Peer-Review 1: UML

Niccolò Brembilla,
Luigi Bruzzese,
Eleonora Cabai,
Milagros A. Casaperalta Garcia

Gruppo AM31
28 marzo 2023

Valutazione del diagramma UML delle classi
del gruppo AM40.

A.A. 2022/2023
Prof. Alessandro Margara

1 Lati positivi

Nel diagramma UML del gruppo AM40 abbiamo identificato i seguenti aspetti positivi:

- la rappresentazione della classe *Bookshelf* come *ArrayList* di *Column* può risultare molto vantaggiosa per i controlli da eseguire per le CommonGoal e per le PersonalGoal;
- l'utilizzo delle mappe per rappresentare le posizioni delle *Tiles* sia sulla Bookshelf, sia sulla Board.
- la creazione di una classe *Position*, che viene sfruttata nelle mappe
- l'utilizzo di un file json per rappresentare i *PersonalGoal*
- la rappresentazione remota della View, in modo da gestire cosa esporre al client
- l'utilizzo degli Observer per l'aggiornamento della *VirtualView* e il conseguente invio dei messaggi ai client
- l'utilizzo delle interfacce *IGame* ed *IGameObserver* per scandire quali metodi riguardano il pattern Observer

2 Lati negativi

Nel diagramma UML ricevuto, innanzitutto, manca una chiara rappresentazione dei package in cui dovrebbero essere suddivise le classi create e questo rende difficoltosa la comprensione della struttura del programma. Abbiamo notato che, in molte classi, ci sono dei metodi con nomi simili e che quindi sembrano dover eseguire le stesse azioni, oppure dei metodi che a nostro parere possono essere "condensati" in un unico metodo, per esempio

- i metodi *freeSpaceCol()* e *isFull()* della classe *Bookshelf*, che controllano lo spazio libero su una determinata colonna e potrebbero diventare un unico metodo;
- il metodo *configureGame()* della classe *Game*, che al suo interno potrebbe contenere le azioni eseguite dai metodi *assignComGoal()* e *assignPersonalGoal()* in quanto sono azioni che vengono eseguite una sola volta all'inizio del gioco.

Alternativamente, tali metodi potrebbero essere resi privati ed essere invocati all'atto della creazione del gioco in successione da metodi interni. Anche alcuni metodi di controllo potrebbero essere resi privati, in modo particolare nei *CommonGoal*, per esempio il metodo *sameColors()* nel *CommonGoal1*.

Nel model, si potrebbe rivedere la struttura della classe *Tile* che prevede, tra gli attributi, una variabile *Position*: quando la *Tile* viene selezionata dalla *Board* e inserita nella libreria, si modifica l'attributo *Position* oppure si crea direttamente un oggetto diverso? Ecco due piccoli suggerimenti su come si potrebbe migliorare questa interazione:

- per la *Bookshelf*, si potrebbe evitare questo attributo mantenendo l'*ArrayList* di *Tile* della classe *Column* ordinato (dal basso verso l'alto o viceversa, in base a come risulta più comodo);
- per la *Board*, si potrebbe utilizzare una rappresentazione matriciale o, sempre mediante *ArrayList*, per mantenere un ordine preciso.

Non ci è molto chiaro, a tal proposito, l'utilizzo della *Map⟨String, Tile⟩*.

Dalla signature dei metodi presenti sul controller, si nota che ogni invocazione effettuata prevede il passaggio della singola *VirtualView* del client: ci chiediamo se sia strettamente necessario, dal momento che poi verrà invocato il model cui, nella maggior parte dei casi, non verrà passata. Quali informazioni della *VirtualView* servono al controller? Se, ad esempio, solo il nickname dell'utente, si potrebbe optare per un passaggio più ristretto di parametri.

Il modello delle connessioni è basato sulla tecnologia socket; infatti, nella documentazione allegata, viene descritto come il *ClientHandler* gestisce i messaggi inviati dal client, che sono parsati prima di chiamare i metodi del model. Nel caso di RMI invece, l'utilizzo dei messaggi standard riduce le potenzialità di tale tecnologia, che dovrebbe chiamare i metodi direttamente sul controller anche nel verso opposto, utilizzando la *VirtualView*. A nostro parere, sarebbe più corretto suddividere le due tipologie di connessione, riflettendo sulle loro differenti modalità di utilizzo. Non è chiaro se, per la creazione dei messaggi da inviare al client, la *VirtualView* utilizza il *ClientHandler* oppure no: se così fosse si potrebbero riscontrare dei problemi nell'implementazione di RMI.

L'invio delle intere classi *Board* e *Player* verso il client, secondo noi, andrebbe evitato. Consigliamo quindi di considerare una delle seguenti modifiche

- l'invio della sola mappa contenente le *Tiles* e le corrispondenti posizioni, in modo da non rendere possibile al client l'utilizzo di tutti i metodi della *Board*;
- l'utilizzo di classi condivise estese nel client e nel socket, in modo da permetterne delle diverse implementazioni.

3 Confronto tra le architetture

Per quanto riguarda il model, la gestione dei *PersGoal* è simile alla nostra (mediante file di configurazione json), mentre per i *CommonGoal* il gruppo ha deciso di intraprendere la via delle 12 sottoclassi: scelta sicuramente valida anche se leggermente meno efficiente della riduzione da noi effettuata. Il consiglio che sentiamo di dare è quello di valutare l'accorpamento tra *CommonGoal* diversi: siamo a disposizione per discutere degli algoritmi che abbiamo implementato.

Il pattern MVC è stato applicato dal gruppo nella variante secondo cui il model dialoga direttamente con la *VirtualView*, non passando per il controller. E' la strada alternativa a quella che abbiamo intrapreso noi (cioè con il controller posto tra model e *VirtualView*) ed è altrettanto valida: analizzarla ci ha consentito di riflettere su eventuali metodi mancanti sul nostro diagramma.

Ci chiediamo se, il metodo di notifica del pattern *Observer* tra *Game* e *VirtualView*, preveda che la *VirtualView*, una volta notificata con il metodo *update()*, richieda gli aggiornamenti manualmente al *Game* per poterli inviare al client (ad esempio quando cambia la *Board*, o una libreria). Suggeriamo di valutare l'utilizzo del più innovativo pattern *Listener*, che consente, tra le altre cose, di notificare gli observer (in questo caso la *VirtualView*) passando anche dei parametri (potreste, così facendo, passare direttamente ciò che avete aggiornato, senza doverlo poi richiedere al Game).

Qualora si volessero implementare alcune funzionalità aggiuntive, suggeriamo di partire dal salvataggio periodico della partita su file; con il diagramma realizzato, infatti, non dovrebbe essere difficile recuperare le informazioni necessarie. Anche salvarle, dato il già presente utilizzo del formato json per i *PersonalGoal*, dovrebbe risultare altrettanto agevole da scrivere da parte vostra.