Peer-Review 1: UML

Riccardo Piantoni, Matteo Rossi, Federico Pinto, Jacopo Sacramone
Gruppo GC12

3 aprile 2024

Valutazione del diagramma UML delle classi del gruppo GC02

1 Lati positivi

La scelta di rappresentare il campo di gioco attraverso una mappa di <Coords, Card> è funzionale e semplifica sia la gestione delle carte sia la navigazione della struttura per validare le operazioni di posizionamento e il riconoscimento dei pattern posizionali presenti all'interno degli obiettivi.

La suddivisione gerarchica delle classi che compongono il Model permette la realizzazione di un progetto molto modulare, teoricamente sufficiente per ottenere il disaccoppiamento tra classi e rispetta un'importante *good practice* nel contesto OOP (Object-Oriented Programming): il Principio di singola responsabilità.

Conformemente alla scelta fatta per il campo di gioco, è ottima la scelta di rappresentare gli obiettivi che richiedono la validazione di un pattern delle QuestCard come un offset di coordinate. La scelta adottata è sicuramente migliorabile, in quanto queste informazioni sono salvate in attributi singoli, invece di essere raggruppate all'interno di una Collection, sulla quale eseguire anche operazioni di scorrimento.

2 Lati negativi

Alcuni nomi dei metodi risultano essere ambigui e non permettono una facile comprensione di quale sia la funzionalità da loro fornita: il nome di un metodo dovrebbe rivelare qual è la motivazione per cui è stato concepito, soprattutto in progetti condivisi e costituiti da molteplici classi.

Alcune scelte di design sembrano contravvenere ai principi della programmazione ad oggetti, limitando la scalabilità del progetto. In particolare citiamo la gestione dei Player all'interno della classe Match: essi dovrebbero essere raccolti in un'unica Collection (per esempio una lista) e non distribuiti in diversi attributi, che peraltro dovranno essere nulli nel caso di partite con un numero di giocatori inferiore rispetto al massimo consentito. Ciò rimuoverebbe anche la necessità di dover creare un costruttore per ogni configurazione possibile di giocatori che intendono giocare, avendo invece un unico metodo generico. Inoltre, citiamo la classe Card, che possiede dei metodi setter e getter per ogni singolo Corner esistente: essi dovrebbero invece essere raccolti in un solo metodo in grado di discriminare il Corner desiderato attraverso degli opportuni parametri (alcuni suggerimenti implementativi possono essere: getCornerAt(int position, bool side) oppure getCornerAt(int position)).

La delega di gestione del giocatore di turno dovrebbe essere di competenza della partita e non dei singoli Player, evitando così in modo più naturale la possibilità che più giocatori risultino essere attivi. Tale aspetto può essere ottenuto aggiungendo un attributo CurrentPlayer alla classe Match e togliendo isTurnPlayer da Player.

La duplicazione delle classi *Deck, le quali possiedono il medesimo comportamento, potrebbe essere evitata mediante la dichiarazione di Deck come aggregazione di Card, oppure, in maniera ancora più funzionale, definendo Deck come classe generica che può essere istanziata solamente con le classi che ereditano da Card (Deck<T extends Card>).

Il Field possiede tre diversi metodi per il piazzamento di una carta, i quali presentano teoricamente il medesimo comportamento. E' possibile unificarli in un unico metodo e declinarne poi l'implementazione determinando il corretto sottotipo di carta piazzata, per esempio attraverso l'ausilio di *instanceof*.

La QuestCard è utilizzabile solamente in questo contesto del gioco è non è scalabile in ottica di future espansioni o modifiche della struttura dei pattern da validare. La classe utilizza infatti degli attributi distinti per ogni elemento che compone il pattern, frazionando l'informazione della posizione della singola carta in una coppia di coordinate senza raccoglierle in una Collection, come viene invece fatto per l'attributo di colore, rappresentato all'interno di un array. Suggeriamo quindi di rendere coerente la rappresentazione di una singola carta del pattern unendo le coordinate e i colori in un unico oggetto. Una possibile scelta implementativa, che non implica la creazione di ulteriori classi, è generalizzare la classe Coords per permetterle di contenere due oggetti di tipo generico: <<Valore X, Valore Y>, Colore>, oppure raggiungere tale comportamento attraverso altre scelte comunque funzionali. Successivamente, gli elementi che costituiscono un pattern possono essere raccolti in un'altra collezione omogenea, come per esempio un array o una lista, senza imporre limitazioni alla dimensione della struttura dati. In questo modo si ottiene un'organizzazione più strutturata delle informazioni e una classe naturalmente predisposta alla scalabilità.
Inoltre, la QuestCard dovrebbe disaccoppiare le carte che richiedono la validazione di un pattern da quelle che valutano il possedimento di determinate risorse, per evitare l'istanziazione di oggetti che presentano attributi nulli, dato che le due condizioni non possono essere vere contemporaneamente.

Vi è una lieve incoerenza nella gestione degli incrementi di punteggio, che assume un comportamento diverso in base alla carta che lo richiede: la classe GoldCard possiede un metodo played(p: Player, c: Coords) che non restituisce alcun valore di ritorno e supponiamo si occupi internamente dell'incremento di punteggio, a differenza del metodo check(p: Player) in QuestCard che restituisce invece un intero come valore di ritorno e probabilmente demanda ad addPoints(points: int) disponibile in player l'incremento. Suggeriamo di uniformare il flusso di questi metodi, dato che logicamente svolgono la medesima operazione.

Per quanto riguarda l'enumerazione GoldObjective, è possibile compiere scelte più eleganti che non offuscano la composizione delle varie condizioni, che peraltro si sovrappongono con alcune delle condizioni delle QuestCard, dato che per alcune carte risultano essere identiche. Compiendo una scelta di questo tipo le condizioni possono essere uniformate in una struttura unica, indipendente dal tipo di carta che la contiene, che sicuramente si integra bene con il suggerimento del punto precedente e ne semplifica la sua implementazione.

3 Confronto tra le architetture

Le architetture dei due gruppi presentano delle differenze sostanziali, poiché il gruppo revisionato non ha indicato di voler implementare alcuna funzionalità aggiuntiva e questo porta a scelte progettuali diverse, per esempio nella gestione delle partite multiple. Infatti, all'interno del modello dell'altro gruppo, non compaiono classi relative alla gestione delle Lobby e neanche la memorizzazione del lato in cui la carta è stata giocata, la quale è inglobata all'interno della Card.

Noi, implementando tale funzionalità, abbiamo ritenuto non ideale istanziare una nuova carta per tenere traccia dell'unico attributo variabile di Card, cioè il lato in cui essa è giocata. La nostra soluzione della problematica ha spostato la gestione della proprietà all'interno della classe Field, personale all'InGamePlayer, che oltre a memorizzare la carta giocata in una determinata posizione può anche mantenere anche il lato scelto dal giocatore. La scelta è coerente con lo svolgimento del gioco in una partita fisica: la carta non possiede un lato finché non è posizionata, quindi un campo booleano non sarebbe stato in grado di definire pienamente la sua rappresentazione quando essa si trova in un mazzo o nella mano del giocatore.

Per quanto riguarda invece la gestione della singola partita, il nostro modello possiede un numero minore di classi:

- La rappresentazione dei Corners, grazie anche agli accorgimenti precedentemente citati, non è realizzata tramite una classe apposita, ma è codificata direttamente all'interno delle sottoclassi della gerarchia di Card che ne possiedono ed è descritta da una struttura annidata, realizzata attraverso una mappa di due mappe, che ne discriminano fronte e retro.

- La mano del giocatore, nella nostro schema, è contenuta all'interno della classe InGamePlayer, in quanto richiede la semplice memorizzazione di carte posizionabili, mentre la carta Obiettivo personale è un attributo del giocatore, in quanto immutabile nel corso di una partita.

- Anche l'inventario di Resources di un giocatore è contenuto all'interno dell'InGamePlayer, in quanto si tratta di un singolo attributo, per cui non abbiamo ritenuto necessario creare un'intera classe che ne declini la sua gestione. Inoltre, poichè la tipologia di risorse è nota dalla struttura del gioco, si è scelto di usare nuovamente una mappa per rappresentare la quantità di risorse correntemente posseduta dal giocatore di qualsiasi categoria.

La classe Coords è logicamente equivalente alla classe GenericPair all'interno del nostro modello: tuttavia, viste le funzioni da essa compiute, noi abbiamo deciso di renderla immutabile, rimuovendone i metodi setter. Non vi è infatti alcun caso d'uso che prevede la modifica delle coordinate di piazzamento all'interno del gioco: quindi abbiamo scelto di optare per un design più sicuro.