DESCRIZIONE UML RETE

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).

- **RMI**

  CLIENT ⇒ SERVER

  Per quanto riguarda la parte di RMI a lato server si ha una classe RMIServer che contiene l'istanza di ServerController mediante il quale si vanno a chiamare i metodi della classe stessa. Questi metodi sono quelli che permettono al giocatore di effettuare le operazioni pre-partita (aggiungersi ad una lobby, creare una lobby, scelta del nickname). La classe RMIClient, infatti, chiama i metodi del RMIServer, che a sua volta chiama i metodi del ServerController. Successivamente, attraverso queste, viene restituito all'RMIClient il GameController, in modo che il Client possa chiamare direttamente i metodi dell'oggetto remoto GameController per eseguire qualunque azione durante la partita (in modo da non intasare ServerRMI).

  SERVER ⇒ CLIENT

  La comunicazione in questa direzione è fondamentale per quanto riguarda gli update delle varie view (clients). Ogni volta che avviene la modifica del model (gli oggetti Observable sono Game e Player), verrà fatta una notify verso gli Observer iscritti alle classi stesse da parte della classe Observable. notifyObservers() in Observable chiama la funzione update() sugli Observer (per aggiornarli). Per RMI abbiamo come osservatori vari WrappedObserver ("wrappano" un oggetto remoto RMIClient), uno per ogni Client connesso con RMI. La funzione update() negli osservatori viene implementata attraverso uno switch case che, in base agli Event contenuti nei messaggi di sistema inoltrati dal model, chiama la funzione sul RMIClient (ad esempio se l'evento è "UPDATED_BOARD", allora viene chiamata la funzione "updateBoard" su RMIClient settando direttamente i nuovi valori degli attributi).

- **SOCKET**

  CLIENT ⇒ SERVER

  Per il protocollo TCP abbiamo lato server una classe ServerSCK che si occupa di accettare la connessione dei vari ClientSCK, generando per ognuno di essi un ClientHandlerThread in modo da gestirli in maniera adeguata. Il ClientSCK invierà

SCKMessage (un'estensione di Message creata per distinguerla dai messaggi di sistema interni) al ClientHandlerThread che contiene i riferimenti a ServerController e, successivamente all'accesso in una partita, allo specifico GameController. ClientHandlerThread riuscirà a chiamare i metodi dei controller (opportunamente sincronizzati per impedire "data-race") effettuando l'azione richiesta dal ClientSCK.

SERVER ⇒ CLIENT

Il percorso contrario, come per RMI, sfrutta il pattern Observer/Observable. Per TCP i nostri osservatori saranno i ClientHandlerThread che verranno registrati alla partita dal GameController stesso (vale lo stesso per RMI). Tramite una funzione update() che conterrà uno switch case, il ClientHandlerThread invierà al ClientSCK vari SCKMessage contenenti gli oggetti e l'Event che indica gli attributi da aggiornare.