# Peer-Review 1: UML

Andrea Tarabotto, Nicola Tummolo, Gabriel Voss, Francesco Zuliani
Gruppo GC36

3 aprile 2024

Valutazione del diagramma UML delle classi del gruppo GC26.

POLITECNICO
MILANO 1863

# 1 Lati positivi

In generale l'UML presentato applica i design patterns e le strutture studiate nei vari corsi, ottimizzando bene e attuando una buona compartimentazione del codice. Di seguito elencati i punti più apprezzati dal gruppo:

- Due classi separate *Game* e *Common_ Table*:
  un'ottima idea di implementazione che separa le funzioni logiche di contenimento delle informazioni generali della partita e di contenimento delle informazioni di più basso livello inerenti agli oggetti comuni come i mazzi e le carte scoperte. Evita la creazione di una singola classe Game molto complicata che dovrebbe unificare i due tipi di funzioni

- Impostazione del flusso di gioco come FSA:
  stratagemma intelligente, lineare e modulare che permette di controllare il flusso di gioco. Facilita non solo la comprensione del codice, ma anche un'eventuale aggiunta di fasi di gioco diverse. Idea originale e degna di lodi

- La classe *Side*:
  la presenza di questa classe aumenta la modularità del codice e permette l'eventuale aggiunta di nuove carte che non seguono la logica "standard" delle carte già esistenti

- La classe *Message*:
  l'inclusione di alcune classi che tengano traccia dei messaggi è fondamentale nel modello dell'applicazione, averne già tenuto conto è un passo avanti notevole in una visione oltre l'orizzonte del percorso di sviluppo del software

- I colori delle pedine:
  può sembrare una banalità ma tenere traccia dei colori delle pedine nel model permette di sviluppare a pieno le funzionalità aggiuntive come la persistenza

## 2 Lati negativi

Nonostante la logica sia magistralmente sviluppata abbiamo notato i seguenti lati negativi che esponiamo al solo scopo di fornire una differente visione che speriamo possa portare al miglioramento di un lavoro già ottimo di per sè:

- Mancanza di metodi "interfaccia" in *Game*:
  per evitare di passare dei riferimenti è utile esporre nella classe *Game* dei metodi che permettano al *controller* di effettuare le azioni senza avere i riferimenti ad ogni altro oggetto diverso da *Game*, potrebbe essere un modo per sfruttare al meglio l'incapsulamento offerto da Java

- La classe *Side*:
  da questa classe ereditano tutti i tipi di facce delle carte, risulta però che ci siano dei metodi che sono presenti in *Side* e che poi non necessitano di esistere in alcune classi "concrete". Si suggerisce qui di implementare come in un "albero genealogico" classi astratte che implementano interfacce così da settorializzare e far ereditare solo i metodi e gli attributi strettamente necessari

- La classe *Gold_Card_Front*:
  essa è estesa da quattro classi counter, si potrebbe ridurle a due unendo sotto una sola classse quelle che contano i punti ottenuti grazie agli oggetti

- Le classi *Deck* e *Hand*:
  essendovi solo un tipo di *Deck*, e avendo entrambe le classi sopracitate riferimenti al tipo astratto *Card*, potrebbe avvenire che nel *goldDeck* si trovino carte risorsa oppure che nella *Hand* si trovino carte obiettivo. Ovviamente con un'opportuna programmazione della parte esecutiva questi spiacevoli scenari si possono evitare, ma per avere la certezza di non poter effettuare nessuna operazione di questo tipo suggeriamo di creare tipi (o sotto-tipi) di *Deck* analoghi a quelli delle carte e di permettere di entrare nella mano solo alle carte dei tipi che possono essere giocati (ad esempio mettendo una sovra-classe di *playableCard*)

In ultima istanza si suggerisce di esplicitare all'interno delle classi gli attributi, non si tratta di un lato negativo ma semplicemente di una piccolezza per una visualizzazione più agevole dell'UML.

# 3 Confronto tra le architetture

Un grandissimo punto di forza rispetto all'architettura da noi implementata è la separazione tra *Game* e *Common_ Table* i cui lati positivi sono già stati trattati in par.1; nel nostro modello ciò non avviene e la classe *GameState* assolve, in maniera molto più articolata i compiti delle due classi. L'implementazione proposta dal GC26 è nettamente più lineare e meno complicata.
Altro importante metodo che abbiamo particolarmente apprezzato è stata la modularià della classe *Side*, questo potrebbe permettere, in fasi avanzate dello sviluppo, come già detto, di uscire ulteriormente dalla logica di gioco. La nostra implementazione è stata ben più aderente alla logica del gioco e per questo meno modulare e adattabile a eventuali cambi di regole. La presenza di questa classe potrebbe essere introdotta anche nel nostro modello.
Il susseguirsi dei turni di gioco come stati di una FSA, da noi non previsto, potrebbe ulteriormente incrementare modularità e persistenza della nostra implementazione. Infine un dettaglio che non abbiamo implementato è stato il colore delle pedine, sarà necessario implementarlo per evitare problemi legati alla disconnessione.