# Peer-Review 1: UML

Gruppo AM08: Francesco Caracciolo, Antonino Ciancimino, Gianluca Di Paola, Davide Greco.

Valutazione del class diagram UML del gruppo AM17.

## Lati positivi

### Generali

- Nel modello è stata prevista a livello concettuale, tramite la classe `GameResources`, la presenza di carte che esistono nel gioco indipendentemente dall'istanza di `GameModel` che le utilizza.

La differenza è evidenziata dal fatto che `GameResources` presenta liste di `PlaceableCard`, mentre il `GameModel` presenta dei deck di carte separati

(che supponiamo prendere le carta proprio da quelle List).

### GameController

- `GameController` rende molto chiaro quali metodi sono mostrati all'esterno e delinea chiaramente le interazioni dell'utente con il gioco.

### ObjectiveCard

- La suddivisione di `ObjectiveCard` in tutte le sue varianti permette di semplificare il funzionamento del codice da un punto di vista algoritmico.

## Lati negativi

### Generici

- In diverse classi alcuni attributi privati non hanno il corrispettivo metodo getter, sebbene servirebbe (e.g. `Token`).

- Vari metodi risultano privi dei parametri che necessiterebbero per funzionare, (e.g. `GameController.setCommonObjective`, `GameController.setObjectiveChoice`).

### ObjectiveCard

- La scelta di creare una sottoclasse per ogni possibile obiettivo risulta troppo frammentaria (infatti sono presenti otto classi con differenze marginali).

Inoltre, troviamo la soluzione poco scalabile (l'inserimento di nuove carte obiettivo implicherebbe la creazione di classi ex novo).

- L'attributo `points` è presente in tutte le sottoclassi di `ObjectiveCard`, e quindi potrebbe essere spostato nella sopraclasse.

### CardCorner

- La gerarchia che ha come sopraclasse `CardCorner` è evitabile, in quanto le sue sottoclassi non presentano sostanziali differenze (tranne per `ResourceCorner`).

- Per evitare un utilizzo eccessivo di `instanceof`, suggeriamo l'utilizzo delle `Enumerations`(e.g. Enum `CornerType`: `Hidden`, `Empty`, ...).

### PlaceableCard

- Supponiamo che esista il metodo `getPermanentResources(...)` in `Back`, solo a fine esemplificativo.

Gli attributi `front` e `back` di `PlaceableCard` sono di tipo statico `Side` dunque non è possibile accedere ai metodi unici delle sottoclassi senza effettuare.

un casting (in Back non sarebbe possibile, invocare getPermanentResources(...)). Una possibile soluzione a questo problema è rendere gli attributi di `PlaceableCard`

(e relativi getter) `front` e `back` rispettivamente di tipo statico `Front` e `Back`.

- Ci sentiamo di condividere un'osservazione: la classe `PlaceableCard` sembra abbia un po' mischiato in sé la struttura

di una carta generica e di una carta piazzata; come denotato dalla presenza degli attributi `front` e `back` (appartenenti al concetto di carta generica)

e `currSide` (appartenente al concetto di carta piazzata).

- In `GoldenCard` non abbiamo identificato con quale meccanismo vengono calcolati i punti in base al numero di angoli coperti.

### GameModel

- I metodi `drawResourceCard` e `drawGoldenCard `di `GameController` non consentono di specificare se la carta va pescata dal deck o dalle carte scoperte.

- `ObjectiveCardsDeck`, `UsableCardsDeck` e `StarterCardsDeck` hanno diversi metodi in comune, ma non appartengono alla stessa gerarchia.

### PlayerModel

- La scelta di utilizzare la classe `Token` per rappresentare il colore del player (o la sua posizione, non risulta molto chiaro dal solo UML), non ci è sembrata una soluzione efficace

in quanto tale classe ha solo un attributo, che potrebbe essere messo all'interno di `Player`.

- L'attributo `objectiveToChoose` risulta non necessario, in quanto esiste già l'attributo `secretObjective`.

### GameController

- La scelta di posizionare `StarterCard` prima di far scegliere il lato non ci è sembrata efficace: una possibile alternativa è di farlo scegliere prima di piazzarla.

## Confronto tra le architetture

A differenza dalla soluzione adottata dal gruppo AM17, noi abbiamo preferito optare per una generalizzazione maggiore, al fine di aderire più strettamente all'OOP.

- Nella nostra architettura non abbiamo fatto utilizzo di ID per le varie entità del gioco, tuttavia riteniamo probabile che l'introduzione di questi possa tornare utile nelle fasi successive dello sviluppo. Soprattutto nella comunicazione client-server, potrà tornare utile al fine di evitare il trasferimento di interi oggetti durante ogni transazione (e.g. ogni player deve comunicare al server quale carta ha piazzato inviando l'intero oggetto carta, oppure, specificatamente nella nostra architettura, inviando l'intero oggetto `Board`); tale scambio di oggetti potrebbe essere fatto durante la sola prima interazione fra client e server.

- Per maggiore scalabilità e chiarezza, abbiamo utilizzato in maniera più intensiva le `Enumeration`, sfruttando la possibilità di generare subset tramite metodi statici.

Ad esempio, l'enumeration `Symbol` contiene tutti i simboli presenti nel gioco, e definisce dei metodi per ottenere i subset validi per ciascuna occasione

- Abbiamo implementato una gestione della partita di gioco attraverso un modello a macchina a stati finiti (design pattern `State`), al fine di distinguere nettamente le fasi di gioco.

- Nella nostra architettura gli obiettivi richiedono un `Requirement` (utilizzato anche per il piazzamento delle carte oro), che si suddivide in due sottoclassi:

`QuantityRequirement` per i requisiti relativi alle risorse e `PositionRequirement` per i requisiti posizionali, evitando così di creare una classe per ogni posizione possibile.

- Abbiamo deciso di esporre all'esterno del model le classi `Match` e `Player`. In questo modo i Controller relativi ai giocatori interagiscono con il solo oggetto `Player`,

mentre il gioco viene gestito in `Match`.

- Il concetto di "carta piazzata" e "carta giocabile" sono da noi considerati diversi e vengono pertanto divisi: abbiamo creato una classe `Card`,

da cui ereditano `InitialCard`, `GoldCard` e `ResourceCard`, e una classe `PlacedCard` composta dal turno in cui viene piazzata e dalla carta che rappresenta.

In questo modo le istanze di `Card` non dipendono da un particolare `Match`, mentre le `PlacedCard` fungono da wrapper di `Card` strettamente dipendenti dal `Match`.

- Per la gestione dei deck abbiamo utilizzato un generic: in questo modo si evita duplicazione di codice, in quanto i mazzi hanno tutti lo stesso funzionamento.