Peer-Review 1: UML

Andrea Biasion Somaschini, Roberto Alessandro Bertolini,
Gabriele Corti, Omar Chaabani
Gruppo AM07

Marzo 2024

Valutazione del diagramma UML delle classi del gruppo AM16.

1 Lati positivi

In generale la struttura del Model che ci è stata presentata è molto ordinata e ben compartimentata tra le varie classi che si occupano delle rispettive funzionalità del gioco. Nello specifico, la suddivisione in package ha facilitato di molto la lettura dell'UML e la sua comprensione. I pattern e le strutture della programmazione orientata ad oggetti sono profondamente radicati nel design del Model, con un frequente uso di interfacce, classi astratte ed ereditarietà per isolare, organizzare e coordinare tutte le sfaccettature del gioco.
In particolare, abbiamo trovato molto ben organizzato e strutturato, nonostante sia una funzionalità avanzata, il sistema di chat, con `ChatManager` e le varie istanze di `Chat`.
Inoltre abbiamo apprezzato la classe `Deck<T extends Card>` e la relativa `DeckFactory`, che si occupa di costruire i mazzi, di ogni tipologia, che andranno poi utilizzati durante il gioco per distribuire le carte iniziali, obiettivo, risorse, ed oro. Un ulteriore motivo di apprezzamento è senza dubbio la gestione della disposizione delle carte sul campo di gioco, che avviene attraverso il meccanismo dei vicini e degli offsets per il calcolo dei pattern. Questa soluzione si distingue per la sua originalità.
Infine, abbiamo apprezzato l'utilizzo della classe statica `FilePaths` contenente i diversi percorsi da cui accedere agli assets e alle risorse di gioco, semplificando così l'organizzazione delle varie risorse durante la fase di implementazione.
Nel complesso riteniamo che il Model sia ben congegnato e ben organizzato, con un'ottima ripartizione dei compiti tra le varie classi.

2 Lati negativi

Dopo un'attenta analisi del diagramma UML del Model abbiamo identificato quelle che secondo noi sono delle criticità nel design, che potrebbero complicare la futura implementazione del gioco:

- `Game`: dal nostro punto di vista, l'interfaccia pubblica che `Game` espone è un po' poco chiara. I metodi `getActivePlayer()` e `getStartingPlayer()` ritornano degli interi invece di istanze della classe `Player`, quindi per ottenere l'active player si è costretti a fare `Player activePlayer = model.getPlayers[model.getActivePlayer()];`.
  Inoltre non ci è chiaro perché il Model dovrebbe esporre i metodi

`triggerFinalRound()` e `checkFinalRound()`, quando, a nostro avviso, dovrebbe essere compito del Model verificare i prerequisiti e poi far scattare il meccanismo dell'ultimo round, per poi notificare i client della cosa.

- `PointMultiplierPolicy`: a nostro avviso la *lambda* `policy` come attributo dell'*enum class* `PointMultiplierPolicy` è solo una maniera più complicata di rappresentare una classe astratta con ereditarietà e polimorfismo. Svolge lo stesso ruolo di una *sealed abstract class* `PointMultiplierPolicy` che espone il metodo astratto `evaluate(playArea: PlayArea): int` e con definite tutte le sottoclassi concrete che la implementano, ma risulta più complicata da implementare e più difficile da comprendere. Inoltre sarebbe stato possibile applicare i principi di ereditarietà e polimorfismo anche sulle facce delle carte per far sì che il metodo di valutazione dei punti venga chiamato solo quando necessario, nelle carte oro, invece che per tutte le generiche carte di gioco.

- `Player`: i metodi `addGamePoints()` e `addObjectivePoints()` sono indicati come pubblici, ma il `Player` dovrebbe poter calcolare i suoi stessi punti internamente. Di conseguenza, i metodi dovrebbeo essere indicati come privati. Inoltre il calcolo dei punti obiettivo avviene solo al termine della partita, quindi l'attributo `currObjectivePoints` rimarrebbe nullo per tutto il lifecycle della classe `Player`.

- `CardRegistry`: il card registry ci appare un sistema molto comodo per memorizzare le informazioni sulle carte lette da un file locale, ma non capiamo perché dovrebbe essere un *lazy singleton* e in quali casi la chiamata a `isInitialized()` potrebbe ritornare `false`, se ogni operazione di parsing del JSON viene fatta nel costruttore della classe, che viene istanziata all'avvio dell'applicazione. Inoltre il `CardRegistry` non dovrebbe aver bisogno di memorizzare come attributi le liste tramite cui costruisce le rispettive mappe, se le mappe sono immutabili e non c'è modo di reinizializzarle tramite un metodo pubblico della classe.

- `ObjectType, ResourceType, CornerType`: abbiamo trovato abbastanza difficile comprendere perché ci debbano essere tre diverse *enum classes* per rappresentare le informazioni dei tipi di oggetti, risorse e corner; dove `CornerType` è inoltre l'unione delle istanze delle altre due classi.

Dal punto di vista del gioco non ci appare strettamente necessaria questa suddivisione: c'è la necessità di identificare se il generico oggetto nel corner è una risorsa di gioco, ma non c'è altra differenza tra un `INSECT` e un `QUILL`.
Questo, dal nostro punto di vista, crea delle dipendenze cicliche tra le varie classi, in quanto ogni corner deve avere una reference all'object e alla resource associati.
Inoltre non ci è chiaro dal punto di vista dell'implementazione il perché dei metodi `bindToCorners()` e `bindToResourceAndObjects()` e quando dovrebbero essere chiamati. Ogni dipendenza tra queste tre classi dovrebbe essere definita a compile time aggiungendo il necessario attributo alle istanze dichiarate delle tre classi, non risolta a runtime, rallentando l'esecuzione e aggiungendo complessità all'implementazione.

- `PlayArea`: la criticità che abbiamo identificato qui si riconduce al punto precedente. Visto che c'è una distinzione tra oggetti e risorse, non capiamo perché tenere una sola mappa `resourceAndObjectCounts` che memorizza i contatori di entrambi i tipi, con la necessità di costruire a runtime una nuova mappa ogni volta che viene chiamato uno dei due getter `getResourceCounts()` e `getObjectCounts()`.

- `RNG e identificatori`: abbiamo notato un po' di inconsistenza tra gli identificatori univoci delle varie classi: in `Player` è un intero, in `Game` è una stringa, in `Card` è la stringa `name`. Inoltre per generare le stringhe alfanumeriche casuali degli id in Java è presente il metodo `UUID.randomUUID()`, che genera un identificatore univoco universale secondo il formato standard; mentre per mischiare i mazzi è presente il metodo statico `Collections.shuffle(List<T> list)`.

3 Confronto tra le architetture

Nel confronto delle architetture, abbiamo individuato diversi punti di forza nell'architettura del gruppo `AM16` rispetto alla nostra, nonché possibili modifiche che potremmo apportare alla nostra architettura per migliorarla.
In primo luogo, abbiamo notato che nel Model del gruppo `AM16` è presente un identificatore univoco (`Id`) associato alla classe `Game`. Questo attributo

potrebbe risultare particolarmente utile durante l'implementazione di funzionalità avanzate, come ad esempio la gestione delle partite multiple, in modo da distinguere i diversi game in modo univoco, problema che non ci eravamo posti durante la fase di progettazione.
Inoltre, ha suscitato particolare interesse la presenza dei metodi `getNumPlayers()` e `getCurrentPlayerCount()`, utili per gestire eventuali disconnessioni durante una partita. Questo è un aspetto che non avevamo considerato nella nostra architettura e che potremmo integrare per migliorare la gestione delle sessioni di gioco.
Un'altra caratteristica che abbiamo ritenuto interessante è l'implementazione del metodo `getStartingPlayer()`, che consente di determinare chi sia il primo giocatore. Inizialmente, avevamo considerato questa informazione come rilevante solo a livello di View per rappresentare la pedina nera, ignorandola lato Model. Tuttavia, dopo una riflessione più approfondita, abbiamo concluso che è doveroso includerlo anche nel Model per garantire la corretta rappresentazione del giocatore iniziale.
Altro aspetto fondamentale che abbiamo notato è la presenza del metodo `peekTop()` sul deck delle carte, che consente di visualizzare il retro delle carte senza rivelarne il fronte. Questa funzionalità è cruciale a livello di gioco perchè fornisce al giocatore la possibilità di conoscere a priori il colore della carta. Tale feature non era stata presa in considerazione, ignorando il fatto che avrebbe potuto modificare le scelte del giocatore finale.
Durante l'analisi dell'UML, analizzando la loro implementazione della classe `ObjectiveCard` ci siamo resi conto di come, nella nostra implementazione, la classe contenga sia un attributo `pattern` che un attributo `requirements`, ma ciascuna carta può avere solo uno dei due attributi. Una soluzione potrebbe essere quella di trasformare `ObjectiveCard` in una classe astratta e creare due sottoclassi concrete, dove una implementa l'attributo `pattern` e l'altra implementa l'attributo `requirements`.
Infine, abbiamo notato che l'implementazione della chat nel Model del gruppo `AM16` è ben strutturata e organizzata. Questo è un aspetto che sicuramente terremo in considerazione durante lo sviluppo di tale funzionalità avanzata.
In generale, l'architettura del gruppo revisionato risulta più orientata agli oggetti e ha permesso di illuminarci sullo sviluppo di alcuni dettagli che ci erano sfuggiti. Questi sono spunti preziosi che potremmo integrare nella nostra architettura per renderla più solida e robusta.