Peer-Review 1: UML

Apollonio Marco, Bossi Giacomo, Alberto Bertazza, Luigi Biasi

Gruppo AM18

Valutazione del diagramma UML delle classi del gruppo AM27.

Lati positivi

La struttura complessiva del modello è ben chiara grazie anche all'utilizzo di interfacce dedicate che vengono poi implementate dalle varie classi interne al model.

Le relazioni tra le varie classi sono state ben descritte tramite le opportune frecce e nel complesso risultano chiare. Il pattern dell'ereditarietà è stato utilizzato in modo opportuno e rende più comprensibili alcune relazioni tra classi.

La struttura risulta chiara ed è ben mostrata la relazione con le classi esistenti.
L'approccio che sembra essere stato utilizzato è quello spostare tutta la parte di controllo nel controller, mentre il model più i componenti del gioco definiscano in modo completo lo stato della partita.

Lati negativi

Nel cercare di simulare una partita abbiamo notato tanti passaggi di oggetti tra diversi metodi di classi per aggiornare i punti di gioco.
Ci sono alcuni riferimenti che non sono strettamente necessari: ad esempio la classe Angle contiene un riferimento alla carta di appartenenza, ma questa stessa carta contiene a sua volta il riferimento ai 4 angoli.
in un'ottica di conteggio dei punti a fine partita, la struttura di ricerca delle carte del giocatore che si viene a creare non è particolarmente semplice da esplorare. Questa risulterebbe infatti essere un grafo in cui è abbastanza complesso andare a verificare quali obiettivi sono stati raggiunti, soprattutto nel caso degli obiettivi di tipo posizionale (in cui bisogna relazionare le carte in base alla loro posizione sul tavolo di gioco). Questo potrebbe risultare difficoltà di implementazione e/o maggiore tempo di calcolo.

Nel modello ci siano classi "troppo specifiche" evitabili con l'utilizzo di strategie soprattutto, polimorfismo e overloading/overriding. Un esempio di ciò sono le due classi GoalDeck e Deck che offrono praticamente la stessa interfaccia ma sono due classi separate. Unire le due classi in un'unica classe in grado di gestire entrambi i tipi la rende la gestione di questi Deck più snella e facile da usare.
Un altro esempio di questo sono le classi Goal, infatti ci sono ben cinque classi diverse, una per tipo di obiettivo, nonostante alcuni siano tra loro simili.
Per esempio, gli obiettivi con le risorse potrebbero essere uniti tenendo come attributo una collection con le risorse richieste e la quantità.
Suggeriamo inoltre di togliere le coordinate dalla carta e di tenerle separate da essa perché una carta nel mazzo esiste senza avere una coordinata assegnata.
In questo modo si possono semplificare notevolmente alcune operazioni.

Per esempio, se nel codex invece di un semplice ArrayList per le carte si usa una struttura di tipo Map o le sue sottoclassi, che usano come chiave le coordinate e come dato una carta, si possono cercare le carte in modo efficiente usando solo le coordinate.
Questo semplifica operazioni come verificare se una carta è effettivamente posizionabile dove richiesto dal client, ma anche cercare completamenti degli obiettivi a L e diagonali.

Abbiamo avuto difficoltà a comprendere la presenza di così tante sottoclassi di “Goals”, anche perché facendo una suddivisione così ampia è necessario un controllo sulla forma e ci sono più funzioni da implementare, tutte molto simili fra loro.
Le classi che ci lasciano più perplessi sono ”EqualsObjectGoals” e “DifferentObjectGoals”, che immagino siano una forma particolare di carte obiettivo basato sul numero delle risorse, quando sarebbe sufficiente utilizzare un'unica classe che contiene all’interno un Dizionario che associa ad una risorsa specifica la quantità richiesta.

Confronto tra le architetture

La differenza più grande che notiamo tra i due UML è che il nostro esegue operazioni di aggiornamento del gioco stesso all’interno del Model, mentre il gruppo AM27 ha optato per un approccio in cui gli aggiornamenti avvengono all’interno del Controller.
Il lato positivo di questo tipo di architettura potrebbe essere quello di semplificare la parte di model (che è in ogni caso già abbastanza popolata di classi) per delegare gran parte della logica di gioco al controller (che nel nostro caso è una struttura abbastanza leggera di mera gestione degli eventi).

Qui è stato inoltre adottato un approccio che permette di accedere immediatamente (tramite reference) alle carte adiacenti sul tavolo di gioco. Nel nostro caso, invece, abbiamo adottato un approccio basato sulle posizioni (coordinate) che quindi implica un piccolo passaggio in più per esplorare le carte adiacenti.

Rispetto al nostro progetto del model si nota un forte utilizzo di interfacce che noi non abbiamo usato ma un più scarso utilizzo di polimorfismo, overriding e overloading.
Mentre noi ci siamo concentrati su come le classi realizzano il loro compito (parecchi metodi per classe) loro hanno favorito un approccio basato su interfacce le quali però sono poco generiche, ad esempio, due interfacce per GoldDeck e Deck.
Le enumerations sono state utilizzate in modo analogo in entrambi i gruppi. L’enum AnglePos, è comprensibile ma non di immediata comprensione, suggeriamo di modificare i nomi degli elementi per rendere immediato il suo scopo.