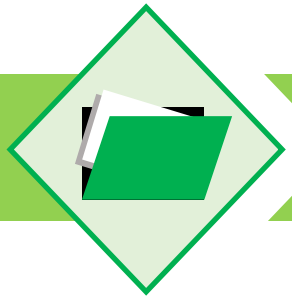
Documentazione Peer Review #2



Documentazione relativa al file Unified Modeling Language (UML) del **Gruppo 04**:
*"**ing-sw-23-saccani-spangaro-sanvito-pedersoli**"*

Abbiamo scelto di implementare sia la comunicazione tramite **Socket** sia tramite **RMI**.
I Messaggi inviati si riferiscono sia al MainController sia al GameController:

- ***MainController***: Gestisce tutte le partite presenti sul server; perciò la creazione e l'entrata di una partita. (Funzionalità avanzata: *"Partite Multiple"*)
- ***GameController***: Gestisce una singola partita

Descrizione sintetica del Protocollo di Comunicazione:

- Lista dei Messaggi che possono essere inviati al **MainController**:
  Implementa interfaccia MainControllerInterface
  - *createGame*
  - *joinFirstAvailableGame*
  - *joinGame*
  - *reconnect*

- Lista dei Messaggi che possono essere inviati al **GameController**:
  Implementa interfaccia GameControllerInterface
  - *playerIsReadyToStart*
  - *grabTileFromPlayground*
  - *positionTileOnShelf*
  - *isThisMyTurn*
  - *setConnectionStatus*
  - *heartbeat*
  - *sentMessage*

Questi sono tutti i messaggi che i clients possono inviare al server sia tramite RMI sia tramite Socket.
In ogni richiesta è sempre presente la *String nick* del giocatore che ha inviato il messaggio (per la lista completa dei parametri di ogni messaggio riferirsi all'UML)

Definizione Protocollo RMI

La classe RMIClient effettua il lookup del registry ottenendo una MainControllerInterface sulla quale effettua le tutte le richieste per entrare a far parte di una partita (crea un game, entra nel primo game disponibile, richiesta di riconnessione, etc)
Le richieste verso il MainController sono sempre accompagnate dal parametro GameListener, il quale, è un oggetto remoto e rappresenta l'oggetto Client sul quale il server andrà a notificare ogni modifica del Model (GameListener è un interfaccia utilizzata per implementare il Pattern Listener)

Ad ogni richiesta verso il MainController, viene restituito dal server un nuovo oggetto remoto di tipo GameControllerInterface. Se questo è diverso da null significa che il RMIClient è riuscito correttamente a collegarsi ad un Game e quindi potrà effettuare tutte le richieste eseguibili durante una partita (dire che è pronto a iniziare, prendere una tile dal playground, posizionare tile sulla propria shelf, etc.)

Definizione Protocollo Socket

Il ClientSocket funziona allo stesso modo del RMIClient con l'unica differenza che non è possibile invocare metodi su oggetti remoti. Si è deciso, di conseguenza, di creare dei messaggi specifici per ogni azione che il client vuole eseguire sul server.
Il Socket si divide in: ClientSocket, SocketWelcome e ClientHandler.

- ClientSocket: Classe utilizzata dal client per fare le richieste Socket
- SocketWelcome: Classe lato server che si occupa di accettare comunicazioni Socket e di inoltrarle al ClientHandler (Un SocketWelcome per tutto il server)
- ClientHandler: Si occupa di ricevere i messaggi (lato server) (in.readObject()) e di gestirli di conseguenza (Basic Rule: One Client One ClientHandler)

Per rendere il codice estremamente flessibile sono state implementate le seguenti classi, opportunamento Serializzate quando necessario:

- **MainControllerMessages (package)**
  Messaggi che il client invia verso il MainController
  - *SocketClientMessageCreateGame*
  - *SocketClientMessageJoinFirst*
  - *SocketClientMessageJoinGame*
  - *SocketClientMessageReconnect*

- **GameControllerMessages (package)**
  - *SocketClientMessageSetReady*
  - *SocketClientMessageGrabTileFromPlayground*
  - *SocketClientMessageNewChatMessage*
  - *SocketClientMessagePositionTileOnShelf*

Tutte le classi messaggi ereditano dalla classe Astratta SocketClientGenericMessage (Serializable) al cui interno sono presenti 2 metodi:

- *GameControllerInterface execute(GameListener lis, MainControllerInterface mainController)*
- *void execute(GameControllerInterface gameController)*

Le varie classi messaggio vanno a fare l'override del metodo execute corrispondente (a seconda se si tratti di un messaggio verso il MainController o verso il GameController) andando a definire quale metodo il server dovrà andare ad invocare.

Ad es:
Per la classe messaggio SocketClientMessageSetReady, questo andrá a fare Override del metodo *execute(GameControllerInterface gameController)* (visto che è un messaggio per il GameController) specificando al suo interno: *"gameController.playerIsReadyToStart(this.nick)"* cosí che il ClientHandler (lato server) a seguito della ricezione del messaggio generico SocketClientGenericMessage si limiterá semplicemente a fare: *msgRicevuto.execute(gameController)*

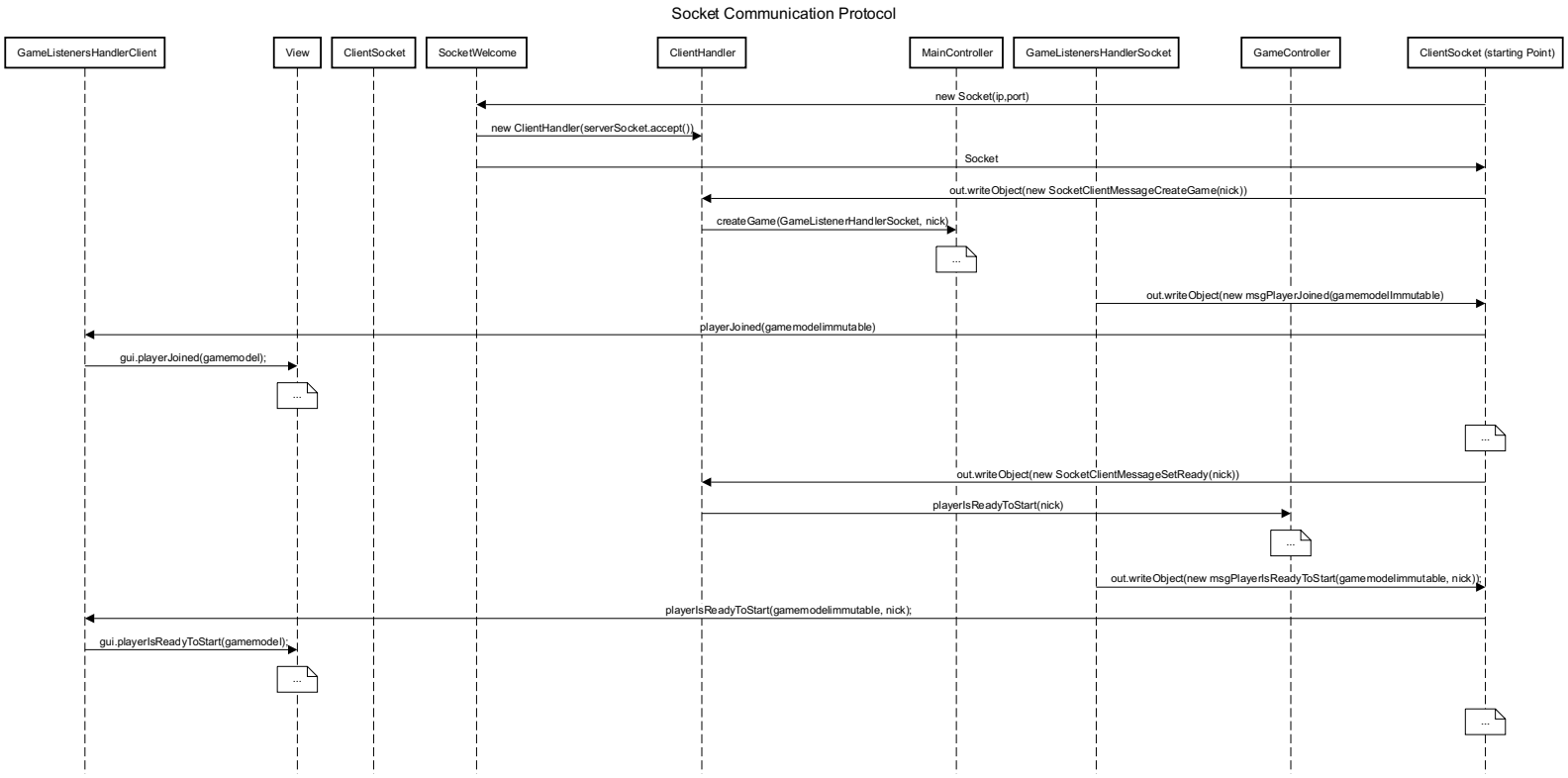
Questo schema utilizzato è un Pattern molto utilizzato in ambito di Socket Networking in quanto non vincola il programmatore a utilizzare gli istanceof per capire quale tipo di messaggio si è ricevuto.

Naturalmente, il ClientSocket invierà la classe messaggio corrispondente a seconda dell'operazione che si vuole svolgere (es: *"out.writeObject(new SocketClientMessageSetReady(nickname));"*)

Stessa identica cosa è stata adottata per i Messaggi scambiati dal Server al Client (a seguito di ogni modifica del model):

- **ServerToClientMessages (package)**
  - msgGameStarted
  - msgPlayerJoined
  - … (per l'elenco completo fare riferimento all'UML allegato)

Riportiamo di seguito, per completezza, un ***Sequence Diagram*** con degli esempi di interazione Client-Server con connessione Socket:



Per implementare la Funzionalitá avanzata *"Gestione delle collisioni lato client"*, sia Socket sia RMI inviano costantemente degli "heartbeat" per segnalare la loro presenza (circa ogni 0.5s).
Il Server memorizza l'ultimo heartbeat ricevuto da ogni client e, periodicamente (ogni circa 3s), verifica che l'ultimo heartbeat ricevuto da ogni client sia recente di almeno 3s.
Se non si riceve un heartbeat da piú di 3s si considera il Client come disconnesso modificando di conseguenza il Model.