Peer-Review 1: UML

Luca Pozzato, Niccolò Puccia, Stefano Romano, Edoardo Saso
Gruppo GC17
Valutazione del diagramma UML delle classi del gruppo GC07

Introduzione

La presente peer review si propone di evidenziare punti di forza, criticità e spunti di miglioramento emersi in seguito all'analisi del diagramma UML fornitoci. Ci teniamo a sottolineare come la visione del solo diagramma delle classi, pur costituendo un documento di rilevante importanza per la descrizione del progetto, omette per sua stessa natura i dettagli implementativi tipici di fasi successive dello sviluppo. Pertanto le osservazioni di seguito elencate mirano a condurre verso una progettazione efficiente e coerente ma potrebbero risentire di assunzioni fatte dal gruppo revisione in casi in cui la sola analisi della sintassi UML risultasse poco chiara o, a tratti, equivoca.

Nel caso in cui doveste riscontrare incongruenze tra la vostra interpretazione del progetto e alcuni degli spunti sotto riportati in seguito al possibile verificarsi di fraintendimenti, in quanto gruppo 17 saremmo felici di incontrarvi per poterne discutere e approfondire durante uno dei prossimi laboratori.

Punti Positivi

Prima di analizzare le criticità individuate, è importante riconoscere i punti positivi presenti nel vostro diagramma UML. Analizzando le vostre scelte progettuali siamo rimasti piacevolmente colpiti da:

- **Utilizzo dello Strategy Pattern**: La scelta di utilizzare il pattern strategy per verificare le condizioni di piazzamento dalle carte e il numero di punti derivanti dallo stesso dimostra una buona comprensione dei principi di design pattern e della modularità del codice. Permette inoltre di gestire in modo agile la scelta dell'algoritmo da utilizzare rendendo scalabile l'architettura.
- **Utilizzo dei costruttori**: Abbiamo avuto modo di apprezzare come le classi del vostro progetto creino istanze passando al costruttore un gran numero di argomenti. L'utilizzo dei costruttori anziché un eccessivo numero di setter è un approccio positivo, poiché promuove l'incapsulamento e la coerenza nell'inizializzazione degli oggetti.
- **Gerarchizzazione delle classi:** Ci piace sottolineare come un punto di forza del vostro progetto consista nell'uso diffuso e corretto dell'ereditarietà tra classi. Siete riusciti ad individuare correttamente tutti e soli i metodi che una sottoclasse può condividere con la sovraclasse senza perciò creare conflitti logici.
- **Intuitività del modello:** Nel complesso il diagramma delle classi del model risulta essere organico e ben strutturato pur mantenendo una notevole chiarezza dovuta anche alla scelta di ridurre al minimo la creazione di classi accessorie.

- # Criticità e Possibili Miglioramenti
  - ## GameField
    - **Costruttore con existingGameField**: Il costruttore che accetta existingGameField potrebbe essere riconsiderato per migliorare la coesione della classe. Abbiamo assunto che le ragioni che rendano conveniente passare al costruttore di un oggetto un altro oggetto della stessa classe del costruito possano riguardare la volontà di creare un clone dell'oggetto passato. Se così fosse occorre tuttavia verificare tramite opportuni controlli che, ad esempio, sia impossibile stanziare più gameFields che giocatori.
    - **Eccezioni:** Riteniamo possa esservi utile l'inserimento di opportune eccezioni che testimonino l'accadere di eventi imprevisti che non sono di competenza del controller. Tali eccezioni dovranno essere gestite in maniera circolare come il pattern MVC impone passando prima dalla view e successivamente dal controller.
  - ## Deck
    - **Utilizzo di existingDeck e sottoclassi**: L'utilizzo di existingDeck e la presenza di sottoclassi di Deck potrebbero generare confusione e potrebbero essere rivisti per migliorare la chiarezza e la coerenza del diagramma.
    - **Passaggio di cardType**: La necessità di passare cardType al costruttore di deck potrebbe essere riesaminata per migliorare la coesione della classe. Infatti nel momento in cui il costruttore riceve uno Stack<T> può automaticamente desumere cardType essendo esso T stesso.
    - **PlayingDeck e faceUpCard**: La presenza di faceUpCard potrebbe richiedere una maggior chiarezza o motivazione nel contesto del gioco. Se, come traspare dal PDF allegato all'UML, le carte obiettivo sono istanza di playingDeck, stanziare una classe il cui unico scopo è quello di provvedere metodi setter e getter per tali carte risulta essere sovrabbondante. Anche il nostro gruppo si è scontrato contro un problema simile e abbiamo risolto tenendo traccia delle due carte obiettivo comune come attributo di una classe più grande (per voi tale classe potrebbe essere GameField). Esse vengono settate alla creazione della partita e quindi dei vari GameFields e non vengono più toccate.
    - **Gestione delle carte iniziali**: Un discorso analogo a quello condotto per le carte obiettivo comune può essere fatto per le carte iniziali. Il fatto di considerare un mazzo delle carte iniziali e dunque istanza di Deck significa esporsi al pericolo che a-run-time una carta iniziale possa diventare di un sottotipo dotandola di operazioni non consentite dalla logica del gioco. Ancora una volta, essendo l'assegnazione delle carte iniziali un unicum nel corso del gioco, è possibile schermarsi dal sorgere di problematiche simili ad esempio assegnando un attributo initialCard alla creazione del nuovo Player.
    - **Necessità delle due sottoclassi per i mazzi giocabili**: La presenza di due sottoclassi potrebbe generare overhead di codice e dovrebbe essere valutata per semplificare la struttura complessiva.
    - **Ereditarietà**: L'ereditarietà delle sottoclassi potrebbe essere valutata per garantire che i metodi e gli attributi ereditati siano pertinenti e necessari. Ad esempio un oggetto di tipo

ResourceDeck eredita anche la lista delle carte obiettivo scoperte e i metodi per accedervi e modificarle. Ciò potrebbe sfociare nel risultato opposto a quello sperato utilizzando l'ereditarietà ovvero in una non corretta separazione dei compiti tra la classi.

- **Condition**
  - **Metodo numTimesMet**: Se non esiste una classe che funge da controllore del flusso di gioco implementandone la logica soggiacente, il metodo numTimesMet potrebbe non essere necessario. Infatti osservandolo attentamente abbiamo notato come il vostro modello risponda al paradigma "Thin model". In quest'ottica potrebbe essere presa in considerazione la possibilità di spostare nel controller anche la parte relativa alla verifica sulla possibilità di piazzare una carta e il calcolo dei punti derivanti dal piazzamento rendendo il model definitivamente più simile a una base di dati con accesso in lettura e in scrittura che a un'unità logica.
- **ScoreTrackingBoard**
  - **Metodo IncrementScore**: La presenza del metodo IncrementScore potrebbe essere riesaminata considerando la presenza già di setScore, per evitare ridondanze e migliorare la coerenza della classe.

Confronto tra architetture

L'analisi del documento propostoci ha innescato un momento di confronto costruttivo all'interno del nostro gruppo da cui è emersa la comune volontà di integrare nel nostro progetto i punti positivi identificati sopra al fine di arricchirlo e migliorarne alcuni aspetti.

Il nostro modello è stato costruito secondo una filosofia "Fat Client" dunque presenta delle differenze strutturali notevoli essendo dotato di una grossa componente che gestisce il corretto flusso del gioco. Nonostante questo abbiamo apprezzato alcuni degli strumenti da voi adoperati per risolvere delle problematiche comuni e intendiamo farne tesoro per le nostre future implementazioni.

In particolare, l'adozione del Strategy Pattern per il calcolo dei punti dalle carte e l'utilizzo dei costruttori anziché dei setter rappresentano due approcci che intendiamo integrare nel nostro UML. Queste pratiche non solo contribuiranno a rendere il nostro modello più flessibile ed efficiente, ma ci permetteranno anche di garantire una maggiore coerenza e chiarezza nella struttura e nel funzionamento del sistema. Integrando tali implementazioni nel nostro UML, miriamo a migliorare la modularità, la manutenibilità e la scalabilità del nostro progetto, assicurando un'implementazione più robusta e conforme ai principi di buona progettazione software.