Peer-Review 1: UML

Simone Boscain, Andrea Croci, Giovanni De Marco, Nicolò di Carlo

Gruppo AM40

Valutazione del diagramma UML delle classi del gruppo AM49.

Lati positivi

Un aspetto positivo è la scelta dell'utilizzo della logica pivot nella classe Pattern che semplifica la visualizzazione dei pattern di disposizione delle carte nella fase di valutazione degli obiettivi. Molto apprezzata è la presenza di una classe DeckLoader che si occupa del parsing con file JSON. Inoltre, molto ben strutturata e organizzata è la gerarchia delle classi relative alle carte con una classe astratta PlaceableCards che racchiude dentro di sé la logica e gli attributi di posizionamento, mentre le classi figlie estendono con gli attributi "stampati" sulle carte da gioco. Infine, l'intento di tenere traccia dello stato del gioco con un attributo apposito è un'ottima idea, che si rivela utile nel caso di check durante le varie fasi di gioco e, eventualmente, nello sviluppo della funzionalità aggiuntiva di "Persistenza".

Lati negativi

Troppi metodi introdotti nell'UML: ci sono molti metodi getter e di check superflui per la visualizzazione ad alto livello. Di conseguenza, nell'insieme si fa fatica a vedere a primo impatto i metodi core che si occupano della logica di gioco; inoltre, in BoardTile vi è una lista di metodi pubblici che, considerando unicamente signature e nome, possono dare a chi osserva un'impressione di ambiguità o di ridondanza. Un problema con i nomi viene riscontrato anche nel package placeables in PlaceableCard e StarterCard. Capiamo la volontà di abbreviare topRight con tr, però questa abbreviazione diventa ambigua in StarterCard dove trB introduce una B di cui non riusciamo a capire il significato. Interpretando, potremmo pensare che trB prenda il posto dell'attributo tr, a questo punto, però, dato che StarterCard è sottoclasse di PlaceableCard essa erediterebbe comunque tr, ma esso rimarrebbe un attributo vuoto?
Nel testo in ausilio all'UML fornito dal gruppo in questione, vengono illustrati perfettamente i meccanismi di parsing JSON con DeckLoader e il check dell'endgame con controllo del pattern illustrato nelle carte obiettivo. Ma al tempo stesso viene tralasciato come gestire la logica di gioco, nella fattispecie sull'interazione tra Player, PlayerBoard e BoardTile e sulla meccanica di piazzamento delle carte non viene detto nulla. Senza una guida, l'analisi risulta complessa e macchinosa a causa dei motivi appena elencati. Ad esempio, con l'overloading del metodo placeTile(…) nella classe PlayerBoard: attraverso una prima analisi risulta ambigua l'effettiva applicazione di ciascuno dei metodi da parte di un'osservatore esterno al gruppo.
Inoltre, è stato difficile associare in modo immediato le enumerazioni ad alcuni attributi di classe vista la scelta della disposizione a package nel diagramma UML. Questa scelta facilita l'organizzazione dei packages in Java, però, a livello di class diagram, sfocia in un'assenza di frecce di riferimento da classi a enumerazioni lasciando, a primo impatto, alcuni interrogativi sul fatto che un'attributo si riferisca ad una classe o ad una enumerazione. Un esempio è gameState dentro Game.
Si consiglia, di asciugare l'UML per la visualizzazione ad alto livello, lasciando solo i metodi essenziali, ovvero da considerare "core" e rivedere la nomenclatura di attributi e metodi.
Si nota, inoltre, che Game possiede molta responsabilità nel gestire la logica di gioco, tra cui la gestione del round, in sé, e il controllo dell'evoluzione degli stati del sistema, ovvero il susseguirsi

dei turni dei giocatori. Essendo una scelta di design, non ci sono scelte giuste o sbagliate, però nel caso sia vostra intenzione snellire Game e creare più modularità si consiglia di creare una classe GameManager direttamente nel controller, oppure da utilizzare per interfacciarsi con il controller a cui far gestire il flusso degli stati del gioco con l'eventuale check dell'endgame ecc. O ancora, creare una Board comune dove inserire i vari deck da cui pescare.
Se il gruppo in analisi scegliesse di procedere creando una classe GameManager nel controller si consiglia di utilizzare il design pattern State al posto di una enumerazione per tenere traccia degli stati del gioco, in modo da garantire più modularità ed elasticità al posto dell'attributo gameState in Game.
Un ulteriore appunto è la scelta della struttura dati "board": allocare quella che ad occhio sembra una matrice statica, di grandezza compresa tra il 50x50 e il 100x100 nel caso pessimo può essere decisamente sconveniente in termini di memoria.

Confronto tra le architetture

Il gruppo in questione ha inserito nell'UML una classe in ausilio per il parsing da JSON, totalmente tralasciata nel nostro UML, nonostante la scelta condivisa di appoggiarsi su file JSON. Di conseguenza, DeckLoader rappresenta un buon punto di partenza per l'inizializzazione dei mazzi da cui prenderemo spunto. Inoltre, l'UML analizzato presenta caratteristiche di logica delle carte molto dettagliate riguardo soprattutto alla gestione delle posizioni durante il piazzamento e dei pattern checker delle carte piazzate nella fase di endgame al momento del conteggio dei punti provenienti dalle carte obiettivo. Sebbene la nostra scelta di design della boardTile risulta diversa, visto l'utilizzo di un meccanismo ausiliare basato su coordinate cartesiane, è possibile comunque prendere spunto da questo gruppo per l'implementazione del meccanismo dei controlli di piazzamento di una carta e per lo sviluppo dei nostri checker per le carte obiettivo. Inoltre analizzando l'enum gameState siamo stati ispirati con l'idea di utilizzare il design pattern State per gestire gli stati di gioco tramite un attributo nel nostro GameManager che fa riferimento allo State, al fine di garantire più modularità tipico dell'object orientation e aprire un varco per un eventuale sviluppo della funzionalità aggiuntiva "Persistenza" come abbiamo evidenziato nei lati positivi.