Peer Review #1: UML

IS24AM14:

- Matteo Delogu,
- Vittorio Pio Remigio Cozzoli,
- Stefan Bogdanovic,
- Niccolò Francesco Basile.

Valutazione dell'UML del Model del gruppo [23].

Lati positivi

La documentazione presentata attesta come il lavoro passato in disamina da parte del gruppo [23] esprima appieno la comprensione da parte di quest'ultimo delle regole di gioco, nonché del radicamento più profondo derivante dai dettagli e dagli aspetti più intrinseci e di basso livello inveterati nella struttura di "Codex Naturalis".

La preponderanza degli elementi di gioco risulta ben implementata, con un particolare accento di apprezzamento circa l'adozione di differenti classi astratte volte a raggruppare i metodi comuni a più classi, una *best-practice* implementativa che favorisce dunque una buona modularità, precipua nel paradigma della programmazione ad oggetti.

Oltre ad aver apprezzato l'alta inclinazione all'*object-orientation*, troviamo particolarmente apprezzabile la scelta della struttura dati adottata dal gruppo [23] per la gestione del manoscritto di gioco: una matrice posizionale che tiene traccia di tutte le carte piazzate nell'area di gioco.

Lati negativi

Osservando la classe `Game`, è possibile notare la presenza dell'attributo `activePlayer`, mentre nella classe `Player` è presente l'attributo `isTurn`: ci domandiamo pertanto se sia necessario avere due attributi (con annessi metodi) in due classi diverse ma che indichino medesimamente di chi sia il turno. Nel caso in cui il gruppo si trovi in linea con l'eventuale ridondanza da noi identificata, potrebbe risultare comodo lasciare soltanto l'attributo `activePlayer` della classe `Game` e far sì che sia il controller ad informare il giocatore quando è il suo turno.

È bene evitare, ove possibile, di avere due classi diverse che si contengono a vicenda come nel caso di `Player` e `Manuscript` (come è possibile creare il *Player* se non ho creato il *Manuscript* e viceversa?). `Manuscript` non ha bisogno di vedere nessun metodo oppure attributo della classe `Player`; pertanto, sarebbe consigliabile rimuovere da essa l'attributo `player` per risolvere il problema.

Inoltre, sempre al fine di evitare la duplicazione dell'informazione e la ridondanza, non risulta necessario avere un attributo `availableCells` quando è di per sé possibile ricavare in maniera diretta dalla matrice quali siano le celle libere.

La classe `Cell` sembrerebbe non aggiungere alcuna informazione al *manuscript* in quanto, per ottenere il numero della riga e della colonna, basterebbe utilizzare il metodo `indexOf()` passando l'oggetto di cui si vuole ottenere le coordinate. Per dare un senso più concreto alle varie sottoclassi della classe `AbstractDeck`, si potrebbe valutare l'idea di creare delle apposite liste *"cards"* con i relativi tipi di carta (es. per il `DeckGold` creare una `List<GoldCard> cards`, e così via).

Sarebbe opportuno rendere pubblici i metodi `setCard` e `getCard` presenti nella classe `Manuscript`, in quanto dovrebbero consentire al controller di interagire direttamente con quest'ultimo.

Dove non necessario, sarebbe consigliabile evitare l'utilizzo degli attributi *"ID"*, dal momento che non si sta avendo a che fare con un database in cui gli oggetti sono record da cercare in un elenco (come nel caso dell'attributo `ID_attachedCard` della classe `Corner`).

Si consiglia poi di cercare, qualora fosse possibile, di generalizzare le `TargetStrategy` anziché utilizzare 16 sottoclassi diverse. In questo caso, 16 classi sono (relativamente) poche, ma se le carte obiettivo fossero un numero molto maggiore questo approccio potrebbe diventare disfunzionale; pertanto, riteniamo utile far notare come le carte obiettivo si possano classificare in tre sottogruppi: carte che assegnano punti in caso il giocatore possieda determinate risorse, carte che assegnano punti in caso egli possieda determinati oggetti e carte che assegnano punti se nella matrice delle carte del giocatore sono presenti tre carte disposte in un determinato modo.

Per ciò che concerne i design patterns utilizzati per la realizzazione degli elementi di gioco, è possibile notare come l'utilizzo del *factory* per la creazione delle carte potrebbe risultare potenzialmente superfluo in quanto tutte le carte verranno ad ogni modo create in un unico punto del codice, vale a dire nella classe astratta `AbstratDeck`. Non è quindi necessario delegare la creazione delle carte ad un'altra classe.

Per concludere, sarebbe opportuno, o quantomeno preferibile, correggere eventuali refusi presenti nell'UML ma soprattutto utilizzare i nomi ufficiali previsti dal regolamento del gioco al fine di migliorare la leggibilità e la chiarezza dell'operato del gruppo (es. *"Fungi"* anziché *"Mushrooms"*, *"Plant"* al posto di *"Vegetal"*, e via discorrendo).

Confronto tra le architetture

Non sono state riscontrate scelte architetturali, design patterns e/o scelte implementative particolarmente difformi dalle linee di pensiero seguite dal nostro gruppo. Tuttavia, abbiamo segnatamente apprezzato l'utilizzo di una matrice di carte per la gestione del manoscritto di gioco; de facto, è a tal fine in valutazione l'adozione di quest'ultima anche per il nostro progetto, in sostituzione della già pensata struttura a grafo, in quanto potrebbe rendere determinati aspetti e dettagli implementativi oltremodo snelli ed efficienti.