppo AM08

Valutazione del diagramma UML delle classi del gruppo AM17.

# Lati positivi

Da una prima analisi fatta sul diagramma UML ne è risultata una buona comprensione del problema nel suo complesso. Infatti, si nota una suddivisione corretta del pattern MVC, anche se la nostra relazione sarà incentrata solo sulla discussione sulle singole parti Model e Controller.

La scelta di integrare subito la parte relativa alla connettività e alla View (anche se solamente come prima bozza) aiuta inoltre a capire il flusso delle informazioni scambiate, anche se a un livello di astrazione molto elevato.

Nel nostro diagramma UML al momento della stesura di questo documento non abbiamo ancora pensato alle parti di connessione e View, ed infatti stiamo riscontrando delle leggere difficoltà ad implementare le funzionalità di rete, date dal fatto che dobbiamo costruirci il modello praticamente da zero.

Relativamente al controller, è interessante la scelta di usare le interfacce “BoardSetter”, “ShelfSetter”, “PlayerManager” (magari le nomenclature sono da rivedere) che espongono al Controller solo alcune delle funzionalità del Model.

La costruzione delle classi e la rispettiva suddivisione dei compiti di ciascuna classe è molto chiarificata, non abbiamo impiegato molte risorse mentali nel capire di che cosa si occupano i metodi scritti nel diagramma. Abbiamo solamente richiesto una veloce discussione orale con il gruppo AM17 per alcuni refusi (probabilmente dati da piccole distrazioni) relativi, ad esempio, a dei metodi getter che ritornavano ‘void’, che sono stati prontamente corretti.

# Lati negativi

Dal diagramma UML, anche se le classi sono ben chiare, il flusso del gioco invece ci è risultato meno intuitivo.

Partendo dal presupposto che magari la classe ‘Game’ potrebbe essere scorporata, con uno State pattern o semplicemente suddividendola in più piccoli componenti adibiti ognuno a una piccola sottofunzione, però è un problema che è già stato notato dal gruppo e ci hanno notificato direttamente nella descrizione del loro UML.

Un suggerimento in quest’ottica può essere quello di spostare il metodo ‘giveShelfie()’ direttamente nella classe ‘Player’, ad esempio nel suo costruttore, e lasciare solo sulla classe ‘Game’ solamente dei metodi invocati dal ‘Player’ che permettono la modifica della libreria.

Questo semplicemente perché ogni libreria è intrinsecamente legata al singolo giocatore.

Un esempio di metodo in ‘Game’ che potrebbe essere creato è ‘Game.makeMove(Player p, int column)’, così che basta chiamare un metodo ‘p.addMoveToShelf(int column)’ (ad esempio) presente nella classe ‘Player’.

Inoltre, relativamente al flusso del gioco, non ci era ben chiaro dove avvengono i controlli sulla correttezza della mossa (ad esempio se la mossa contiene tre pezzi in ordine tutti con almeno un lato libero), anche se erano presenti dei metodi come ‘checkLegalChoice’ e ‘checkLegalInsert’. Parlandone con il gruppo AM17 ci è stato comunicato che la loro intenzione è quella di effettuare i controlli lato client, così da alleggerire la mole di dati scambiati.

Il nostro consiglio è comunque quello di lasciarli solamente sulla parte server, perché richiamando il modello MVC è solamente il componente Model che contiene tutta la logica di gioco e permette l’avanzamento della partita.

Inoltre, prendendo sempre come esempio il metodo ‘giveShelfie()’, si nota che esso può essere lasciato privato, visto che nessuna classe oltre al ‘Game’ stesso (sia in Model che in Controller) dovrà accederci per nessun motivo.

Quindi consiglieremmo di rivedere le visibilità dei metodi, soprattutto quelli modificatori, e quelli che non devono essere chiamati da nessun’altra classe sarebbe preferibile tenerli privati oppure protected, così da avere un’ulteriore sicurezza relativa all’information hiding della classe.

Da un punto di vista di design invece, è questionabile la scelta di trattare i badge che corrispondono al punteggio come classi che implementano l’interfaccia ‘Badge’.

Questo perché ci sembra eccessivo trattare il punteggio relativo alla fine della partita (+1) come un oggetto a sé, perlopiù statico. Semplicemente si potrebbe aggiungere come un +1 al giocatore che finisce per primo la libreria nel calcolo dei punteggi, come abbiamo fatto noi nel nostro programma.

L’idea non è sbagliata in sé, ci sembra solamente una “iper-ingegnerizzazione” del problema, visto che per un semplice +1 al punteggio viene fatto tutto un giro di classi non banale.

Inoltre, l’altra classe ‘BadgeScore’ che implementa ‘Badge’ ci ha fatto dubitare della correttezza della scelta.

Infatti, prima di tutto è una enumerazione che implementa un’interfaccia, e anche se le nostre competenze di buona programmazione in Java sono molto limitate, non ci sembra corretto dal punto di vista logico correlare il concetto di enumerazione (lista di oggetti/componenti statici) e interfaccia (contratto per l’implementazione di metodi).

Secondariamente, non ci è chiaro come viene effettuato l’assegnamento dei punteggi: su cosa si basa il metodo ‘givePoints()’? Cosa succede al suo interno quando viene chiamato?

Il metodo, infatti, come fa a conoscere lo ‘stato’ della pila dei punteggi (visto che è presente solo nella classe ‘CommonCards’)? Come fa a sapere se a un giocatore il punteggio per l’obiettivo comune è già stato assegnato (e quindi non deve essere riassegnato un’altra volta)?

Sono domande secondo noi importanti alle quali però il diagramma UML non risponde.

Un primo suggerimento per concludere la parte è quello di togliere la classe ‘BadgeScore’ e nella classe ‘CommonCards’ trattare ‘badges’ come una semplice lista di interi e gestirli come se fosse una pila di numeri (per partite con numero di giocatori diversi si potrebbe caricare queste informazioni da file, come fa il nostro gruppo).

Per concludere l’analisi vorremmo sottolineare un grosso punto interrogativo. L’interfaccia ‘PlayerManager’ presenta un metodo ‘getCurrentPlayer(): Player’, che espone al Controller una proprietà intrinseca del Model, ovvero la possibilità di accedere direttamente alla classe ‘Player’.

Ecco, secondo noi questa cosa viola leggermente il concetto di information hiding che offre il pattern MVC: si sta cercando di esporre qualcosa che è nel ‘rep’ (astratto) a un utilizzatore (il Controller).

Per ovviare a questo problemino basterebbe rimuovere il metodo getter ed esporre solamente nell’interfaccia dei metodi presenti nella classe ‘Game’ (ad esempio un metodo ‘Game.updateScoreForCurrentPlayer()’ che richiama il metodo ‘Player.updateScore()’ al giocatore corrente).

# Confronto tra le architetture

Da un confronto tra le due architetture (guardando solamente ai diagrammi UML) si notano molte similarità sull’uso delle classi e delle loro relazioni.

La prima differenza che spicca particolarmente è la scelta di suddividere gli obiettivi comuni in 12 classi (loro) rispetto alla scelta di raggruppare gli obiettivi in 6 classi comuni (più 3 di utilità varie legate alla lettura di file di configurazione JSON).

Premettendo che anche la nostra prima idea è stata quella di avere 12 classi diverse, che poi sono state raggruppate unendo tra loro quelle concettualmente simili e configurando l’obiettivo preciso tramite un file di configurazione json, come ha detto il professor Margara a lezione “non esistono scelte di design più corrette o più sbagliate, esistono solo scelte” (parafrasata leggermente). La scelta del gruppo AM17 risulta decisamente meno scalabile, ma è comunque ragionata visto che nella loro presentazione ci è stato comunicato che secondo loro, visto che non deve essere presente nessuna “feature” del gioco che richieda più obiettivi da implementare, preferiscono avere una divisione netta delle classi.

Un’altra differenza è la scelta dell’utilizzo di interfacce per la comunicazione tra Model e Controller. Il nostro gruppo infatti ha una classe (chiamata ‘GameModel’) che è composta sia da metodi pubblici che possono essere chiamati dal componente Controller, sia da metodi privati che servono per la configurazione iniziale della partita e per utilità varie.

Crediamo che entrambe le scelte siano un’alternativa valida, è interessante la scelta di interfacce perché obbligano sia il Model che il Controller a rispettare il contratto (metodi) che portano con sé. Dall’altra parte, avere il riferimento alla classe è egualmente rispettabile visto che il nostro contratto sono tutti i metodi pubblici che vengono esposti (sono simili per entrambi i gruppi).

La vera forza del loro UML, che il nostro non possiede ancora, è però sicuramente l’idea concreta di tutti gli aspetti dell’applicazione, Model, Controller, protocollo di comunicazione, View. Come già detto in precedenza è ovviamente una sola prima bozza, però avere già in mente di astrarre l’azione che il client dovrà inviare al server (classe Action) e di come viene effettuata la comunicazione è sicuramente un punto a favore visto che la nostra produzione di codice sta rallentando proprio perché stiamo investendo la maggior parte del tempo a pensare le classi che vogliamo usare per la comunicazione.

Concludendo, il diagramma UML del gruppo AM17 ci è sembrato buono, di sicuro è stato pensato e non buttato a caso giusto per avere qualcosa. Si vede che hanno una buona idea di come proseguire e speriamo che le nostre osservazioni vengano accolte con un occhio critico. Abbiamo cercato di essere il più oggettivi possibile, cercando di argomentare perché secondo noi ci sono alcune parti del diagramma da rivedere e ripensare un attimo, e cercando di mettere esempi, anche tratti dal nostro diagramma, per dare uno spunto di pensiero verso quella che è secondo noi una scelta di design migliore.

Se comunque rimangono dubbi, critiche alle nostre critiche, o qualsiasi cosa siamo comunque aperti a discussioni e spiegazioni.