Peer-Review 1: UML

<Chiara Auriemma>, <Francesco Benelle>, <Giacomo Ballabio>, <Alberto Cavallotti>

Gruppo <AM04>

Valutazione del diagramma UML delle classi del gruppo <AM13>.

Lati positivi

Il principale aspetto positivo è il raggruppamento delle carte obiettivo comuni in classi accumunate da un modo simile di fare il check dei punteggi, si evita cosi ridondanza e anche il testing diventa più veloce ed efficace dal momento che si hanno meno classi da osservare. Lo stesso ragionamento si applica alle carte obiettivo personali, a loro volta raggruppate in una sola classe.
Inoltre, riteniamo che sia una buona idea aver creato la classe CommonGoal che si occupa della gestione dei punteggi delle carte comuni. Avete così alleggerito il carico di Player. Consigliamo di sfruttare la classe anche per salvarsi i punteggi legati alle carte obiettivo personali.

Lati negativi

La classe Game, indicata come il controller del pattern MVC, non svolge effettivamente le funzionalità del Controller, ma contiene metodi e attributi atti alla gestione della partita, quindi parte del Model. La classe Game è quindi utile per la gestione della partita, ma manca una classe che effettivamente agisca come un Controller, ovvero che faccia da tramite tra model e input che arrivano dalla View.

La classe Game memorizza solo il “currentPlayer”, non viene salvata una lista di giocatori o simili, utili per la gestione del sistema a turni.

Da ciò che si comprende nell'UML, le classi collegate all'interfaccia AbleToGetPoints svolgono autonomamente tutte le funzionalità (come riportato nel file di spiegazione inviato dal gruppo revisionato: "*AbleToGetPoints” è un’interfaccia implementata dalle classi che possono dare punteggio. In realtà questo viene calcolato, per la sedia dell’ultimo giocatore, con un semplice controllo di un attributo booleano*"), dunque è superfluo aggiunge l’interfaccia in questione.

Quando un utente entra nell'applicazione, sembra che entri direttamente in una partita, manca una Lobby o qualcosa di simile che consenta di scegliere se creare una partita, entrare in una esistente oppure recuperare un vecchio salvataggio.

La classe UsableCells serve solamente a distinguere i casi che possono capitare all'interno del costruttore già dichiarato in Board, quindi può essere evitata spostando il contenuto del costruttore di UsableCells direttamente in Board.

La classe Item contiene, oltre al colore, solamente l’attributo che indica qual è il numero “seriale” della tessera. Non ci sono né metodi né attributi che tengono conto di quante delle 132 tessere iniziali sono già state estratte, nè che tengono conto di quante tessere di ogni colore possono essere estratte dal sacchetto.

Confronto tra le architetture

Ci sono sostanziali differenze nel modo in cui vengono utilizzate le classi (metodi e attributi) e anche la spartizione dei compiti tra le classi è molto diversa. Entrando più nel dettaglio, noi utilizziamo il concetto di ereditarietà per la Board: a seconda del numero di giocatori istanziamo direttamente il tabellone adeguato alla partita. Il gruppo AM13 utilizza la classe “UsableCells” per inizializzare il tabellone. Potrebbe essere interessante implementare la loro idea, ma sarebbe più sensato utilizzare direttamente un file Json al posto di creare un’altra classe che, di fatto, non svolge compiti (non possiede metodi).
Nel nostro UML, la classe chiamata Game si occupa del calcolo dei punteggi, delegando alle singole classi solo l’operazione di check. Nel loro design tutte le classi che implementano l’interfaccia “AbleToGetPoints” sono in grado di calcolarsi i propri punteggi. In questo caso entrambe le possibilità ci sembrano valide.
Il principale punto di contatto è la gestione delle carte, entrambi gli UML raggruppano le carte obiettivo comuni e utilizzano una sola classe PersonalCard che verrà inizializzata a seconda del bisogno.