Peer-Review 1: UML

Poidomani Davide, Riboni Andrea, Volpentesta Edoardo
Gruppo GC51

1 aprile 2022

Valutazione del diagramma UML delle classi del gruppo GC61.

1 Lati positivi

- **Le classi di gestione**: ottima l'idea di delegare i compiti onerosi ad apposite classi (es.: *AssistantDeck, Team, Islands*) e di non sovraccaricare le classi più *semplici*
  - In particolare abbiamo apprezzato la classe *Islands* per gestire al meglio l'unione di isole adiacenti: l'introduzione di questa classe supervisionante semplifica il calcolo dell'influenza
- **Attributi ridondanti che semplificano**: altra nota positiva è data dalla presenza di alcuni attributi che potrebbero essere calcolati sul momento ma che si è preferito memorizzare, come ad esempio *Island.mothernature* e *Island.ownership*
- **I colori come enum**: per quanto esista già un'apposita enum propria di *java.awt*, apprezziamo l'idea di mantenere una propria enumerazione di colori, in particolare relativamente alla facilità di creare proprie funzionalità di gestione e per mantenere limitato il numero di colori, memorizzando i soli necessari ai fini del gioco
- **La logica del turno** [CurrentTurnState] può risultare una strategia utile per concentrare la logica del turno in un'apposita classe evitando

di spalmarla nel controller in quanto (i) quest'ultimo risulterà semplificato e (ii) il flusso di esecuzione è facilmente individuabile (e di conseguenza debuggabile)

2 Lati negativi

- **La mancata presenza di interfacce**: alcune funzionalità vengono riprese in diverse classi (*Pouch, Island e Cloud*, per esempio). Per evitare una ridondanza nella scrittura del codice, si potrebbe pensare di inserire delle interfacce o delle classi astratte

- **Le carte personaggio**: nonostante sia una buona idea quella di usare strategy per le carte personaggio, si potrebbe pensare ad una soluzione più efficiente, come ad esempio una generalizzazione delle sottoclassi (in modo tale da non doverne gestire 8 molto simili). Si potrebbero dividere i Character in base al loro comportamento, evitando così di creare molteplici classi simili. Ciò potrebbe permettere, inoltre, una maggior facilità di implementazione di ulteriori carte in futuro

- **L'utilizzo di poche classi**: per quanto sicuramente un numero ridotto di classi possa facilitare la gestione della parte grafica e di rete, questo approccio potrebbe diventare problematico nel momento in cui si volessero aggiungere funzionalità. Basti pensare alla gestione delle aree scolastiche (dining-room, entrance) come array: molto C-like, ma potrebbe perdere in scalabilità risultando troppo terra terra (potenzialmente si è persa l'occasione per rendere questi array sottoclassi di una classe atta a gestire gli studenti). Utilizzando gli array, il rischio potrebbe essere quello di centralizzare troppa logica su *School*.

  Inoltre, appare strano come questa scelta sia stata fatta per le sale ma non per le nuvole, che dunque potevano tranquillamente essere gestite come Student[] nella Board. Analogamente, una scelta poco chiara è data dal fatto di istanziare una classe *Student* che di fatto è un solo wrapper di *Color* ma di non farlo nel caso di *Tower*, nonostante lo scopo fosse il medesimo.

  Infine, troviamo scomoda la gestione dei prof: essendo un array di boolean, ipotizziamo che verrà determinata la presenza o assenza di un professore in base al valore asserito in un dato indice; indici che però

non sono noti e andranno tenuti a mente "manualmente". Si può pensare, qualora fosse effettivamente questo l'approccio, di memorizzare con delle costanti intere questi indici, per rendere il codice più leggibile (nel diagramma UML, non si fa invece cenno a questi attributi)

3 Confronto tra le architetture

Abbiamo particolarmente apprezzato

- l'introduzione di classi del modello che non rappresentano concetti concreti ma sono più di utility, come dichiarato in *Le classi di gestione*: noi ad ora abbiamo preferito distribuire la logica sulle singole entità, ma terremo conto di questo approccio

- L'assenza di alcune classi (come ad esempio Tower o Professor). Effettivamente, lavorando sul codice, ci siamo resi conto di come classi quali *Student*, *Professor* e *Tower* (ma anche la nostra *LockCard*) possano essere considerate superflue e come una loro assenza potrebbe rendere il codice più facilmente gestibile, nonostante diverrebbe meno object-oriented