Peer-Review 1: UML

Beghetto Pietro, de Donato Simone, Dimaglie Gregorio
Gruppo **AM46**

Valutazione del diagramma UML delle classi del gruppo **AM19**.

Lati positivi

- **Gestione di diverse tipologie di partite**: è intelligente ed efficiente la scelta di usare il pattern Decorator per la creazione di una partita in modalità esperto. Per distinguere fra partite con due o tre giocatori, invece, è stata implementata la classe astratta `AbstractMatch` (che implementa l'interfaccia `Match`) con due classi concrete.
- **Interfaccia `MoveStudent`**: avere un'interfaccia che accomuni tutti i "contenitori di studenti", cioè tutte le classi che acquistano o perdono studenti, è secondo noi un'ottima pratica che tutti i gruppi dovrebbero seguire e che molto probabilmente hanno seguito.
- **Strategy per i personaggi**: anche questa è un'ottima scelta di design, in quanto come le precedenti è generalizzata (può accomodare nuovi elementi di gioco in eventuali aggiornamenti successivi), intuitiva, "leggera" e sicura sia per i personaggi che per le classi, le quali possiedono le `Strategy`, ma non si devono preoccupare dell'implementazione.
- **Suddivisione di Controller** in `ActionPhase` e `PlanningPhase` comandati da `Turn`, più `MatchController` e `EndMatchHandler`. La divisione dei ruoli aumenta la testabilità e la chiarezza dell'information flow e delle chiamate a funzione. Inoltre è importante che il Controller agisca più come "traduttore" tra Model e View, che come agente attivo, ovvero "*you want to keep your controllers thin and your models fat*".
- **Centralizzazione di `Match` e alleggerimento del carico**: parzialmente correlato al punto precedente, ormai uno dei chiodi fissi del progetto visto che è stato ripetuto spesso dai tutor. Il design revisionato rispetta entrambe le linee guida: `Match` è composto solo da metodi chiamati direttamente dal Controller (cioè traduzioni dei comandi della View), i quali non conterranno al loro interno, per via del design delle classi sottostanti, logica o algoritmi implementativi, quanto piuttosto semplici chiamate a funzione perlopiù atomiche. Il Controller d'altro canto non interagirà (preferibilmente) con i componenti del gioco, ma solo con la sua facciata principale, il `Match`.

Lati negativi

- **Privatezza ed incapsulamento**: abbiamo notato che alcune funzioni pubbliche possono essere rese private con un po' di refactoring. Prendendo l'esempio di `Match`, si possono salvare i parametri di inizializzazione in `Match` (oppure passarli direttamente fra costruttori) evitando così di sovrascrivere `initializeClouds()` o `initializePlayers()` (per il numero di studenti all'entrata) a seconda del numero di giocatori.
  Inoltre si potrebbero evitare alcuni riferimenti (sintomo di accoppiamento fra classi), per esempio quello di `Match` a `MotherNature` (si può delegare a `IslandsManager` e pensiamo semplificherebbe anche le chiamate a funzione) o quello doppio fra `GameBoard` e `Player`, entrambi posseduti da `Match` (riteniamo migliore a questo punto una gerarchia `Match` -> `Player` -> `GameBoard`). Altro esempio, la delega della logica di controllo di fine partita da `EndMatchHandler` a `Match`.
  Un altro esempio è il fatto che i `Character` abbiano un riferimento a `Match` (e viceversa), che ci è stata indicata come pessima idea dai tutor.
- **Scelta di `PieceColor` per i parametri delle funzioni**: si tratta di passare `PieceColor` come parametro quando si spostano studenti da e per isole, nuvole, ecc. Non è del tutto sbagliato visto che uno studente vale l'altro fintantoché hanno lo stesso colore, tuttavia pensiamo che non sia un'esperienza ottimale dal lato del giocatore, che non può mai "prendere in mano" uno studente specifico, ma solo cliccarne uno di quel colore.
- **Chiamata di `Character`**: probabilmente è stata solo una svista, ma attualmente non c'è modo di chiamare i metodi di `Character` (`activateEffect()`) direttamente dalla View per mezzo del Controller. Pensiamo sarebbe preferibile in realtà che `Match` chiamasse/controllasse i `Character` e non viceversa, per avere migliore incapsulamento e gerarchia. Ma attualmente, `playCharacterCard()` di `ExpertMatch` non passa i parametri richiesti dai personaggi.
  Si può anche notare che alcuni `Character` possono essere raggruppati fra loro visto che hanno la stessa facciata (cambia solo l'implementazione del metodo principale, e di poco), mentre ad altri è stato dato un metodo di interfaccia `activateEffect(Island island, PieceColor c)` che sì riesce a raggruppare tutti i personaggi, ma lo fa a costo di rendere l'interfaccia insensata per personaggi che non usano neanche uno dei due parametri (mentre molti richiedono di conoscere il Player che li sta chiamando per agire).
  Infine, ci è stato fatto notare dai tutor che creare una classe per ogni personaggio è una cattiva pratica: pensiamo sia fattibile raggrupparle in base alla loro funzione (se modificano strategie, spostano studenti ecc.).

Confronto tra le architetture

- Non avevamo pensato all'idea, semplice ma sensata, di rendere `Bag` un singleton, visto che l'avevamo già pensato per Madre Natura.
- La suddivisione del Controller in varie sezioni, in modo da massimizzare semplicità d'uso, disaccoppiamento e incapsulamento, è un problema che il nostro gruppo ancora non ha risolto. Per esso possiamo trarre spunto dalla suddivisione presente in questo design.
- La classe `Game` del nostro gruppo (chiamata `Match` in questo design) risponde alla diversificazione delle modalità di gioco con una `GameFactory`, mentre `Match` ha due realizzazioni concrete per il numero di giocatori, e un decoratore per la difficoltà di gioco. Troviamo che entrambe le scelte possano essere considerate valide: la `GameFactory` trasforma i parametri raccolti dal controller in un'istanza ben precisa di `Game` in maniera univoca e pulita, mentre l'`ExpertMatchDecorator` è una struttura invisibile al resto del Model ma funzionale, ed estendibile per natura senza bisogno di modificare codice esistente.
- Nonostante ci siano alcune criticità riportate nel paragrafo precedente, dare un'occhiata a questa versione del design dei `Character` ci ha chiarito molti dei dubbi che avevamo sul nostro stesso design, molto simile a questo. Ad esempio la funzione `setPieceColor()` nella strategia di influenza `NoColorInfluence`, ci ha mostrato un modo di risolvere il problema di modellare l'unica carta personaggio, fra le nostre sei che usano il pattern Strategy, che richiedeva esplicitamente input aggiuntivo da parte dell'utente oltre alla semplice selezione del personaggio da usare.

Grazie per l'attenzione.