Peer-Review 1: UML

Bordignon L., Aliberti A., Barcellini L.

Gruppo AM-52

Valutazione del diagramma UML delle classi del gruppo AM-09.

Lati positivi

Dall'analisi del diagramma UML del gruppo AM-09, sembra che tutte le funzionalità di gioco siano correttamente modellate. Un aspetto positivo è il numero contenuto di classi utilizzate: infatti sono meno di una ventina di classi (più qualche enumerazione). Inoltre, dallo schema appaiono chiare le interazioni tra le varie classi. Questo aspetto, però, risulta anche essere un punto di debolezza del modello, in quanto sembra che ci sia poca granularità nella gerarchia delle classi, come spiegato nel paragrafo "lati negativi".
Un ulteriore aspetto positivo è la modellazione delle classi per l'implementazione della chat di gioco, che è una funzionalità aggiuntiva che il gruppo revisore non ha modellato.

Lati negativi

Alcuni aspetti negativi riscontrati sono i seguenti:

- In ottica di un design OOP, riteniamo più opportuno definire delle classi più specifiche nella gerarchia, che implementino al loro interno il comportamento specifico dell'oggetto. Nel UML revisionato, infatti, sembra che ci siano alcune classi generali con degli attributi da usare per l'algoritmo interno, piuttosto che un metodo astratto implementato nello specifico da sottoclassi. Per esempio, per gli obiettivi, la classe GoldCardsArrangements serve per tutti gli obiettivi legati alla disposizione delle carte (diagonali e "torri"), e distingue i vari casi con il campo privato arrangementType. Si deduce che la classe abbia un metodo di calcolo dei punti generico per tutti gli obiettivi, che utilizza il campo privato per distinguere i diversi casi. Dal nostro punto, di vista, sarebbe stato preferibile creare due classi separate, derivate dalla stessa classe base, per l'obiettivo diagonale e "torre", con un metodo implementato a doc per il caso specifico. Oltre ad una semplificazione del codice, un'eventuale aggiunta di nuovi obiettivi richiederebbe solo la definizione di una nuova classe, senza toccare il codice "generico" della classe base.

- Nella modellazione delle classi ci sono anche attributi legati all'aspetto grafico (o quantomeno "visivo") del gioco. A nostro parere, è meglio tenere nettamente separato il modello di gioco dall'aspetto grafico. Per esempio, il colore del segnaposto non ha nessun impatto sulla dinamica di gioco, ma ha solo una funzionalità grafica, che riteniamo preferibile definire solo nel codice della view. In questo modo, in caso di restyling del gioco, non è necessario modificare il codice del modello, ma solo quello della view.

- Da un punto di vista di implementazione generale, alcune classi si sarebbero potute definirle come immutabili (per esempio le carte), modificando il design delle classi stesse e del loro utilizzo.

- Dallo schema sembra che alcuni metodi prendano come parametro un riferimento a degli oggetti mutabili, anche se lo scopo del metodo non è quello di modificare l'oggetto: ad esempio, il metodo calculatePoints() della classe PointsCalculator prende un parametro di tipo Player, ma ovviamente non dovrà usare i metodi useCard(), addCards() o incrementSymbols() sull'oggetto

player. Questo a nostro avviso è un design “pericoloso”, perché si consente ad un metodo di una classe di poter accedere a metodi modificatori di un’altra classe senza che sia necessario, rischiando di interferire con lo stato interno degli oggetti. Sarebbe stato più opportuno definire un’interfaccia con solo metodi osservatori, oppure passare un oggetto immutabile con le info necessarie, oppure ripensare l’implementazione invertendo chiamato con chiamante: la classe Player offre un metodo che accetta un oggetto Card (avendo però implementato la card come classe immutabile)

Confronto tra le architetture

Lo schema generale della funzionalità del modello di gioco è simile per la nostra architettura e per quella del gruppo revisionato (AM-09). La grande differenza consiste nell’impostazione dei comportamenti delle classi. Noi abbiamo preferito definire delle gerarchie di classi molto profonde, in cui la classe base definisce metodi astratti che vengono implementati molto in basso nella gerarchia. Questo ha il costo di un maggior numero di classi, ma ciascuna con metodi molto semplici e specifici. Inoltre, l’aggiunta o modifica di funzionalità non richiede la revisione del codice in alto nella gerarchia, ma solo l’aggiunta di nuove classi o la revisione del codice di una classe specifica. Il gruppo revisionato, invece, sembra aver preferito un approccio opposto, in cui un algoritmo generico performa la funzionalità, gestendo la diversità di comportamenti in base a degli attributi privati (di tipo enum).

Anche al punto di vista dell’implementazione, il nostro approccio ha prestato molta attenzione nel definire classi immutabili, riducendo al minimo il numero di classi con metodi modificatori. Questa attenzione è stata posta anche nel passaggio dei parametri ai metodi per la comunicazione tra oggetti, evitando di passare al metodo chiamato dei parametri di tipo mutabile, se non strettamente necessario (cioè, se effettivamente il metodo ha la funzione di modificare l’oggetto ricevuto). A questo scopo il nostro gruppo ha fatto uso di interfacce con metodi osservatori, o oggetti immutabili definiti ad hoc. Questa direzione è stata intrapresa per semplificare l’implementazione dei metodi, evitare di introdurre bugs (side effects indesiderati), rendere più chiaro lo scopo dei metodi, ridurre problematiche di sincronizzazione in ottica multithreading.

Alla luce di quanto illustrato sopra, il gruppo revisore (AM-52) trova come spunto di miglioria, per il proprio modello, la definizione delle classi per la funzionalità della chat di gioco.
Per il resto del modello, il gruppo AM-52 non trova particolari aspetti nella modellazione del gruppo revisionato (AM-09), da poter prendere come spunto per migliorare il proprio modello di gioco, in quanto:

- lo schema della funzionalità generale è molto simile
- la modellazione delle singole classi risulta troppo grossolana (poca granularità delle gerarchie)