Peer-Review 1: UML

Vaccarino, Vacis, Viganò, Zacchi

Gruppo GC05

Valutazione del diagramma UML delle classi del gruppo GC52.

Lati positivi

Game

- metodo *startGame()*: “blocca” tutte le modifiche, come l’aggiunta di un giocatore o una seconda chiamata a *startGame()* → utile per prevenire errori di modifiche

Decks

- utilizzato factory method per *DeckLoader* → utile per definire in modo scalabile il loading delle carte nei diversi deck
- Utilizzo efficace dei generics per aumentare la flessibilità del sistema

Card

- utile *ItemCollection* come classe per definire in maniera generale il totale di items
- metodo *placedAfter(Card)* utile per definire se una carta è piazzata sopra o sotto → per capire quali angoli sono esposti sulla board

Lati negativi

Game

- poca astrazione della chat → usato un attributo per rappresentare la chat e non una classe singola → non permette flessibilità in caso di modifiche future
- il metodo *isGameStarted()* deve ritornare bool
- avere la map che gestisce le chat non permette di definire un ordine in maniera semplice
- composizione di game da diversi messaggi è errata logicamente, un game non è composto da messaggi ma ad un game è associata una chat che è a sua volta composta da messaggi

Decks

- il giocatore non dovrebbe mai pescare a caso dal deck ma solamente la carta “topOfTheStack”. Non serve perciò definire un attributo *topOfTheStack: T* in quanto basterebbe definire la lista con un ordine specifico e prendere la carta in posizione 0 o len()-1
- Gestione poco efficace del tabellone centrale tramite solo la classe Deck, che viene “inquinata” nel suo concetto originale (mazzo di carte) dal metodo Deck.next per supportare un utilizzo che non le si addice (rappresentare anche il tabellone centrale)
- Generics non meglio specificati, sarebbe consigliabile inserire dei vincoli sul tipo T modificando, ad esempio, la definizione di Deck da Deck<T> a Deck<T extends Card> (applicabile in tutti i casi in cui sono definiti generics liberi nel resto del diagramma)

Goals

- complessa gestione dei pattern di carte tramite matrice se ogni carta è salvata tramite hashmap → ricercare un pattern di una matrice in una hashmap richiederebbe una bassa efficienza computazionale/temporale

Card

- gestione disordinata del piazzamento di una carta, l’informazione è distribuita tra due classi (Card e CardLocation) senza una chiara linea di separazione delle responsabilità. Sarebbe consono che tutte le informazioni sul piazzamento di una carta siano incapsulate in un singolo oggetto (la specifica dei metodi Card.flip e Card.place non è chiarissima in merito)
- (Legato al punto precedente) La gestione della sovrapposizione delle carte rende la ricostruzione della Board contorta
- complessa gestione delle facce della carta → la carta dovrebbe essere composta da due oggetti facce che renderebbero la gestione dei metodi molto più lineare
- tutte le carte dovrebbero avere dei *backCorners* e non solo le carte start → gestione poco robusta delle carte
- per semplificare si potrebbe spostare l’attributo *int scoreperCoveredCorner/freeScore/scorePerItem* all’interno di *PlayCard* senza doverlo scrivere in maniera ridondante all’interno delle diverse classi di *ScoringStrategy*
- nonostante gli attributi *PlayCard* e *StartCard* siano di tipo final, sarebbe meglio renderli private per non farli visualizzare al di fuori della classe → usare solo i metodi get
- a cosa serve definire il numero di corner empty ci sono nella *ItemCollection*?

PlayCard

- Perché non creare due classi separate GoldCard e ResourceCard? Aumenta la flessibilità dell’architettura e inoltre imposta l’estensibilità del sistema in caso di aggiornamenti futuri (da preferire rispetto alla modifica di classi già esistenti)

In generale sull’architettura

- Utilizzo di classi intermedie come astrazione di oggetti primitivi (ad esempio nel caso di Chat) per aumentare i punti di estensibilità del sistema in caso di modifiche future

Confronto tra le architetture

- Attributo *isStarted* in Game utile per definire quando partita creata ma ancora in cerca di tutti i player
- Classe *GameSelector* che crea e gestisce le partite multiple
- Definizione dei goal in due sottoclassi differenti
- Usare matrice per definire i pattern delle posizioni delle carte - dipende da implementazione
- Definizione attributo *disconnected* per il player, in maniera da poter gestire se un utente si scollega o meno
- Gestione con più attenzione dei package ed eventuali attributi protected
- L’idea di *ItemCollection* come classe per definire in maniera generale il totale di items
- Per lo scoring l’uso di *scoringStrategy* è una soluzione robusta e scalabile