Documentazione Prova Finale
“Lorenzo Il Magnifico”

Castelnuovo Carlo, Cerioli Alessandro, Tosi Matteo

20 Giugno 2017

Sommario

1 Introduzione 3

1.1 Obbiettivo e Specifiche 3

1.2 Tecnologie e Architettura 3

1.3 Struttura del Documento 4

2 Model 5

2.1 Il giocatore ed i componenti del gioco 5

2.2 UtilEffetto 6

2.3 Carta 6

2.4 Le Tessere Scomunica 7

2.5 SpazioAzione e Famigliare 7

2.6 La classe Partita 9

3 Server e Comunicazione 10

3.1 Server 12

3.2 Flusso di esecuzione 14

3.3 Comunicazione 15

4 Client 19

5 Grafica 20

5.1 I Listener 20

5.2 Aggiornamento e AggiornamentoInterfaccia 20

6 Test 21

6.1 GameTest 21

6.2 GiocatoreTest 21

6.3 PartitaTest 22

7 Futuri Sviluppi 22

1 Introduzione

La seguente sezione spiega brevemente le specifiche preposte per il progetto, le decisioni prese nella scelta di strumenti, librerie e altri componenti, e la struttura del seguente documento.

1.1 Obbiettivo e Specifiche

Il progetto consiste nello sviluppo di una versione software distribuita del gioco da tavolo *Lorenzo il Magnifico* (gioco con regole semplificate). Il sistema è composto da:

- **Server** singolo in grado di gestire più partite simultaneamente
- **Client** multipli (uno per giocatore) che possono partecipare ad una sola partita alla volta

L'**interfaccia grafica** è posta sui soli Client per poter interagire con il Server e monitorare l'avanzamento della partita.

Al fine di poter supportare più partite contemporaneamente, il server è stato progettato per:

- attendere richieste di login di nuovi giocatori
- attendere un time-out (in secondi) prima dell'avvio di una partita dopo che si raggiunge il minimo numero di giocatori (da un minimo 2 fino a massimo 4, dopo il quale la partita inizia automaticamente)

1.2 Tecnologie e Architettura

In questo paragrafo vengono elencate le motivazioni che hanno portato alla scelta di alcune tecnologie o strumenti, così come sono fatte alcune considerazioni sulle scelte effettuate nella struttura generale del progetto.

Architettura di Rete paradigma Client/Server nel quale il Server funge da gestore della logica del gioco e delle connessioni tra i singoli terminali che si limitano a fornire le mosse disponibili al giocatore.

Linguaggio di programmazione per lo sviluppo dell'interno progetto è stato utilizzato il seguente linguaggio: JavaSE (ver. 8)

Libreria grafica per quanto riguarda l'interfaccia grafica del Client, la specifica, permetteva di decidere tra JavaFX e Swing, poiché uno dei componenti del gruppo aveva tra le esperienze pregresse la conoscenza di una tra le due si è optato per: Swing

Server di seguito la lista dei requisiti tecnici per il lato server:

- implementa la comunicazione Client/Server sia via Socket sia via RMI
- supporto a partite in cui i giocatori utilizzano tecnologie diverse (Socket/RMI)

Client di seguito la lista dei requisiti tecnici per il lato client:

- supporto per la comunicazione Client/Server sia via Socket sia via RMI (all'avvio permette al giocatore di selezionare la tecnologia da utilizzare per l'intera sessione di comunicazione)
- supporta l'interfaccia sia testuale (CLI) che grafica (GUI) (all'avvio permette al giocatore di selezionare il tipo di interfaccia da utilizzare)

1.3 Struttura del Documento

Visto che la distinzione dei vari componenti dell'applicazione si è riflessa in buona parte nella suddivisione dei compiti all'interno del gruppo, verranno analizzati gli stessi da ogni punto di vista: "**model**" (logica del gioco, Server), "**network**" (comunicazioni, Client/Server), "**ui**" (interfaccia grafica, Client).

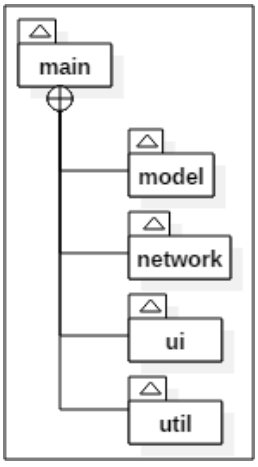


Figura 1: Struttura dei package

2 Model

In questa sezione vengono spiegate come sono state strutturate e implementate le classi che modellano le componenti e la logica del gioco (per le regole semplificate si rimanda al manuale “regole_del_gioco.pdf”).

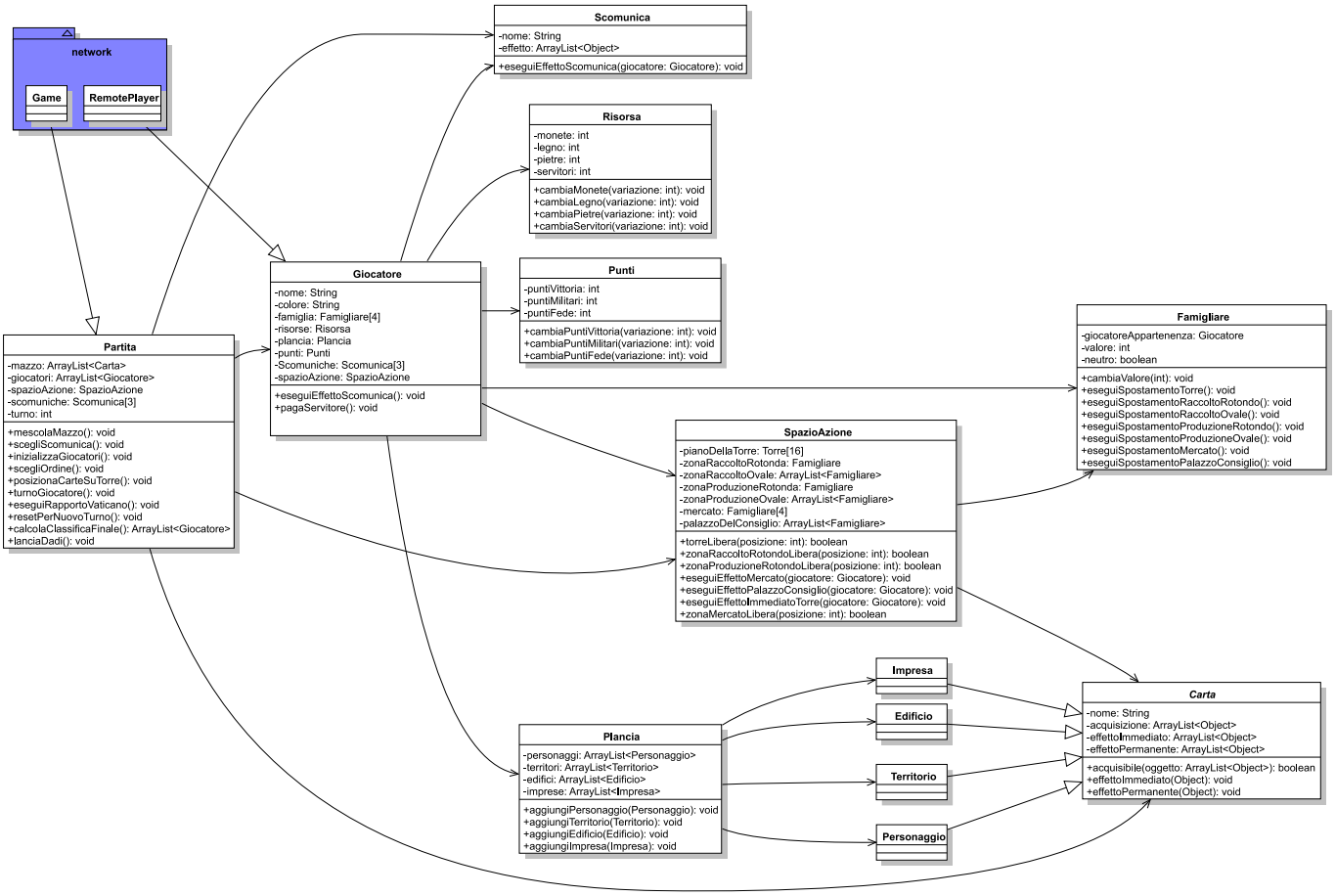


Figura 2: Diagramma delle classi del package model (vengono inserite anche le classi “Game” e “RemotePlayer” per mostrane le connessioni con il package “network”)

2.1 Il giocatore ed i componenti del gioco

I componenti del gioco, come le carte sviluppo ed i famigliari, sono strettamente correlati al singolo giocatore e svolgono una parte fondamentale nel gioco. Il giocatore è rappresentato dall'omonima classe `Giocatore`, dentro alla quale sono presenti i riferimenti alle scomuniche associate al giocatore, ai propri famigliari ed alla propria

plancia, dove sono raggruppate le carte in proprio possesso. Per rappresentare le carte sopra alla plancia sono utilizzati quattro `ArrayList`, uno per tipo di carta. I tipi delle carte sono espressi tramite classi con nomi diverso ma che ereditano tutte dalla classe `Carta`, dove sono presenti anche i principali metodi di gestione degli effetti delle carte e dei loro costi.

2.2 UtilEffetto

Per potere riutilizzare il codice associato a determinati effetti, è stata effettuata la scelta di dividere le carte dai propri effetti, implementando una apposita classe `UtilEffetto`, dove sono presenti tutti gli effetti delle carte sviluppo ed alcune tessere scomunica. Nella classe sopra citata, ad ogni effetto corrisponde un metodo identificato da un intero. Per il passaggio dei parametri a tali metodi, essendo essi variabili da effetto ad effetto, sono stati utilizzati degli `Array` della classe predefinita `Object`. In tal modo è possibile passare in una sola volta tutti i parametri, anche se di tipo differente. Il numero del metodo da attivare fa parte dei parametri appena presentati.

Figura 3: Funzionamento della classe “UtilEffetto”

2.3 Carta

La classe carta presenta al suo interno svariati `ArrayList` di significato differente:

- **`acquisizione`** rappresenta i costi della carta. E' stato utilizzato un Arraylist in quanto alcune carte presentano un costo doppio
- **`effettoImmediato`** rappresenta gli effetti atomici associati alla carta
- **`effettoPermanente`** rappresenta gli effetti atomici associati alla carta

Tali `ArrayList` sono utilizzati prevalentemente per la gestione della parte di logica del gioco.

Figura 4: Classe “Carta”

Ancora all'interno di carta è possibile trovare altri attributi di supporto alla parte di comunicazione del progetto, facenti riferimento a quante scelte spettano al giocatore nel pagamento di una carta o nell'attivazione di un effetto permanente e quali siano le scelte possibili.

2.4 Le Tessere Scomunica

Le tessere scomunica sono gestite in modo similare alle carte sviluppo. Le uniche differenze sono i metodi di controllo sulle azioni attivanti effetti di scomuniche.

2.5 SpazioAzione e Famigliare

Così come `Carta` e `UtilEffetto`, anche `SpazioAzione` e `Famigliare` sono strettamente correlati, in quanto `SpazioAzione` contiene tutte le informazioni relative al tabellone fisico del gioco, ad esclusione dei vari punti del giocatore, associati direttamente al giocatore stesso, e `Famigliare` rappresenta il famigliare fisico che spesso interagisce con

il tabellone. Tutti i metodi per lo spostamento del famigliare sono raggruppati in `Famigliare` e fanno tutti riferimento a `SpazioAzione`.

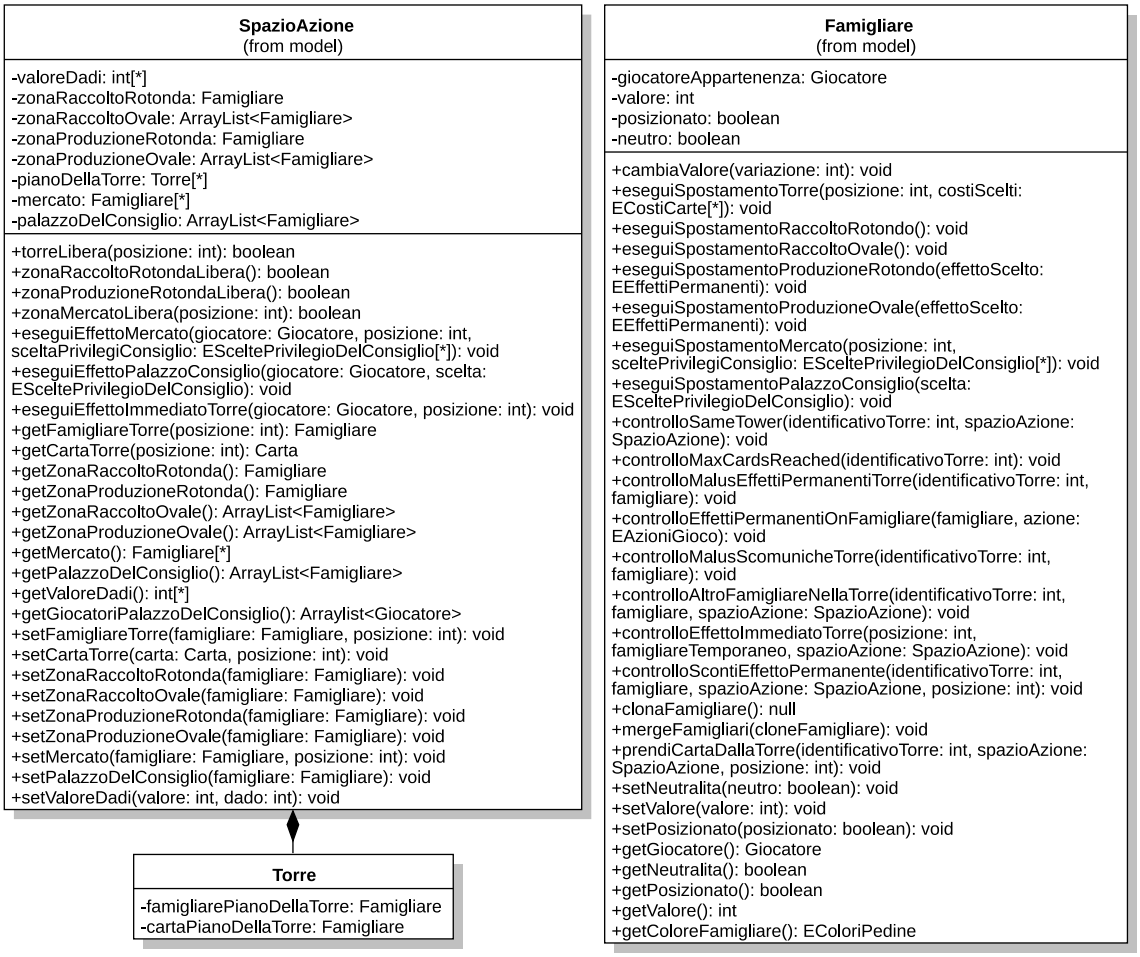


Figura 5: Classi “SpazioAzione” e “Famigliare”

Ogni famigliare ha all'interno un riferimento al proprio giocatore possessore che a sua volta ha un riferimento al tabellone in uso nel gioco. Di conseguenza, il tabellone è apertamente accessibile al famigliare, che può così mantenere i propri metodi relativi allo spostamento ed ai controlli applicati nel corso di quest’ultimo. Le variazioni temporanee da applicare come controllo prima di eseguire l’effettivo spostamento del famigliare sono applicate ad un clone del `Famigliare` stesso che, se i controlli vanno a buon fine, copierà i propri valori modificati all’interno del `Famigliare` originale. Da qui l’utilizzo di **`cloneFamigliare`** e **`MergeFamigliare`**, metodi che si occupano rispettivamente di creare il clone e distruggerlo copiando il contenuto all’interno del Famigliare d’origine.

A loro volta, gli spazi presenti sul tabellone sono riassunti in `SpazioAzione` tramite delle variabili che assumono valore **`null`** quando lo spazio non è occupato ed il valore del famigliare quando invece sono occupate. Nel caso di spazi con più pedine si utilizzano degli `ArrayList` di variabili `Famigliare`.

All'interno di `SpazioAzione` vi sono i principali metodi che assieme agli altri metodi contenuti in `Famigliare`, concorrono a gestire tutte le conseguenze dello spostamento di un `Famigliare` sul tabellone.

2.6 La classe Partita

La classe `Partita` è il cuore della parte logica: in esse sono contenute le scomuniche da usare durante la partita, il tabellone, l'ordine dei giocatori e i metodi gestori delle varie fasi di gioco. Tramite `Partita` si rende disponibile la gestione del gioco dalle parti di comunicazione e grafica, in quanto contiene i metodi chiave della logica del gioco. La classe `Game`, definita nel package network, eredita da questa classe.

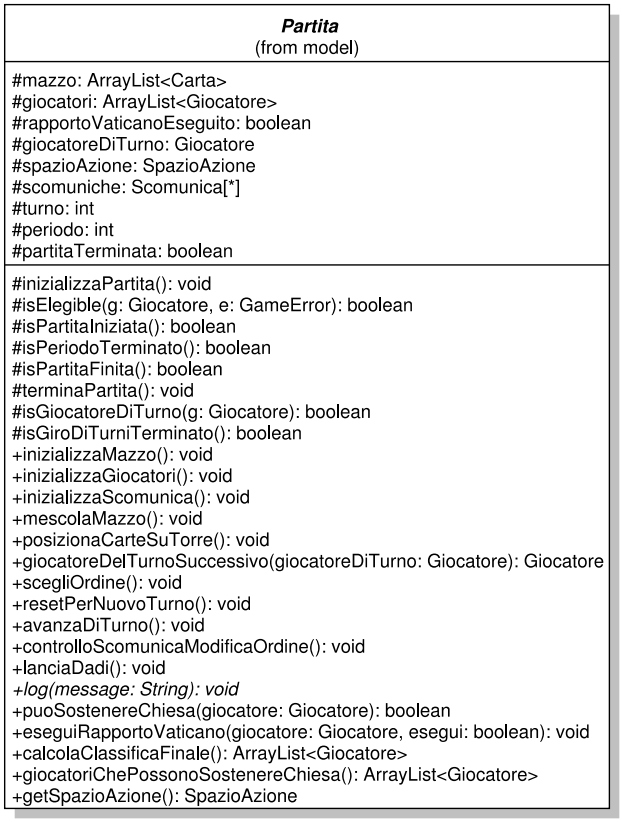


Figura 6: Classe "Partita"

3 Server e Comunicazione

In questa sezione vengono spiegate come sono state strutturate e implementate le classi che modellano la comunicazione e il flusso di esecuzione del gioco"), per la struttura completa delle classi si rimanda alla Fig.9 (pag. 11).

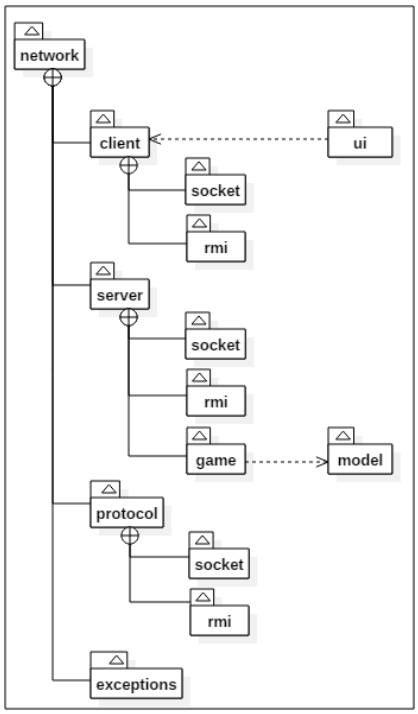


Figura 7: Struttura dei package "network"

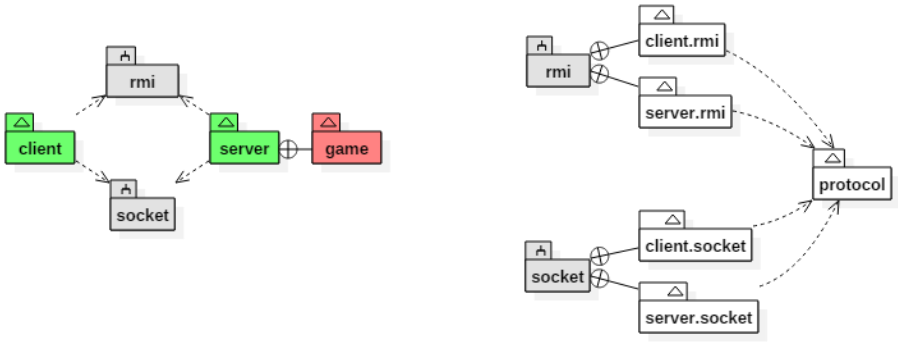


Figura 8: Dipendenze dei package "network"

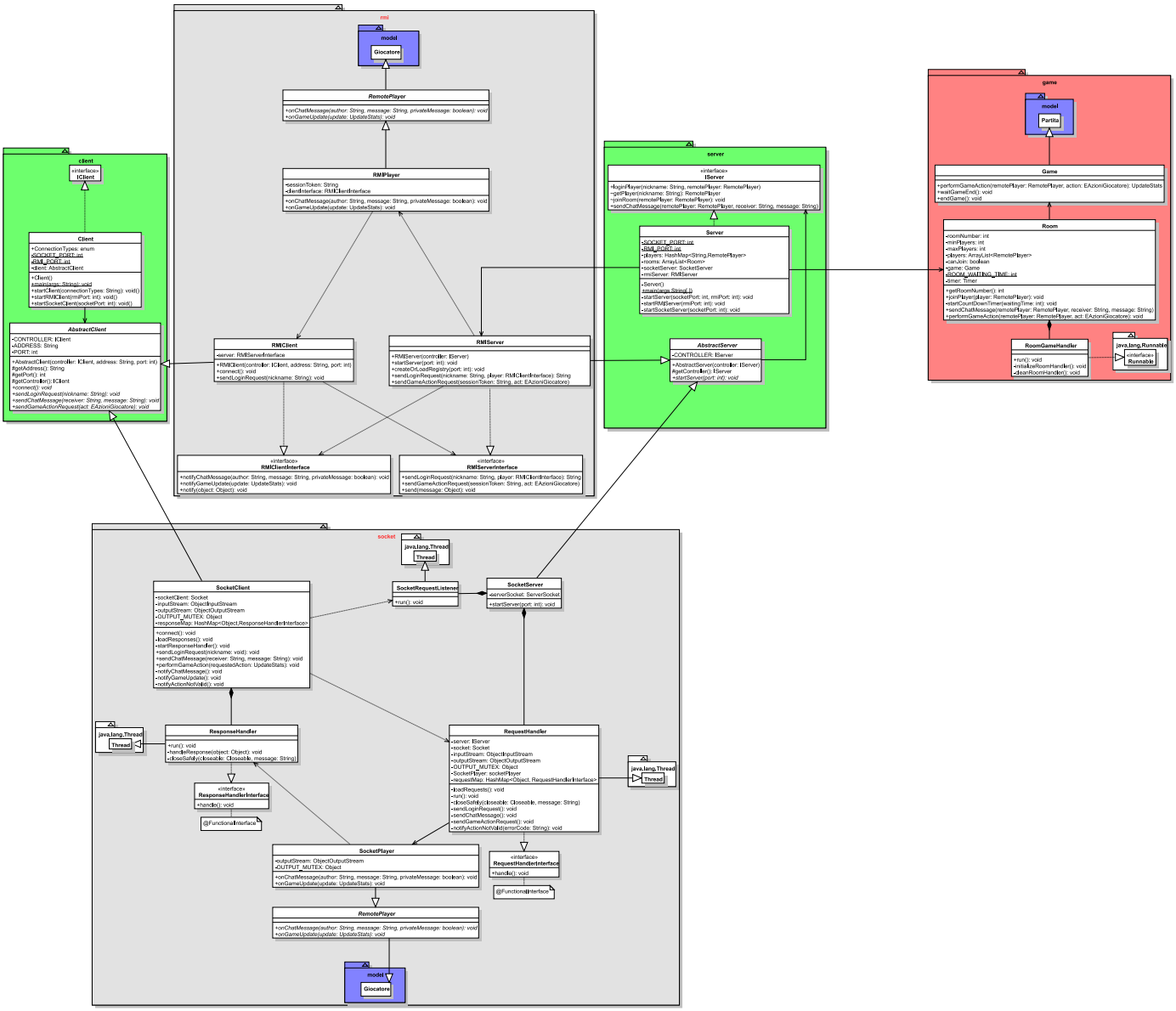


Figura 9: Diagramma delle classi del package model (vengono inserite anche le classi “Partita” e “Giocatore” per mostrane le connessioni con il package “model”)

3.1 Server

L'avvio del server può avvenire rispettivamente in due modi:

- "**Command Line**", in questo caso l'applicazione si avvierà dal relativo `main(String[] args)` il quale farà partire una singola istanza del Server secondo i parametri predefiniti (`Server Address: "127.0.0.1", RMI Port: "1099", Socket Port: "1098"`) oltremodo, se sono stati passati parametri di avvio, userà quelli definiti dall'utente (`SocketPort = args[0], RMIPort = args[1]`)

- "**startServer**", in questo caso è necessario aver precedentemente istanziato un oggetto di tipo Server, per far avviare il Server sarà necessario usare il metodo `startServer(int socketPort, int rmiPort)` per far partire entrambi i tipi di comunicazione supportati (RMI e Socket), altresì è possibile scegliere di inizializzare solamente una delle due (RMI o Socket) `startRMIServer(int rmiPort)` (per il solo Server RMI) oppure `startSocketServer(int socketPort)` (per il solo Server Socket), dai quale si può facilmente evincere il significato dei parametri.

In tutti i casi sarà possibile avviare una sola istanza del Server (localhost).

Una volta avviato il server è quindi in grado di gestire i due tipi di comunicazione già citati:

- "**RMI**", in questo caso il server gestirà le richieste provenienti dai client che supportano la connessione RMI.

- "**Socket**", in questo caso il server gestirà le richieste provenienti dai client che supportano la connessione Socket.

In entrambi i casi il tipo di connessione utilizzata per la comunicazione è trasparente sia alla "Partita" che alla "Stanza" nel quale i Client verranno aggiunti.

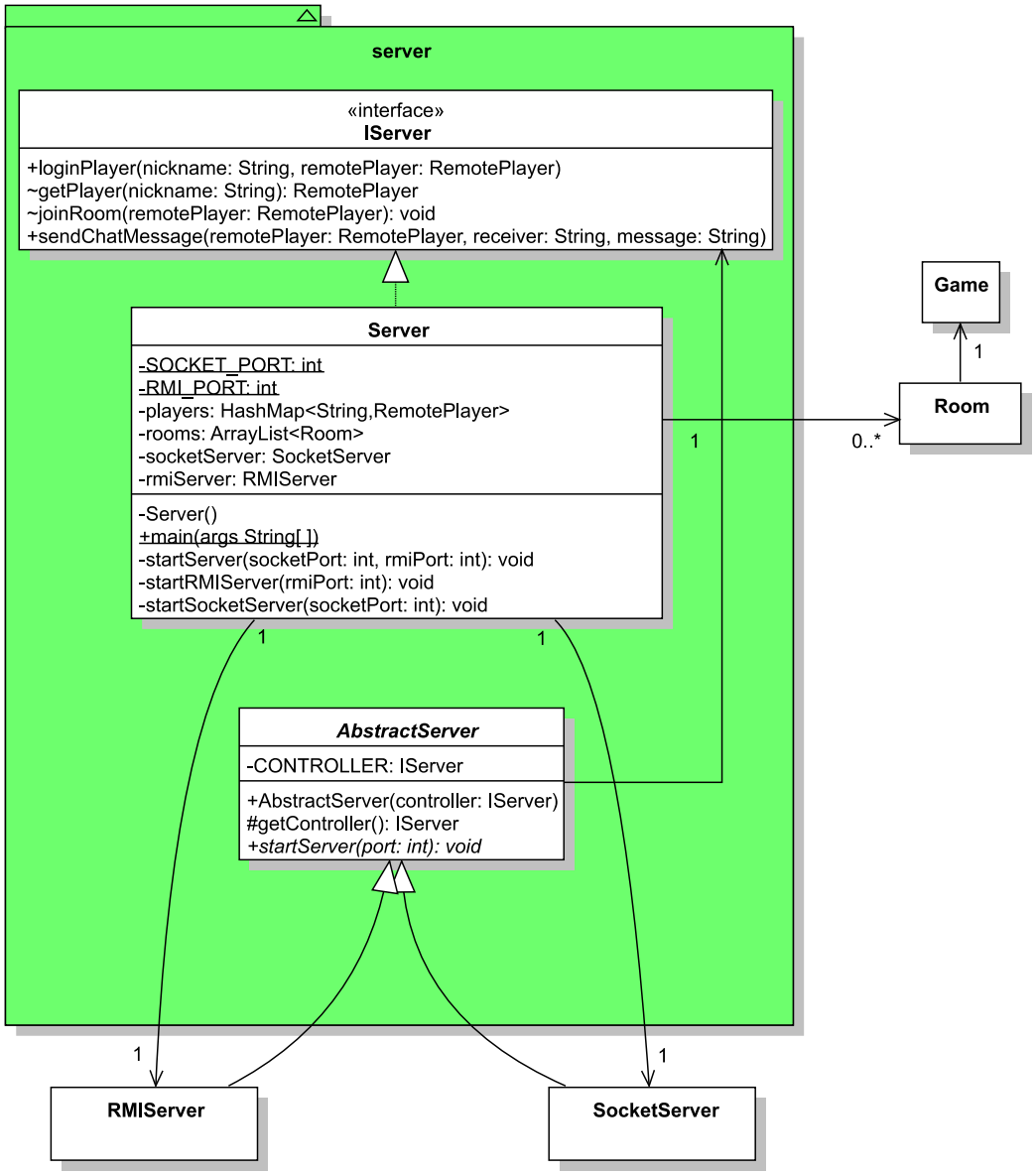


Figura 10: Diagramma delle classi (Server)

3.2 Flusso di esecuzione

Di seguito viene analizzato il flusso di esecuzione della comunicazione Client/Server:

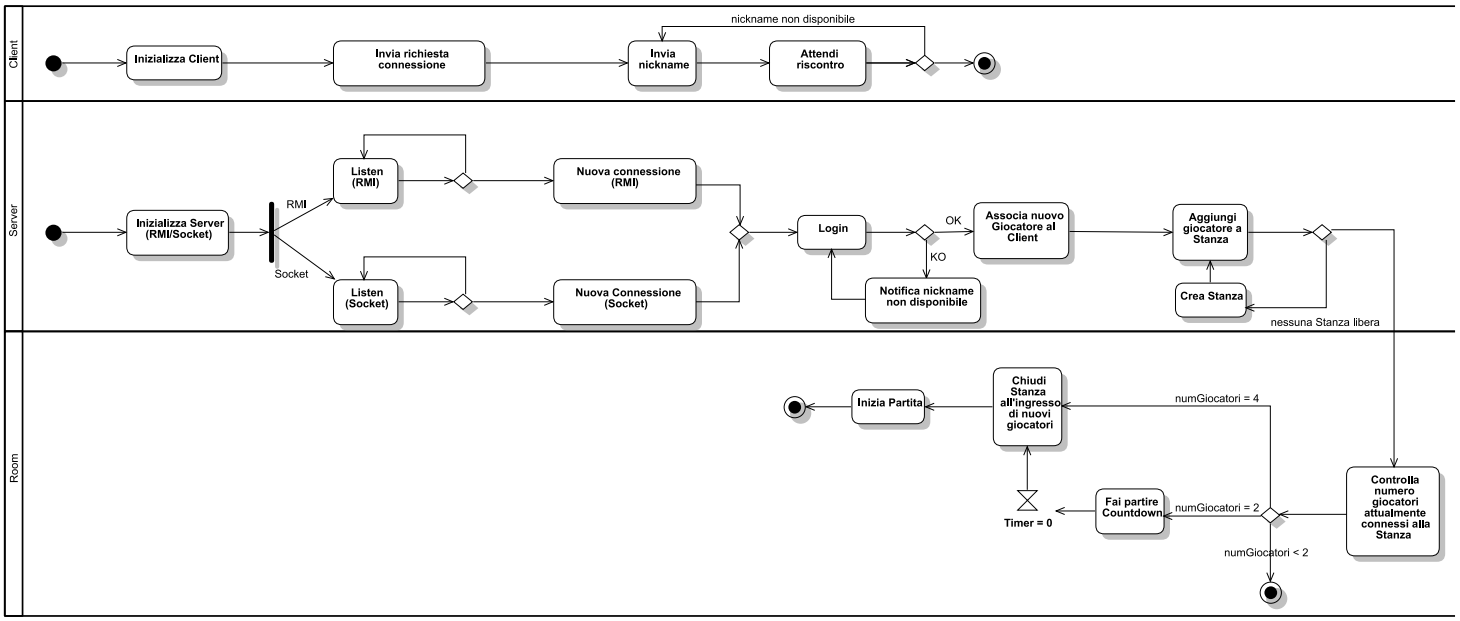


Figura 11: flusso di esecuzione del server (sequenza di Login)

1. Dopo averlo fatto partire il server rimane in attesa di nuove richieste di connessione da parte dei client.

2. La procedura di identificazione presso il server consiste nell'invio di una semplice stringa contenente il "nickname" (nome giocatore) con il quale il client vorrebbe essere registrato presso il server:

   a nel caso nome di utente già presente sul server, il server sospende la procedura di login e notifica al client di ritentare con un altro nickname

   b altresì la procedura di login continua associando al client un nuovo giocatore (un "`RemotePlayer`" coerentemente al tipo di connessione usata, ovvero, "`RMIPlayer`" nel caso connessione RMI e "`SocketPlayer`" nel caso connessione Socket).

3. Conseguentemente il giocatore creato sul server ("`RMIPlayer`" o "`SocketPlayer`") viene aggiunto alla prima Stanza ancora disponibile (nel caso nessuna Stanza sia disponibile ne viene creata una nuova in automatico).

4. Da questo punto in poi il client conclude la procedura di Login, ed il resto delle comunicazioni avverrà direttamente con la Stanza a cui è stato associato.

5. Ogni volta che viene aggiunto un giocatore alla Stanza essa controlla che si sia raggiunto il numero minimo di giocatori affinché possa partire la partita (in questo caso al secondo giocatore aggiunto parte un countdown allo scadere del quale la partita inizia automaticamente), in ogni modo, al raggiungimento del numero massimo di giocatori supportati (4 per partita) la Stanza viene chiusa immediatamente e il Gioco ha inizio.

3.3 Comunicazione

La comunicazione Client/Server può avvenire indiscriminatamente tramite Socket e/o RMI, in particolare:

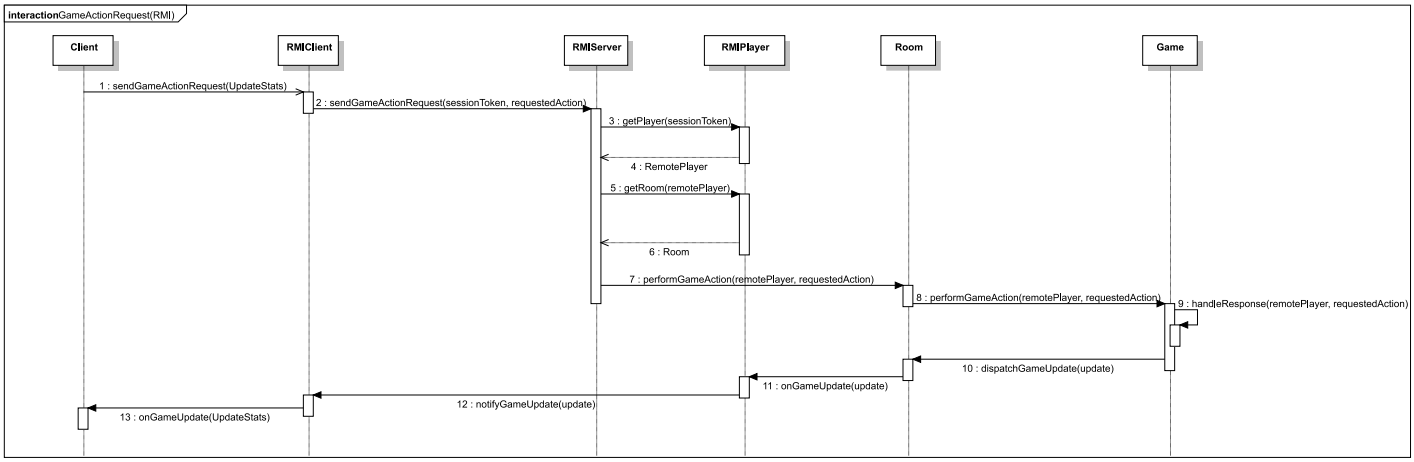


Figura 12: diagramma di sequenza per la richiesta di svolgimento di un'azione legale da parte del Client presso il Server (RMI)

1. per lo svolgimento della sessione di gioco i client (a turno) inviano una richiesta di esecuzione di un'azione di gioco (**`GameActionRequest`**) codificata all'interno di un messaggio `UpdateStats` all'interno del quale è codificato l'aggiornamento che vorrebbero effettuare allo stato della partita (l'azione richiesta al server)

2. il server riceve la richiesta e, attraverso la Stanza (`Room`) nel quale è inserito il giocatore, la inoltra presso la relativa partita (`Game`)

3. la partita provvede a verificare che il giocatore associato al client (`RemotePlayer`) sia idoneo a effettuare l'azione e, in caso positivo, aggiorna lo stato interno della partita

(provvedendo a sua volta, inviando un messaggio di `UpdateStats`, a notificare i giocatori connessi dell'avanzamento della partita) contrariamente se risulta che il giocatore stia tentando di eseguire un'azione illegale scatenerà un eccezione presso il singolo client

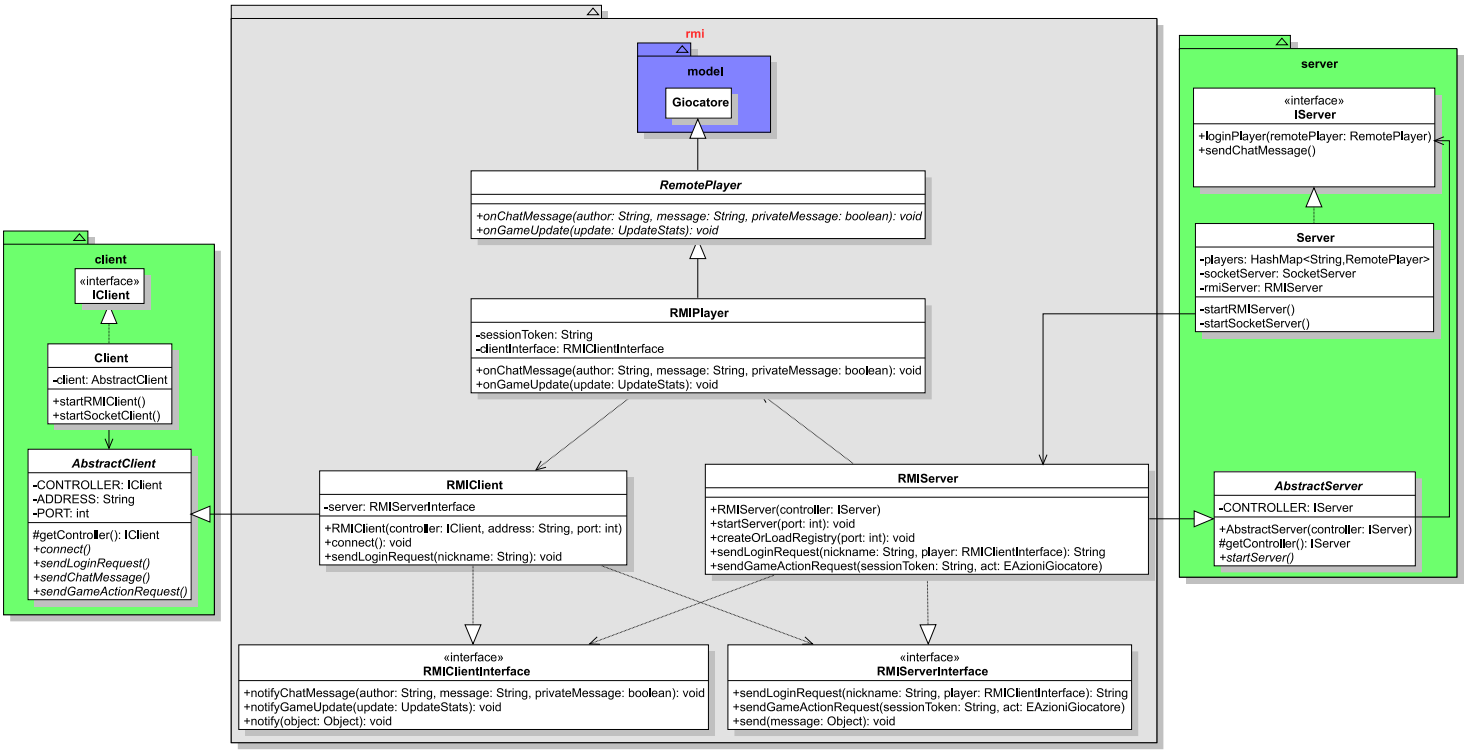


Figura 13: Diagramma delle classi (RMI)

Dal diagramma delle classi è facilmente possibile evincere che la comunicazione RMI è sostanzialmente basata sul Pattern Observer/Listener nel quale i client inviano richieste di esecuzione di azione al Server ("senders") scatenando un "evento" associato sul server, dopodiché, si mettono in attesa dell'elaborazione della richiesta da parte del server il quale a sua volta si preoccuperà di notificare l'evoluzione dello stato della partita ai client connessi (scatenando a sua volta degli "eventi" sul client).

Nel caso di comunicazione Client/Server attraverso Socket, ovviamente, l'invio delle richieste non può avvenire tramite chiamate a metodi remoti (da Client a Server, e viceversa), ma solamente attraverso un preciso protocollo di comunicazione. Specificatamente, il protocollo Socket è per semplicità fortemente basato su quello definito dalle interfacce "RMIClientInterface" e "RMIServerInterface" per la comunicazione attraverso RMI (vedi package "protocol"); in particolare, ogni qualvolta si intenda "simulare" l'invocazione di in metodo tramite Socket sarà necessario inviare in sequenza: ["NOME_METODO", "PARAMETRI", "EVENTUALE_CODICE_ERRORE"], ad esempio per la richiesta di svolgimento di un'azione (**`GameActionRequest`**) il client invierà:

1. "`gameAction`": stringa identificativa del "metodo" da attivare presso il server
2. "`requestedAction`" `UpdateStats` nel quale è codificata l'azione richiesta

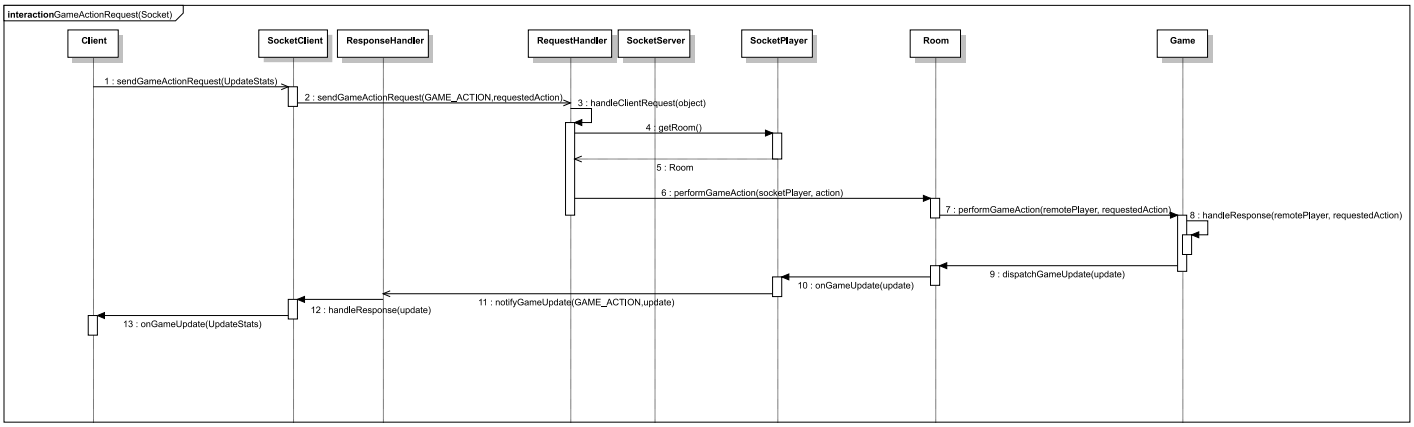


Figura 14: diagramma di sequenza per la richiesta di svolgimento di un'azione legale da parte del Client presso il Server (Socket)

Nel caso di azione illegale il server notificherà al client il fatto "attivando" il metodo **`notifyActionNotValid`** metodo (passando come parametro una stringa contenente il codice di errore analoga all'eccezione che si sarebbe scatenata effettuando una richiesta RMI).

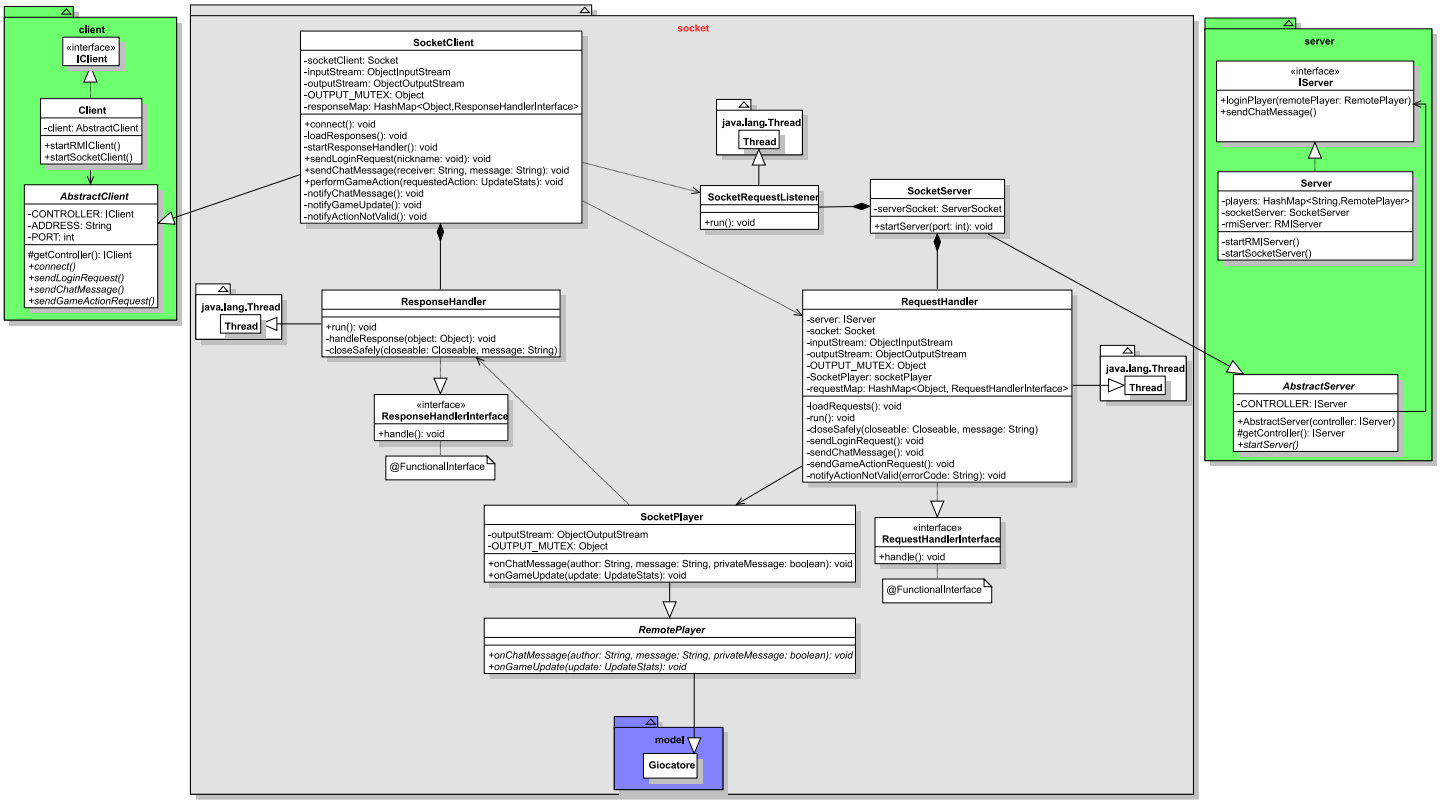


Figura 15: Diagramma delle classi (Socket)

Dal diagramma delle classi è facilmente possibile evincere che anche la comunicazione Socket è sostanzialmente basata sul Pattern Observer/Listener, ma con le dovute differenze da RMI, infatti, a causa del funzionamento intrinseco del `serverSocket.accept()` (bloccante) si rende necessario l'utilizzo di due Thread distinti: il primo per l'accettazione di nuove richieste di login (`SocketRequestListener`) ed il secondo per la gestione delle richieste dei client (`RequestHandler`).

4 Client

Entrambe le modalità (CLI e GUI) implementano l'interfaccia `IClient`, attraverso il quale il Server è in grado di scatenare gli eventi di notifica direttamente sull'interfaccia utente, per esempio: l'evento **`onGameStarted`** viene prima "attivato" su `Client` il quale non fa altro che inoltrare l'evento all'interfaccia associata eseguendo `ui.onGameStarted(Update)`.

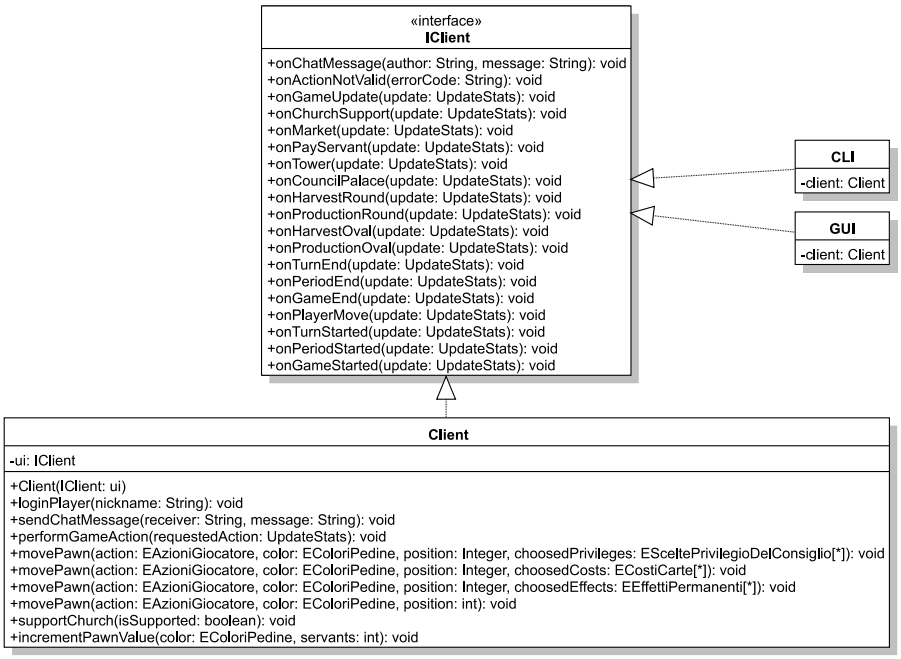


Figura 16: Diagramma delle classi che mostra la connessione tra il package "network" (`Client`) ed il package "ui" (`CLI` e `GUI`)

Per quanto riguarda l'invio di richieste verso il Server, il funzionamento è analogo, ad esempio: per la richiesta di spostamento di una pedina **`movePawn`** l'interfaccia utente (CLI o GUI) notificherà il fatto tramite l'oggetto Client eseguendo una `client.movePawn(Request)`.

5 Grafica

Per quanto riguarda la parte dell'interfaccia grafica, sono stati utilizzati AWT e Swing. La struttura dell'interfaccia è composta da più frame, di seguito i principali:

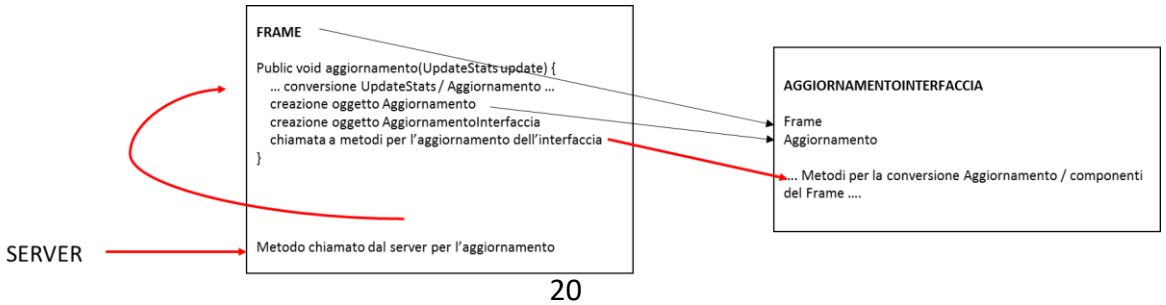
- **`GUI`**, frame utilizzato come principale schermata al momento della partita, in particolare, comprende numerosi oggetti per lo sviluppo dell'interfaccia quali i JPanel: `Tabellone`, `Plancia`, `PlanciaAvversario`.
- **`UsernameFrame`**, utilizzato al momento del login del giocatore
- **`PrivilegioConsiglio`**, consente al giocatore di scegliere tra le varie opzioni proposte dal privilegio del consiglio (chiamata tramite la classe privata `ChiediPrivilegioDelConsiglio` all'interno di `GUI`)
- **`ClassificaFinaleFrame`**, dove vengono stampati a video i risultati finali della partita (chiamata tramite la classe privata `ApriPaginaFinePartita` all'interno di `GUI`).

5.1 I Listener

Sono stati aggiunti all'interno di Frame i Listener agli oggetti che devono essere in grado di permettere all'utente di interagire col gioco. Di seguito le "zone" più rilevanti usate ai fini del funzionamento del gioco: **`FamigliariPlancia`**, **`TorreTabellone`**, **`RaccoltoTabellone`**, **`ProduzioneRotondoTabellone`**, **`PalazzoConsiglioTabellone`**, **`MercatoTabellone`**, **`PanelServitore`**.

5.2 Aggiornamento e AggiornamentoInterfaccia

L'aggiornamento dell'interfaccia è il punto nevralgico per il corretto funzionamento della stessa. Utilizza un package detto "aggiornamento", il quale contiene varie classi "fittizie" che rappresentano le varie componenti del Frame. Partendo delle informazioni ricevute dal Server viene generato un oggetto `Aggiornamento` attraverso l'apposito metodo `aggiornamento(UpdateState)`, il quale assieme al Frame `GUI`, verrà successivamente passato alla classe `AggiornamentoInterfaccia` che non farà altro che modificare opportunamente i vari elementi dell'interfaccia.



6 Test

I test implementati sono basati principalmente sul controllo delle fasi avanzate di gioco e sulla correttezza dei metodi che legano le scelte del giocatore alle dovute conseguenze nel gioco. E' stata fatta la scelta di non implementare test parametrici in quanto il numero di scelte effettuabili dall'utente sono molteplici e molto simili tra loro.

6.1 GameTest

La classe `GameTest` contiene i test più significativi relativi alle possibili azioni nel gioco da parte dell'utente ed alla gestione dell'avanzamento dei turni, nonché della gestione della fase del Rapporto col Vaticano. Di seguito vengono descritti gli ambiti dei test compiuti:

- Test sulla gestione corretta dell'avanzamento del turno, del periodo e relativa fase del Rapporto col Vaticano (`testGiroDiTurniTerminato, testFinePeriodo, testRapportoVaticano`)
- Test sulla corretta gestione delle azioni indicate dall'utente nelle varie fasi di gioco (`testOnMarket, testOnPayServant, testOnTower, testOnCouncilPalace, testOnHarvestRound, testOnHarvestOval e simili`)
- Test sulla corretta gestione delle eccezioni scatenate da un errore dell'utente (`testOnMarketExceptionSpaceTaken, testOnMarketExceptionInvalidPosition, testOnMarketExceptionInsufficientFamiliarValue, testOnMarketExceptionInvalidChoice`)

6.2 GiocatoreTest

La classe `GiocatoreTest` ha l'obiettivo di verificare la correttezza dei metodi legati al giocatore da un punto di vista prettamente logico, non concernente la gestione di comunicazione o grafica. I test di `GiocatoreTest` sono i seguenti:

- Test sul controllo del posizionamento di tutti i famigliari di un giocatore sul tabellone (`testCheckPosizionato`)
- Test sulla gestione dell'aumento del valore di un famigliare (`testPagaServitore`)
- Test sulla corretta gestione delle eccezioni scatenate da un errore dell'utente (`testOnMarketExceptionSpaceTaken, testOnMarketExceptionInvalidPosition, testOnMarketExceptionInsufficientFamiliarValue, testOnMarketExceptionInvalidChoice`)

6.3 PartitaTest

La classe `PartitaTest`, a differenza di Game dove è presa in considerazione l'interazione da parte dell'utente, verifica principalmente la corretta gestione dal punto di vista logico delle fasi di raccordo tra i turni di gioco. I test di `PartitaTest` sono i seguenti:

- Test sul controllo della corretta modifica dell'ordine dei turni per il successivo turno di gioco (`testScegliOrdine`)
- Test sull'applicazione corretta delle scomuniche e relative a seguito dell'intenzione di un giocatore di supportare o meno il Vaticano (`testEseguiRapportoVaticano`)
- Test sulla segnalazione della fine di una fase di gioco (`testFineTurno`)

7 Futuri Sviluppi

Di seguito vengono elencati un insieme di miglioramenti o implementazioni di funzionalità future:

1. Migliorare la gestione del fine Partita, ad esempio: permettendo agli utenti di conoscerne lo stato e in seguito decidere se intraprendere o meno una nuova partita.

2. Migliorare la gestione delle Stanze, ad esempio: al momento le stanze chiuse (quelle in cui la partita è in corso/terminata) non vengono controllate in fase di aggiunta di un nuovo giocatore, anzi, il giocatore viene sempre aggiunto all'ultima stanza creata (se disponibile), altrimenti, ne viene creata sempre una nuova.

3. Migliorare la gestione dei Giocatori, ad esempio: al momento un giocatore quando esegue il login deve usare obbligatoriamente un "nickname" diverso da quelli presenti sul server, in questo modo diventa impossibile per un giocatore ricollegarsi ad una partita a seguito di errore di connessione.

4. Migliorare la Struttura del Progetto, ad esempio: bisognerebbe eseguire il "refactoring" di alcuni metodi perché non sono stare rispettate alcune buone pratiche di programmazione OOP.

5. Aumentare la Copertura e la Completezza dei Test.