Peer-Review 1: UML

Lorenzo Aicardi, Federico Arcelaschi, Giovanni Arriciati
Gruppo AM01

5 aprile 2022

Valutazione del diagramma UML delle classi del gruppo AM31.

1 Lati positivi

- suddivisione delle fasi di gioco utilizzando una enumerazione;
- raggruppamento delle costanti in un'unica enumerazione, in modo tale che siano facilmente modificabili a livello di codice attraverso una singola riga;
- utilizzo di un'interfaccia per generalizzare i personaggi;
- la suddivisione degli elementi di gioco in classi è stata fatta, secondo la nostra opinione, in maniera corretta.

2 Lati negativi

- non vi è differenziazione sulla classe `Player` tra modalità di gioco base o esperto: tale classe, infatti, contiene metodi e attributi per la gestione delle monete, che sono quindi presenti anche in caso di partita senza "regole esperto";
- tralasciando l'errore di inserire degli attributi non costanti nell'interfaccia character, l'idea di impiegare un'interfaccia per generalizzare i personaggi è buona, ma nel diagramma UML non viene specificato come i personaggi vengano implementati;

- la struttura ad array per rappresentare gli studenti contenuti in un oggetto in alcuni casi ha poco senso: sarebbero sufficienti un colore o una lista.
  Esempi:
  - il metodo pickstudent della classe `Bag` potrebbe restituire soltanto il colore dello studente estratto;
  - l'attributo studentson`Cloud` potrebbe essere una lista/array di 3 o 4 colori.
- le classi principali (`Player`, `Board`, `Cloud`, ecc.) vengono utilizzate come semllici contenitori di dati, privi di una logica interna che permetta di utilizzare e/o modificare i dati stessi.
  L'intera logica, infatti, viene concentrata nella classe game: questo è un errore, non solo dal punto di vista del paradigma di programmazione ad oggetti, ma impedisce anche la modifica di attributi di una classe da parte di una classe esterna.
  Esempi:
  - le classi `Player` e `Board` contengono alcuni metodi getter, che permettono di mostrare all'esterno una copia degli attributi privati della classe; mancano però dei metodi che permettano, seguendo la logica di gioco, di modificare gli attributi stessi da parte di classi esterne;
  - la classe `Cloud` contiene metodi semplici per la modifica degli attributi, ma questi non seguono la logica di gioco: più metodi base potrebbero essere accorpati in metodi più complessi che coincidono con mosse/operazioni di gioco.

3 Confronto tra le architetture

Entrambe le architetture centralizzano istanze di tutte le classi in una unica, la quale permette al controller di interagire con il model.
Riteniamo tuttavia migliore la nostra suddivisione della logica applicativa nelle varie classi, che non necessita di esporre attributi privati tramite metodi getter.
Nella nostra implementazione sono inoltre presenti delle sottoclassi e dei

decorator, che permettono di separare le classi utili alla modalitá di gioco "base" da quelle riservate alla modalitá "esperto".
Visti i lati positivi riscontrati, riteniamo utile implementare nell'architettura da noi utilizzata i seguenti elementi:

- suddivisione delle fasi di gioco nella nostra classe di gestione dei turni, mediante l'impiego di una enumerazione;
- raggruppamento delle costanti in un'unica enumerazione.