Peer-Review 1: UML

Pietro Agnoli, Pietro Benecchi, Daniel Carozzo, Arturo Amoretti

Gruppo AM26

Valutazione del diagramma UML delle classi del gruppo AM35.

Introduzione

In generale il design sembra soddisfacente e completo. Con questo design risulta possibile gestire correttamente tutte le fasi del gioco. Tuttavia, sono giunte perplessità sul design delle carte e di come possiate visualizzare e "muovervi" tra le carte giocate. Per alcune classi i metodi elencati sembrano avere firme scorrette: quelli più evidenti sono stati riportati.

Lati positivi

Game

La classe game risulta ben fatta. Abbiamo simulato il gioco e possiede tutti i metodi necessari per effettuare una partita e gestire correttamente ogni fase di gioco. I metodi sembrano essere ben divisi e unitari, favorendo il riutilizzo del codice.
Come avete riportato nel documento, l'invocazione dei vari metodi ci risulta corretta e completa.
Suggeriamo però di introdurre un'enumerazione per gestire le fasi di gioco. Per esempio, cosa succede se un player prova a posare due carte in una giocata? Creando un attributo apposito, potete lanciare delle eccezioni se avvengo richieste da parte client scorrette.

Market

Il design dei vari deck è ottimo. Dividere le tipologie di carte favorisce la loro distribuzione nelle varie fasi di gioco e market possiede tutti i metodi per creare le carte e distribuirle nei vari deck.
Creare una classe apposita per i deck è una soluzione elegante e semplice. È anche molto utile dividere le carte nel mazzo con quelle disponibili sul tavolo.

Funzione calcolo punti

L'implementazione di calcolo punti è ottimale. Utilizzando un bit Mask che memorizza la disposizione delle carte che soddisfano l'obiettivo semplifica notevolmente l'implementazione di uno dei metodi più complessi del design. Si consiglia perciò di mantenere tale questa parte del design.

Score track e Token

Implementare una classe per realizzare il tabellone è molto utile in quanto semplifica la realizzazione della view. Abbiamo notato che avete realizzato solo una lista di interi. Perché allora non realizzare una lista di <int(1-29), list(token)> (lista di token poiché più giocatori possono avere gli stessi punti) così da salvare i punti dei vari giocatori?

Lati negativi

Il design fornito utilizza poco la programmazione ad oggetti. In particolar modo il design delle carte sembra carente da questo punto di vista. Inoltre, suggeriamo alcune semplificazioni alle varie classi. Alcune firme risultano, secondo nostra analisi, scorrette e proponiamo un breve commento per risolvere le problematiche.

Carte

Il design delle carte sembra non utilizzare in modo efficace la programmazione ad oggetti. Si nota il fatto che utilizzando un'unica classe per le carte si lasciano inutilizzati (null) numerosi parametri.

1. L'utilizzo di front e black per tutte le carte è ridondante. Le carte risorsa e oro presentano sempre lo stesso retro (quattro angoli liberi + una risorsa), perciò è inutile salvare le informazioni del lato retro per questa tipologia di carte.

2. Il salvataggio di tre ResourceType sul back è anch'esso superfluo. Solo starting card possiede due o tre risorse. Utilizzare questi parametri per tutte le carte comporta metterli a null sia per le carte oro che per le carte risorsa(queste carte hanno solo una risorsa).

3. Abbiamo compreso che per salvarvi il lato di carta giocato utilizzate solo un parametro tra front e back mettendo l'altro a null. La soluzione però non sembra delle migliori inquanto prima che una carta sia giocata (messa sul tavolo) dovete salvarvi comunque entrambi i parametri. Inoltre, dovete ripetere le informazioni dei corner su entrambe le strutture dati, il che è ridondante. Potete trasferire queste informazioni sulla classe carta e semplificare l'utilizzo della memoria.

Si suggerisce perciò la seguente modifica: utilizzare una classe astratta per la carta in generale, e suddividere in due sottoclassi diverse le carte risorsa/oro con quelle starting. Così facendo non è più necessario avere salvati sia un front che un back per tutte le carte, ma solo per le starting card. Per salvarvi il lato giocato è sufficiente un booleano o una enumerazione (Fronte, retro, non giocata).
Inoltre, per evitare l'utilizzo di vari attributi simili come il conteggio delle risorse di una carta, l'utilizzo di una hash o una lista può semplificare l'implementazione e limitare gli attributi per classe.

ObjectiveCard

Come per altre classi, si potrebbe utilizzare una struttura dati avanzata per salvare la risorsa necessaria per la carta obiettivo. Si potrebbe utilizzare una enumerazione per distinguerne la tipologia (Tris, forma a L, risorse) e un intero per sapere quante risorse avere.

Corner

Utilizzare una classe corner sembra una valida soluzione. Tuttavia, si ricorda che un corner può avere un solo tipo di risorsa/oggetto o essere vuoto. Perché non crearsi una enumerazione ad hoc ed avere un solo attributo per salvarsi la tipologia di angolo? Si potrebbe adottare la stessa soluzione anche per la tipologia di angolo (HL, HR, …).

Player

La classe player è simile al nostro design e non abbiamo particolari commenti. Alcuni metodi e firme sembrano però essere scorretti. Elenchiamo in seguito quelli più evidenti:

Metodo	Commento
+Card drawCard(Player player, Type type)	È inutile chiamare l'oggetto player in quanto è il chiamante.
+placeCard(TableArea tableArea)	Bisogna passare anche la carta da piazzare. Non serve passare anche la table perché è attributo del chiamante.
+int modifyPoints (Player player)	Non serve passare l'oggetto player, è il chiamante

+void throwCard()	Non è stato specificato nel documento inviato e sembra avere un comportamento ridondate al metodo placeCard.

TableArea

Analizzando questa classe sono giunte perplessità riguardo a come muoversi tra le varie carte, in quanto crediamo che con il design scelto non sia possibile.

Nel vostro design nella mappa (tableArea) salvate unicamente i vari angoli di una carta.
Un angolo è identificato da una sola carta, ma una carta non è identificata da solo quattro angoli. Infatti, nella view bisognerà stampare gli angoli, ma anche le risorse, i punti e la tipologia di Kingdom di ogni carta. Come fate a fare tutto ciò? Gli angoli salvati nella matrice non possiedono nessun riferimento a che carta appartengono perciò risulta problematico stampare ulteriori informazioni oltre a quelle degli angoli.

Inoltre, non comprendiamo come possiate contare i punti obiettivo segreto. Salvando solo i corner nella matrice saprete certamente le risorse che ogni giocatore possiede ma non il resto: la disposizione dei vari Kingdom delle carte. Gli angoli non possiedo riferimenti alla carta o attributi che indichino a quale tipologia di Kingdom la loro carta appartiene.
Come si fa perciò a sapere se la disposizione soddisfa la configurazione di Kingdom necessaria per la carta obiettivo? Per gli obiettivi come il tris in diagonale oppure la L capovolta non importa la tipologia di angoli, ma il tipo di Kingdom (colore) di ogni carta. Utilizzando una matrice di corner sembra non abbiate accesso a queste informazioni.

Per sistemare queste problematiche si potrebbero adottare varie soluzioni. Ne riportiamo due:

1. È possibile modificare la matrice di corner con una matrice di card. Così facendo avrete tutte le informazioni necessarie per la view e per il calcolo degli obiettivi finali. Potete così preservare anche le informazioni degli angoli avendo la carta quattro attributi per gli angoli.
2. Mantenere una matrice di angoli ma aggiungere una serie di attributi come un riferimento id a ciascuna carta. Questa soluzione crediamo sia meno preferibile poiché più laboriosa ma da tenere in considerazione se avete già sviluppato questa parte del model.

Confronto tra le architetture

Si procede a confrontare la vostra architettura con la nostra. Elenchiamo prima i vantaggi e poi gli svantaggi della vostra architettura, suggerendovi alcune modifiche simili al nostro design.

Game

La classe Game, come la nostra, gestisce le varie fasi del gioco. Nel nostro design abbiamo realizzato una classe Lobby per salvare i vari giocatori mentre i vari deck delle carte rimangano nella classe Game. La soluzione da voi proposta risulta più semplice ed elegante. Avendo Game meno attributi, risulta una classe più piccola e semplice da utilizzare.

Tabellone punti

Nel nostro design, non abbiamo realizzato nessuna struttura per il tabellone. Usare la vostra soluzione semplifica la gestione della view ed è una soluzione che certamente implementeremo. Suggeriamo però la modifica in una struttura dati più complessa capace di salvare sia i numeri da 1 a 29 ma anche dove si trovano i vari token. Una lista di generics è più che sufficiente.

Player

Per la classe player gli attributi e i metodi di entrambi design sono molto simili. Nel nostro design abbiamo delegato maggior responsabilità alla classe Game Master (equivalente del vostro Game): la classe Player mantiene le varie informazioni come carte in mano e punti, ma è game master che fa scegliere le carte da pescare e quelle da giocare tramite la view. La vostra soluzione invece, permette alla classe player di pescare e giocare le varie carte tramite i metodi opportuni specificati nel documento inviato. Inoltre, il conteggio punti è affidato totalmente alla classe Game.
Riteniamo che siano soluzioni entrambe valide e nessuna abbia un particolare svantaggio o vantaggio.

Funzione obiettivo segreto

Utilizzare una matrice per gestire la carta obiettivo può essere una valida soluzione. Non considerando il problema esplicato nella sezione lati negativi, una matrice può semplificare l'implementazione di numerose funzioni per il calcolo degli obiettivi.
Tuttavia, non comprendiamo come possiate posizionare le carte sulla matrice. Per preservare le coordinate sulla mappa fisica, posizionare Starting card sulla posizione (0, 0) sarebbe naturale. Con questa soluzione però è impossibile mettere a sinistra ulteriori carte in quanto avrebbero ascissa negativa. Inoltre, una matrice non rispecchia il contenuto della mappa, che non è necessariamente quadrata, e questo comporta uno spreco di memoria.
Attualmente noi stiamo implementando un grafo per gestire la table. Così facendo è possibile posizionare le carte nella posizione naturale senza usare un sistema di coordinate complesso o spreco di memoria.
Crediamo perciò che la matrice possa essere una valida soluzione, ma vi consigliamo di ragionare sulle nostre perplessità. Il nostro gruppo non è riuscito a risolvere queste problematiche usando questa struttura dati: potreste risolvere le coordinate mettendo starting card in una posizione sufficientemente grande come (30,30), avendo però un enorme spreco di memoria.
Il nostro design probabilmente più difficile da implementare risulta più efficiente in termini di spazio e vi consigliamo di valutarlo.

Carte

Per le carte preferiamo la nostra soluzione. Non avendo utilizzato sottoclassi avete per numerose carte, attributi vuoti o ridondanti. Invece, la nostra soluzione prevede una carta astratta con solo attributo id, dalla quale si sviluppano varie sottoclassi:

1. Starting card che gestisce gli angoli di fronte e di retro.
2. Resource card che possiede solo angoli di fronte.
    a. Da cui si sviluppa Gold card con attributi punti e risorse necessarie. Infine, da Gold card sviluppiamo special Gold card con l'attributo oggetto speciale.
3. Objective card che possiede attributi per indicare la tipologia di obiettivo.

Per comprendere come le varie carte sono posizionate (fronte o retro) invece di avere due attributi front o back, abbiamo scelto di incapsulare il contenuto di un solo lato della carta in una nuova classe PlayedCard, così da rendere impossibile l'accesso al lato non giocato.
La nostra soluzione utilizza maggiormente il paradigma della programmazione ad oggetti e definendo le varie sottoclassi, si eliminano vari riferimenti a valori nulli e si favorisce il riutilizzo del codice.