Andrea Guglielmini
Matricola 826160

PROGETTO SOFTWARE

Gioco di carte “Dubito”

PROGETTO SOFTWARE

Gioco di carte “Dubito”

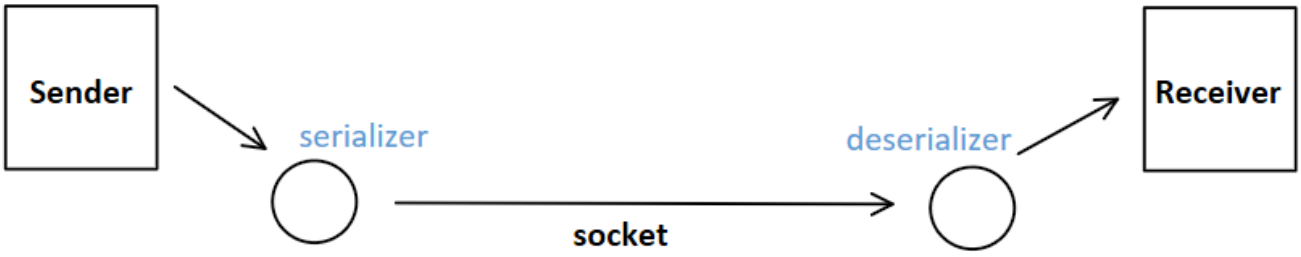
Introduzione

Il progetto consiste nella realizzazione del gioco di carte “Dubito” mediante una architettura distribuita client-server utilizzando come linguaggio di programmazione Java.

Il programma è strutturato in due file **eseguibili,** compatibili con i principali sistemi operativi (MacOS, Linux, Windows).
In particolare nel pacchetto di installazione si trovano un *client* ed un *server.*
Il primo permette di usufruire e partecipare al gioco stesso connettendosi ad una istanza di un server che può essere avviato sia da un computer remoto sia in locale.
Attualmente la dimostrazione del programma viene effettuata connettendosi ad un server remoto.

Comunicazione

Il *client* comunica con il server creando una *socket* verso la porta 8080 dello stesso; una volta che la connessione è stata istanziata lo scambio di informazioni avviene mediante la *JavaScript Object Notation.* Ogni azione che viene eseguita durante il gioco è rappresentata mediante un oggetto.
Vi sono diverse classi che estendono la generica *Action* e permettono una comunicazione chiara ed efficiente tra le due parti.
Dato che mediante una socket possiamo inviare soltanto informazioni *binarie o testuali*, è necessario utilizzare uno strumento che ci permette di *serializzare* l’oggetto azione, trasformandolo in una stringa, questa operazione di codifica viene effettuata utilizzando la libreria *gson* di Google.
In particolare viene impiegato un *adapter (utils.MySerializer)* che permette di generare un *JSONElement* dove viene mantenuta traccia della tipologia originale dell’ oggetto che viene serializzato. In questo modo l’applicazione che riceve la stringa *serializzata* riesce a conoscere la tipologia dell’ oggetto serializzato permettendo così una *deserializzazione efficace.*



Ciò che viene effettivamente inviato mediante socket è una stringa che assomiglia alla seguente:

{"type":"game.action.JoinServer","data":{"username":"Mario Rossi"}}

Nell'attributo type notiamo la tipologia dell'oggetto di partenza mentre nell'attributo data è presente la rappresentazione degli attributi dell' oggetto.

Azioni Disponibili

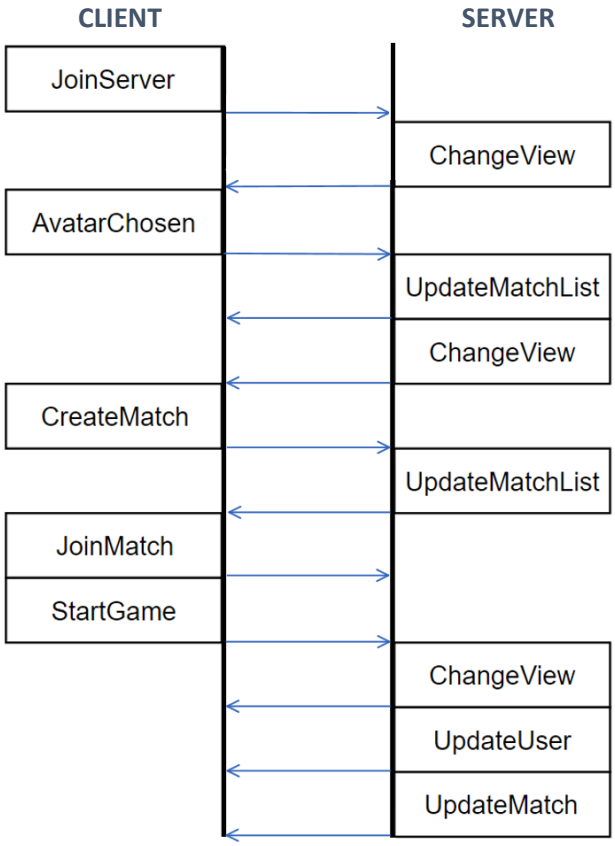
Le varie e possibili *Action* sono implementate in un package condiviso tra client e server *(game.action).* All'interno di questo package sono presenti diversi files che estendono una *classe astratta (game.action.Action)*. Ogni singola *azione* presente effettua l' *override* del metodo *doAction()* specificando cosa deve effettivamente succedere quando viene ricevuto un oggetto di quel tipo. Di seguito un esempio: l'azione **StartGame**.

```
1.  public class StartGame extends Action {
2.      /**
3.       * Creates payload
4.       */
5.      public StartGame() {
6.      }
7.
8.      @Override
9.      public void doAction(User user) throws ActionException {
10.         try {
11.             Match match = user.getMatch();
12.             GameLogic.getInstance().startMatch(match);
13.         } catch (Exception e) {
14.             GameLogic.getInstance().sendDangerMessageTo(user, e.getMessage());
15.             return;
16.         }
17.     }
18. }
```

ESEMPIO: LA FUNZIONE STARTGAME

Un oggetto **`StartGame`** viene inviato dal *client* al *server* quando il giocatore vuole iniziare a giocare.
Il server non appena riceve la rappresentazione dell' oggetto via socket la deserializza ed esegue il metodo *doAction()* riportato sopra.

Di seguito è riportato il flusso di azioni che avviene durante l'inizializzazione di una partita.



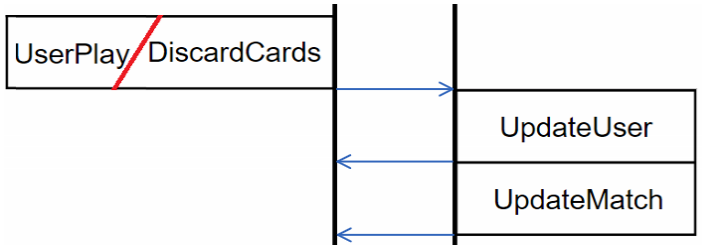
Il client invia una richiesta di ***JoinServer*** proponendo un *username;* se accettato il server, mediante un ***Change-View,*** modifica lo *stage* visualizzato nel client.

Successivamente il client notifica al server l'avatar scelto, il server risponde con le stanze disponibili nelle quali è possibile giocare.

L'utente, a sua discrezione, può decidere se entrare in una stanza (*match)* preesistente o creare una nuova partita. Quando un *match* termina o rimane senza giocatori viene automaticamente cancellato; ogni modifica alla lista delle stanze attive viene notificata a tutti gli utenti connessi mediante un oggetto ***UpdateMatchList***.

Quando il *match* ha raggiunto un numero sufficiente di utenti *(>=2),* l'utente può chiedere di iniziare a giocare.Il server, una volta ricevuto l'oggetto ***StartGame***, chiede a tutti i client di mostrare l'interfaccia di gioco ed invia ad ognuno di essi una copia aggiornata dell' oggetto *Utente* e dell' oggetto *Match.*

Durante il turno di gioco, ogni *player* può effettuare due tipologie di mossa: ***UserPlay*** oppure ***DischargeCards***:



- ***DiscardCards****:* viene inviata quando un utente vuole scartare quattro carte presenti nel suo mazzo dello stesso seme.
- ***UserPlay****:* si divide a sua volta nella possibilità di *dubitare* l'ultima mossa avvenuta o giocare nuove carte

Vi sono poche altre azioni non trattate nel diagramma:

- ***GameOver****:* notifica la fine di una partita.
- ***Croupier****:* si occupa dello spostamento grafico delle carte nei clients.
- ***Alert****:* mostra un avviso sullo schermo del giocatore.
- ***SendStats****: Utilizzata per inviare informazioni sullo stato del server via HTTP.*

Analisi del Model

Le varie *Actions* che clients e server si scambiano lavorano su diversi oggetti comuni ai due software contenuti nel package *game.model*.

Le classi contenute nel model comune sono estese all' interno del server-model (server.model), questa necessità deriva dal fatto che il server ha bisogno di molte più informazioni di quelle che effettivamente sono necessarie ai singoli clients; di seguito è discussa la struttura delle principali classi;

	ATTRIBUTI	

USER
CONTIENE LA VARIE INFORMAZIONI SUL GIOCATORE
	`String username;`
	`UserState userState;`
	`String avatarURL;`
	`List<Card> cards;`
	`SocketHandler socket;`
	`Match match;`
		Username del giocatore (*default: undefined)*
		Stato del player: (Lobby, Playing, Waiting)
		Nome del file avatar.
		Carte possedute dal giocatore.
		Socket attiva per raggiungere il client.
		Match al quale l'utente ha deciso di partecipare.

MATCH
RAPPRESENTA LA STANZA DA GIOCO DISPONIBILE NEL SERVER.
	`String name;`
	`MatchState matchState;`
	`User whoseTurn;`
	`boolean isFirstTurn;`
	`List<User> users;`
	`static List matches;`
	`List<Card> tableCardsList;`
		Nome della stanza.
		Stato del gioco (Playing, Waiting)
		Utente a cui tocca giocare.
		Indica se questo è il primo turno dopo che qualcuno ha appena perso.
		Lista degli utenti attivi nel match.
		Lista di tutti i Match (stanze) presenti nel server.
		Carte che sono state '*buttate*' *(Throw Card)* sul tavolo.

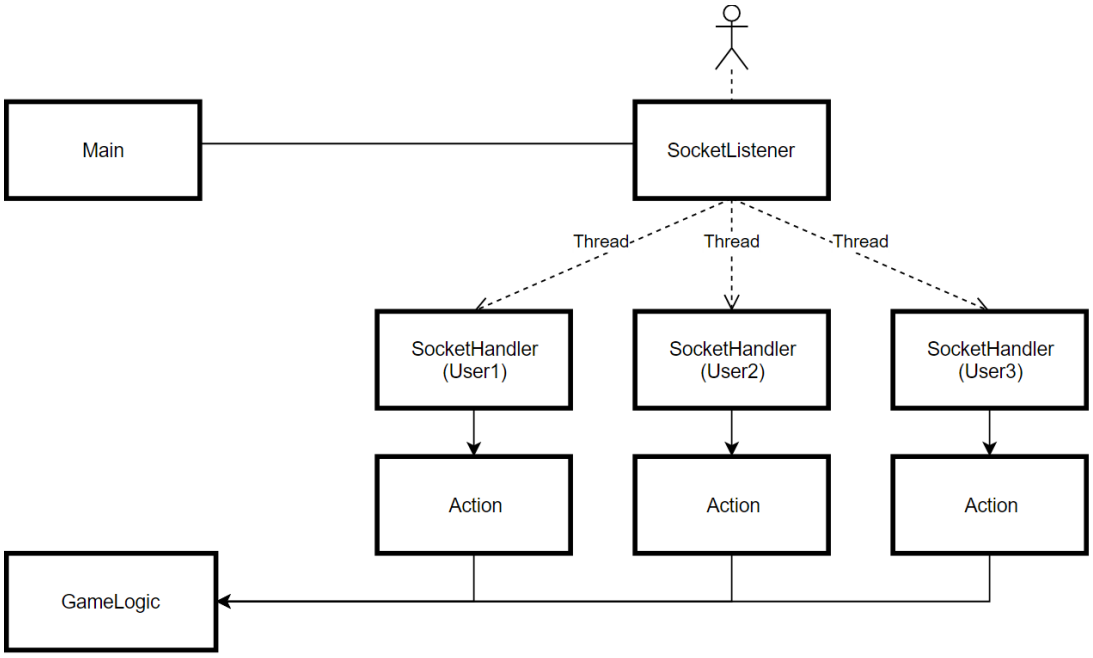
CARD
FORNISCE UN' ASTRAZIONE DELLA CARTA DA GIOCO
	`CardSuit cardSuit;`
	`CardType cardType;`
	`String locale;`
		Seme della carta (bastoni, coppe, etc)
		Valore della carta (asso, due, tre etc)
		Tipologia di carte con cui si vuole giocare.

DECK *CLASSE PRESENTE SOLO NEL SERVER, FORNISCE UN'ASTRAZIONE DEL MAZZO DI CARTE.*	`List cards`	Carte presenti nel mazzo, inizialmente 40, vengono distribuite agli utenti di un match mediante `giveCards(List<User> users)`

Gli attributi sottolineati sono estesi nel package server e sono soltanto disponibili per quest'ultimo package.

Architettura del Server

Gestione Socket



Il server, appena viene istanziato, resta in ascolto delle nuove connessioni; quando viene aperta una *nuova socket* viene creato un *thread* che si occupa della comunicazione con quel determinato client. SocketHandler, appena inizializzato, crea un *PrintWriter* ed uno *Scanner* per permettere la comunicazione in entrambi i sensi versi l'utente, viene inoltre creato un nuovo oggetto *User* che rappresenterà l'utente all'interno del gioco.

Quando viene ricevuta una stringa di testo dal client, *SocketHandler* utilizza *gson* per ricostruire l'oggetto ricevuto. I vari oggetti *Action* mediante override del metodo *doAction()* definiscono una sequenza di funzioni dell'oggetto ***GameLogic*** da invocare. Di seguito la porzione di SocketHandler che *deserializza ed esegue.*

```
1.  public void run() {
2.      try {
3.          while (true) {
4.              String line = in.nextLine();
5.              try {
6.                  Action oggetto = gson.fromJson(line, Action.class);
7.                  oggetto.doAction(this.user);
8.              } catch (Exception e) {
9.                  MyLogger.println("Invalid JSON: " + line);
10.             }
11.         }
12.     } catch (Exception e) {
13.         //Usually we get here if user disconnects
14.     } finally {
15.         GameLogic.getInstance().onUserDisconnect(this.user);
16.         //Close all open streams
17.         if (in != null) in.close();
18.         if (out != null) out.close();
19.     }
20. }
```

GameLogic

GameLogic si occupa dell'esecuzione delle varie azioni in gioco; questo oggetto è un *singleton* ed al suo interno mantiene una lista di tutti gli utenti connessi al server al fine di *housekeeping*.
Questa classe definisce al suo interno tutte le funzioni necessarie al corretto svolgimento del gioco stesso. Non sono necessari attributi con riferimento ai vari *match* in atto o alle carte in gioco poiché la classe è *stateless:* ogni richiesta viene trattata come transazione indipendente e non dipende da ciò che è successo prima. Tutta l'informazione riguardo allo stato del giocatore ed il riferimento al relativo *match* sono contenuti nell' oggetto *User* che viene passato come argomento ai metodi necessari.

Tra i metodi più importanti presenti in questa classe vi sono, per informazioni esaustive consultare JavaDoc.

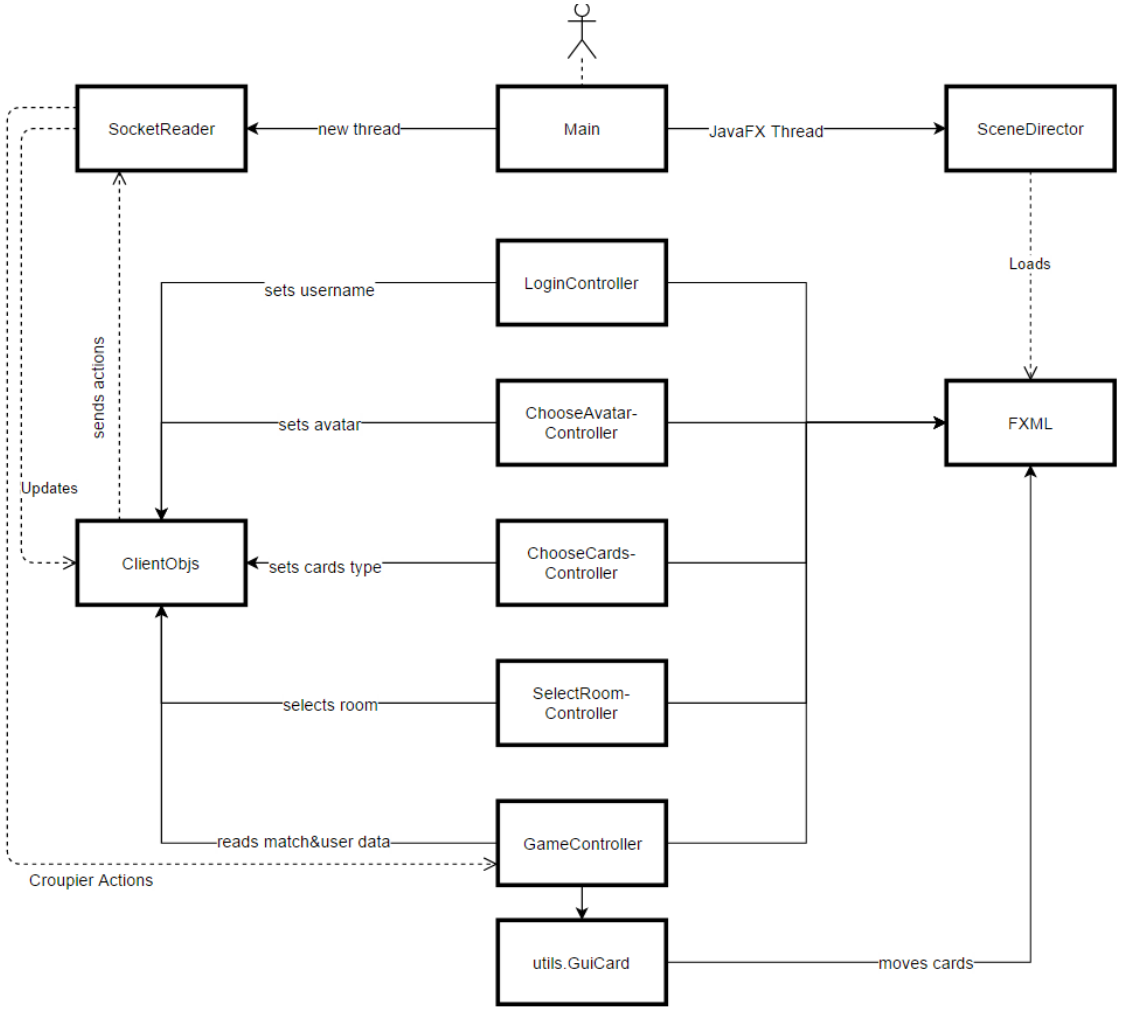
```
void executeDubitoPlay(User actualPlayer, UserPlay lastMove)
```

Questo metodo è chiamato quando l'utente *dubita* la mossa di un altro giocatore;
Il metodo non restituisce alcunché ma invia direttamente a tutti gli utenti del *match* un aggiornamento (Actions: *UpdateUser, UpdateMatch, Croupier)*

```
void moveCardsToTable(List<Card> cards, User source)
```

Questa funzione permette di spostare le carte dal mazzo dell'utente verso il tavolo da gioco. Il metodo si occupa di inviare agli utenti i rispettivi oggetti aggiornati e di avviare gli effetti grafici sui clients.

Client



Il client è stato realizzato utilizzando i componenti della libreria grafica **JFoenix,** all'interno del software distribuito agli utenti non vi è traccia della *logica di gioco:* ogni cosa che accade sull'interfaccia grafica dell'utente finale è decisa dal server in via remota; l'amministratore potrebbe quindi decidere di cacciare un utente dal gioco semplicemente utilizzando una azione *ChangeView.*

Tutti i dati ricevuti dal server vengono salvati nel singleton *ClientObjs* che funge da *database locale.*
Per alternare le cinque *views* presenti nel client viene utilizzato un singleton d'appoggio: **SceneDirector.**
Ogni volta che viene cambiata la view il nuovo controller collegato all' FXML si registra all'interno di **SceneDirector,** in questo modo si ha sempre un riferimento all'istanza del controller in esecuzione.

I vari *controllers* alterano i dati presenti all' interno di **ClientObjs** e li sincronizzano con il server mediante le già discusse *Actions.*
Il thread *SocketReader* compie l'operazione inversa: attende informazioni remote ed aggiorna i corrispettivi locali eseguendo i vari metodi *doAction()* degli oggetti ricevuti. Dato che è sempre presente un riferimento al *controller* collegato alla grafica, il server riesce ad invocare metodi da remoto gestendo lo spostamento di ogni carta mostrata sullo schermo.
Le carte sono mostrate sullo schermo mediante varie *ImageViews,* viene mantenuto un collegamento tra la carta mostrata graficamente e quella ricevuta dal server mediante una *hashmap.*