DESCRIZIONE UML MODEL

MEMBRI DEL GRUPPO

- Polizzi Chiara
- Raimondi Francesco
- Rubagotti Andrea
- Santagata Maria Concetta (REFERENTE)

FUNZIONALITÀ AGGIUNTIVE

Sono presenti classi anche per le 2 funzionalità aggiuntive che abbiamo deciso di implementare (chat, partite multiple).

DESCRIZIONE UML

N.B: L'UML è di alto livello, di conseguenza è privo di tutti i metodi getter e setter di ogni classe in esso presente (li consideriamo sottintesi).

La classe **Game** si occupa di gestire l'intera partita dall'inizio (distribuzione di tutte le carte iniziali, gestione delle **Pawn** (pedine) del singolo giocatore, ecc.) alla fine (controllo del vincitore con relativi conteggi dei punti). Ogni Game è contraddistinto da uno stato di gioco (**GameState**), il quale può assumere diversi valori in base alla fase in cui lo stesso si trova: STARTED, ENDING(settato appena un giocatore raggiunge 20 punti o entrambi i mazzi di pesca terminano), ENDED, WAITING_FOR_START). Il Game, inoltre, possiede un attributo di tipo lastEvent, il quale sarà spiegato meglio nel file di descrizione dell'UML di rete. In breve, questo attributo contiene lo stato dell'ultimo evento avvenuto durante il gioco (ad es. OK se operazione andata a buon fine, NOT_YOUR_TURN se il giocatore non in turno sta provando a fare qualche azione, ecc…). La scelta di posizionare questo attributo nella classe Game risiede nel fatto che tutti i giocatori dovranno essere successivamente informati (attraverso il pattern Observer-Observable) dello stato dell'ultima azione compiuta da un certo giocatore. Ci sarà quindi una notify nel metodo setLastEvent().

La classe Game contiene una lista di **Chat** (che possono essere da Player a Player, oppure un'unica chat comprendente tutti i membri della partita). Ogni chat è composta a sua volta da una lista di **ChatMessage**, che contengono in una stringa il testo scambiato dai giocatori attraverso la piattaforma di gioco.

Ogni Game è composto anche da una lista formata da 2 a 4 **Player**. Questi ultimi sono implementati in modo da avere funzionalità di pesca della carta (dai mazzi o dalle carte già scoperte sul tavolo) e funzionalità per posizionare (giocare) una carta in una specifica posizione sulla Board. Inoltre ogni Player è contraddistinto da un suo stato (**PlayerState**) che può essere: IS_PLAYING, IS_WAITING oppure IS_DISCONNECTED e da una propria pedina (**Pawn**).

Ogni Player è legato alla propria **Board** la quale è composta principalmente da una tabella (matrice) che contiene tutte le carte (Playable Card) presenti sul tavolo del giocatore ad essa relativo. La dimensione della matrice viene calcolata dinamicamente in base al numero di giocatori che formano la partita (da 2 a 4) e al numero di carte giocabili, che in questo gioco sono sempre 80. Dentro la board ci sono due set di **Coordinates** (formate da int x, int y) che permettono di stabilire dove il giocatore possa effettivamente giocare una carta, la mappa di playedCards, invece, ci consente nella fase finale della partita di calcolare i punti ottenuti completando gli obiettivi "pattern" delle carte obiettivo (in modo più rapido ed efficace rispetto alla matrice "table").

Per quanto concerne la gestione delle carte, la classe **Card** è una classe astratta che contiene un ID e abbiamo da questa due classi figlie: Playable e Objective.

- La classe **Playable Card** rappresenta l'insieme delle carte oro, risorsa e base (da qui il grande numero di attributi che abbiamo preferito tenere distinti per motivi di leggibilità). Le Playable Cards (risorsa, oro, base) sono raggruppate in diverse istanze di **Deck** (mazzi) che sono degli Stack (quindi offrono metodi di libreria). Dalla classe Deck, ereditano sia **PlayableDeck**, sia ObjectiveDeck.
- Le **Objective Card** (che sono raggruppate in **ObjectiveDeck**) sono caratterizzate da due metodi fondamentali: addPatternPointsToPlayer() e addNumberPointsToPlayer(). In base alla tipologia di carta obiettivo (se essa attribuisce punti per un pattern o per delle risorse presenti sul tavolo) verrà chiamata la funzione opportuna.

I metodi goldDeck(), baseDeck(), resourceDeck() (in PlayableDeck) e objectiveDeck() (in ObjectiveDeck) sono i metodi utilizzati per creare i mazzi leggendo dai files JSON.

Per la gestione degli angoli delle carte (e delle risorse al centro) è stata usata un'enumerazione: **AngleType**. Questa contiene tutte le tipologie di angoli possibili (un angolo può anche essere vuoto, o può anche non essere presente).