

# PROVA FINALE ING. SOFTWARE

A.A. 2015/2016

*Sviluppo di una versione Software del gioco da Tavolo Council Of Four (Cof)*

# SOMMARIO

## Sommario

PRESENTAZIONE	1
SERVER	2
ARCHITETTURA NETWORK	14
CLIENT	22
View	24
CLI (Command Line Interface)	25
GUI	30
Informazioni	36
Informazioni sulla società	36

# COUNCIL OF FOUR

## PRESENTAZIONE

Il gioco è una versione distribuita del gioco da tavolo Council of Four.

Sono state implementate tutte le specifiche di base:

Il Client supporta la scelta del metodo di comunicazione (RMI o Socket) e la scelta dell'Interfaccia (CLI o GUI).

In base al metodo di comunicazione il Server comunicherà con il client mediante quella tecnologia.

Il gioco può ospitare fino a 10 client assieme (numero scelto prettamente a caso per avere un limite in quanto altrimenti si aveva il bisogno di creare nuove carte)

Contiene la possibilità di scegliere tra un numero finito di mappe (8, ossia le 8 permutazioni di ogni faccia di ogni regione). È stato comunque implementato un sistema di generazione di mappa: se data un nuovo file di configurazione con relativa immagine è possibile aggiungerla senza ulteriori cambiamenti.

Alla fine di ogni giro di turni viene aperto il market in cui si possono comprare e vendere carte politiche, carte permesso e aiutanti.

Implementata la regola addizionale per la partita con solo due giocatori.

Costruito un percorso della nobiltà random ogni volta che si avvia la partita e mostrato.

Implementati tutti i 10 bonus presenti nel vero gioco, da quelli trovabili sulle carte permesso a quelli nel percorso della nobiltà.

L'interfaccia grafica è totalmente resizable, in ogni suo aspetto.

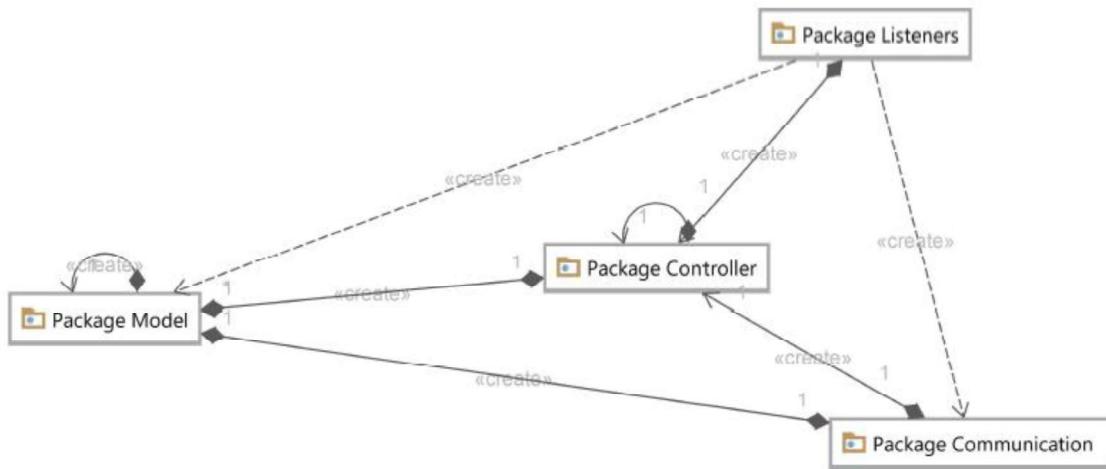
Abbiamo preferito costruire un gioco sulla quale sia possibile il gioco e che possa soddisfare l'utente finale. Abbiamo, quindi, aumentato il valore possibile dei bonus per rendere più dinamico il gioco.

# COUNCIL OF FOUR

## SERVER

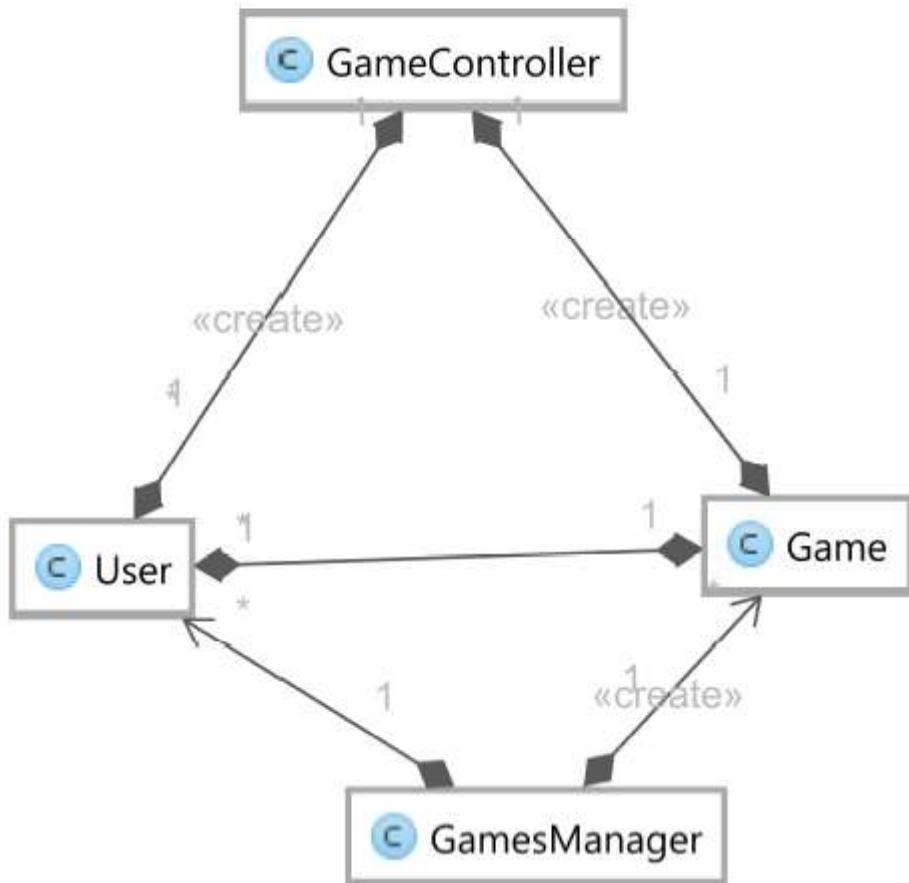
Il server è strutturato in 3 Package:

- Model
- Controller
- NetworkInterface



Il package Controller contiene la vera e propria logica del gioco. E' strutturato in questo modo:

# COUNCIL OF FOUR



## GAMESMANAGER

GamesManager è la classe che gestisce le varie partite.

E' un singleton e al momento della sua creazione avvia i Listeners (Socket e RMI). Quando un utente si connette decide se aggiungerlo ad una partita esistente (nel caso non sia ancora iniziata) oppure se creare una nuova partita. Quando un utente cerca di effettuare il login questo oggetto cerca se è già presente un altro utente con lo stesso nome, in questo caso manda un messaggio di errore perché lo username deve essere unico.

Questa classe è anche la classe incaricata di cancellare la partita quando questa è finita.

# COUNCIL OF FOUR

## GAMECONTROLLER

La classe GameController contiene la vera logica del gioco.

Gestisce la singola partita, quindi ogni partita ha il riferimento al proprio GameController e viceversa.

GameController quindi gestisce i turni, il market e manda gli aggiornamenti e gli eventi ai client.

Questa classe, gestendo i turni decreta anche il vincitore della partita.

## GAME

Game è l'oggetto che rappresenta la partita. Contiene i riferimenti a tutti gli oggetti della partita e a tutti gli utenti all'interno della partita. In questo modo si ottiene un punto di accesso semplificato a tutti gli oggetti.

Game viene modificato da GameController, dalle azioni e dai bonus. In questo modo è stato possibile distribuire la logica del gioco in più classi.

```
1. public class Game implements Serializable {
2.
3.     /**
4.      * True if game is full (There game is started and the players are playing)
5.      */
6.     private boolean started;
7.
8.     /**
9.      * All users in the game with their name
10.     */
11.    private HashMap<String, User> usersInGame = new HashMap<>();
12.
13.    /**
14.     * All cities in undirectedgraph
15.     */
16.    private Map map;
17.    private HashMap<RegionName, Region> regions = new HashMap<>();
18.
19.    /**
20.     * King
21.     */
22.    private King king;
23.
24.    // PATHS
25.    private VictoryPath victoryPath;
26.    private NobilityPath nobilityPath;
27.    private MoneyPath moneyPath;
28.    private GameController gameController;
29.    private Bank bank;
30.
```

# COUNCIL OF FOUR

```
31.    // PERMIT DECK
32.    private HashMap<RegionName, PermitDeck> permitDecks;
33.
34.    // POLITIC CARD
35.    private PoliticDeck politicCards;
36.    private HashMap<RegionName, RegionBonusCard> regionBonusCard = new HashMap<>();
37.
38.    private HashMap<String, ColorBonusCard> colorBonusCard = new HashMap<>();
39.    private Stack<KingBonusCard> kingBonusCards = new Stack<>();
40.
41.    //list of buyable wrapper, all object that user can buy
42.    private ArrayList<BuyableWrapper> marketList = new ArrayList<>();
43.
44.    public Game() {
45.        this.started = false;
46.        gameController = new GameController(this);
47.        gameController.startTimer();
48.        //create Region
49.        bank = new Bank();
50.        createRegion();
51.        createDecks();
52.        // create nobility, victory and moneyPath
53.        createPaths();
54.        //create regionBonusCard, kingBonusCards, colorBonusCard
55.        createBonusDeck();
56.        //create Politic Card Deck
57.        createPoliticCards();
58.
59.    }
```

Nel costruttore vengono creati tutti gli oggetti necessari per il gioco.

## AZIONI

Ogni azione che il client può fare corrisponde ad una classe e tutte estendono la classe astratta action.

Quindi quando il server riceve un'azione dal client non deve fare altro che chiamare il metodo doAction sull'azione e sarà lei a implementare la propria logica.

In questo modo il server si disinteressa del tipo dinamico dell'azione e dei particolari controlli in quanto essa si "autocontrolla".

```
1.  public abstract class Action implements Serializable {
2.
3.      protected String actionPerformed;
4.
5.      // real behaviour of action
6.      public abstract void doAction(Game game, User user) throws ActionNotPossibleException;
7.
```

# COUNCIL OF FOUR

```
8.     boolean checkActionCounter(User user) throws ActionNotPossibleException {
9.         switch (actionType) {
10.             case Constants.FAST_ACTION:
11.                 if (user.getFastActionCounter() <= 0) {
12.                     throw new ActionNotPossibleException("Non hai azioni veloci!");
13.                 } else return true;
14.             case Constants.MAIN_ACTION:
15.                 if (user.getMainActionCounter() <= 0) {
16.                     throw new ActionNotPossibleException("Non hai azioni principali!");
17.                 } else return true;
18.             default:
19.                 throw new ActionNotPossibleException("Azione non possibile");
20.         }
21.
22.     }
23.
24.     void removeAction(Game game, User user) {
25.         switch (actionType) {
26.             case Constants.MAIN_ACTION:
27.                 user.setMainActionCounter(user.getMainActionCounter() - 1);
28.                 break;
29.             case Constants.FAST_ACTION:
30.                 user.setFastActionCounter(user.getFastActionCounter() - 1);
31.                 break;
32.         }
33.         // send a snapshot to all player
34.         game.getGameController().sendSnapshotToAll();
35.     }
36.
37.     protected void removePoliticCard(ArrayList<PoliticCard> politicCards, User user,
38.                                     Game game) {
39.         for (int i = 0; i < politicCards.size(); i++) {
40.             for (int j = 0; j < user.getPoliticCards().size(); j++) {
41.                 if (politicCards.get(i).equals(user.getPoliticCards().get(j))) {
42.                     game.removeFromMarketList(new BuyableWrapper(user.getPoliticCards().get(j),
43.                                                               user.getUsername()));
44.                     user.getPoliticCards().remove(j);
45.                     user.decrementPoliticCardNumber();
46.                     break;
47.                 }
48.             }
49.
50.     protected void checkRegionBonus(City city, User user, Game game) throws Action
51.     NotPossibleException {
52.         if (game.getRegion(city.getRegion()).checkRegion(user.getUsersEmporium()))
53.     {
54.         game.getRegionBonusCard(city.getRegion()).getBonus(user, game);
55.         // check king bonus and get it
56.         KingBonusCard kingBonusCard = game.getKingBonusCard();
57.         if (kingBonusCard != null) {
58.             kingBonusCard.getBonus(user, game);
59.         }
60.     }
61.
```

# COUNCIL OF FOUR

```
60.
61.     protected void checkColorBonus(City city, User user, Game game) throws ActionN
62.         otPossibleException {
63.             if (city.getColor().checkColor(user.getUsersEmporium())) {
64.                 game.getColorBonusCard(city.getColor()).getBonus(user, game
65.             );
66.             // check king bonus and get it
67.             KingBonusCard kingBonusCard = game.getKingBonusCard();
68.             if (kingBonusCard != null) {
69.                 kingBonusCard.getBonus(user, game);
70.             }
71.         }
72.
73.     protected int calculateMoney(int correctPoliticCard, ArrayList<PoliticCard> po
74.         liticCards, int bonusCounter) throws ActionNotPossibleException {
75.         // calculate multicolor:
76.         int bonusNumber = 0;
77.         for (PoliticCard politicCard : politicCards)
78.             if (politicCard.isMultiColor()) {
79.                 bonusNumber++;
80.             }
81.         // calculate money
82.         int newPositionInMoneyPath = 0;
83.         if (correctPoliticCard == politicCards.size()) {
84.             if (correctPoliticCard < Constants.FOUR_PARAMETER_BUY_PERMIT_CARD && c
85.                 orrectPoliticCard > 0)
86.                 newPositionInMoneyPath = Constants.TEN_PARAMETER_BUY_PERMIT_CARD -
87.                     3 * (correctPoliticCard - Constants.ONE_PARAMETER_BUY_PERM
88.                         IT_CARD);
89.             else if (correctPoliticCard == Constants.FOUR_PARAMETER_BUY_PERMIT_CAR
90.                 D)
91.                 newPositionInMoneyPath = 0;
92.                 newPositionInMoneyPath += bonusNumber;
93.             }
94.             return newPositionInMoneyPath;
95.     }
96.
97.     protected int countCorrectPoliticCard(GotCouncil gotCouncil, ArrayList<Politic
98.         Card> politicCards, int bonusCounter) {
99.         int correctPoliticCard = 0;
100.        // count all correct and bonus card
101.        Queue<Councilor> council = gotCouncil.getCouncil().getCouncil();
102.        for (PoliticCard politicCard : politicCards) {
103.            if (politicCard.isMultiColor()) {
104.                bonusCounter++;
105.                correctPoliticCard++;
106.            } else {
107.                for (Councilor councilor : council) {
108.                    if (councilor.getColor().equals(politicCard.getPoliticC
109.                        olor())) {
110.                            correctPoliticCard++;
111.                            council.remove(councilor);
112.                        }
```

# COUNCIL OF FOUR

```
108.                     break;
109.                 }
110.             }
111.         }
112.     }
113.     return correctPoliticCard;
114. }
```

Le classi che estendono Action fanno Override del metodo doAction e implementano la loro logica. La classe Action fornisce metodi comuni a più azioni.

## BONUS

La struttura dei bonus è simile a quella delle azioni.

Tutti i bonus estendono l'interfaccia Bonus che espone il metodo getBonus.

Quindi quando in qualunque momento si ha bisogno di ricevere un bonus non si fa altro che chiamare il metodo getBonus e l'implementazione dipende dal tipo di bonus.

```
1. public interface Bonus {
2.
3.     void getBonus(User user, Game game) throws ActionNotPossibleException;
4.
5.     String getBonusName();
6.
7.     ArrayList<Bonus> getBonusArrayList();
8.
9.     ArrayList<String> getBonusURL();
10.
11.    ArrayList<String> getBonusInfo();
12.
13.
14. }
```

Inoltre siccome ci sono casi in cui possono presenti più bonus abbiamo implementato anche la classe MainBonus che contiene un array di bonus e implementa Bonus, quando viene chiamato il suo getBonus essa chiama getBonus su tutti i suoi bonus.

```
1. @Override
2.     public void getBonus(User user, Game game) throws ActionNotPossibleException {
3.
4.         for (Bonus bonus : bonusArrayList) {
5.             bonus.getBonus(user, game);
6.         }
7.     }
```

# COUNCIL OF FOUR

Per le azioni e i bonus abbiamo utilizzato il Pattern **Strategy** incapsulando all'interno dell'oggetto l'algoritmo da eseguire.

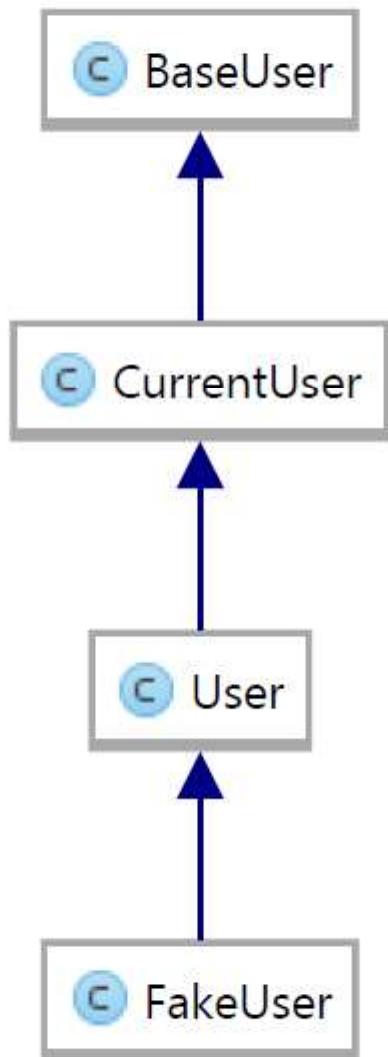
## USER

User è la classe che rappresenta gli utenti.

Contiene tutte le informazioni e gli oggetti corrispondenti agli utenti.

La gerarchia degli utenti è questa:

# COUNCIL OF FOUR



BaseUser è l'utente base con visibilità molto limitata. Viene utilizzato per mandare informazioni sommarie e non dare visibilità delle cose più importanti.

CurrentUser rappresenta l'utente all'interno del client.

User è l'utente come viene visto dal server quindi ha tutti gli oggetti associati a quel determinato utente.

FakeUser invece rappresenta uno User finto. E' stato necessario per l'implementazione di una regola che abbiamo notato a progetto già inoltrato.

La regola è la seguente:

## Preparazione di una partita per due giocatori

In una partita a 2 giocatori, dopo aver eseguito le normali azioni di preparazione, pescate a caso una tessera permesso per ogni regione, e posizionate un emporio (di uno dei colori non scelti dai giocatori) in ognuna delle città che compaiono nei permessi estratti. Rimescolate poi le tessere nelle pile di appartenenza.

*NB. Si possono avere da un minimo di 3 ad un massimo di 9 empori presenti sulla mappa a inizio partita.*

Per noi, avendo gli empori all'interno dell'utente risultava impossibile posizionare empori non appartenenti ai giocatori all'interno della partita. Per questo abbiamo implementato il FakeUser: agli occhi del server è uno User sempre disconnesso che non fa niente.

Anche in questo caso abbiamo quindi utilizzato il Pattern del null Object.

```
1. public class FakeUser extends User {  
2.  
3.     public FakeUser() {  
4.         this.usersEmporium = new ArrayList<>();  
5.     }  
6.  
7.     @Override  
8.     public boolean isConnected() {  
9.         return false;  
10.    }  
11.  
12.    @Override  
13.    public BaseCommunication getBaseCommunication() {  
14.        return new FakeCommunication();  
15.    }  
16.  
17.    @Override  
18.    public int getPoliticCardSize() {  
19.        return 0;  
20.    }  
21.  
22. }
```

In questo modo siamo riusciti a implementare la regola aggiuntiva.

## MAPPA

La mappa è rappresentata dalla classe Map.

```
1. public class Map implements Serializable {  
2.  
3.  
4.     private ArrayList<Link> links;
```

# COUNCIL OF FOUR

```
5.     private ArrayList<City> city;
6.     private String mapName;
7.     private String mapPreview;
8.     private String realMap;
9.
10.    private transient SimpleGraph<City, DefaultEdge> mapGraph = new SimpleGraph<>(
11.        DefaultEdge.class);
12.
13.
14.    private transient SimpleGraph<City, DefaultEdge> mapGraph = new SimpleGraph<>(
15.        DefaultEdge.class);
```

Essa contiene un ArrayList di Link che sono coppie di città collegate: links.

Un ArrayList di città che sono le città che appartengono alla mappa: city.

MapGraph contiene la vera e propria rappresentazione della mappa. E' un grafo costituito da archi non direzionali: in questo modo si riesce a riprodurre la topologia della mappa.

E' possibile creare una topologia a piacere salvando un file .json nella cartella ConfigurationFile.

Il gioco all'avvio carica tutte le mappe salvate nei file e tramite il contenuto dei file riproduce la mappa. Per creare una nuova mappa è quindi sufficiente aggiungere il file di configurazione e aggiungere le immagini della preview e della mappa reale. Si possono configurare i link tra le città, i colori delle città e le regioni delle città.

L'utilizzo di SimpleGraph ha permesso di implementare un semplice algoritmo di visita tramite la classe CityVisitor utilizzando il [Pattern Adapter](#).

```
1. public class CityVisitor {
2.
3.     private UndirectedGraph<City, DefaultEdge> cities = new SimpleGraph<City, Defa
4.         ultEdge>(DefaultEdge.class);
5.     private HashMap<City, Boolean> alreadyVisited = new HashMap<>();
6.     private ArrayList<City> usersEmporium;
7.     private NeighborIndex neighborIndex;
8.
9.     public CityVisitor(UndirectedGraph<City, DefaultEdge> cities, ArrayList<City>
10.         usersEmporium) {
11.         //this.usersEmporium = user;
12.         this.cities = cities;
13.         this.usersEmporium = usersEmporium;
14.         neighborIndex = new NeighborIndex(cities);
15.     }
16.
17.     public ArrayList<City> visit(City city) {
18.         ArrayList<City> visitedCity = new ArrayList<>();
19.         alreadyVisited.put(city, true);
20.         for (Object city1 : neighborIndex.neighborListOf(city)) {
21.             City realCity = (City) city1;
22.             if (!alreadyVisited.containsKey(realCity) && usersEmporium.contains(re
23.                 alCity) && !realCity.equals(city)) {
```

# COUNCIL OF FOUR

```
21.          alreadyVisited.put(realCity, true);
22.          visitedCity.add(realCity);
23.          visitedCity.addAll(visit(realCity));
24.      } else {
25.      }
26.  }
27.  return visitedCity;
28.
29.
30. }
```

City Visitor viene creato ogni volta che si vuole ricevere le città di proprietà di un utente collegate ad una città appena comprata. L'algoritmo utilizza il metodo neighborListOf per ottenere tutte le città collegate ad una città.

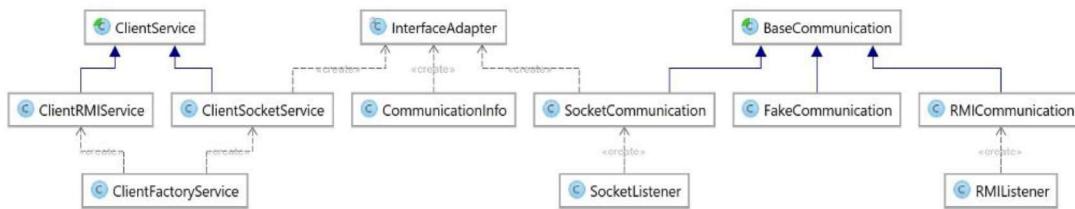
# COUNCIL OF FOUR

## ARCHITETTURA NETWORK

L'architettura network del progetto è stata fortemente influenzata dalle specifiche di.

L'utente deve poter scegliere il metodo di comunicazione con il server (Socket o RMI) all'avvio dell'applicazione.

Per questo è necessario, lato Server, avere due listener in grado di accettare connessioni socket o RMI. Nel caso del listener per la connessione Socket questo crea un thread dedicato alla comunicazione con l'utente, nel caso del listener per la connessione RMI questo crea un oggetto remoto ad-hoc per l'utente. Nel particolare questa è la struttura network del client (sinistra) e del server (destra):



Come si può notare il server ha due Listener che creano le comunicazioni Socket e RMI.

Le classi di comunicazione hanno al loro interno un riferimento all'utente con cui sono connesse, in questo modo si riesce a capire quale utente ha mandato quel messaggio.

## LATO SERVER

### LISTENER

Come già detto in precedenza i Listener sono oggetti che creano le comunicazioni reali e servono per avere un punto di accesso unico per tutti i client.

SocketListener ha una struttura molto semplice:

```
1. @Override
2.     public void run() {
3.         Socket clientSocket;
4.         ExecutorService executorService = Executors.newCachedThreadPool();
5.         while (true) {
6.             try {
7.                 clientSocket = serverSocket.accept();
8.                 SocketCommunication socketCommunication = new SocketCommunication(c
9.                     lientSocket);
10.                User user = new User(socketCommunication, gamesManager);
11.                socketCommunication.setUser(user);
```

# COUNCIL OF FOUR

```
11.             gamesManager.AddToUsers(user);
12.             executorService.execute(socketCommunication);
13.         } catch (IOException e) {
14.             e.printStackTrace();
15.         }
16.     }
17. }
```

Implementa Runnable perché deve essere sempre in esecuzione. Quando riceve una connessione esso crea un utente con la comunicazione SocketCommunication e un thread dedicato alla comunicazione con quel client.

RMIListener ha una struttura ancora più semplice:

```
1. public class RMIListener implements RMIListenerInterface {
2.
3.     private int clientNumber = 0;
4.     private GamesManager gamesManager;
5.
6.     public RMIListener(GamesManager gamesManager) throws RemoteException {
7.         UnicastRemoteObject.exportObject(this, 0);
8.         this.gamesManager = gamesManager;
9.     }
10.
11.    @Override
12.    public String Connect() {
13.        String name = "ClientHandler" + clientNumber;
14.        System.out.println("ClientPackage connected in RMI");
15.        try {
16.            RMIClientHandler rmiHandler = new RMICommunication(name);
17.            Registry registry = LocateRegistry.getRegistry();
18.            registry.rebind(name, rmiHandler);
19.            User user = new User((BaseCommunication) rmiHandler, gamesManager);
20.            ((BaseCommunication) rmiHandler).setUser(user);
21.            gamesManager.AddToUsers(user);
22.            clientNumber++;
23.        } catch (RemoteException e) {
24.            e.printStackTrace();
25.        }
26.        return name;
27.    }
}
```

Implementa RMIListenerInterface che è un'interfaccia condivisa tra client e server per la comunicazione.

Il client che vuole connettersi chiama il metodo connect quindi il server binda sul registro un oggetto di tipo RMIClientHandler che si occupa della comunicazione con quel preciso client e restituisce il nome dell'oggetto remoto al client.

## COMMUNICATION

Le classi di comunicazione lato server sono tre: SocketCommunication, RMICommunication e FakeCommunication.

# COUNCIL OF FOUR

Implementano tutte la classe astratta BaseCommunication in modo che lato server sia trasparente il tipo di comunicazione utilizzata per da quel client.

```
1. public abstract class BaseCommunication {  
2.  
3.     public abstract void setUser(User user);  
4.  
5.     public abstract void sendSnapshot(SnapshotToSend snapshotToSend);  
6.  
7.     public abstract void changeRound();  
8.  
9.     public abstract void sendAvailableMap(ArrayList<Map> availableMaps);  
10.  
11.    public abstract void sendSelectedMap(SnapshotToSend snapshotToSend);  
12.  
13.    public abstract void finishTurn();  
14.  
15.    public abstract void sendStartMarket();  
16.  
17.    public abstract void sendStartBuyPhase();  
18.  
19.    public abstract void disableMarketPhase();  
20.  
21.    public abstract void selectPermitCard();  
22.  
23.    public abstract void selectCityRewardBonus(SnapshotToSend snapshotToSend);  
24.  
25.    public abstract void moveKing(ArrayList<City> kingPath);  
26.  
27.    public abstract void sendMatchFinishedWithWin(ArrayList<BaseUser> finalSnapshot);  
28.  
29.    public abstract void ping();  
30.  
31.    public abstract void selectOldPermitCard();  
32.  
33.    public abstract void sendUserDisconnect(String username);  
34. }
```

Tramite queste classi il server comunica con il client e riceve informazioni da esso.

## SocketCommunication

La classe SocketCommunication è incaricata della comunicazione tramite socket.

La comunicazione avviene tramite oggetti di tipo CommunicationInfo serializzati tramite Gson.

```
1. public class CommunicationInfo {  
2.  
3.     //The code of the Request  
4.     private String code;
```

# COUNCIL OF FOUR

```
5.     //Information associated to the request
6.     private String info;
7.
8.     public CommunicationInfo(String code, String info) {
9.         this.code = code;
10.        this.info = info;
11.    }
12. }
```

L'attributo code rappresenta il codice della richiesta e tramite questo client e server sanno come interpretare in modo corretto l'informazione associata.

Info è a sua volta un oggetto serializzato tramite Gson.

Siccome Gson non supporta la deserializzazione di classi astratte o interfacce, cioè non si riesce a deserializzare correttamente oggetti sapendo solo l'interfaccia, abbiamo dovuto aggiungere degli oggetti InterfaceAdapter. Gson espone il metodo registerTypeAdapter(Type type, Object typeAdapter), in questo modo quando viene chiamato per effettuare serializzazione di un oggetto del tipo registrato utilizza l'oggetto typeAdapter per la serializzazione. Consente quindi di definire una propria politica di serializzazione.

```
1. /**
2.  * Interface for deserialize and serialize object implementing a common interface
3.  * Created by Emanuele on 19/05/2016.
4. */
5. public final class InterfaceAdapter<T> implements JsonSerializer<T>, JsonDeseriali
zer<T> {
6.
7.     /**
8.      * Called when you call gson.fromJson(object,yourClass.class)
9.      */
10.    @Override
11.    public T deserialize(JsonElement elem, Type type, JsonDeserializationContext c
ontext) throws JsonParseException {
12.        final JsonObject wrapper = (JsonObject) elem;
13.        final JsonElement typeName = get(wrapper, "type");
14.        final JsonElement data = get(wrapper, "data");
15.        final Type actualType = typeForName(typeName);
16.        return context.deserialize(data, actualType);
17.    }
18.
19.    /**
20.     * Called when you call gson.toJson(jsonObject)
21.     */
22.    @Override
23.    public JsonElement serialize(T object, Type type, JsonSerializerContext con
text) {
24.        final JsonObject wrapper = new JsonObject();
25.        wrapper.addProperty("type", object.getClass().getName());
26.        wrapper.add("data", context.serialize(object));
27.        return wrapper;
28.    }
29.
30.    /**
```

# COUNCIL OF FOUR

```
31.     * @param typeElem
32.     * @return the Type of the object
33.     */
34.    private Type typeForName(final JsonElement typeElem) {
35.        try {
36.            return Class.forName(typeElem.getAsString());
37.        } catch (ClassNotFoundException e) {
38.            throw new JsonParseException(e);
39.        }
40.    }
41.
42.    /**
43.     * @return a member of json object
44.     */
45.    private JsonElement get(final JsonObject wrapper, String memberName) {
46.        final JsonElement elem = wrapper.get(memberName);
47.        if (elem == null)
48.            throw new JsonParseException("no '" + memberName + "' member found in
what was expected to be an interface wrapper");
49.        return elem;
50.    }
51. }
```

In questo modo i socket emulano il comportamento di rmi che mantiene il tipo dinamico dell'oggetto passato come parametro.

Socket communication è un oggetto che implementa Runnable e sta continuamente in ascolto sullo stream di input. In questo modo è semplice capire quando un client si disconnette in quanto viene lanciata un'eccezione.

## RMICommunication

RMICommunication estende BaseCommunication e implementa RMIClientHandler che definisce i metodi che possono essere chiamati dal client.

Al suo interno ha un attributo di tipo RMIClientInterface che rappresenta l'oggetto remoto esportato dal client e tramite il quale è possibile chiamare i metodi del client (per esempio per notificare qualcosa).

Tutti gli oggetti inviati tramite RMI devono implementare Serializable e avere quindi un costruttore vuoto per permettere la deserializzazione.

RMI non consente di sapere quando un client effettua la disconnessione, per questo la classe RMICommunication ha un metodo ping() che viene chiamato per testare la connessione con il client. Lato client questo metodo è vuoto. Se il client è connesso quindi non succede nulla mentre se il client è disconnesso viene lanciata una RemoteException e parte la routine di disconnessione.

## FakeCommunication

FakeCommunication è una comunicazione finta necessaria per l'utente di tipo FakeUser e viene utilizzata anche nei Test.

# COUNCIL OF FOUR

Abbiamo utilizzato il Pattern **null Object**:

Gli utenti che non necessitano di una vera connessione (utenti finti) utilizzano un oggetto di tipo FakeCommunication. Invece che mettere a null l'oggetto BaseCommunication ed effettuare il check ogni volta che si tenta di invocare un metodo si utilizza la FakeCommunication. Essa estende BaseCommunication e fornisce un'implementazione vuota di tutti i metodi.

## LATO CLIENT

Lato client gli oggetti incaricati della comunicazione con il server vengono istanziati tramite il FactoryMethod. In base al parametro passato a questo metodo esso ritorna un oggetto di tipo ClientSocketService o ClientRMIService. Entrambi implementano l'interfaccia ClientService, in questo modo risulta trasparente al Controller del client il tipo di comunicazione utilizzata.

```
1. public class ClientFactoryService {  
2.  
3.     /**  
4.      * @param method RMI or Socket  
5.      * @param serverIP ip of server  
6.      * @param clientController client logic  
7.      * @return ClientService for communication with server  
8.      * @throws RemoteException  
9.      * @throws NotBoundException  
10.     */  
11.    public static ClientService getService(String method, String serverIP, ClientController clientController) throws RemoteException, NotBoundException {  
12.        if (method.equals(Constants.RMI)) {  
13.            return new ClientRMIService(Constants.SERVER, serverIP, clientController);  
14.        } else {  
15.            return new ClientSocketService(serverIP, clientController);  
16.        }  
17.    }  
18. }
```

### ClientRMIService

ClientRMIService estende ClientService ed implementa RMIClientInterface, interfaccia comune tra client e server, tramite la quale quest'ultimo può chiamare i metodi del client, per notifiche push per esempio.

Nel costruttore viene effettuata la lookup dell'oggetto remoto e viene chiamato il metodo UnicastRemoteObject.exportObject(this, 0) in modo da rendere l'oggetto esportabile.

# COUNCIL OF FOUR

Nella connect viene chiamata la connect dell'oggetto remoto del server che ritorna il nome dell'oggetto remoto dedicato a quel client, in questo modo il client fa una nuova lookup per il suo oggetto dedicato.

Successivamente viene chiamato il metodo sendRemoteClientObject in modo che sia possibile chiamare i metodi del client da server.

```
1. public class ClientRMIService extends ClientService implements RMIClientInterface
{
2.
3.     private String serverName;
4.     private RMIListenerInterface rmilistenerInterface;
5.     private Registry registry;
6.     private String rmiHandlerName;
7.     private RMIClientHandler rmiClientHandler;
8.     private ClientController clientController;
9.     private ExecutorService executorService = Executors.newCachedThreadPool();
10.
11.    ClientRMIService(String serverName, String serverIP, ClientController clientCo
ntroller) throws RemoteException, NotBoundException {
12.        this.serverName = serverName;
13.        this.clientController = clientController;
14.        registry = LocateRegistry.getRegistry(serverIP, Constants.RMI_PORT);
15.        rmilistenerInterface = (RMIListenerInterface) registry.lookup(serverName);
16.
17.        UnicastRemoteObject.exportObject(this, 0);
18.    }
19.
20.    /**
21.     *
22.     * @return true if connected
23.     */
24.    @Override
25.    public boolean Connect() {
26.        try {
27.            rmiHandlerName = rmilistenerInterface.Connect();
28.            rmiClientHandler = (RMIClientHandler) registry.lookup(rmiHandlerName);
29.
30.            rmiClientHandler.sendRemoteClientObject(this);
31.            return true;
32.        } catch (RemoteException e) {
33.            e.printStackTrace();
34.        } catch (NotBoundException e) {
35.            e.printStackTrace();
36.        }
37.        return false;
37.    }
37. }
```

Tutti gli altri metodi di comunicazione (qui non riportati) vengono eseguiti all'interno di un esecutore in modo che non vi siano rallentamenti dell'applicazione.

# COUNCIL OF FOUR

## ClientSocketService

Questa classe è molto simile alla classe SocketCommunication lato server. Gli oggetti vengono mandati e interpretati allo stesso modo per mantenere la simmetria. Anche questa classe implementa Runnable in modo da stare continuamente in lettura sullo stream di input.

I messaggi del server vengono decodificati in un DecoderTask (nested class che implementa Runnable) in modo da poter leggere anche più messaggi in successione senza subire rallentamenti.

```
1. @Override
2.     public void run() {
3.         System.out.println("ClientSocketService Started");
4.         String line;
5.         try {
6.             while ((line = in.readLine()) != null) {
7.                 // create a new runnable and use a executor service in order to ex
ecute this task
8.                 class DecoderTask implements Runnable {
9.                     private String lineToDecode;
10.
11                     private DecoderTask(String line) {
12                         this.lineToDecode = line;
13                     }
14.
15                     @Override
16                     public void run() {
17                         decodeInfo(lineToDecode);
18                     }
19.                 }
20.                 DecoderTask decoderTask = new DecoderTask(line);
21.                 executorService.execute(decoderTask);
22.             }
23.         } catch (IOException e) {
24.             e.printStackTrace();
25.         }
26.     }
```

# COUNCIL OF FOUR

## CLIENT

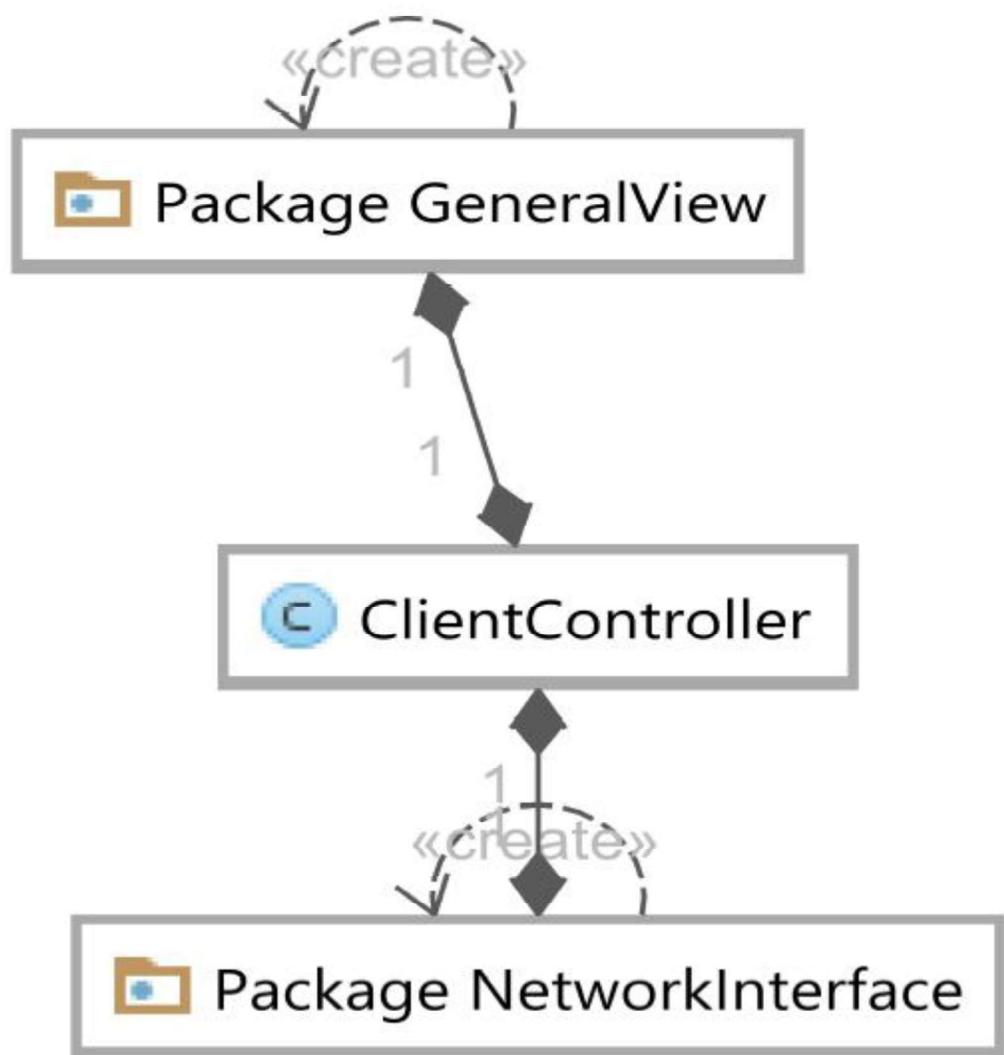


Figura 1 UML Client Alto Livello

Il client è strutturato in tre package dalle funzionalità distinte:

# COUNCIL OF FOUR

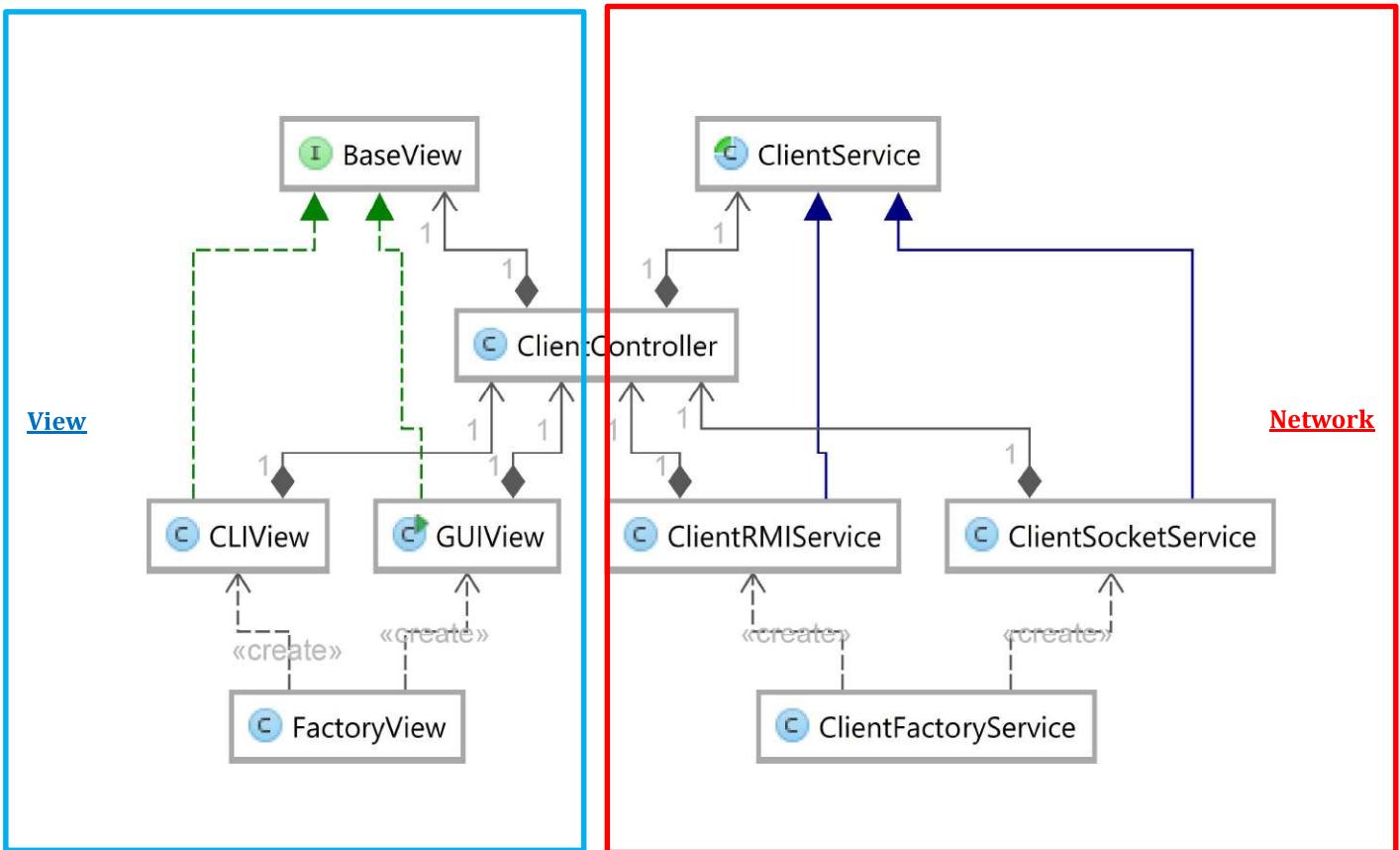


Figura 2 View Client

Figura 3 - Network Client

Il package View e il package network sono perfettamente distinte e comunicano mediante il ClientController che manda a server le azioni dell'utente e notifica alla View i cambiamenti del model.

Abbiamo utilizzato il pattern **MVC**:

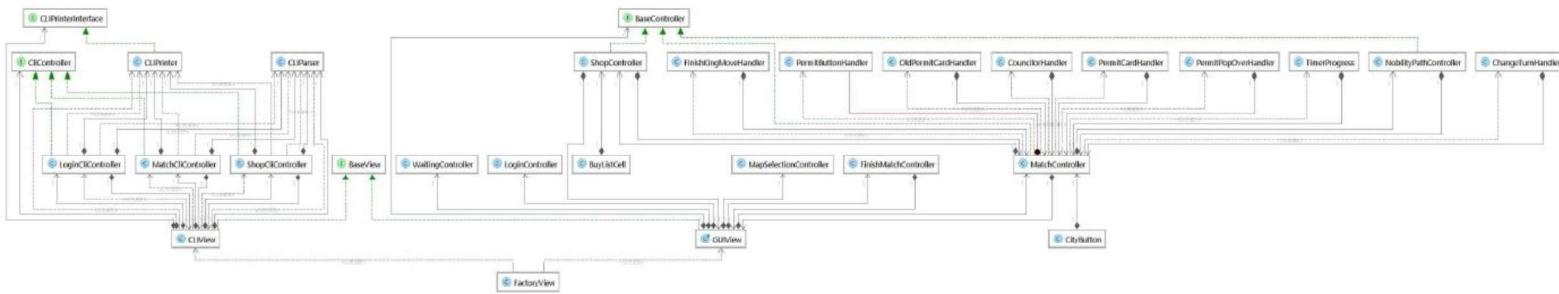
Quando avviene un cambiamento il server manda uno snapshot (che è una "fotografia" della situazione attuale) al client. L'interfaccia network del client (Socket o RMI) riceve lo Snapshot e comunica al ClientController. Questo notifica alla view l'avvenuto cambiamento. La View (CLI o GUI) quindi si aggiorna in base alle informazioni contenute nel nuovo snapshot.

Questo avviene per tutti gli eventi che succedono lato server (fine turno, inizio market, ecc...).

# COUNCIL OF FOUR

## View

Abbiamo progettato la View nel seguente modo:



In questo modo si può vedere come ci sia una classe generica chiamata FactoryView che consente, grazie alla scelta dell'utente, di generare una view definita dall'utente attraverso una scelta dal terminale.

```
1. public static BaseView getBaseView(String viewType, ClientController clientController) throws ViewException {
2.     switch (viewType) {
3.         case Constants.GUI:
4.             return new GUIView(clientController);
5.         case Constants.CLI:
6.             return new CLIView(clientController);
7.     }
8.     throw new ViewException("ViewType Not Supported");
9. }
```

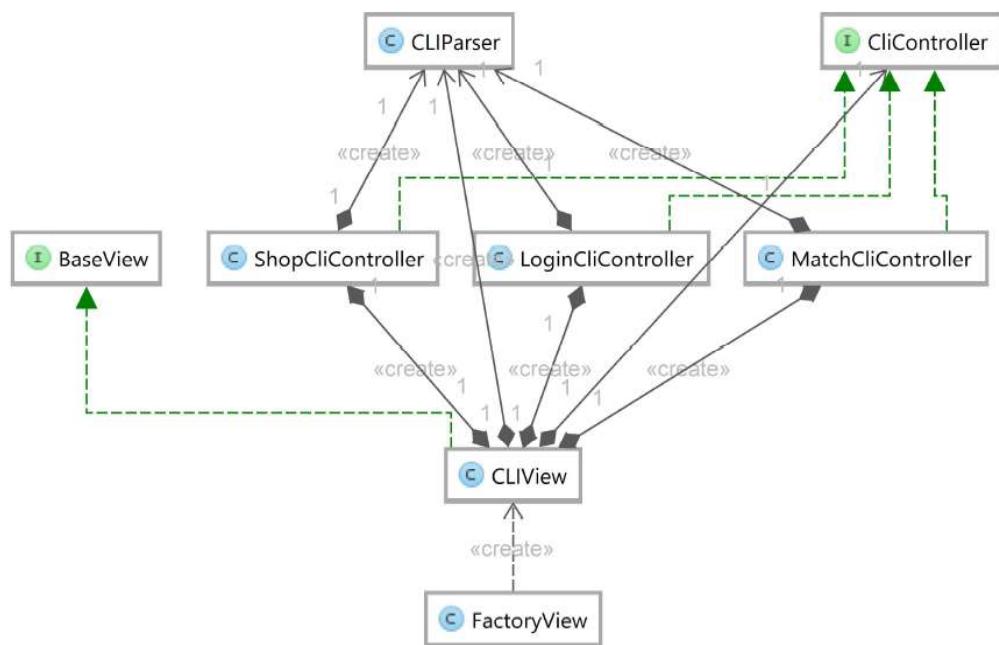
In questo modo abbiamo creato la View vera e propria.

Attraverso, quindi, questo pattern possiamo scegliere quale UI lanciare tra CLI e GUI.

# COUNCIL OF FOUR

## CLI (Command Line Interface)

La Command Line Interface è strutturata in questo modo:



Nel progettare la parte relativa alla Command Line Interface abbiamo seguito la struttura della GUI in modo da rendere tutto più simmetrico.

CliView è la classe che crea tutti i controller, che implementano l'interfaccia CliController.

CliView, come GUIView, estende BaseView in modo che sia trasparente alla classe ClientController il tipo di UI utilizzata.

In base allo stato corrente CliView chiama il metodo parseLine del controller corrente.

Ogni controller chiama quindi il metodo parseLine di cliParser che a sua volta chiama il metodo corretto del controller a cui è riferito.

Per chiamare il metodo corretto viene utilizzata la **Reflection** e le **Annotation**.

```
1. public class CLIParser {
2.
3.     private Class mClass;
```

# COUNCIL OF FOUR

```
4.     private HashMap<String, Method> methodHashMap = new HashMap<>();
5.     private HashMap<String, String> methodDescription = new HashMap<>();
6.     private HashMap<String, String> shortMethodDescription = new HashMap<>();
7.
8.     public CLIParser(Class mClass) {
9.         this.mClass = mClass;
10.        initMethods();
11.    }
12.
13.    private void initMethods() {
14.        for (Method method : mClass.getMethods()) {
15.            for (Annotation annotation : method.getAnnotations()) {
16.                if (annotation instanceof Command) {
17.                    methodHashMap.put(((Command) annotation).abbrev(), method);
18.                    methodHashMap.put(((Command) annotation).name(), method);
19.                    String description = CLIColor.ANSI_RED + " short: "
20.                            + ((Command) annotation).abbrev() + CLIColor.ANSI_RESET
21.                            + "\t Description: " + ((Command) annotation).description();
22.                    Annotation[][][] annotations = method.getParameterAnnotations();
23.
24.                    if (method.getParameterCount() > 0)
25.                        description += CLIColor.ANSI_GREEN + "\n \t Parameter\n" +
CLIColor.ANSI_RESET;
26.                    for (Annotation[] ann : annotations) {
27.                        for (Annotation paramAnn : ann) {
28.                            if (paramAnn instanceof Param) {
29.                                description += " \t" + CLIColor.ANSI_CYAN + ((Param)
paramAnn).name() + " - " + ((Param) paramAnn).description() + CLIColor.ANSI_RESET
+ "\n";
30.                            }
31.                        }
32.                    methodDescription.put(((Command) annotation).name(), description);
33.                    shortMethodDescription.put(((Command) annotation).abbrev(), description);
34.                }
35.            }
36.        }
37.    }
38.
39.    void parseInput(String line, Object object, CLIPrinter cliPrinter) {
40.
41.        String[] lines = line.split(" ");
42.
43.        if (lines[0].equalsIgnoreCase("help")) {
44.            printHelp(lines);
45.        } else {
46.            Method method = methodHashMap.get(lines[0]);
47.            if (method != null) {
48.                try {
49.                    if (method.getParameterCount() != lines.length - 1)
50.                        System.out.println("Check parameter number!");
51.                    else {
52.                        Object[] objects = new Object[lines.length - 1];
```

# COUNCIL OF FOUR

```
53.             System.arraycopy(lines, 1, objects, 0, lines.length - 1);
54.             method.invoke(object, objects);
55.         }
56.
57.         } catch (InvocationTargetException e) {
58.             System.out.println(e.getMessage());
59.         } catch (IllegalAccessException e) {
60.             System.out.println(e.getMessage());
61.         }
62.     } else {
63.         cliPrinter.printError("Sorry, method not found");
64.     }
65. }
66. }
```

CliParser alla sua creazione popola le HashMap con i nomi dei metodi e i metodi da chiamare della classe passata come parametro.

Utilizza solamente i metodi annotati con @Command in modo da evitare di esporre altri metodi.

Inoltre popola le altre hashMap con la descrizione del metodo e i parametri necessari.

In particolare per essere chiamato da cliParser un metodo deve avere queste annotazioni:

```
1. @Command(description = "|FAST ACTION|
  Change permit card", name = "changePermit", abbrev = "cp")
2.   public void changePermitAction(@Param(name = "region", description = "Region of
  the permit card that you want to change") String arg)
```

con il parametro description si inserisce la descrizione del metodo che sarà poi vista dall'utente. Con name il nome del metodo, nome che l'utente dovrà digitare. Con abbrev viene inserita un'abbreviazione per quel metodo.

I parametri sono annotati con @Param e contengono un nome e una descrizione.

Quando l'utente digita sulla tastiera il parser divide l'input in base agli spazi. Cerca nell'hashmap dei metodi un metodo con nome uguale alla prima parola digitata dall'utente. Una volta trovato il metodo effettua il controllo dei parametri, cioè se i parametri corrispondono ai parametri che si aspetta il metodo, se questo avviene invoca il metodo sull'oggetto passato come parametro al metodo parseLine (che rappresenta il controller corrente).

Abbiamo deciso di implementare in questo modo la CLI per renderla più scalabile. Una volta impostato il parser occorre solamente annotare i metodi che si vuole esporre con la sintassi definita in precedenza. Inoltre in questo modo si possono anche passare parametri ai metodi e rendere le azioni del client più veloci.

Per vedere la lista de metodi che è possibile chiamare l'utente deve solo digitare help.

# COUNCIL OF FOUR

Ogni oggetto che implementa CliController ha il proprio parser e in questo modo in base al controller corrente l'utente vedrà solo determinati metodi da chiamare.

## LETTURA DA INPUT ED EVENTI

La lettura da System.In viene effettuata dalla CliView in questo modo:

```
1. private void getInput() {
2.     Runnable runnable = () -> {
3.         String input = "";
4.         if (first) {
5.             currentController.printHelp();
6.             first = false;
7.         }
8.
9.         while (true) {
10.             try {
11.                 while (!bufferedReader.ready() || !needToRead.get()) {
12.                     Thread.sleep(200);
13.                 }
14.                 input = bufferedReader.readLine();
15.                 String finalInput = input;
16.                 currentController.parseLine(finalInput);
17.             } catch (IOException e1) {
18.
19.             } catch (InterruptedException e) {
20.             }
21.         }
22.     };
23.     futureTask = executorService.submit(runnable);
24. }
```

Viene definito un Runnable che viene poi eseguito tramite un executorService.

Il Thread "dorme" se o non è ancora pronto il bufferedReader oppure se non deve leggere.

In questo modo è possibile "stoppare" la lettura da input in questo metodo ed effettuarla in un altro metodo, per esempio quando l'utente riceve dei bonus che necessitano una scelta.

Ogni controller, come si può capire dal nome è adibito ad una funzione specifica:

- LoginCliController effettua il login
- MatchCliController è il controller del Match
- ShopCliController è il controller del market

Abbiamo quindi cercato di seguire il pattern State:

Ogni controller rappresenta uno Stato e in base allo stato corrente viene interpretato l'input dell'utente.

In base agli eventi che arrivano da server si cambia stato.

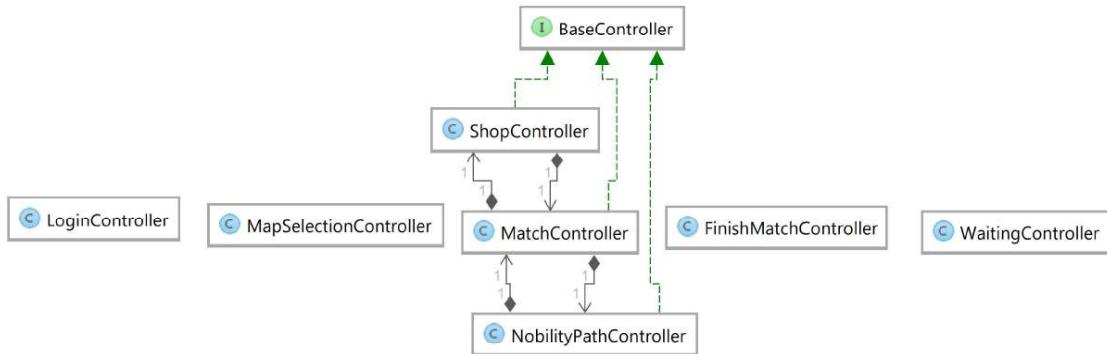
# COUNCIL OF FOUR

Avremmo potuto implementare meglio questa parte vedendo ogni singola azione dell'utente come uno stato.

# COUNCIL OF FOUR

## GUI

La GUI è strutturata nel modo seguente:



Nelle prime schermate semplici, come quella di Login, quella di attesa della selezione o della selezione stessa della mappa, ci siamo limitati a costruire semplicemente un classico modello composto in generale da 2 classi:

- Un controllerFXML
- Un loader

### LOADER

La classe che fa da loader è GUIView che ha diverse funzioni. La prima di tutte, e più importante, è quella di caricare l'FXML e costruire la scena e lo stage su cui poi verranno inseriti gli altri node. Un'altra funzione importante è, al momento di questa inizializzazione, la chiamata al metodo che definisce cos'è un clientController nel controllerFXML. Così facendo riusciamo a conoscere oggetti e metodi nel client anche nel controllerFXML.

Qui possiamo vedere un classico caricamento della schermata fxml da parte di GUIView.

Viene creato un nuovo FXMLLoader, un nuovo Parent, una Scene e assegnato uno Stage.

Alla riga 7 si può vedere come si chiama il metodo setClientController passandogli il clientController, facendo conoscere, quindi, le due classi.

```
1. @Override
2.     public void start(Stage primaryStage) throws Exception {
3.         this.stage = primaryStage;
4.         FXMLLoader loader = new FXMLLoader(getClass().getResource(Constants.LOGIN_
FXML));
5.         Parent screen = loader.load();
6.         loginController = loader.getController();
7.         loginController.setClientController(clientController);
8.         primaryStage.getIcons().add(new Image("/ClientPackage/View/GUIResources/Im
age/Icon.png"));
9.         primaryStage.setTitle("COFFee");
```

# COUNCIL OF FOUR

```
10.     scene = new Scene(screen);
11.     stage.setOnCloseRequest(new EventHandler<WindowEvent>() {
12.         @Override
13.         public void handle(WindowEvent event) {
14.             System.exit(0);
15.         }
16.     });
17.     this.stage.setScene(scene);
18.     this.stage.show();
19.     stage.setMinHeight(550);
20.     stage.setMinWidth(600);
21. }
```

Un'altra funzione fondamentale di questa classe è quella cosiddetta di “bridge”. Essa infatti, funge soprattutto da ponte tra il clientController e il controllerFXML. In questo modo posso collegare sempre un cambiamento avvenuto nel server ad un cambiamento grafico e visibile sul lato client.

Un esempio:

```
1. @Override
2. public void showLoginError() {
3.     Platform.runLater(() -> {
4.         loginController.showLoginError("Username già scelto");
5.     });
6. }
```

In questo modo vediamo come riesca a comunicare al controller un errore che mi arriva dal server.

Fondamentale è il metodo Platform.runLater() che mi permette di fare un update di un qualche componente da un non-GUI thread.

## CONTROLLER

Il controller è ciò che collega la UI alle modifiche del Server, ciò che fonde l'FXML e il clientController. Ad ognuno di esso è infatti rappresentato un fxml diverso che è la struttura dell'interfaccia grafica, assieme al css. In esso quindi avvengono tutti i metodi associati al tale oggetto sulla UI del client.

Avremo quindi variabili di un genere particolare e con uno scope particolare.

```
1. @FXML
2. private ImageView prevImageView;
```

Con questa annotazione si dice al compilatore che questa variabile è già presente nel foglio fxml e che quindi ha già alcuni parametri definiti.

Infatti, associato al controller, che opera prettamente per modifiche a livello UI e quindi non si occupa assolutamente di cambiare qualcosa nel clientController o nel loader, abbiamo un foglio fxml che può essere anche vuoto, se si crea tutto a runtime a codice, oppure utilizzato per creare i nodes.

Ciò non toglie quindi, che io possa aggiungere un'immagine da fxml oppure a codice e successivamente aggiungerla al rootPane.

# COUNCIL OF FOUR

Se prima nel loader avevamo una funzione di bridge per collegare UI e clientController, nel controllerFXML, grazie al metodo setClientController, possiamo chiamare metodi direttamente sul clientController senza bisogno di passare dalla GUIView.

## CONFIGURAZIONE DEL MATCH

All'interno del match abbiamo dovuto aggiungere un'interfaccia BaseController in quanto ci serviva per riuscire, con un solo metodo, a chiamare un metodo in ogni controllerFXML. Ad esempio, quando arriva lo snapshot, dobbiamo aggiornare tutti i valori presenti sull'interfaccia della mappa, ma anche nel market nel caso in cui fosse questo turno.

```
1. @Override
2. public void updateSnapshot() {
3.     for (BaseController baseController : baseControllerList) {
4.         Platform.runLater(new Runnable() {
5.             @Override
6.             public void run() {
7.                 baseController.updateView();
8.             }
9.         });
10.    }
11. }
```

In questo modo quindi, possiamo modificare più controller contemporaneamente perché in ogni BaseController avrà @Override del metodo updateView();

MatchController:

```
1. @Override
2. public void updateView() {
3.     this.currentSnapshot = clientController.getSnapshot();
4.     nobilityPathText.setText(currentSnapshot.getCurrentUser().getNobilityPathPosition().getPosition() + "");
5.     richPathText.setText(currentSnapshot.getCurrentUser().getCoinPathPosition() +
6.     "");
7.     helperText.setText(currentSnapshot.getCurrentUser().getHelpers().size() + "");
8.     victoryPathText.setText(currentSnapshot.getCurrentUser().getVictoryPathPosition() + "");
9.     mainActionText.setText(currentSnapshot.getCurrentUser().getMainActionCounter() +
10. );
11.     fastActionText.setText(currentSnapshot.getCurrentUser().getFastActionCounter() +
12. );
13.     reprintCouncilor();
14.     reprintPermitCard();
15.     setEmporiaVisibility();
16.     populateField(clientController.getSnapshot().getUsersInGame().get(usersComboBox
17. .getValue()));
18.     createHand();
19.     createOldPermitCard();
20.     if (needToSelectOldPermitCard.get() && myTurn)
21.         selectOldPermitCardBonus();
22. }
```

# COUNCIL OF FOUR

ShopController:

```
1. @Override
2. public void updateView() {
3.     updateList();
4. }
```

## CARICAMENTO DELLE IMMAGINI

Dato il forte impatto nel consumo di RAM e il rallentamento tra il load di una schermata e l'altra abbiamo pensato di ottimizzare le prestazioni grazie all'utilizzo del proxy ImageLoader, una classe fatta ad hoc che carica le immagini attraverso parametri che ne velocizzano il tempo.

Abbiamo quindi utilizzato soprattutto il .setCache(true) delle immagini ed inoltre abbiamo messo a true l'upload di queste in background.

```
1. public Image getImage(String path) {
2.     if (!this.imageHashMap.containsKey(path)) {
3.         Image image = null;
4.         if (!path.contains("Map")) {
5.             image = new Image(path, true);
6.         } else {
7.             image = new Image(path, true);
8.         }
9.         this.imageHashMap.put(path, image);
10.    }
11.    return this.imageHashMap.get(path);
12. }
```

Abbiamo, infine, creato un altro metodo, nel caso in cui width e height erano note, con cui potevamo settare a true la smooth.

```
1. public Image getImage(String path, double width, double height) {
2.     if (!this.imageHashMap.containsKey(path)) {
3.         Image image = null;
4.         if (!path.contains("Map")) {
5.             image = new Image(path, width, height, true, true, true);
6.         } else {
7.             image = new Image(path, width, height, true, true, true);
8.         }
9.         this.imageHashMap.put(path, image);
10.    }
11.    return this.imageHashMap.get(path);
12. }
```

# COUNCIL OF FOUR

## RESIZABLE

Abbiamo reso l'interfaccia resizable grazie all'utilizzo di diversi pane:

- GridPane
- StackPane
- VBox
- HBox

In questo modo siamo riusciti, detectando la posizione relativa attraverso un metodo che restituiva tramite il click la sua X e la sua Y relative alla larghezza del rootPane.

```
1. private void placeRegionBonus() {
2.     ImageView coastImage = new ImageView(ImageLoader.getInstance().getImage(Constants.IMAGE_PATH + "/CoastBonusCard.png", background.getWidth() * 0.89, background.getHeight() * 0.82));
3.     ImageView hillImage = new ImageView(ImageLoader.getInstance().getImage(Constants.IMAGE_PATH + "/HillBonusCard.png", background.getWidth() * 0.89, background.getHeight() * 0.82));
4.     ImageView mountainImage = new ImageView(ImageLoader.getInstance().getImage(Constants.IMAGE_PATH + "/MountainBonusCard.png", background.getWidth() * 0.89, background.getHeight() * 0.82));
5.     coastImage.setCache(true);
6.     hillImage.setCache(true);
7.     mountainImage.setCache(true);
8.     coastImage.layoutXProperty().bind(background.widthProperty().multiply(0.089));
9.     coastImage.layoutYProperty().bind(background.heightProperty().multiply(0.82));
10.    hillImage.layoutXProperty().bind(background.widthProperty().multiply(0.447));
11.    hillImage.layoutYProperty().bind(background.heightProperty().multiply(0.82));
12.    mountainImage.layoutXProperty().bind(background.widthProperty().multiply(0.80));
13.    mountainImage.layoutYProperty().bind(background.heightProperty().multiply(0.82));
14.    coastImage.fitWidthProperty().bind(background.widthProperty().divide(20));
15.    hillImage.fitWidthProperty().bind(background.widthProperty().divide(20));
16.    mountainImage.fitWidthProperty().bind(background.widthProperty().divide(20));
17.    coastImage.setPreserveRatio(true);
18.    hillImage.setPreserveRatio(true);
19.    mountainImage.setPreserveRatio(true);
20.    Graphics.addShadow(coastImage);
21.    Graphics.addShadow(hillImage);
22.    Graphics.addShadow(mountainImage);
23.    Graphics.bringUpImages(coastImage, hillImage, mountainImage);
24.    background.getChildren().addAll(coastImage, hillImage, mountainImage);
25.    Tooltip.install(coastImage, new Tooltip("Bonus della regione"));
26.    Tooltip.install(hillImage, new Tooltip("Bonus della regione"));
27.    Tooltip.install(mountainImage, new Tooltip("Bonus della regione"));
28. }
```

# COUNCIL OF FOUR

Così siamo riusciti a bindare le relative X e Y e le Width e le Height a dimensioni relative. È stato molto più semplice per le immagini in quanto hanno la possibilità di preservare la ratio.

Per il background invece abbiamo utilizzato sempre un gridPane facendo fissare l'immagine al centro ed utilizzando ai bordi dei "pattern" dello stesso colore in modo tale che non ci fosse troppo distacco per la schermata.

# COUNCIL OF FOUR

## Informazioni

Gruppo costituito da ;

EMANUELE GHELF  
BACKEND AND FRONTEND  
DEVELOPER  
SOFTWARE ARCHITECT



Tel. 3288219218

GIULIO MELLONI  
BACKEND AND FRONTEND  
DEVELOPER  
UI ARCHITECT



Tel. 3407531919

## Informazioni sulla società

A Beautiful Mind  
Polesine Zibello