Peer-Review 1: UML

Bergagna Matteo, Brun Antony, Chen Jie, Chiroli Lorenzo - Gruppo 32

Valutazione del diagramma UML delle classi del gruppo 41.

Lati positivi

- Implementabile: non si notano criticità che possano impedire a questo UML di rappresentare lo stato di gioco. La squadra si è impegnata a rispettare la naturale divisione del gioco fisico. Hanno rispettato tutte le convezioni tipiche dell'UML.
- Comprensibile: la maggior parte dei metodi relativi alla logica di gioco è stata riportata correttamente nell'UML. Le idee principali della squadra in questa fase di sviluppo preliminare sono state trasmesse. I metodi impiegano una nomenclatura chiara che permette di comprendere chiaramente la loro funzione.

- Inoltre, si può notare che:
  1. Il Paradigma Object-Oriented è generalmente rispettato, con una separazione logica di ruoli e metodi valevole per il tipo di programma da implementare. Si evidenzia particolarmente la divisione del modello in Game, Player, Hand, Deck, Card, PersonalBoard attenendosi in maniera generale alla struttura del gioco fisico.
  2. Vengono correttamente utilizzate enumerazioni al posto dei cosiddetti "Magic Numbers"; le enumerazioni seguono la struttura del gioco fisico e si possono trovare: *Resource*, *Object* e *OtherElement.*

Lati negativi

- In generale:
  1. Il diagramma UML risulta non facilmente interpretabile data l'elevata quantità di classi e relazioni. Sarebbe consigliabile dividere il diagramma in packages e organizzare la disposizione delle classi per ottenere una migliore leggibilità.
  2. L'eccessiva rigidità nell'attenersi alla struttura del gioco fisico e il tentativo di trasporre queste strutture in relative classi e/o sottoclassi potrebbe provocare complicazioni durante la scrittura del codice e rende difficile l'utilizzo dei pattern tipici del ObjectProgramming (es: GoldDeck e ResourceDeck potrebbero essere la stessa classe).
  3. Le relazioni tra Player e Hand, e tra Player e PersonalBoard, dovrebbero essere delle composizioni, dato che, concettualmente, ne Hand, ne PersonalBoard, possono esistere senza il player.
- "Magic Numbers":
  1. Per quanto concerne il metodo *getAmountOfResource,* in classe *PersonalBoard*, potrebbe essere più conveniente passargli un'enumerazione, invece che un intero, facilitando la comprensione del codice in fase d'implementazione: per esempio, *getAmountOfResources(RESOURCE.ANIMAL)* è più leggibile di *getAmountOfResources(3).*
- Classi e interfacce vuote:

1. L'uso di classi e interfacce vuote potrebbe indicare problemi nel design: ad esempio, la classe *ResourceCard* non aggiunge nessuna funzionalità in più rispetto alla classe padre *ColouredCard*. Lo stesso vale per l'interfaccia *Element*.

- ### Variabili non necessarie:
  1. Si potrebbe evitare la creazione di variabili che hanno valori facilmente deducibili dalle strutture dati: ad esempio, i valori dell'attributo *placedCardsNum,* nella classe *PersonalBoard,* possono essere dedotti dalla mappa (ricavando il numero di chiavi presenti in essa); il metodo *getPlacedCardsNumber()* non sarebbe più un "getter" banale ma si rimuove un attributo superfluo.
- ### Gestione del "verso" delle carte:
  1. Non è intuitivo come il modello gestisca il verso con cui le carte vengono piazzate nel campo: in particolare, non è esplicito che sequenza di metodi porti al piazzamento della carta nel field.
- ### Uso di Interfaccie:
  1. Si potrebbe creare un'interfaccia per la classe Game che sottolinei i metodi che essa deve esporre al controller; ciò potrebbe tornare utile per suddividere il lavoro. Avere garanzia che la classe Game implementi certe funzioni può infatti permettere la scrittura anticipata di test in modalità blackbox.
- ### Possibile uso/abuso di getClass() e instanceOf():
  1. Nella classe ObjectivePointsCalculator vengono salvati gli obbiettivi comuni e privati (nel Player) a compile-time come ObjectiveCard. Per calcolare i punti da aggiungere ai player si fa uso però di metodi che per parametro richiedono sottoclassi di ObjectiveCard, questo implica che sia necessario a runtime un casting ma anche di verificare, prima di richiamare i metodi, la natura degli obbiettivi attraverso metodi come getClass() o isIstance(). Questa necessità potrebbe essere un campanello di allarme per quanto riguarda un design non buono di questa fase del gioco. È consigliato fare uso del pattern Strategy per casi come questi.
  2. Per come è strutturato il modello, le carte piazzate sono immagazzinate in formato Placeable: sorge però il problema che, quando si fa una qualsiasi operazione su queste carte, in quanto di tipo Placeable, non sarà possibile utilizzare vari i metodi di Card, StarterCard, ColoredCard o GoldCard senza prima verificare il tipo di istanza con getClass o istanceOf per poi fare casting. Si ha quindi un uso massivo di questi metodi e di controlli di condizioni nel codice; questo è un segno di un possibile errore di design.

Confronto tra architetture

Le due architetture, la nostra e quella del gruppo revisionato, presentano varie similarità e differenze.

- ### Field: Map vs ArrayList

  Il gruppo revisionato ha deciso di utilizzare una mappa Map<Position,Placeable> per memorizzare le carte piazzate sul campo sfruttando una coppia key-value di tipo coordinate-carta; noi abbiamo adottato un approccio simile utilizzando un ArrayList <cardPlaced> i cui oggetti sono degli aggregati che contengono al loro interno le coordinate delle carte, il riferimento alla carta stessa e un attributo per indicarne il verso di piazzamento.

- ### Verso delle carte piazzate: collocazione dell'attributo nelle classi
  Si nota inoltre che il gruppo revisionato ha deciso di mettere il flag del verso di piazzamento nella classe Card: concettualmente è possibile considerare tale flag come una “proprietà” intrinseca della carta ma, quando questa non ancora stata piazzata, questo attributo è privo di significato. Si potrebbe valutare l’uso di una classe extra contenente un attributo Placeable e il verso, con conseguente modifica della Map nella classe Player.
- ### Hand: nuova classe vs ArrayList
  Il gruppo revisionato ha creato una classe Hand per rappresentare la mano del giocatore Player. Noi, al posto di creare una nuova classe, abbiamo deciso di aggiungere la mano del giocatore direttamente nella classe Player come una semplice ArrayList<nonObjectiveCard>. Entrambi gli approcci sono ragionevoli, però la creazione di una classe separata per la Hand del giocatore, aumenta sia la modularizzazione del codice sia la complessità dell’implementazione.
- ### Divisione delle carte: divisione dettagliata seguendo il gioco fisico vs divisione sulle base delle funzionalità
  Mentre il gruppo revisionato ha deciso di separare completamente le carte Resource, Gold, e Objective, creando classi apposite per ogni tipo di carta, noi abbiamo deciso di mantenerle unite, avendo notato un alto livello di similarità tra gli attributi e i metodi dei vari tipi di carte.
- ### Punti delle carte: calcolatori separati vs unico calcolatore
  Il gruppo revisionato ha deciso di separare il calcolo dei punti per le carte obiettivo da quello per le carte “colorate” (gold and resource), mentre noi, utilizzando un strategy pattern, abbiamo omogeneizzato tale operazione, assegnando a ogni carta una strategia di calcolo diversa.
- ### File di Config e inizializzazione delle carte
  Nel modello presentato dal gruppo revisionato si nota che i deck vengano inizializzati dal metodo createDeck () nella classe Game: data la mancanza di maggiori dettagli su come avviene questo processo e la mancanza di una classe a sé stante che compie tale compito, sembra che l’inizializzazione sia hardcoded. Per confronto nel nostro schema facciamo uso di una classe che crea i deck e le carte da un file JSON; questo porta diversi vantaggi come il poter variare le carte e il loro numero senza ricompilare il programma ma semplicemente modificando il file e riavviando l’applicazione.
  Molto similmente si consiglia di valutare l’uso di un file di Config (a piacere JSON, XML, etc.) per variare alcuni parametri di gioco rapidamente come, ad esempio, maxPlayer senza dover ricompilare. A questo proposito, anche il nostro schema iniziale del modello non considerava l’uso della parametrizzazione del programma attraverso un file di Config, perciò, è una modifica di cui valuteremo l’implementazione.

In conclusione, entrambi i gruppi hanno individuato gli stessi elementi cardine del gioco (come la differenziazione in partita, giocatori, carte, ...) ma l’effettiva implementazione, a livello di classi, è sostanzialmente differente (specialmente sulle tipologie di carte).