Peer-Review 1: UML

Valeria Amato, Francesco Barisani, Nicolò Caruso
Gruppo AM38

4 aprile 2022

Valutazione del diagramma UML delle classi del gruppo AM11.

1 Lati positivi

- Riteniamo che un sicuro punto di forza di questo UML sia la predisposizione di una solida architettura del Model che permette la creazione e gestione di partite multiple, con anche la possibilità di giocare partite a 4 giocatori.
- Tale approccio è reso particolarmente efficace dal design delle classi GameManager e ComplexLobby, che risultano ricche di metodi specifici per gestire tutte le opzioni e le preferenze di gioco (e.g.: metodi deckRequest e createGame).
- In generale, riteniamo ben costruita l'architettura GameManager – ComplexLobby – Game
- Riteniamo sia interessante e positivo l'uso delle interfacce, in particolare le interfacce Moovable e Board.
- In questo caso arricchiscono l'architettura del Model, introducendo un maggiore livello di astrazione concettuale, che si sposa bene con la scelta di creare più classi "specifiche" e distinte aventi, però, una semantica di base comune e, di conseguenza, un comportamento simile.
- È questo il caso delle classi SchoolBoard, IslandCard e CloudCard e del gruppo di classi che implementano Moovable.

- L'ulteriore gerarchia che specializza la superclasse Piece distinguendo gli specifici "pezzi mobili" del gioco rende questa idea architetturale, a nostro avviso, ancora più concreta e chiara.

- Pensiamo sia positiva la modellazione dei diversi mazzi di carte assistente (AssistantDeck), distinguibili in base al diverso "mage". Questa scelta è stata gestita in modo particolarmente consistente e coerente nelle varie classi, grazie alla possibilità di richiedere uno specifico mage sia a livello di player (metodo chooseMage()), sia a livello di GameManager con il metodo deckRequest.

2 Lati negativi

- La classe GameComponents rappresenta, a nostro avviso, semplicemente lo stato interno di alcuni elementi del gioco (ad es. la bag, le SchoolBoards e le CloudCards) e, di conseguenza, contiene solo getters e setters semplici, che immaginiamo siano stati lasciati impliciti, ma non conferiscono abbastanza "peso" e specificità a questa classe. Di conseguenza, pensiamo sarebbe potuto essere più efficace evitare la classe GameComponents ed inserire gli attributi (e quindi lo stato) di GameComponents all'interno della classe Game.

- Ci sembra manchi in molti casi l'uso di costruttori personalizzati, che dovrebbero aiutare a capire come settare ed instanziare classi "non-banali", come ad esempio SchoolBoard, IslandCard e CloudCard.

- Il pattern Factory per le Character Cards ci sembra leggermente "scarno" o comunque poco chiaro. Non risulta evidente come vengono gestite le carte speciali e se faranno riferimento ai metodi del Game o saranno gestite in modo diverso.

- Riteniamo che la modellazione di Coin e CoinReserve sia da rivedere. A nostro avviso il Coin non ha l'importanza semantica sufficiente per essere una classe autonoma che istanzia oggetti a sé stanti . Si è resa troppo complicata la gestione di un'entità che, non avendo un sufficiente contenuto informativo, ha valore solo come "quantità". Si potrebbe valutare di assegnare dei coin come interi a ciascun giocatore e alle relative carte speciali per identificarne il costo.

- Riteniamo discutibile la scelta di avere vari attributi dichiarati come public: ad esempio professors in GameComponents, deckManager in Player, characters in CharacterDeck. A nostro avviso sarebbero potuti essere friendly, in modo da massimizzare l'approccio di information hiding.

- L'attributo "position" nella classe "Professor" è duplicato e sembra riferirsi sia all'interfaccia Board che alla classe SchoolBoard. Tale soluzione risulta poco chiara a livello di associazione logica tra classi.

- La map nella classe Game contiene un solo parametro: non risulta chiaro l'intento quale sia il valore indicizzato da tale parametro.

- Nella classe Game ci sono 3 diversi metodi "generateBoard" che fondamentalmente, a nostro avviso, assolvono lo stesso compito. Suggeriremmo di usare un solo metodo sfruttando come parametro un intero o una enum.

- Osservando l'architettura e i metodi delle varie classi, ci risulta poco chiara la gestione del calcolo dell'influenza. In particolare vediamo due metodi islandDominance() e colorDominance() nella classe Game, ma non comprendiamo come potranno essere usati dal model per il calcolo dell'influenza.

- Abbiamo notato la presenza di alcuni "anelli di referenza" a livello di informazioni e dati contenuti nelle varie classi, che provocano ridondanza informativa. Un esempio si ha nel ciclo esistente tra le classi GameComponents, SchoolBoard, DiningRoom e Professor. L'associazione tra GameComponents e Professor risulta potenzialmente ridondante, poiché può essere ricavata direttamente dall'attributo Professors in GameComponents oppure può essere ricavata percorrendo la catena di referenza GameComponents -> SchoolBoard -> DiningRoom -> Professor

3 Confronto tra le architetture

- Pensiamo che un punto di forza dell'UML revisionato sia la scelta di distribuire la logica delle fasi di gioco in classi diverse, coerentemente con il loro ruolo specifico, invece che avere, come nel caso del nostro

UML, pressochè un'unica classe GameBoard (e relative sottoclassi) che gestisce tutta l'orchestrazione dei turni e delle mosse. Ad esempio, nell'UML revisionato vediamo che la turnistica è gestita nella classe ComplexLobby, mentre il calcolo dell'influenza e delle condizioni di vittoria/fine della partita sono gestite nella classe Game. Pensiamo che questo approccio leggermente più "distribuito" della logica di gioco possa essere preso in considerazione per rivedere il nostro UML ed eventualmente spostare parti di logica di gioco al di fuori della classe GameBoard, gestendo tali dinamiche nel Controller o in un'altra classe.

- Una significativa differenza tra il nostro model e il model dell'UML revisionato sta nel diverso approccio architetturale seguito nella creazione delle classi. Nel caso dell'UML revisionato abbiamo un approccio orientato più fortemente alla presenza di molte classi, spesso di piccole dimensioni o di ridotto contenuto informativo e logico, come Coin, CoinReserve, MotherNature, Student, Professor. Il nostro approccio è stato diverso ed è stato maggiormente orientato all'implementazione del model, secondo la quale abbiamo pensato di incapsulare il contenuto informativo di queste "piccole classi" all'interno dello stato di classi più grandi, tramite attributi e metodi getters/setters. Un esempio lampante di questo approccio differente è la gestione degli Students: nell'UML revisionato sono stati promossi a classe autonoma, mentre nel nostro caso sono stati gestiti come semplice "quantità" usando dei "contatori" interni alle classi. Sebbene rimaniamo convinti della nostra scelta, pensiamo di poter considerare un approccio intermedio alle due soluzioni, con l'eventuale creazione di alcune classi più piccole a ridotto contenuto semantico, ma tali da arricchire l'architettura complessiva.

- Un'altra differenza sta nella gestione diversa della fase di game setup e scelta delle opzioni di gioco. Nell'UML revisionato è chiaro l'intento di gestire in modo dedicato e completo questa fase di setup attraverso apposite classi e metodi (si veda quanto detto su ComplexLobby e GameManager nelle note positive), mentre nel caso della nostra architettura abbiamo inizialmente pensato di delegare questo compito al Controller. Riteniamo che sia totalmente sensato rendere le lobby parte del Model, in quanto contengono effettivamente parte dello stato della partita e delle preferenze dei giocatori. Poiché riteniamo che la scelta del gruppo revisionato sia solida ed interessante, potremmo pensare di

introdurre anche nel nostro model il concetto di lobby o, perlomeno, di spostare parte della logica di creazione e personalizzazione delle partite dal Controller all'interno del Model.

- L'architettura del model revisionato gode di un uso delle interfacce che porta verso una maggiore ramificazione/specializzazione delle classi e ci sembra efficace. La nostra architettura, al contrario, fa un uso più limitato dell'ereditarietà, anche in virtù dell'approccio "orientato all'implementazione" di cui si è detto sopra.

  Ciononostante, pensiamo che le interfacce "Moovable" e "Board" possano essere un buono spunto per noi al fine di poter spostare "più in alto" alcuni dei nostri metodi all'interno di interfacce, senza però andare a snaturare le nostre scelte architetturali di base.