Peer-Review 1: UML

Samuele Allegranza, Matteo Arrigo, Lorenzo Battini, Federico Bulloni
Gruppo AM13

29 marzo 2024

Valutazione del diagramma UML delle classi del gruppo AM22.

1 Lati positivi

Riteniamo che i metodi esposti verso il controller tramite l'interfaccia del Model specificata da `Model_Controller` risultano chiari e ben definiti, delimitando in modo netto il distacco tra Model e Controller.
Inoltre, i metodi di maggior interesse per le dinamiche di gioco sono presenti nel diagramma UML. Provando a simulare una partita abbiamo trovato tutti i metodi necessari per avanzare nelle varie fasi di gioco.
Abbiamo apprezzato la chiarezza riguardo alla divisione delle classi che compongono il modello e il tentativo di non renderlo troppo articolato.

2 Lati negativi

Elenchiamo gli aspetti che riteniamo possano essere migliorati riguardo al design UML del Model di gioco:

- La classe `Player` non presenta un attributo che rappresenti il nickname del giocatore, contrariamente a come richiesto dalle specifiche di gioco, le quali impongono che venga scelto lato client.
  Un appunto analogo, e più puntiglioso, riguarda il colore che, per quanto siamo riusciti ad intuire, non viene scelto dal giocatore, bensì in modo automatico dal server. Le regole specificano diversamente.

- Nella classe `Player` gli attributi `rows` e `columns` possono essere rimossi e le informazioni riguardo alle dimensioni sono ricavabili direttamente chiamando il metodo `size()` di `ArrayList`. Il problema può essere risolto alla radice usando la soluzione proposta al punto successivo.

- A nostro avviso utilizzare una matrice dinamica per memorizzare le carte piazzate dal giocatore ne complica la gestione, dato che richiede di aggiungere colonne o righe sia all'inizio che alla fine, con eventuale shift degli indici. Per esempio, questo richiede di dover accettare anche coordinate negative in `PlayTheCard()`, che poi dovranno essere rese positive in seguito all'aggiunta di una riga e/o una colonna. Inoltre è inefficiente a livello di memoria e spazio utilizzati.

A tal proposito si vedano, ad esempio, rappresentazione di matrici sparse mediante hashmap o rappresentazione di grafi mediante liste di adiacenza [1].

- In generale, il valore di ritorno di un metodo non dovrebbe avere a che fare con la gestione di eventuali errori durante l'esecuzione. Per situazioni di questo tipo può essere più consono usare le `Exceptions` di Java. Abbiamo identificato più metodi che ritornano dei valori interi per indicare il successo/insuccesso degli stessi (a titolo di esempio riportiamo i metodi `Game.PlayTheCard()` e `Player.placeCard()`).

- Abbiamo identificato metodi che usano valori di ritorno interi ma che potrebbero essere rimpiazzati con dei booleani (a titolo di esempio riportiamo il metodo `Player.haveCard()`).

- A nostro avviso, la classe `ShownCards` dovrebbe essere un attributo di `DeckToChoose` e non una sua sotto-classe, perché altrimenti erediterebbe l'attributo `ArrayList<> deck` da `Deck`, ed in ogni caso rappresentano due concetti distinti. Questa soluzione permette inoltre di spostare in `Deck` il metodo `shuffle()`, sfruttando al meglio la gerarchia.

- Gli attributi di `GoalCard` non sono, secondo noi, sufficienti a rappresentare la complessità e diversità delle carte obiettivo. Un discorso analogo può essere fatto per l'attributo `details` della classe `FGold`.

- A nostro avviso, `GoldCard.returnTypeCond()` dovrebbe ritornare un array di `Resource` come nel relativo metodo di `FGold`.

- Mantenere in due `Enumeration` separate `Obj` e `Resource` costringe a fare un casting a `Object` (si veda `MCard.getElementCorner()`), oltre a dover tenere separate le informazioni relative alle due (per esempio in `Angle`). Potrebbero piuttosto essere unite in un'unica `Enumeration`.

- `setCornersCovered` e `getCornersCovered` e il relativo attributo `cornersCovered` in `MCard` possono essere spostati in `GoldCard` dato che tale informazione può essere utile solamente nel caso in cui bisogna contare i punti istantanei delle carte gold che danno punti per ogni angolo coperto.
  Inoltre, a nostro avviso, questo attributo non serve dal momento che il calcolo può avvenire senza l'uso di questa "variabile temporanea".

- Riteniamo che ci sia un abuso di attributi booleani per distinguere la tipologia di carte, e che vengano sfruttati poco il polimorfismo e l'ereditarietà, in stile più *Object Oriented*. A conferma di ciò citiamo l'esistenza di `StartingDeck` senza una corrispondente `StartingCard`.

Riportiamo, inoltre, una serie di dubbi che abbiamo riscontrato nel provare a simulare una partita di gioco:

- I metodi `Game.StartGame()` e `Game.EndGame()` ritornano `void`, quindi come fanno a comunicare l'esito del controllo che effettuano? Forse dovrebbero ritornare un valore booleano, oppure gestiscono l'esito con una `Exception`?

- Come fa `EndGame()` a sapere che è l'ultimo turno? Quale metodo o attributo indica che un giocatore ha superato i 20 punti?

3 Confronto tra le architetture

Analizzando questo UML, ci siamo accorti di alcune dimenticanze sulla nostra implementazione, come ad esempio un metodo per verificare che le carte dei mazzi siano finite, con conseguente conclusione del gioco.
Un altro punto di forza di questo schema è la compattezza delle gerarchie, che comunque mantengono un livello di dettaglio sufficiente. Prendendo ispirazione da questo approccio abbiamo deciso di eliminare delle classi inutili sul nostro UML.
Tra le maggiori differenze di gestione dei dati, abbiamo notato che noi teniamo tra gli attributi delle carte tutte le informazioni di utilità nel gioco, mentre l'architettura analizzata ricava alcune informazioni indirettamente (per esempio l'eventuale risorsa nel retro delle carte, se questa è una carta oro o risorsa). Questo permetterebbe di risparmiare spazio in memoria, anche se renderebbe più complessa la gestione.

Riferimenti bibliografici

[1] Cormen, T. H., Stein, C., Rivest, R. L. & Leiserson, C. E. Introduction to Algorithms (3rd ed.), Chapter VI - Graph Algorithms