Relazione progetto Reti WORTH

https://reti.fulvio.dev

Fulvio Denza, 544006, f.denza@studenti.unipi.it

January 24, 2021

1 Introduzione

WORTH è un modo per condividere il workflow di un progetto. Esso rappresenta una delle metodologie AGILE che si sono sviluppate negli ultimi anni. Il modello di WORTH è un modello semplificato di applicativi del campo, quali easy redmine, trello e molti altri. In WORTH vediamo l'esistenza di una sequenza di stati dei vari task, quale todo, in progress, to be revised e done, la presenza di una chat per ogni progetto, azioni basilari sulle card (o task) quali change status, add card, get card history ed altre.

Il modus operandi per presentare i comandi del progetto sarà bilaterale, ossia verrà descritto l'azione del client e la risposta del server, mentre le strutture dati e le scelte di implementazione delle astrazioni saranno presentate nelle rispettive sezioni del server o del client.

2 Testing

Per testare il progetto basta andare nel percorso: worth/src/ ed avviare il server con

```
./runServer.sh
```

ed il client con

```
./runClient.sh
```

Dal client basterà iniziare ad usare il programma secondo le specifiche.

3 Package Server

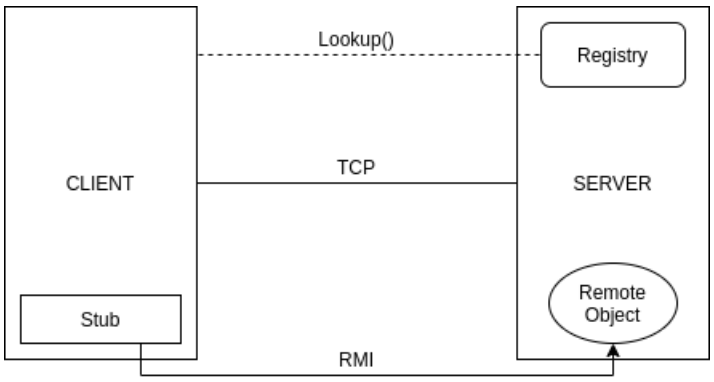
Il server contiene la logica del nostro applicativo, contiene le Classi per descrivere le componenti di WORTH, esso parte da una classe Main.

3.1 Main

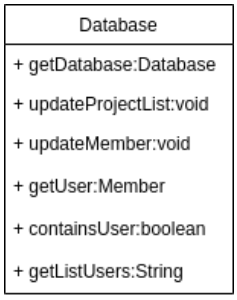
Il main presenta le istanziazioni delle classi per la connessione TCP e per la RMI Callback

```
//Main.java
//REGISTER
UserRegister ur = new UserRegister();
Runtime.getRuntime().exec("rmiregistry 2020");
System.setProperty("java.rmi.server.hostname","0.0.0.0");
ur.RemoteHandler(5455);

//LOGIN
ServerNotification serverCB = new ServerNotification();
ServerNotificationInterface stubCB = (ServerNotificationInterface)
    UnicastRemoteObject.exportObject(serverCB, 0);
String name = "notification";
LocateRegistry.createRegistry(7001);
Registry registryCB = LocateRegistry.getRegistry(7001);
registryCB.bind(name, stubCB);

//TCP Connection
TCPConnection connection = new TCPConnection(serverCB);
connection.start(5456);
if(Thread.interrupted()) connection.stop();
System.out.println("Server Started");
```

Le righe di codice sopra riportate presentano semplicemente l'istanziazione delle classi Server per le nostre RMI Callback che verranno invocate nel momento in cui il client effettuerà una register e una login. La TCP Connection è il modo in cui viene istanziato il server TCP per aprire il canale di comunicazione lato server implementato per mezzo di una ServerSocket sulla porta 5456.



3.2 Database



Il database è stato sviluppato usando la libreria di Google gson, esso consiste in un file .json contenente tutti gli utenti nella forma:

```
n:
        username: "username"
        password: "password"
        projectList:
                0: "project"
                1: "project"
                ...
```

in cui n è l'indice, nel file json, della posizione dell'utente, il campo username e password sono i campi in cui vengono conservati le relative informazioni, projectList contiene, invece, la lista di progetti a cui appartiene l'n-esimo utente. Una miglioria, nonchè best practice, sarebbe potuta essere implementare un algoritmo di crittografia per salvare le password cifrate, anzichè in chiaro, tuttavia ai fini del progetto non era strettamente necessario.
Il database contiene i campi

```
private static volatile Database database;
private static Path dbFile;
private static final ConcurrentHashMap<String, Member> db = new
    ConcurrentHashMap<>();
```

Ho dichiarato Database database volatile perchè avevo bisogno di una garanzia per il database che il suo valore sia uguale ovunque, sia nella cache sia nella memoria principale affinchè non ci siano problemi di lettura/scrittura qualora consumassero o producessero il database più thread alla volta, la variabile database contiene la struttura stessa del database. la ConcurrentHashMap db contiene le associazioni (Username, Member) che verranno usate per le ricerche su database, ogni qualvolta la ConcurrentHashMap viene modificata, viene modificato anche il database.json. È stata scelta una struttura dati concorrente in modo da gestire gli accessi dei Thread, questa scelta migliora sensibilmente le performace del nostro applicativo.

getDatabase

getDatabase è il metodo usato per usare il database esistente o, se non esistente, crearne uno nuovo.

updateMember

updateMember è il metodo per aggiornare il file json degli utenti del database.

updateProjectList

updateProjectList server per aggiornare la lista dei progetti a cui l'utente indicato nei parametri è iscritto.

getUser

getUser è il metodo getter per ottenere dal file database.json l'utente che come username ha quello indicato tra i parametri. Il metodo effettua una ricerca sulla ConcurrentHashMap del database rispetto alla chiave String username.

containsUser

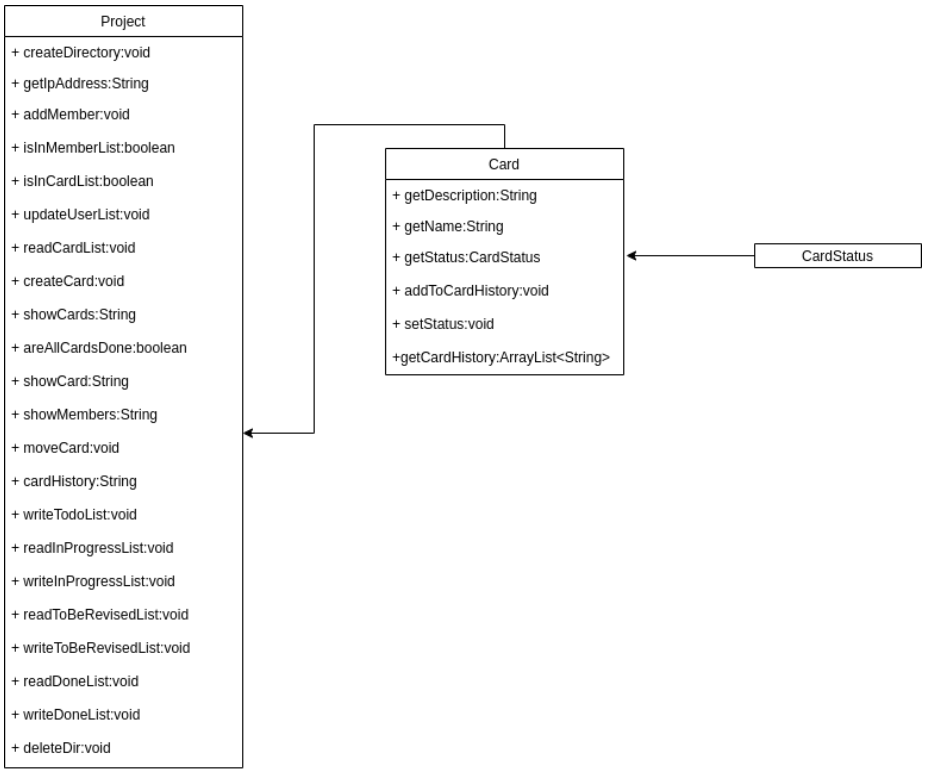
containsUser è il metodo per restituire una risposta alla domanda: "l'utente si trova nel database?". Questo metodo usa la getUser per capire per effettuare la ricerca, se il metodo getUser non throwa un'eccezione MemberNotFoundException, restituisce true, altrimenti false.

getListUser

Il metodo getListUser viene usato per restituire una stringa contente tutti gli utenti separati, nella stringa, da un carattere speciale $ e ogni utente è descritto da una coppia username:status

3.3 CardStatus, Card, Project

CardStatus è l'enum per definire gli stati in cui si può trovare una Card, Card, a sua volta, è la definizione di una Card come data nelle specifiche, Project è la composizione delle Cards, in Project vengono definite svariate strutture per organizzare le card.



3.3.1 CardStatus

CardStatus è un semplice enum che contiene i possibili stati di una Card:

- TODO
- IN_PROGRESS
- TO_BE_REVISED
- DONE

3.3.2 Card

Card è l'elemento fondamentale di ogni Progetto, una Card è un task, un pezzo di lavoro che si trova in una certa situazione tra quelle descritte in CardStatus. Una Card ha 4 campi principali:

```
private String name;
private String description;
private CardStatus status;
private ArrayList<String> cardHistory;
```

i primi 3 sono autoesplicativi, il quarto, cardHistory, è la lista di Stati che ha attraversato la carta in questione. Ognuno di questi campi ha un metodo getter, mentre hanno un metodo setter status e description. Il metodo addToCardHistory è il metodo utilizzato per aggiornare la Card.

3.3.3 Project

Project è la classe più grande del progetto WORTH. Essa contiene tutta la logica relativa ai progetti all'interno di WORTH.

```
public String projectName;
public ArrayList<String> memberList;
public ArrayList<Card> taskList;
public ArrayList<Card> TODO_List;
public ArrayList<Card> IN_PROGRESS_List;
public ArrayList<Card> TO_BE_REVISED_List;
public ArrayList<Card> DONE_List;
```

Ogni Status possibile ha una lista associata, in aggiunta a questi abbiamo una taskList, ossia la lista di tutte le Card in un progetto, memberList, ossia la lista dei membri di un progetto e, ovviamente, projectName, il nome identificativo del progetto. Il metodo costruttore di Project non è altro che l'istanziazione delle varie liste, rappresentate da $Arraylist < String >$ che contengono semplicemente i nomi delle varie Card presenti nel singolo Progetto e la creazione dei file per ogni singola lista, in modo da garantire la persistenza. La struttura di una cartella progetto è questa:

Listing 1: tree projects

```
projects/
+-- project
|-- cards.json
|-- done.json
|-- in_progress_list.json
|-- to_be_revised.json
|-- ip_address.json
|-- memberList.json
+-- todo_list.json
```

Oltre a questo, il metodo costruttore contiene anche la generazione di un ip usando il metodo statico

Listing 2: IPGenerator class

```
public class IPGenerator {
	public static String generateIPAddress() {

		return (ThreadLocalRandom.current().nextInt(224, 239+1) +
		    "." + ThreadLocalRandom.current().nextInt(0, 256) +
```

```
                "." + ThreadLocalRandom.current().nextInt(0, 256+1) +
                "." + ThreadLocalRandom.current().nextInt(0, 256+1));
        }
}
```

Il quale, con la classe ThreadLocalRandom, genera una stringa composta da 4 parti separate da un punto, ogni parte rappresenta una porzione di indirizzo IP partendo da 224.0.0.0 e finendo all'indirizzo 255.255.255.255, ossia il subset di indirizzi utilizzabili come indirizzi multicast.

createDirectory

Il Metodo createDirectory è composto da un if·else per descrivere l'esistenza o meno del progetto nella directory dei progetti. Quindi, se non esiste il progetto, viene creata la cartella e il file con, all'interno, l'ip address, altrimenti stampa a schermo "Project exists in the project folder"
Da adesso, molti metodi saranno della forma:

```
Gson gson = new Gson();
BufferedReader br;

try {
        File f = new File(path+"/file.json");
        if(!f.exists()) {
                f.createNewFile();
        }
        br = new BufferedReader(new FileReader(path+"/file.json"));
        Type type = new TypeToken<InterestedType>() {
        }.getType();
        dataStructure = gson.fromJson(br, type);
} catch (IOException e) {
        e.printStackTrace();
}
return dataStructure;
```

per la lettura aggiornata del contenuto di file.json e il suo salvataggio in

$$< InterestedType > dataStructure$$

al fine di utilizzare la versione aggiornata del file json e,

```
Writer writer;
try {
        writer = new FileWriter(path+"/file.json");
        Gson gson = new GsonBuilder().setPrettyPrinting().create();
        gson.toJson(dataStructure, writer);
        writer.flush();
        writer.close();
} catch (IOException e) {
```

```
        e.printStackTrace();
}
```

per la scrittura di dataStructure in file.json.
Questi due metodi verranno utilizzati in coppia per deserializzare e serializzare una qualsiasi struttura dati di cui avremo bisogno in futuro. I due metodi sono l'equivalente di una update dataStructure e update file.json, una che scrive nella struttura dati e una che scrive nel file json.
I metodi della prima forma (**Reader**) sono:

- readCardList, per salvare nella struttura dati dell'oggetto Project la lista di Cards;
- readTodoList, per salvare nella struttura dati dell'oggetto Project la lista di Cards nello stato TODO;
- readInProgressList, per salvare nella struttura dati dell'oggetto Project la lista di Cards nello stato IN_PROGRESS;
- readToBeRevisedList, per salvare nella struttura dati dell'oggetto Project la lista di Cards nello stato TO_BE_REVISED;
- readDoneList, per salvare nella struttura dati dell'oggetto Project la lista di Cards nello stato DONE.

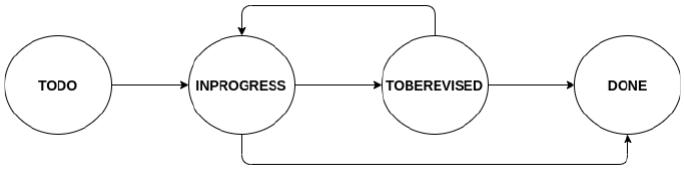
I metodi della seconda forma (**Writer**) sono:

- updateCardList, per rendere persistenti i cambiamenti effettuati nella struttura dati taskList;
- writeTodoList, per rendere persistenti i cambiamenti effettuati nella struttura dati TODO_List;
- writeInProgressList, per rendere persistenti i cambiamenti effettuati nella struttura dati IN_PROGRESS;
- writeToBeRevisedList, per rendere persistenti i cambiamenti effettuati nella struttura dati TO_BE_REVISED;
- readDoneList, per rendere persistenti i cambiamenti effettuati nella struttura dati DONE.

Un miglioramento che si sarebbe potuto fare sarebbe stato creare un singolo metodo per ogni tipologia (Writer e Reader) in cui si indicava la struttura dati e il file da Scrivere o Leggere, in modo da non avere ripetizioni di codice.
Abbiamo poi dei metodi ausiliari

- getIpAddress, per restiture l'ip address dal file json, andando a prendere l'ip dal file ipAddress.json;

- isInMemberList, per restituire la risposta alla domanda "l'utente è nella lista utenti del progetto?" andando a controllare in memberList.json;
- isInCardList, per restituire la risposta alla domanda "la card è nella lista di cards del progetto?" andando a controllare nel file cards.json;
- addMember, per aggiungere un membro nella struttura dati e aggiornare il json all'ultima versione della struttura dati;
- createCard, per creare una nuova card e alla fine dell'operazione aggiornare il json all'ultima versione della struttura dati.
- showCards, per restituire una stringa contente tutte le cards separate da un carattere speciale $;
- areAllCardsDone, per effettuare una visita al file cards.json e controllare se tutte le cards siano nello stato DONE, se lo sono, restituisce true, altrimenti restituisce false;
- showCard, per cercare la card con il cardName dato nel parametro, effettua una visita per controllare che quella card ci sia e in tal caso restituire una stringa della forma:
  NAME:name$STATUS:status$DESCRIPTION:description, ossia i campi descrittivi della card richiesta che sarà manipolata dal client.
- showMembers, per restituire una stringa della forma:
  member1$member2$...$memberN, ossia la lista di membri appartenenti a un progetto, che sarà poi manipolata dal client.
- moveCard, infine abbiamo la move card che segue il seguente automa:



La moveCard contiene il seguente algoritmo: per ogni operazione ammissibile, legge il file json in cui si trova la card e ne salva il contenuto nella struttura dati, effettua la rimozione dalla struttura dati, scrive la struttura dati aggiornata nel json, legge il file json di destinazione (indicata dall'utente) e ne salva il contenuto nella struttura dati, ne aggiunge la Card, se esiste, e carica il contenuto della struttura nel file json interessato.

3.4 TCPConnection e ConnectionHandler

Il server, come abbiamo visto nel main, istanzia l'oggetto TCPConnection e "starta" il task passandolo a un thread della threadpool, che nel nostro server è implementata come una CachedThreadpool, visto che non sappiamo a priori quanti server eseguiranno. Lo fa istanziando una ServerSocket presso la porta indicata nel main e, all'interno di un ciclo infinito, chiama il metodo execute del , passandogli come parametri serverSocket.accept() e serverNotification,

```
while(true) {
       pool.execute(new ConnectionHandler(serverSocket.accept(),
           serverNotification));
}
```

questi due parametri serviranno al gestore della connessione per mandare sulla socket le risposte ai comandi invocati dal client e la serverNotification per utilizzare la Callback RMI per inviare la lista di utenti ONLINE e OFFLINE presenti nel sistema.
Sulla serverSocket il server riceverà il comando mandato dal client conservandolo nell'array *command*, l'array command avrà nella posizione 0 il comando stesso, nella posizione 1 conterrà i suoi parametri, separati da un ":". In questa implementazione *command[0]* verrà inserito all'interno di uno switch...case per gestire ogni singolo comando che arrivi al server e all'occorrenza verranno separati i parametri di *command[0]* in ogni case a seconda di quanti parametri il *command[0]* prenda.
Torneremo dopo su ogni case al fine di capirne meglio l'implementazione dei comandi.

3.5 RMI: ServerNotification e UserRegister

UserRegister

La classe UserRegister viene usata per gestire il comando di registrazione dell'utente ed implementa i metodi dell'interfaccia.
Per l'implementazione del metodo register ho creato un Registry sull'host presso la porta 5455 e ho bindato il nome "RegisterUserInterface" all'oggetto UserRegister, in modo che fosse localizzabile dal client. Il metodo register non fa altro che aggiungere l'utente al file database.json se l'utente con il nome utente inserito dal client non esiste già, altrimenti skippa l'azione stampando "SERVER: user already registered!".

ServerNotification

La classe ServerNotification viene usata per gestire il servizio di notifica degli utenti e del loro stato. ServerNotification ha un campo
$List < ClientNotificationInterface > clients$ e un campo $List < String > usersList$, il suo metodo costruttore è semplicemente l'istanziazione di queste

due liste. Quando si effettua una chiamata al metodo register, si iscrive semplicemente il client al servizio di notifica aggiungendolo alla lista e si aggiunge la stringa relativa al suo username alla lista di Stringhe, quando si effettua una unregister si rimuove il client e la stringa relativa al suo username dalle liste relative. sendMemberList effettua una singola chiamata al metodo doCallbacks il quale, a sua volta, genera un iteratore di clients e finchè questo iteratore non giunge al termine, istanzia un oggetto della classe ClientNotificationInterface client ed effettua una chiamata al metodo notify di quest'ultima classe.

4 Package Client

4.1 Main

Il main del client contiene innanzitutto l'istanziazione di TCPClient che vedremo dopo, oltre a ciò contiene un ciclo che continua finchè non viene interrotto il thread in cui prende come input i commandi passati dall'utente e passa questi ultimi al metodo compute di clientConnection, ossia la nostra connessione TCP.

4.2 TCPClient

La classe TCPClient contiene il cuore della logica del client. In questa classe abbiamo 3 campi importanti:

```
private Socket clientSocket;
private PrintWriter out;
private BufferedReader in;
public static boolean alreadyLogged
```

La prima variabile viene usata per far connettere la socket del client a quella del server in modo da stabilire la connessione, il secondo campo è il canale per scrivere sulla socket e il terzo campo è il canale per ricevere dalla socket. Il metodo cardine del client è il metodo compute, questo prende in input una stringa cmd e decide cosa fare in base al contenuto di quella stringa. Nel capitolo 4 descriveremo i comandi.

4.3 CLICommand

CLICommand è una classe astratta che definisce la struttura di un comando base. Ogni classe che estende CLICommand deve implementare un metodo manage(Scanner scanner) per gestire l'input dell'utente in cui, in base al comando, prende i parametri di cui quest'ultimo ha bisogno e li impacchetta nel modo seguente:
Sia N $\geq$ 1,

$$command@parameter0 : ... : parameterN$$

Oppure, qualora il comando non ammetta parametri:

$$command$$

Quindi, ogni classe che estende CLICommand, non fa che creare un formato standard di impacchettamento dei comandi con i loro parametri affinchè il server possa usare un modo univoco per spacchettare le stringhe che riceve e gestirle in modo appropriato. In questa relazione non ci soffermeremo su ogni sottoclasse, vista la grande somiglianza tra tutte. Nel capitolo 4 ci soffermeremo sugli altri metodi di questa classe.

4.4 ClientNotification e ClientNotificationInterface

ClientNotificationInterface è un'interfaccia che contiene un solo metodo: notify. ClientNotification implementa l'omonima interfaccia con un singolo metodo. La notify non fa altro che venire chiamata dal server tramite il registro RMI in modo da stampare sul client la lista di utenti che, all'inizio è impacchettata in una stringa della forma:

$$user1\$user2\$...\$userN$$

e al posto del carattere speciale $ vi sostituisce un \n in modo da separare con un new line ogni utente e renderlo leggibile per l'utente.

4.5 RemoteHandlerClient

La classe RemoteHandlerClient contiene un solo metodo "registerStub" che viene chiamato al momento della registrazione di un nuovo utente. All'interno di questo metodo si effettua una LocateRegistry sulla porta 5455, poi si effettua un lookup del nome del registro e, in base alla risposta del server RMI, stampa il risultato della registrazione, che può essere: "*You have been successfully registered*" o "*You're already present in the Database!*"

5 Comandi

L'intento di questa sezione è mostrare come il comando viene incapsulato e quali sono le operazioni, lato server o client, che vengono eseguite, tuttavia è possibile che ci siano ridondanze con quanto già scritto nelle specifiche del progetto.

register

Il comando register prende in input due parametri: username e password, esso istanzia, lato client, un oggetto RemoteHandlerClient, attraverso *scanner.next()* prende gli altri due parametri ed effettua la registerStub della classe suddetta. Il server gestirà questo commando nel metodo register di UserRegister, aggiungendo, se l'utente non esiste già, al Database, tramite la chiamata:

```
Member m = new Member(nickname, password);
Database.getDatabase().updateMember(m);
```

Restituisce 1 o 0, rispettivamente se il server non riesce ad aggiungere l'utente indicato o se ci riesce.

login

L'operazione di login incapsula (come dicevamo dei comandi che estendono CLICommand) l'intero comando in una stringa della forma
login@username:password. Se l'utente non inserisce caratteri speciali all'interno

delle stringhe, il client manda la stringa al server, inoltre setta la variabile alreadyLogged a true, in modo da far sapere al client UDP quando interrompere il ciclo while (che vedremo in seguito). Lato server, la login effettua un controllo per vedere se l'utente esiste nel Database, in tal caso salva l'utente con lo stesso username ricevuto dal client, successivamente effettua un controllo per capire se la password coincida con quella ricevuta e, se quell'utente non è già loggato, setta il suo stato a ONLINE, setta la variabile logged a true, manda la stringa [OK SERVER: Logged in!] e effettua la sendMemberList, che abbiamo trattato in paragraph 3.5. Il risultato della login arriva sotto forma di stringa "[OK] message" o "[KO] message", per decidere se stampare o meno la lista degli utenti il client effettua una ricerca della sottostringa "[OK]" e se questa sottostringa esiste, stampa la lista chiamando il comando ListUsers che vedremo dopo.

logout

Logout prende un solo parametro, ossia il nickname dell'utente di cui si vuole fare il logout, quindi incapsula il comando in logout@username. Effettuo prima il controllo che il client sia loggato, se lo è, controlliamo che l'utente loggato da quel client sia lo stesso di quello di cui vuole fare il logout, poi settiamo la variabile logged e lo stato dell'utente a false e successivamente scriviamo sullo stream in output il risultato dell'operazione.

create_project

La struttura di create_project è create_project@projectName. Quando la stringa arriva nel server, aggiunge innanzitutto il creatore del progetto alla lista utenti, poi istanzia un nuovo progetto, ne crea la cartella e manda il messaggio di riuscita creazione al client.

add_card

La struttura di add_card è la seguente:
add_card@projectName:cardName:cardDescription, una volta spedita questa stringa al server, quest'ultimo controlla che l'utente sia nella lista membri del progetto, se appartiene a questa lista, controlla che la card inserita dall'utente non esista già, dopo aver fatto questi controlli provvede ad aggiungere la card creandola con il metodo createCard di Project, aggiunge la card direttamente nella Todo List e manda il messaggio sulla chat udp (di cui parleremo dopo)

add_member

Lato Client la add_member incapsula il comando in questo modo:
add_member@projectName:usernameToAdd. Una volta che la stringa arriva al server, questo effettua prima di tutto il controllo che l'utente da aggiungere esista, successivamente controllo che l'utente che ha mandato il comando sia effettivamente nella memberList del progetto che ha indicato, se l'utente da

aggiungere è già nel progetto, il server manda al client la stringa *"User already present in the project"* altrimenti lo aggiunge. Se l'utente loggato non è nel progetto, il server stampa a schermo *"You cannot add a member to a project if you're not in that project"*

show_members

Il formato di questo comando è show_member@projectName. Il server gestisce questo comando mandando al client il risultato del metodo showMembers() di Project, il client, una volta ricevuta la stringa, la stampa sostituendo ogni occorrenza del carattere "$" con "\n" in modo da stamparlo a schermo, lato client, in maniera leggibile.

show_cards

Il comando show_cards serve a mostrare tutte le cards di un progetto. Quindi incapsula il comando con show_cards@projectName. Il procedimento è lo stesso di show_members, con la differenza che qui si scansiona la lista delle cards di un progetto.

list_projects

list_projects serve a mostrare tutti i progetti a cui appartiene l'utente loggato. La procedura per mostrare i progetti è: nel database tengo salvata una lista di progetti a cui l'utente appartiene e li incapsulo in una stringa in cui ogni progetto è separato dal carattere speciale "$", come per gli altri comandi, il client sostituisce ogni occorrenza del carattere "$" con "\n" in modo da stamparlo a schermo, lato client, in maniera leggibile.

show_card

Incapsulato come show_card@projectName:cardName, questo comando è utilizzabile solo per chi appartiene al progetto, quindi prima di tutto effettuo un controllo per assicurarmi che l'utente loggato appartenga al progetto, successivamente stampo tutte le informazioni della card indicata nei parametri con il metodo showCard di Project.

change_status

Questo comando è il comando più lungo del progetto, infatti ha 4 parametri: change_status@projectName:cardName:oldList:newList. Se l'utente loggato appartiene al progetto effettuo semplicemente l'operazione di moveCard definita dal metodo omonimo nella classe Project (3.3.3) e successivamente scrive sulla chat l'operazione appena effettuata.

get_card_history

Il comando get_card_history è incapsulato nel seguente modo: get_card_history@projectName:cardName. Esso consiste nello stampare la lista di stati attraverso cui è passata la card, si effettua questa operazione grazie al metodo cardHistory di Project e stampando a schermo, lato client, il risultato.

delete_project

Il comando delete_project viene incapsulato così: delete_project@projectName, se c'è un utente loggato in quel terminale, il server procede al controllo della presenza dell'utente nel progetto, se questo controllo va a buon fine, il server controlla che tutte le Cards siano nello stato done, per farlo chiama il metodo areAllCardsDone della classe Project di cui si è parlato sopra (3.3.3).

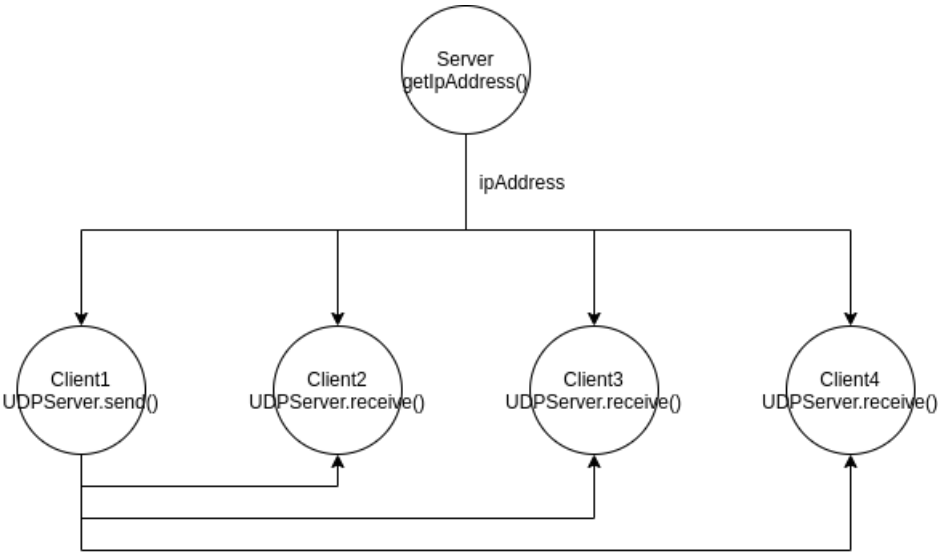
exit

Il comando exit serve a chiudere il client e tutte le connessioni con il server ad esso relative. È stato scelto di sfruttare l'implementazione di logout, qualora il client fosse loggato con un account e poi, in ogni caso, il sistema chiude le connessioni utilizzando il metodo *stopConnection* così implementato:

```
public void stopConnection() throws IOException {
       in.close();
       out.close();
       clientSocket.close();
}
```

6 UDP

Sia la classe Client che la classe Server UDP sono lato Client per evitare una piena centralizzazione del progetto, l'unica azione in cui è coinvolto il server è l'ottenimento dell'indirizzo ip del progetto da parte di ogni client.



6.1 UDP Server

Contiene un singolo metodo statico

```
public static void send(String msg, String ip, String username)
```

che viene usato dal client per mandare i messaggi ogni qualvolta venga invocata la sendMessage da un utente. Esso prende in input il messaggio, l'ip su cui è situata la chat a cui si vuole mandare il suddetto messaggio e l'username che manda il messaggio. Come prima cosa aggiungo al messaggio dell'utente l'username con il carattere ":" in modo da far sapere a tutti i destinatari l'autore del messaggio, successivamente si eseguono tutte le azioni di default per mandare un DatagramPacket: si definisce il gruppo identificato per mezzo dell'ip, che verrà utilizzato per il multicasting, si istanzia l'oggetto MulticastSocket sulla porta 20005, si inserisce il messaggio in un byte array, si incapsula il tutto in un DatagramPacket e si spedisce per mezzo della socket.

6.2 UDP Client

Il client contiene due metodi:

```
private void receive()
public void run()
```

Il client si occupa sostanzialmente della ricezione dei messaggi ricevuti dal gruppo, si istanzia la socket sulla porta 20005, si effettua la joinGroup presso l'InetAddress del gruppo e si avvia un ciclo while che ha lo scopo di ricevere i DatagramPacket dalla socket precedentemente istanziata finchè l'utente non rimane loggato, converto in String i byte che arrivano e lo stampo a schermo. Una volta terminato il ciclo, visto che un utente può partecipare anche a più chat in una sola esecuzione del programma, chiamo la leaveGroup per chiudere la connessione con quel determinato gruppo relativo al progetto, qualora l'utente abbia la necessità di partecipare ad un altro gruppo. L'altro thread, è semplicemente l'override del metodo run per startare il thread che contiene il solo metodo di ricezione dei messaggi sulla chat del client.