Peer-Review 1: UML

Massimo Albino Sandretti, Cristiano Sartori, Mario Valente, Alberto Francesco Visconti

Gruppo GC52

Valutazione del diagramma UML delle classi del gruppo GC42.

Lati positivi

Apprezziamo l'inserimento nel diagramma, già a questo livello di dettaglio, della logica Observer-Observable, oltre all'introduzione di alcuni metodi QoL (come, ad esempio, kickPlayer) e l'individuazione di quelli che sono i metodi utilizzati dal controller.

Come esporremo nella sezione finale, troviamo particolarmente interessante l'idea di utilizzare la classe astratta Objective per rappresentare un'ampia varietà di concetti.

Lati negativi

Inizieremo parlando di quelli che secondo noi sono decisioni "implementative" (ad alto livello perlomeno) che potrebbero risultare problematiche, se direttamente errate.

- L'utilizzo di un ArrayList per il PlayField è una scelta che valutiamo inefficiente dal punto di vista temporale e di facilità di implementazione di algoritmi (ad esempio quelli relativi ai pattern). Il guadagno spaziale è trascurabile data la ridotta portata dell'applicativo da sviluppare. Una matrice o una Map che associa a una carta la sua posizione potrebbero essere degli esempi di strutture dati più auspicabili. Per quanto riguarda l'ordine, basta aggiungere un campo a Card con il turno in cui è stata piazzata.
- In Deck, le carte scoperte dipendono dal mazzo iniziale, questo crea problemi in quanto è impossibile aggiungere carte da un altro mazzo quando quello iniziale finisce, dato che sul piano devono sempre esserci 4 carte scoperte (fare riferimento alle discussioni relative su Slack; è anche deducibile dal funzionamento della regola secondo cui il turno finale viene innescato al terminare dei mazzi: se è il primo giocatore a finire il primo mazzo e a lasciare solo 2 carte visibili, allora il quarto giocatore rimarrebbe senza carte da pescare e giocherebbe il turno finale con 2 carte in mano).
- In PlayField, c'è una mappa che rappresenta l'inventario del giocatore e, secondo il commento, viene utilizzato per calcolare i punti degli obiettivi di conteggio; però non è chiaro come questo possa avvenire (dato che il metodo che prende in ingresso un ArrayList è protetto e quello chiamato dal controllore non ha parametri). Probabilmente un problema di rappresentazione nel diagramma di un'implementazione in Java funzionante.

Per quanto riguarda gli Objective, notiamo prevalentemente dei problemi di flessibilità. Per l'applicazione in esame, nella versione attuale del gioco, non ci sono problemi; ma se in futuro il gioco dovesse essere aggiornato con delle espansioni, molto codice andrebbe riscritto (di conseguenza, ritenetevi pure liberi di non considerare i seguenti punti se non intendete perseguire l'obiettivo della flessibilità a situazioni oltre la specifica). In particolare:

- Quasi tutti gli obiettivi sono definiti come classi diverse. Ognuna deve implementare una logica "complessa" da zero. Questo porta inevitabilmente ad avere molto codice ripetuto. Le due classi KingdomCountObjective e ItemCountObjective sarebbero particolarmente facili da unire usando un'interfaccia comune agli enum (dettagli in seguito). Espandere il concetto per unire anche SetItemCountObjective sarebbe il passo successivo. Per come sono implementati ora, queste 3 classi sono in grado di rappresentare i soli 3 stili di obiettivi di conteggio presenti in gioco e non c'è flessibilità (es. si decide di introdurre un ulteriore obiettivo che prevede il conteggio di 2 Piante e 2 Piume).
- Gli obiettivi PlacementObjective sono molto vincolati: oltre a non ammettere pattern diversi da quelli standard, non è agevole estendere i pattern individuabili.

Di seguito riportiamo quelle che secondo noi sono progettazioni migliorabili (escluse chiaramente quelle già trattate nei punti precedenti):

- L'architettura non isola le carte giocabili dalle carte obiettivo: in linea teorica si potrebbero piazzare delle carte obiettivo sul PlayField. Questo perché le ObjectiveCard vengono trattate come delle Card; cosa che, come avete giustamente sottolineato nel commento, porta alcuni campi e metodi ad essere inutili (nella fattispecie, frontSide, il relativo getter e le coordinate (x,y)).
- Strettamente legato al punto precedente c'è la gestione delle carte da gioco in 3 classi diverse. Non abbiamo obiezioni su questa scelta implementativa, ciò che ci sembra problematico è la (possibile) necessità di utilizzo di Casting e/o instanceof. Ad esempio, avendo un oggetto Card non è immediato sapere con che tipo di carta si ha a che fare (questo chiaramente a meno che i metodi non siano opportunamente interfacciati nella classe Card, ma questo ci riporta al problema del punto precedente: troppi metodi inutili per il tipo ObjectiveCard). Questi ultimi due punti portano al suggerimento della separazione di Card in due classi astratte, una che rappresenti puramente il concetto di elemento grafico (frontImage, backImage, flip...) e una invece che rappresenti le carte effettivamente giocabili e che abbia dei metodi di interfaccia opportunamente ridefiniti che renda possibile evitare l'utilizzo di casting e/o instanceof.
- La struttura delle classi Corner (e sottoclassi) è complicata. Si usano essenzialmente due classi "wrapper" per incapsulare un valore enumerato. Inoltre, utilizzare una classe vuota per rappresentare una costante non ci sembra una soluzione ottimale. Un modo potenzialmente migliore per gestire questo oggetto sarebbe creare un'interfaccia Item (nome di esempio) che venga implementata dai due enum (KingdomResource e Resource) e poi inserire una variabile di tipo Item all'interno di Corner. Far diventare Corner concreto e eliminare tutte le sottoclassi. A questo punto, EmptyCorner può essere rappresentato dal valore nullo di questo attributo Item.

Confronto tra le architetture

Trattare i vincoli delle carte dorate e le strategie di valutazione come Objective è interessante (nel nostro diagramma sono individuate come 3 elementi ben distinti). Nel nostro diagramma, sostituire l'attuale definizione del vincolo come un oggetto Objective (nel nostro caso Goal) opportunamente configurato, permetterebbe vincoli per le carte più espandibili e flessibili.

Apprezziamo l'esistenza di una sola classe Deck (fatta eccezione per PlayingDeck, ma l'esistenza di questa è stata oggetto di discussione), responsabile della gestione di tutti i possibili mazzi del gioco. Noi abbiamo optato per una soluzione leggermente più complicata, ma che alla fine svolge il compito in maniera molto simile. Abbiamo voluto sfruttare una natura più "type-sensitive" e una logica di caricamento separata (e isolata) al fine di rendere facilmente identificabili errori logici e minimizzare l'accesso al disco.

Nell'ottica di semplificare il diagramma (e il corrispondente codice), è apprezzabile l'esistenza di due istanze di CardSide, a identificare tutti gli 8 Corner di una carta, all'interno della classe Card. Nel nostro diagramma abbiamo deciso di sfruttare il fatto che carte dorate e risorsa hanno sempre angoli vuoti e visibili sulla faccia posteriore per evitare di definire gli oggetti che descrivono i Corner in questa faccia, definendolo solo all'interno della classe che identifica le classi iniziali. Questa "ottimizzazione" permette di risparmiare un paio di migliaia di Byte in memoria per partita.

ResourceCard e GoldCard nella nostra architettura sono rappresentate dalla stessa classe. Siamo arrivati a questa scelta progettuale interpretando le carte risorsa come delle carte dorate con criteri di valutazione a punteggio costante (eventualmente nullo) e vincoli vuoti. A nostro avviso, nessuna delle due strutture vince veramente l'una sull'altra (motivo per cui ne stiamo parlando in questa sezione); supponiamo che entrambe offrano spunti di riflessioni e abbiano pro e contro diversi.

Volendo fare una breve riflessione sul metodo checkEndGame, notiamo che le sue chiamate non sono ordinate dal controller, ma da dei listener che, appunto, ascoltano i valori che possono causare l'innesco del turno finale. Questa soluzione è indubbiamente interessante. Noi siamo arrivati alla conclusione che, a livello di pura logica di gioco, è sufficiente verificare l'avvento del turno finale all'inizio del turno successivo (o alla fine del corrente). Tuttavia, la soluzione adottata nel diagramma in esame permette di conoscere "in tempo reale" l'innesco della condizione che porterà alla fine del gioco, rendendo possibile una notifica ai giocatori (tramite qualche aggiornamento della vista).