Documentazione Prova Finale

“Lorenzo Il Magnifico”

Castelnuovo Carlo, Cerioli Alessando, Tosi Matteo

20 Giugno 2017

Sommario

[1 Introduzione 3](#_Toc7246)

[1.1 Obbiettivo e Specifiche 3](#_Toc7247)

[1.2 Tecnologie e Architettura 3](#_Toc7248)

[1.3 Struttura del Documento 5](#_Toc7249)

[2 Model 6](#_Toc7250)

[2.1 I pezzi e la scacchiera 6](#_Toc7251)

[2.2 Altri componenti 9](#_Toc7252)

[3 Server e Comunicazione 12](#_Toc7253)

[3.1 Flusso di esecuzione 12](#_Toc7254)

[3.2 Comunicazione 14](#_Toc7255)

[4 Client 17](#_Toc7256)

[4.1 Flusso di esecuzione 17](#_Toc7257)

[4.2 SessionManager e NetworkManager 17](#_Toc7258)

[5 Grafica 18](#_Toc7259)

[5.1 Gli FXML e i CSS 19](#_Toc7260)

[5.2 Le immagini 19](#_Toc7261)

[5.3 I controller 20](#_Toc7262)

[5.4 I popup 20](#_Toc7263)

[6 Futuri sviluppi 20](#_Toc7264)

# 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.

## 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 quindi composto da:

* un singolo **Server** che puó gestire più partite simultaneamente
* multipli **Client** (uno per giocatore) che possono partecipare ad una sola partita alla volta

L’**interfaccia grafica** è posta sul solo Client per poter interagire con il Server e monitorare l’avanzamento della partita.

Il gioco è progettato al fine di:

* poter caricare da file tutte le carte: “carte sviluppo”, “carte scomunica”
* attendere un timeout (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)

## 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** come richiesto, si è scelto il paradigma Client/Server nel quale il Server funge da gestore della logica del gioco e delle connessioni tra i singoli terminali Client che si limitano a fornire le mosse disponibili al giocatore.

**Linguaggio di programmazione** come richiesto, per il 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 quindi optato per: Swing

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

* instanziabile una sola volta, per poter gestire più partite contemporaneamente
* implementa la comunicazione Client/Server sia via Socket sia via RMI
* supporto a partite in cui i giocatori utilizzano tecnologie diverse (Socket/RMI)
* implementato con JavaSE (ver. 8)

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

* instanziabile piú volte (una per giocatore)
* supporta 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)
* implementato mediante Swing (solo GUI) e JavaSE (ver. 8) (CLI e GUI)

## 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).

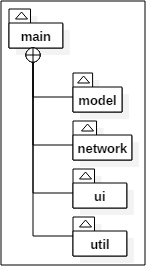


Figura 1: Struttura dei package

# 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”).

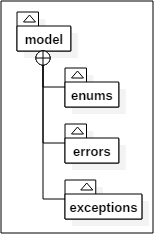


Figura 2: Struttura dei package “model”

## I pezzi e la scacchiera

I pezzi Tutti i tipi di pezzi sono rappresentati da una classe, e ognuna di queste estende la stessa *classe astratta* Piece.In particolare ogni classe fa un *overriding* della funzione accessiblePositions che definisce le caselle che quel pezzo può raggiungere, e quindi in ultima analisi il suo comportamento. Questo permetterà poi di trattare ogni pezzo come riferimento ad un tipo Piece, e tramite *binding dinamico* ottenere il comportamento che ci si aspetta. Per capire come e dove un pezzo può muoversi deve sapere:

* il proprio colore del pezzo
* la scacchiera su cui si trova
* la sua posizione nella scacchiera

Come si vede anche dalla figura 2 queste condizioni si traducono immediatamente in attributi della classe che sono riferimenti ad oggetti rispettivamente di tipo:

* PieceType, che non è nient’altro che un enum per distinguere il colore dei pezzi
* Chessboard, che approfondiremo nel prossimo paragrafo
* Coordinate, che sostanzialmente è una classe che fa da *wrapper* ad una coppia di indici interi, con la comodità di avere controlli sul valore e altri metodi utili incapsulati nella classe

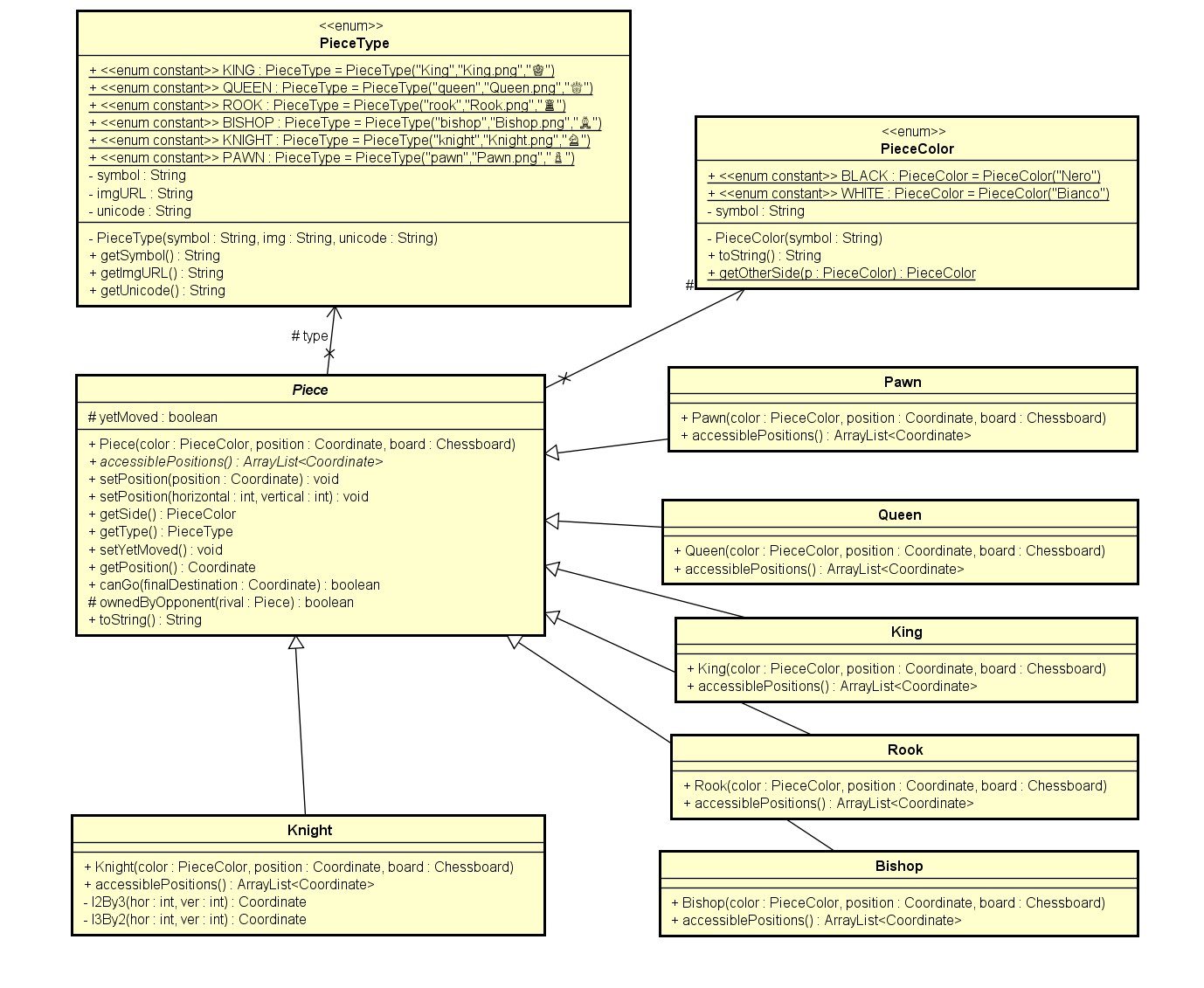


Figura 3: Struttura classe Piece

La scacchiera La scacchiera (la classe Chessboard) non è nient’altro che un *incapsulamento* di un array bidimensionale di tipi Piece. Questa classe non contiene ancora la logica del gioco con la gestione dei turni e la verifica dello stato del gioco, quello verrà analizzato nella sezione 2.2. Comunque la classe permette di fare tutte le operazioni che si potrebbero fare *"fisicamente"* alla scacchiera, cioè spostare i pezzi senza regole, eliminarli, modificarli, aggiungerli, vedere quali mosse questi potrebbero fare.

## Altri componenti

Le classi Coordinate e Move Queste classi servono solo per fare un *wrap* rispettivamente di una coordinata intera bidimensionale, e di due Coordinate per rappresentare una mossa. Queste classi contengono alcuni

I messaggi I messaggi sono le classi le cui istanze vengono scambiate tra il client e il server. Attualmente non andrò nei dettagli della parte di networking che preferisco spiegare nella sezione 3. Attualmente però si può dire che la struttura è identica a quella dei pezzi, ovvero una classe Message che poi viene specializzata a mano a mano. In questo caso però l’overriding avviene per due metodi:

* serverAction, definisce le azioni da compiere sul server, ed è esso stesso che la chiama non appena arriva un messaggio
* haveEffect, che definisce in modo duale le azioni da compiere sul client La struttura dei messaggi è rappresentata nel seguente UML (figura 3):

Il giocatore e il MoveController Nel package del modello sono presenti anche la classe Player che contiene le informazioni base per descrivere un giocatore, cioè il suo nome e la sua fazione. Una versione contenete maggiori informazioni è la classe ServerPlayer contenuta nel package server che verrà analizzato nella sezione 3.

inoltre c’è anche la classe MoveController che controlla l’evoluzione del gioco, in particolare: l’alternarsi dei turni, la fattibilità delle mosse, le condizioni di scacco e di scacco matto. Centrale è il metodo move, che riceve una istanza della classe Move, e restituisce un messaggio che poi potrà essere mandato o al client che rappresenterà una azione valida o meno o azioni speciali.

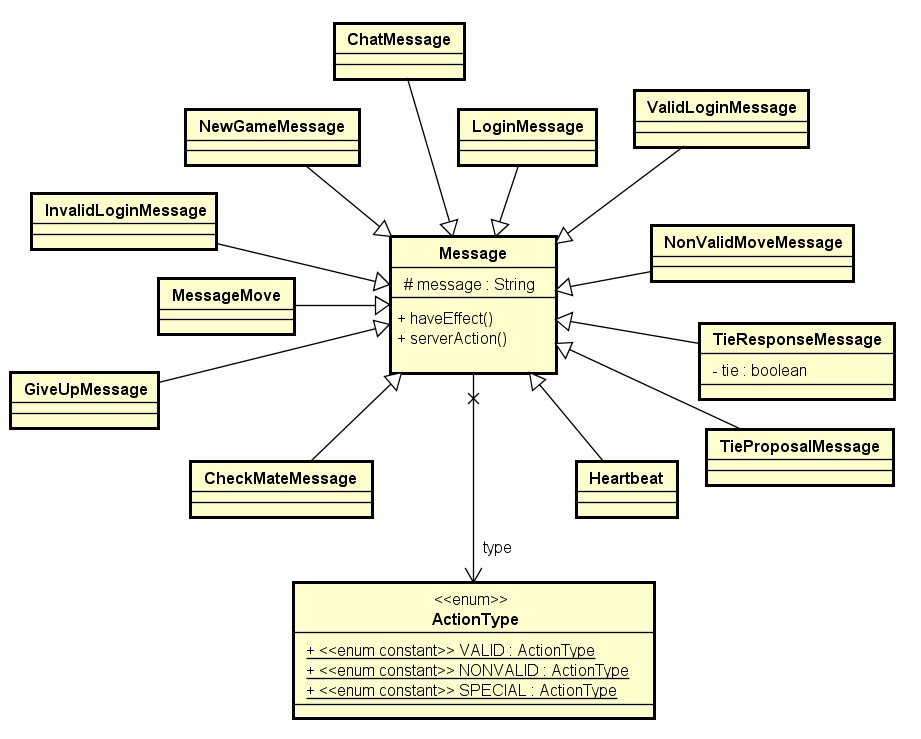


Figura 4: Struttura e interdipendenze tra i messaggi, vengono mostrati i metodi solo della superclasse Message

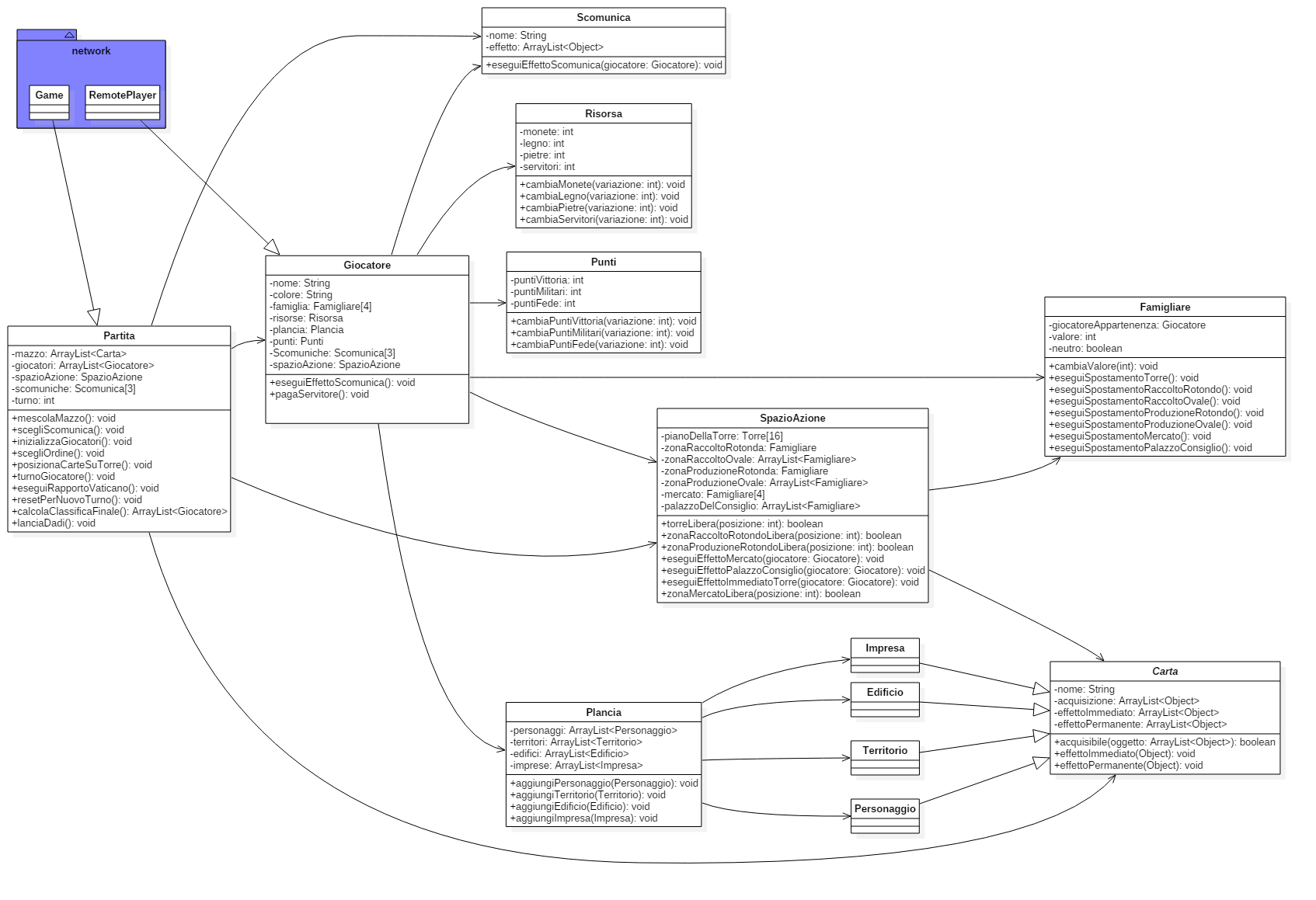


Figura 5: Diagramma delle classi del package model (vengono inserite anche le classi “Game” e “RemotePlayer” per mostrane le connessioni con il package “network”)

# 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.

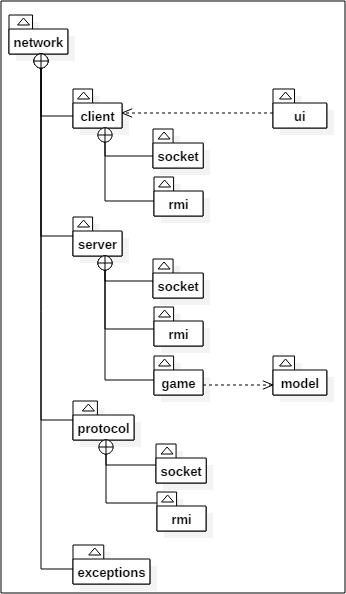


Figura 6: Struttura dei package “network”

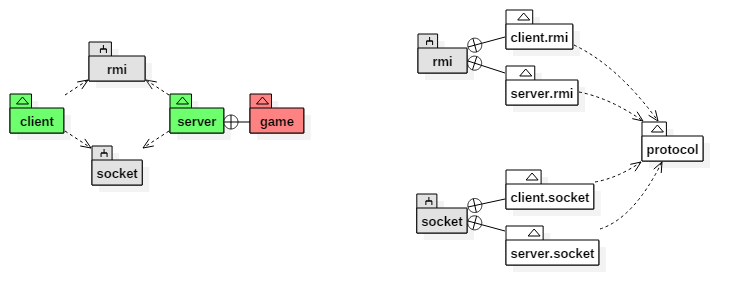


Figura 7: Dipendenze dei package “network”

## Server

L’avvio del server può avvenire in due modi:

* “**Stand Alone**”, in questo caso bisogna far partire l’applicazione 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])
* “**Async Start**”, in questo 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 (locale).

Una volta avviato il server è quindi in grado di gestire due i 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.

C:\Users\Matteo\Desktop\ClassDiagram(Server).emf

Figura 8: Diagramma delle classi (Server)

## Flusso di esecuzione

Qui si analizzano le varie parti della esecuzione del server:

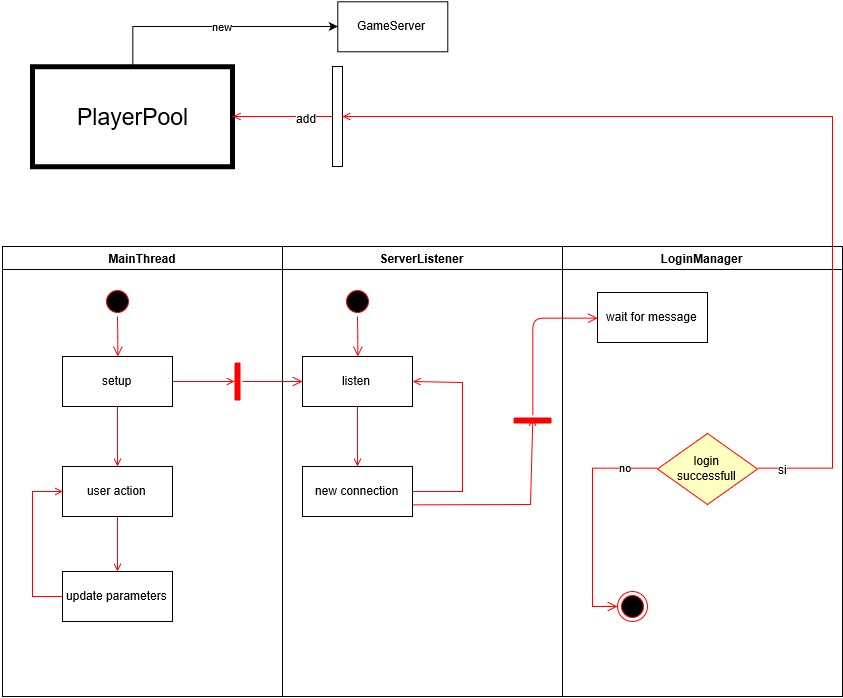


Figura 7: flusso di esecuzione del server

1. Il thread main viene lanciato nella classe ServerMain che compie l’inizializzazione del server, e dopodichè attende degli input da tastiera per abilitare o disabilitare alcune funzionalità di dignostica o informative. L’inizializzazione corrisponde a far partire la classe ServerListener su un differente thread.
2. Il thread ServerListener non fa altro che ascoltare sulla porta predefinita eventuali connessioni da accettare. Questo thread (come del resto tutti gli altri) è di tipo daemon così che la chiusura del thread Main comporta la chiusura di tutti i thread attivi. Una volta ricevuta una connessione il thread continua ad ascoltare sulla porta.
3. Quando un client si connette si fa partire un ulteriore thread a partiredalla classe LoginManager che si occupa dalla procedura di login, che se va a successo inserisce il giocatore nella PlayerPool, cioè l’insieme dei giocatori che stanno aspettando di fare una partita, mentre se il login fallisce la connessione viene chiusa e il client dovrà rifare la procedura di login dall’inizio.
4. come già detto se il login ha successo il giocatore finisce nella PlayerPool, da qui non appena ho almeno un giocatore per fazione libero faccio partire una partita (la cui esecuzione verrà spiegata nel paragrafo 3.2).
5. alla fine della partita i due giocatori vengono entrambi disconnessi, eper giocare nuovamente dovranno ripetere l’operazione dall’inizio

PlayerPool La classe PlayerPool rappresenta la coda d’attesa. In realtà questa è implementata tramite due code, una per i giocatori di fazione bianca e l’altra per quelli di fazione nera. Queste due code seguono una politica FIFO, e le due teste della coda vengono estratte per far partire una partita non appena sono non null.

## Comunicazione

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

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 sever 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

C:\Users\Matteo\Desktop\GameActionRequest(RMI).emf

Figura 9: diagramma di sequenza per la richiesta di svolgimento di un’azione legale da parte del Client presso il Server (RMI)

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 protcollo 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

C:\Users\Matteo\Desktop\GameActionRequest(Socket).emf

Figura 10: 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).

Come già anticipato si inviano messaggi che estendono la classe Message (vedi 2.2), che vengono prima serializzati utilizzando lo standard JSON, in particolare, vengono creati una coppia nome\_attributo:valore per ogni proprietà della classe, e poi viene creato un attributo type che contiene il tipo della sottoclasse, così che al momento della deserializzazione è possibile specificare la sottoclasse da istanziare e quindi riottenere lo stesso oggetto. Un esempio potrebbe essere:

{

"type":"GiveUpMessage.class",

"message": "text",

"type" : "SPECIAL"

}

flusso di messaggi

1. Tutto inizia con il messaggio di NewGameMessage che viene mandato dal server a tutti i giocatori, e contiene tutte le informazioni riguardanti la partita appena inziata.
2. Dopo di che vengono inviati messaggi di tipo MoveMessage che propongono una mossa al server, e il MoveController presente sul server genera una risposta in base all’esito della mossa. Che può essere di successo o insuccesso, e in quest’ultimo caso viene riportato anche il motivo nel messaggio.
3. Ci sono poi una serie di messaggi che sono asincroni, nel senso che nonsono dettati dal valore del turno. Questi sono per esempio:
   * GiveUpMessage che serve per abbandonare la partita
   * ChatMessage per lo scambio di messaggi in chat
   * TieProposalMessage per proporre una parità
   * TieResponseMessage per rispondere alla parità
4. La fine della partita può avvenire con uno dei sopracitati messaggi,oppure tramite il CheckMateMessage che definsce la fine della partita per scacco matto.
5. solo della supeclasse Message

C:\Users\Matteo\Desktop\ClassDiagram.emfFigura 6: Diagramma delle classi del package model (vengono inserite anche le classi “Partita” e “Giocatore” per mostrane le connessioni con il package “model”)

# Client

## Flusso di esecuzione

Come per il sever si analizzerà il flusso di esecuzione del client:

1. Tutto parte dal main presente nella classe Client che inizializza i due *singleton* NetworkManger e SessionManger
2. Poi viene chiamta la funzione launch che fa partire il thread di JavaFX
3. Il thread di JavaFX come prima cosa carica tutte le risorse necessarie (FXML e immagini) e poi mostra la schermata iniziale.
4. nel caso in cui ci si connetta al server per una partita online vienelanciato un thread dalla classe ClientListener che riceve i messaggi e poi ne utilizza il già citato metodo haveEffect per compiere tutte le azioni necessarie. Per poi accedere al controller e alla scena di cui bisogna modificare i componenti, il metodo utilizza i due singelton SessionManager e NetworkManager
5. il resto del flusso è demandato alle scelte dell’utente sull’interfacciagrafica.

## SessionManager e NetworkManager

queste due classi sono due singleton che servono per

* gestire la visualizzazione delle varie schermate
* gestire le componenti necessarie al networking
* fornire un single access point per tutte le alre classi e soprattutto per l’esecuzione delle azioni associate ai messaggi

In particolare SessionManger si occupa di della gestione delle scene e della loro visualizzazione nonchè del mantenimento di alcune informazioni per il gioco in locale, mentre NetworkManager si occupa di contenere le informazioni della sessione di gioco online e delle strutture necessarie alla comunicazione. Metodi e attributi sono presenti nell’UML in figura 6, i loro nomi sono abbastanza autoesplicativi.

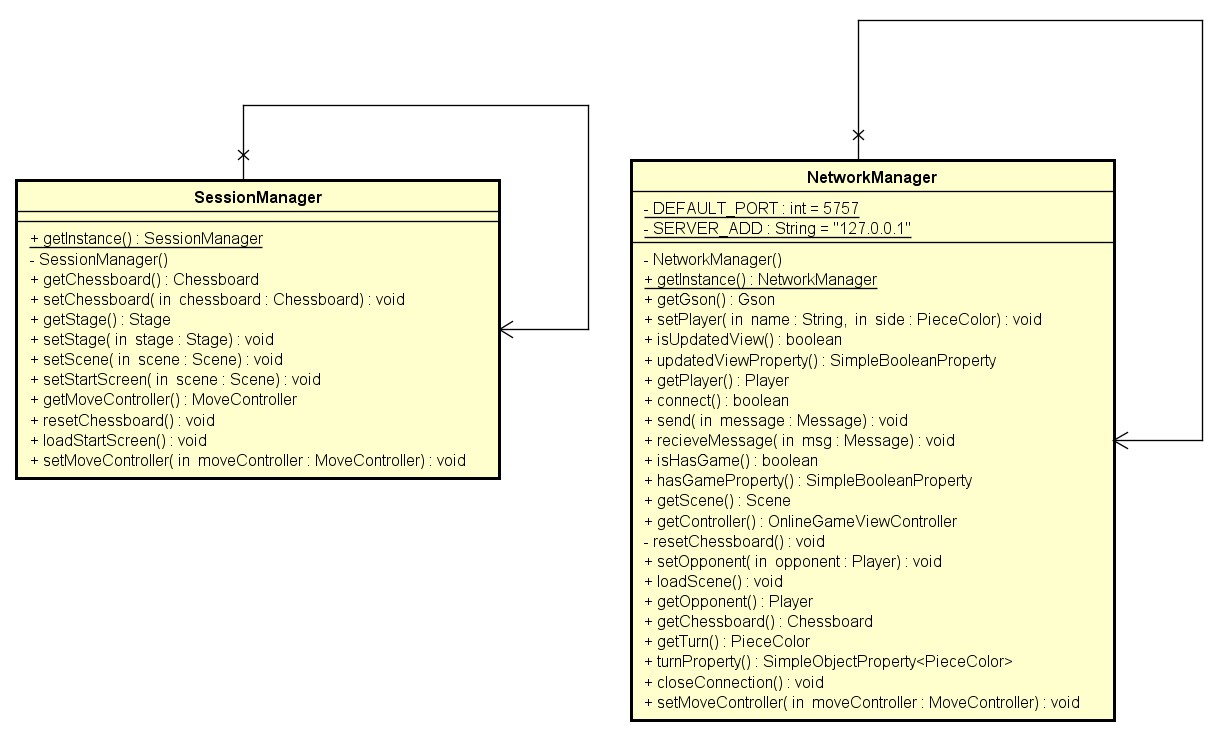


Figura 6: Struttura delle classi SessionManager e NetworkManager

# Grafica

Come già accennato nella introduzione si è scelto di utilizzare JAVAFX per la parte di grafica, per via della sua estrema flessibilità e potenza. I componenti della grafica sono raggruppati nel package view, e comprendono:

* file FXML, che definiscono la struttura della finestra grafica
* gli stylesheet associati ad essi
* Classi controller, che appunto controllano uno specifico FXML
* Un insieme di classi che mostrano dei Popup personalizzati
* le immagini utilizzate

Molti degli elementi grafici utilizzati sono parte della libreria JFoenix che "reinterpreta" alcuni elementi nativi di JAVAFX seguendo le regole dettate dal material design per avere un aspetto più moderno. [[1]](#footnote-1).

## Gli FXML e i CSS

I file chre rappresentano le varie schermate sono soltanto 3:

StartScreen.fxml che rappresenta la schermata iniziale

GameView.fxml che rappresenta la schermata di gioco in locale

OnlineGameView.fxml che invece è relativa ad una partita in rete a cui poi sono associati 2 file CSS:

StartScreen.css che definisce lo stile della schermata d’inizio

GameView.css che si applica agli altri due FXML

La trattazione dei controller delle varie schermate è rimandata al paragrafo 5.3, cioè dopo le componenti statiche.

## Le immagini

Tutte le pedine sono rappresentate da una relativa immagine. L’associazione avviene tramite la classe ImagePicker che utilizza una HashMap per associare un valore dell’enum PieceType ad un oggetto di tipo Image che ne è la rappresentazione. L’inizializzazione della HashMap avviene all’inizio dell’applicazione, in cui appunto si caricano le immagini che devono vengono selezionate in base al loro nome dalle cartelle whites e blacks. In questo modo è possibile sostituire facilmente le pedine, ma non tanto per scegliere tra un particolare set di icone(più nella sezione 6).

Le altre immagini utilizzate come sfondo o per il titolo della schermata iniziale sono raccolte nella cartella images.

## I controller

Le classi che fanno da controller si trovano nella stessa cartella dei rispettivi FXML e ne riprendono il nome, in particolare sono:

StartScreenController controlla ala schermata iniziale

GameViewController controlla la schermata di gioco in locale

OnlineGameViewController per le partite online

## I popup

Questi sono classi che che servono per far comparire dei popup sullo schermo. Fanno utilizzo della classe JFXDialog come base per il dialog box e poi vengono riempiti con i vari elementi in base alla casistica. Tra queste classi non c’è una relazione gerarchica per via delle operazioni molto diverse tra loro che devono compiere e che richiedono diversi parametri.

# Futuri sviluppi

Qui viene elencato un insieme di miglioramenti o implementazioni di funzionalità future:

1. Aggiungere un’interfaccia grafica al server per rendere più leggibili i log
2. Permettere di tornare indietro nelle mosse nella partita il locale, e magari anche in quella online entro un certo timeout. Questa funzionalità necessita di modificare abbastanza la struttura di alcune classi, poiché attualmente il gioco non mantiene informazione riguardo le mosse precedenti
3. Con l’implementazione del punto precedente sarebbe anche possibileimplementare la mossa *en passant*
4. Si potrebbe inoltre implementare la possibilità di giocare con un utentespecifico.
5. Anche l’introduzione di un menù contestuale in cui scegliere magari icolori della scacchiera o un set di pedine specifiche.
6. per quanto riguarda la struttura del progetto, bisognerebbe riordinare le classi del client perché non sono stare rispettate alcune buone pratiche dell’incapsulamento, sia alcune strutture che tendono più al procedurale che all’OOP.

1. Per la documentazione e altre informazioni sulla libreria si può consultare il sito [jfoenix.com.](http://www.jfoenix.com/) [↑](#footnote-ref-1)