# GC10 Network

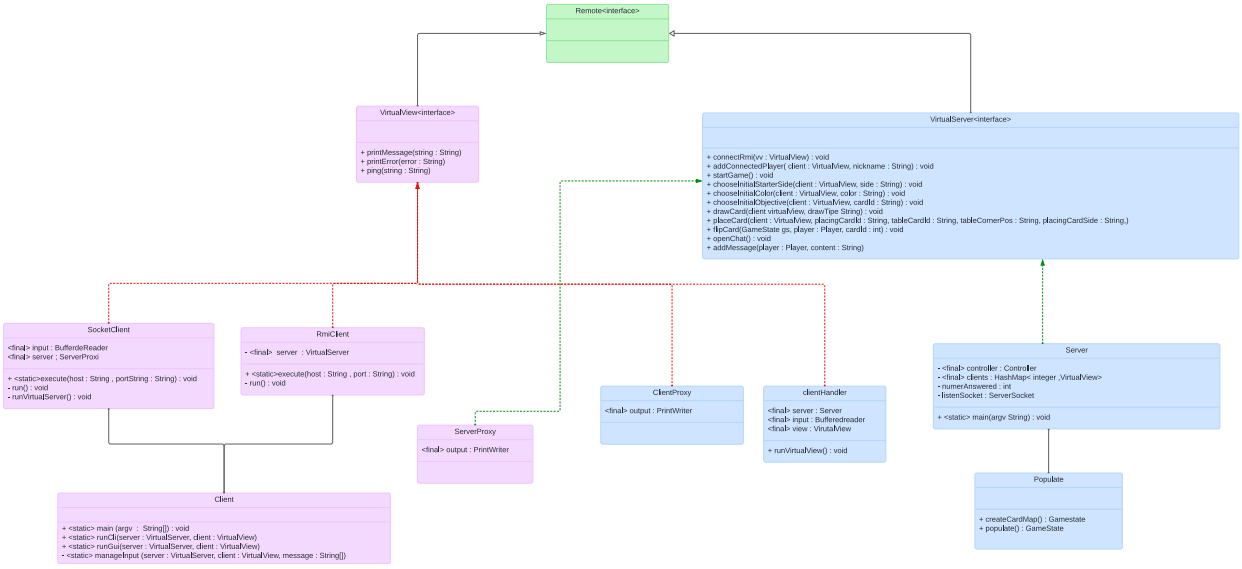
Riccardo Polelli Filippo Pozzi Federico Quartieri Giacomo Tessera

## Indice

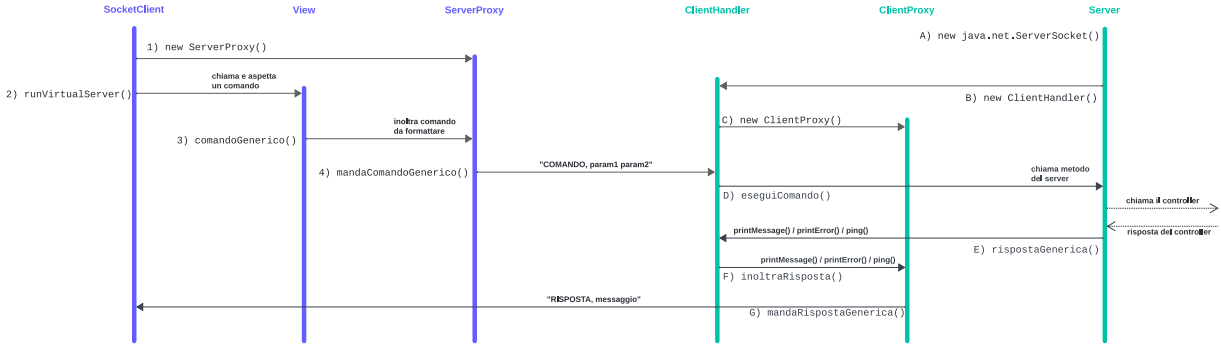
# Introduzione

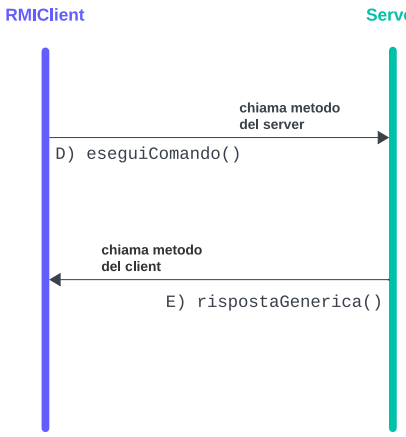
## Diagramma UML Classi Network

Iniziamo a mostrare le classi che abbiamo introdotto per gestire la comunicazione tra server e client.




## Flusso di Comunicazione



## Integrazione tra RMI e Socket

Per poter facilitare lo sviluppo della parte `Server`, le chiamate di metodi al `Controller` vengono *unificate*, sia che siano provenienti dal socket, sia che siano provenienti da RMI. Questo comportamento viene realizzato chiamando direttamente i metodi del `Server` dall'`RMIClient` nel caso di RMI e dal `ClientHandler` nel caso di socket, che, una volta ricevuto un nuovo comando nel suo buffer di ricezione, se nota che è un comando disponibile, chiamerà lo stesso identico metodo chiamato da RMI. Per questo motivo sia RMI che Socket utilizzano metodi che hanno come argomenti stringhe, le quali vengono convertite nelle strutture dati apposite nei metodi del `Server`, in modo da chiamare i metodi corrispondenti del `Controller`. Infatti, i metodi presenti nell'interfaccia `VirtualServer` sono gli stessi metodi del `Controller` con gli argomenti di tipo Stringa al posto delle strutture dati usate nel `Model`

## Formattazione dei messaggi

### Client to Server

I messaggi *Client to Server* sono del tipo: `"COMANDO, parametro1, ..., parametro"`.

- `COMANDO` indica il tipo di metodo che il client desidera chiamare
- I parametri del metodo sono codificati in formato di stringhe

Lista di comandi possibili: `ADDUSER`, `START`, `CHOOSESTARTER`, `CHOOSECOLOR`, `CHOOSEOBJECTIVE`, `PLACECARD`, `DRAWCARD`, `FILPCARD`, `OPENCHAT`, `ADDMESSAGE`.

### Server to Client

Analogamente, nella comunicazione *Server to Client* i messaggi sono formattati come: `"COMANDO, MESSAGGIO"`.

- `COMANDO` specifica se la comunicazione è riferita ad un messaggio generico, di errore o di ping
- `MESSAGGIO` è codificato in forma di stringa

Lista di comandi possibili: `MESSAGE` , `ERROR` , `PING`.

- `MESSAGE` : serve ad inviare alla view un feedback positivo (aggiornamento del model)
- `ERROR` : serve ad inviare alla view un errore
- `PING` : serve a verificare la connessione del client prima di inviargli un messaggio, ad un client viene inviato un ping prima di ogni messaggio effettivo, e vengono inviati dei ping periodici a tutti i client in modo che il `Model` sia a conoscenza dei giocatori connessi, per poter attribuire le fasi del gioco in modo corretto

## Server

- All'avvio, il Server istanzia un `Controller` del gioco e gestice la rete RMI e Socket, istanziando un `java.net.ServerSocket` per Socket e un RemoteObject per RMI sulla porte specificate dalla riga di comando. L'istanziazione del Controller avviene passandogli il `GameState`, istanziato con l'utilizzo di `populate()`, dove viene usato un JSON per generare le carte e tutto quello che serve al `GameState`
- Il `Server` si pone in attesa di una richiesta di associazione da parte di un client che ha intenzione di comunicare con il server sulla porta appena aperta
- Il `Server` salva i `Client` che si connettono, in ordine di arrivo, all'interno di una lista (condivisa sia per i client RMI che per quelli Socket). L'ordine dei `Client` di questa lista corrisponde all'ordine dei `Player` nella lista dei giocatori del model. Se un giocatore si disconnette prima di aver scelto il nickname, non viene incluso nella lista dei clients. Al contrario, se un giocatore si disconnette dopo aver scelto il nickname, rimane nella lista dei clients sia nel server che nel model. Le informazioni sulla sua connessione vengono conservate e aggiornate nel model. Il giocatore ha la possibilità di riconnettersi successivamente tramite il suo nickname.
- RMI
  - I metodi del `Server` sono messi a disposizione diretta da parte del client
- Socket
  - Se l'associazione ha successo, viene istanziato un `ClientHandler` che richiede il server corrente e due stream, uno di ricezione e uno di trasmissione (di conseguenza esiste un `ClientHandler` per ogni client)
    - il `ClientHandler`, quando viene istanziato, a sua volta istanzia il `ClientProxy`, che richiede lo stream di output per essere istanziato
    - lo stream di output del `ClientProxy` è utilizzato dal server per scrivere i dati, che attraverso il protocollo TCP/IP raggiungono il buffer di ricezione del `SocketClient`
  - Dopo l'istanziazione del `ClientHandler`, viene creato un Thread il cui compito è eseguire `ClientHandler.runVirtualView`, che legge dal suo stream di ricezione per individuare nuovi comandi da processare
  - Una volta ricevuti dei dati sul buffer, il `ClientHandler` controlla la corretta formattazione dei parametri e chiama il metodo del server corrispondente al comando ricevuto, in modo da semplificare l'integrazione con RMI, che utilizza lo stesso server.
  - Dopo aver ricevuto un comando, il Server chiama il metodo relativo al comando nel Controller
  - Nel caso di feedback da comunicare al client, il metodo `ClientHandler.printMessage(String)` viene chiamato. Questo metodo, a sua volta, chiama `ClientProxy.printMessage(String)`, il quale formatta il messaggio secondo il nostro protocollo proprietario e lo scrive sul suo stream di output. Successivamente, il messaggio viene inviato al client attraverso il protocollo TCP/IP. La stessa cosa avviene nel caso di errori da comunicare al client, con le chiamate di `ClientHandler.printError(String)` e `ClientProxy.printError(String)`

# Client

È presente un Client che, a seconda che l'utente abbia scelto di utilizzare Socket o RMI, chiama il metodo Execute della classe corrispondente

## Socket Client

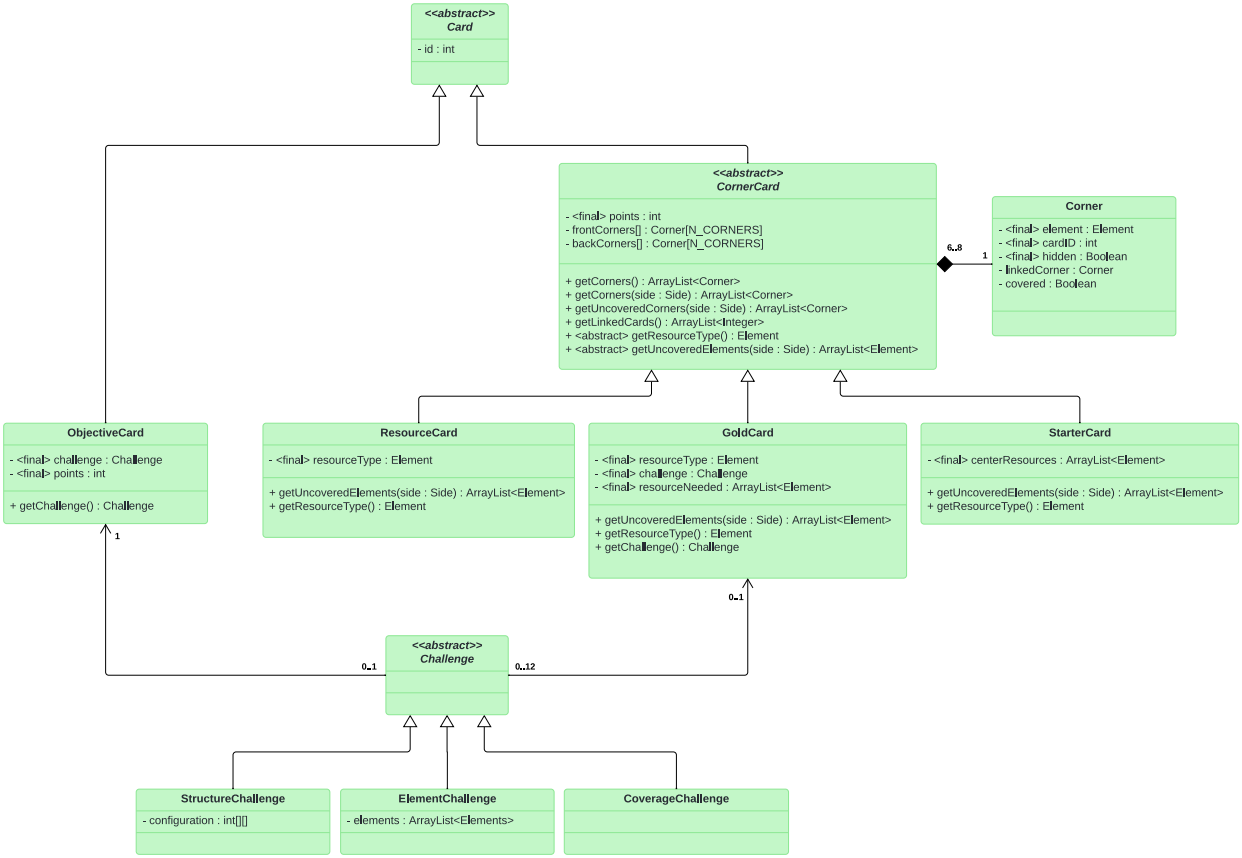
- Il `SocketClient` quando viene eseguito crea due buffer derivati da due stream: uno per la ricezione e uno per la trasmissione dei dati. Inoltre, istanzia un `ServerProxy`
  - Il `ServerProxy` richiede lo stream di trasmissione per scrivere le informazioni da inviare al server (al `ClientHandler` del server)
- Viene istanziato un `Thread` che esegue `SocketClient.runVirtualServer()`, responsabile dell'ascolto dei messaggi provenienti dal server
- Simultaneamente, il `SocketClient` avvia una delle due interfacce utente (CLI/TUI o GUI)
- la view riceve i comandi per giocare/creare un utente/... dall'utente
- Una volta riconosciuto un comando correttamente formattato, Il comando, insieme ai relativi parametri, viene scritto secondo una sintassi proprietaria sul buffer di uscita del `ServerProxy`
- Dopo la scrittura del comando, questo viene trasferito tramite il protocollo TCP/IP al buffer di ingresso del `ClientHandler` e sarà "catturato/letto" da `ClientHandler.runVirtualView`

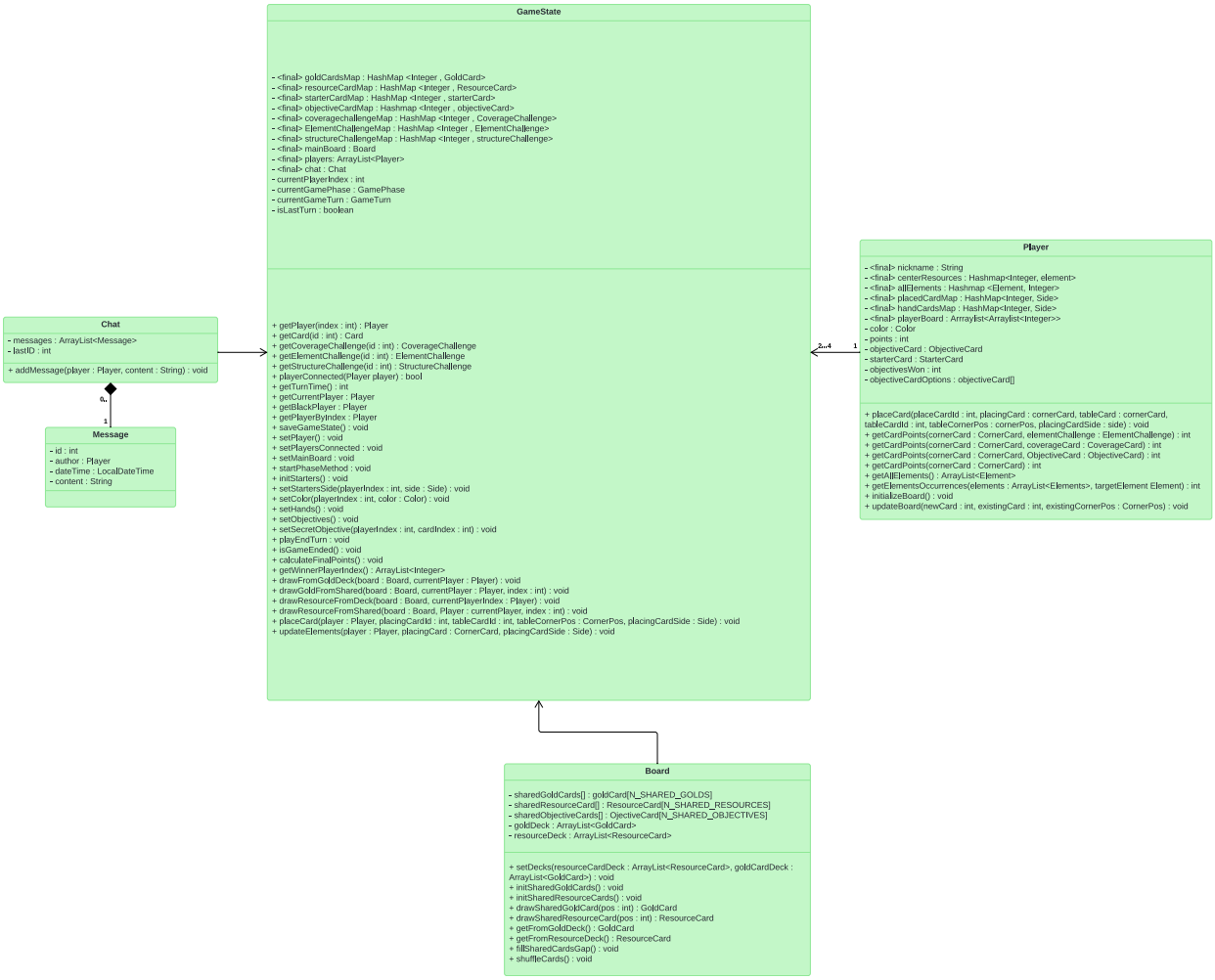
## RMI Client

Il client RMI come da definizione, cerca sull'indirizzo del server il registry corrispondente e una volta instaurata la connessione chiama la *view* corrispondente per interpretare i comandi dell'utente e chiamare i metodi del server corrispondenti.

# Appendice

## Diagramma UML Classi Model





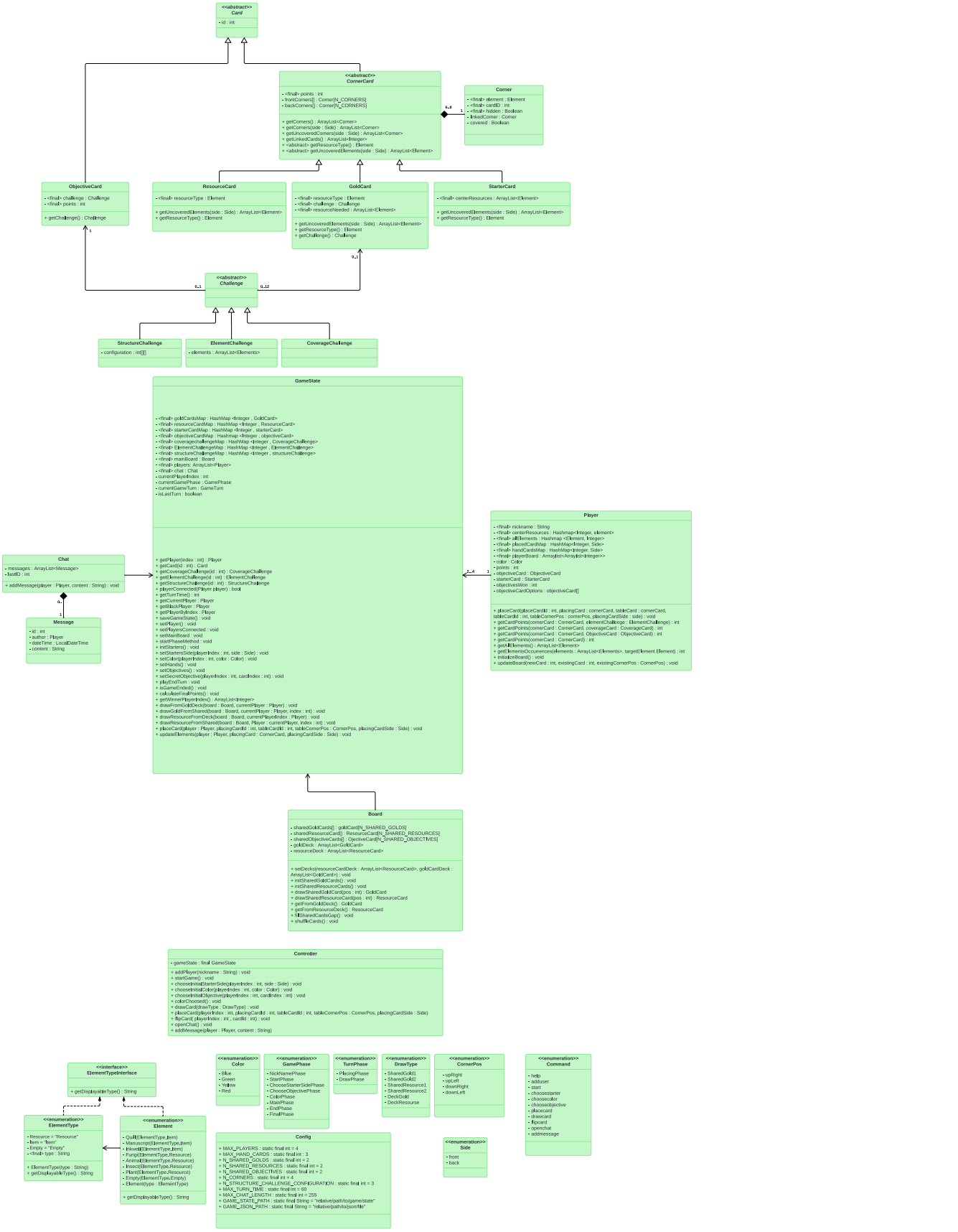

# Diagramma UML Classi Controller

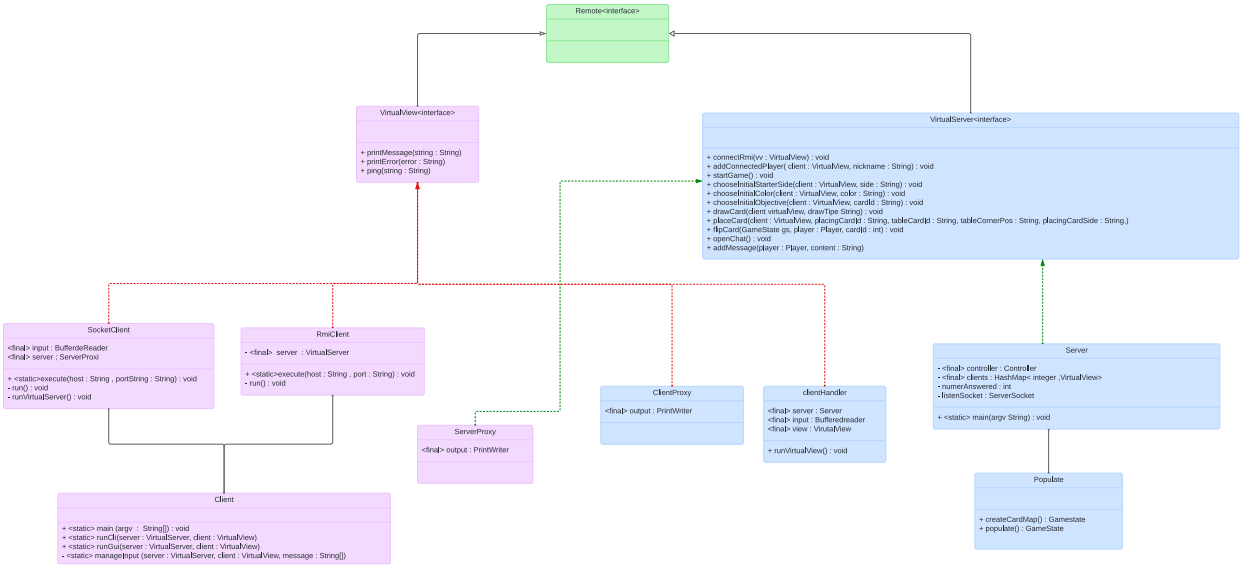
**Controller**

- gameState : final GameState

+ addPlayer(nickname : String) : void
+ startGame() : void
+ chooseInitialStarterSide(playerIndex : int, side : Side) : void
+ chooseInitialColor(playerIndex : int, color : Color) : void
+ chooseInitialObjective(playerIndex : int, cardIndex : int) : void
+ colorChoosed() : void
+ drawCard(drawType : DrawType) : void
+ placeCard(playerIndex : int, placingCardId : int, tableCardId : int, tableCornerPos : CornerPos, placingCardSide : Side)
+ flipCard( playerIndex : int , cardId : int) : void
+ openChat() : void
+ addMessage(player : Player, content : String)

# Diagramma UML Classi Completo




Remote<interface>
VirtualView<interface>
+ printMessage(string : String)
+ printError(error : String)
+ ping(string : String)
VirtualServer<interface>
+ connectRmi(vv : VirtualView) : void
+ addConnectedPlayer( client : VirtualView, nickname : String) : void
+ startGame() : void
+ chooseInitialStarterSide(client : VirtualView, side : String) : void
+ chooseInitialColor(client : VirtualView, color : String) : void
+ chooseInitialObjective(client : VirtualView, cardId : String) : void
+ drawCard(client virtualView, drawType String) : void
+ placeCard(client : VirtualView, placingCardId : String, tableCardId : String, tableCornerPos : String, placingCardSide : String,)
+ flipCard(GameState gs, player : Player, cardId : int) : void
+ openChat() : void
+ addMessage(player : Player, content : String)
SocketClient
<final> input : BufferdeReader
<final> server ; ServerProxi
+ <static>execute(host : String , portString : String) : void
- run() : void
- runVirtualServer() : void
RmiClient
- <final> server : VirtualServer
+ <static>execute(host : String , port : String) : void
- run() : void
ServerProxy
<final> output : PrintWriter
ClientProxy
<final> output : PrintWriter
clientHandler
<final> server : Server
<final> input : Bufferedreader
<final> view : VirutalView
+ runVirtualView() : void
Server
- <final> controller : Controller
- <final> clients : HashMap< integer ,VirtualView>
- numerAnswered : int
- listenSocket : ServerSocket
+ <static> main(argv String) : void
Populate
+ createCardMap() : Gamestate
+ populate() : GameState
Client
+ <static> main (argv : String[]) : void
+ <static> runCli(server : VirtualServer, client : VirtualView)
+ <static> runGui(server : VirtualServer, client : VirtualView)
- <static> manageInput (server : VirtualServer, client : VirtualView, message : String[])