Descrizione UML Model – GC42

Introduzione

Per il nostro progetto abbiamo scelto di implementare un'architettura di tipo "*Fat Model*", ovvero in cui la maggior parte della logica di funzionamento è implementata all'interno del *Model*, ed in cui il *Controller* (di tipo "*Thin Controller*") si occupa solamente di chiamare nell'ordine corretto una serie di metodi pubblici di alto livello presenti nelle varie classi che compongono il *Model*, le quali eseguono internamente le chiamate ai metodi che effettuano le modifiche vere e proprie sui dati.

Questa decisione architetturale è stata presa in ottica di semplificare l'implementazione del *Controller*, quasi a renderlo uno strumento puramente di *scripting*. Questo ci consente, inoltre, di implementare eventuali modifiche future alle implementazioni di gioco senza dover toccare il *Controller*, in quanto tutta la logica è localizzata nel *Model*.

Per semplificare il lavoro di revisione, sia in questo documento che nell'UML, i metodi di alto livello che vengono chiamati dal *Controller* saranno colorati di rosso.

Game

Il nostro progetto si basa principalmente sulla classe Game, che gestisce la partita ed i giocatori presenti in essa (questi ultimi nell'attributo *players*). Questa classe contiene anche un *Deck* (*starterDeck*) contenente le carte iniziali e 3 *PlayingDeck* (*resourceCardDeck, goldCardDeck, objectiveCardDeck*)
La scelta di implementare anche i "contenitori" delle carte iniziali e delle carte obiettivo come un mazzo è stata presa in ottica di riutilizzo del codice, nello specifico per poterle mischiare e pescare. Le carte Obiettivo, inoltre, sono contenute in un *PlayingDeck*, in modo che i due slot contengano gli Obiettivi Comuni.

Gli attributi booleani *isResourceDeckEmpty* e *isGoldDeckEmpty* segnalano lo stato dei rispettivi mazzi (aggiornati mediante un *listener*), mentre il booleano *playerHasReachedTwentypoints* viene aggiornato, sempre mediante un *listener*, dal primo giocatore che supererà i 20 punti.
L'aggiornamento di uno qualunque dei 3 valori innesca una chiamata al metodo privato *checkEndGame*, che verifica se è il caso di avviare la fase finale del gioco.
I turni sono gestiti mediante un attributo di tipo *int* che rappresenta semplicemente il numero del giocatore a cui spetta la mossa (indice del giocatore nell'*Arraylist* +1, in modo da numerarli da 1 a 4 per gli utilizzatori esterni).

Il metodo *initPlayingDecks* inizializza tutti e 4 i mazzi mediante il metodo *initDeck* della classe *Deck,* il quale non soltanto crea i mazzi, ma li riempie anche inizializzando le carte tramite dati contenuti in un file JSON, e posiziona due carte negli slot dei *PlayingDeck* dove previsti. Questo metodo viene chiamato automaticamente dal costruttore della classe *Game*, in modo che la creazione di un nuovo oggetto *Game* porti automaticamente all'inizializzazione delle carte.

Questa classe non è pensata, però, per essere utilizzata direttamente, ma è previsto un *GameController*, il quale andrà a chiamare i metodi *startGame*, *endGame*, *addPlayer*, *kickPlayer*, ecc, in modo appropriato per automatizzare lo svolgimento della partita.

Player

Ogni giocatore avrà le proprie carte in mano contenute in un *Arraylist* chiamato *hand*.
Questo *ArrayList* non è visibile dall'esterno, bensì accessibile mediante metodi *getCard* e *setCard*, non esplicitati nell'UML in quanto affini a banali *getter* e *setter*.

Sono presenti anche un booleano per indicare se il giocatore è il primo del giro, un *int* per contenere i punti che quel giocatore ha accumulato (ovvero la posizione sul tabellone), una *ObjectiveCard* che corrisponde all'Obiettivo Segreto pescato dal giocatore (conserviamo l'intera carta in modo da poterla mostrare nella GUI), ed un *PlayField*, che corrisponde al proprio "tavolo".

Il metodo *drawStartingHand* viene chiamato automaticamente nel momento in cui viene avviata la partita, e si occupa di pescare le 3 carte iniziali (2 *ResourceCard* e 1 *GoldCard*), che vengono posizionate nella mano.

In modo simile il metodo *drawSecretObjectives* pesca due carte obiettivo, proponendo al giocatore di sceglierne uno da tenere come obiettivo segreto.

playCard aggiunge la carta passata come argomento nell'*Arraylist* di *PlayField*, impostandone prima le coordinate x e y passategli come parametro nei rispettivi attributi.

drawCard pesca una carta dal mazzo contenuto nel *PlayingDeck* passato come parametro, e la aggiunge alla mano del giocatore.

grabCard prende una delle due carte scoperte posizionate negli slot del P*layingDeck* passato come parametro (l'*int* indica da quale dei due slot prendere la carta).

Player implementa l'interfaccia *Observable*: quando i punti raggiungono e/o superano il 20 viene notificato il *Game*, che imposta a *true* l'attributo *playerHasReachedTwentyPoints* (se è il primo giocatore a notificarlo).

PlayField

Tramite un *Arraylist* contiene tutte le carte giocate dal giocatore, ovvero posizionate sul tavolo.
È stato scelto di utilizzare un *Arraylist* nello specifico per poter tenere in memoria l'ordine di piazzamento delle carte, la cui posizione è gestita mediante delle coordinate x e y memorizzate negli gli attributi di ciascuna carta (*Card*).

L'idea dietro al posizionamento delle carte nel campo è basata su un piano cartesiano ruotato di 45° (in senso antiorario), in modo che qualunque coppia di coordinate intere corrisponda ad una posizione valida sul tavolo, e risulti invece impossibile piazzare una carta di fianco ad un'altra.

Per tenere conto delle risorse e degli oggetti giocati è stata implementata una *HashMap*, che contiene al suo interno una coppia <*String, int*> che associa ad ogni risorsa/oggetto un numero.
Sono previsti anche metodi per leggere ed incrementare/decrementare questi valori, omessi dall'UML in quanto affini a dei banali *getter* e *setter*.

PlayingDeck

Un *PlayingDeck* è un "contenitore" che ingloba un mazzo e due carte dello stesso tipo.
Questo consente di raggruppare un mazzo dello stesso tipo (ad esempio quello delle carte risorsa) insieme alle sue due carte scoperte, in modo da rendere più comodo il pescare queste carte, il

rimpiazzamento delle carte scoperte (che avviene in automatico quando una viene presa), ed il riutilizzo del codice.

Nota: è previsto un *PlayingDeck* anche per le carte Obiettivo. Questo viene utilizzato solo ad inizio partita per pescare prima gli Obiettivi Segreti dei giocatori (che pescano dal mazzo al suo interno), e poi per decidere i due Obiettivi Comuni per tutti (le carte che vengono posizionate nei due Slot).
Per tutto il resto della partita quel *PlayingDeck* rimane inaccessibile, e viene nuovamente utilizzato solo ed esclusivamente per il conteggio dei punti finale.

Deck

Il *Deck* è un *Arraylist* di carte (*Card*).
Ogni *Deck* ha un *CardType*: un oggetto proveniente da un'enumerazione contenente i 4 tipi di carte esistenti nel gioco.

Il metodo *initDeck* inizializza il mazzo, riempiendolo con le carte importate da un file JSON e successivamente mescolate tramite il metodo *shuffle*.

getCorner, *getKingdom* e *getObjective* sono metodi di supporto a *initDeck* per trasformare le informazioni lette dal file in oggetti.

draw ritorna la prima carta dell'*Arraylist* (tramite *getFirst*) e la rimuove da esso.

Deck implementa l'interfaccia *Observable* e notifica *Game* quando il numero di carte contenuto raggiunge lo zero, ovvero il mazzo si svuota: *Game* imposta un *listener* solo per il *Deck* delle carte risorsa e per quello delle carte oro, e quando loro notificano l'evento "Mazzo vuoto" viene settato a *true* il rispettivo booleano.

Card

La classe *Card* è la classe da cui discendono i vari tipi di carte implementati nel gioco.

isFrontFacing è un booleano che indica se la carta è girata mostrando il fronte (*true*) o il retro (*false*). Ogni faccia è rappresentata da un *CardSide*, un "contenitore" di angoli, chiamati *frontSide* e *backSide*.

id rappresenta un numero univoco associato ad ogni carta nel file JSON.

x e *y* rappresentano le coordinate che verranno associate alla carta una volta piazzata mediante il metodo *playCard*.

flip inverte lo stato del booleano (*isFrontFacing*). Può essere chiamato direttamente dal *GameController*.

Card ha 4 sottoclassi, una per le carte risorsa, una per quelle oro, una per le iniziali e una per le carte obiettivo.

CardSide

"Contenitore di angoli (*Corner*). Contiene 4 angoli al suo interno. Ogni carta contiene due *CardSide*: uno per il fronte, con un insieme in genere eterogeneo di angoli, e uno per il retro, inizializzato sempre con angoli vuoti (*EmptyCorner*). Questo lato è stato comunque incluso per permettere una futura aggiunta di carte con angoli diversi anche sul retro.

Gli angoli o *Corner* in esso contenuti possono essere di 4 tipi: angoli con risorse (*KingdomCorner*), angoli con oggetti (*ResourceCorner*), angoli vuoti (*EmptyCorner*) o *null*, che indica che l'angolo non è

presente (corrisponde all'angolo "non disegnato", sopra il quale non è possibile piazzare un'altra carta).

Corner

È una classe astratta che rappresenta gli angoli contenuti nella faccia della carta.

isCovered è un booleano che indica se l'angolo è coperto da un'altra carta.

Quando viene chiamato il metodo *playCard* di *Player*, tra le altre cose, viene effettuata una ricerca di carte "vicine" alle coordinate della nuova carta piazzata, ed i loro angoli (in base alla posizione) vengono "coperti".

Corner possiede delle sottoclassi concrete, una per gli angoli contenti le risorse (*KingdomCorner*), una per quelli contenenti gli oggetti (*ResourceCorner*) e una rappresentante l'angolo vuoto (*EmptyCorner*). L'angolo non presente è indicato come *null*.

ResourceCard

Classe che rappresenta le carte risorsa.

Contiene una *permanentResource*, la risorsa permanente posizionata sul retro, che viene anche usata per indicare il colore della carta.

Contiene anche un *int* per indicare i punti che la carta fa guadagnare al giocatore quando viene piazzata, 0 se non ne dà.

GoldCard

Come la *ResourceCard*, contiene *earnedPoints* e la *permanentResource*, ma contiene anche una *HashMap* che indica i costi per ogni tipo di risorsa: il costo indica quante risorse di un certo tipo è necessario avere visibili nel campo per poterla piazzare.

Le *GoldCard* possono avere una condizione per poter calcolare i punti, ad esempio "2 punti per ogni angolo coperto" oppure "1 punto per ogni risorsa del tipo X". Queste condizioni sono state unite al concetto di Obiettivi, in quanto altro non sono che Obiettivi di conteggio.

Objective

Classe astratta che rappresenta gli obiettivi.

Ogni obiettivo ha un numero di punti che il giocatore guadagna ogni volta che viene completato, e ha una stringa di descrizione (usata per spiegare la carta di Obiettivo Segreto nella GUI).

Il metodo *check* è un metodo astratto che prende un *ArrayList* come parametro (ovvero le carte giocate dal giocatore), e su cui conta quante volte l'obiettivo è stato completato.
Ritorna un *int*: il numero di volte in cui l'obiettivo è stato completato (0 se non è mai stato completato).
Essendo un metodo astratto, ognuna delle sottoclassi lo implementa a modo proprio, compatibilmente con l'obiettivo che rappresenta.

Il metodo *calculatePoints* è pubblico ed è quello che viene chiamato dall'esterno: ritorna il numero di punti moltiplicato per il numero di volte che l'obiettivo è stato completato, ovvero il numero di punti che il giocatore guadagna.

CountObjective

Classe astratta che rappresenta gli obiettivi di tipo “di conteggio”, usati sia dalle carte obiettivo che dalle carte oro (come condizione).

L’attributo *number* indica il numero di volte in cui un oggetto deve essere contato per considerare completato l’obiettivo: se 1 indica gli obiettivi di tipo “per ogni …”.

Ha quattro sottoclassi concrete: *KingdomCountObjective*, *ItemCountObjective*, *SetItemCountObjective* (che conta la terna di oggetti diversi) e *CornerCountObjective* (che conta gli angoli coperti dalla carta sulle carte vicine, da usare solo come condizione delle carte oro).

PlacementObjective

Classe astratta che rappresenta gli obiettivi di tipo “di posizionamento”.

Ha un attributo *primaryType* che indica il colore principale delle carte che compongono il pattern: per *DiagonalPlacementObjective* indica l’unico colore, per *LShapedPlacementObjective* indica il colore delle due carte disposte verticalmente, mentre il colore della terza carta è indicato dal suo attributo esclusivo *secondaryType*, insieme al *positionCorner* che indica dove è posizionata la terza carta.

DiagonalPlacementObjective ha anche un booleano che indica la disposizione: \ se *true*, / se *false*.