Progetto di Laboratorio di Programmazione di Rete

WordQuizzle – 2019/2020

Pinna Matteo

UNIVERSITÀ DI PISA

Indice

1. Introduzione

Il progetto ha lo scopo di sviluppare un sistema Client-Server per la gestione di sfide di traduzione IT-ENG tra gli utenti che si registrano al servizio. Le due principali componenti, Client e Server, comunicano tra di loro tramite l'utilizzo dei protocolli UDP, RMI e TCP per le operazioni di:

- **UDP:** richiesta/gestione sfida
- **RMI:** registrazione nuovo utente
- **TCP:** le restanti

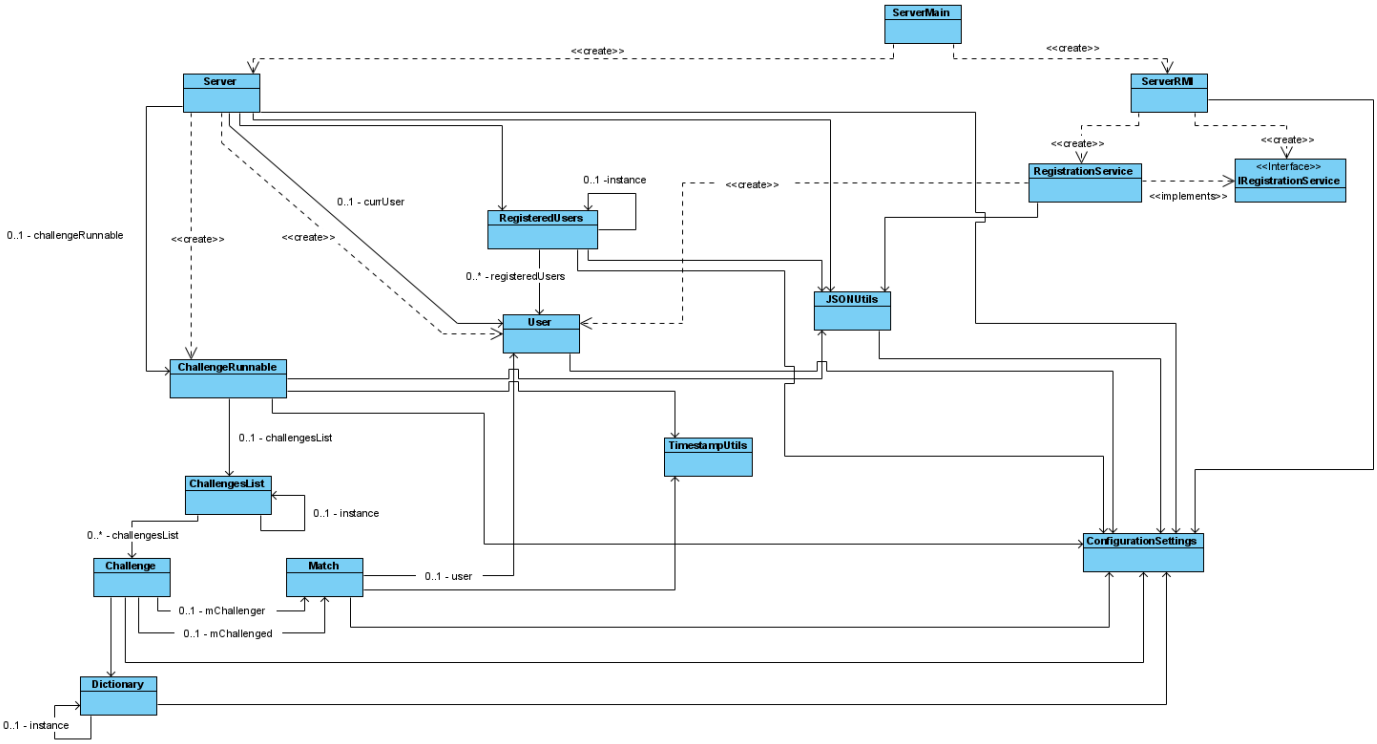
Per facilitare sia la comprensione che la modifica dei più importanti valori di default del progetto, è stata implementata una classe che li mantiene (*ConfigurationSettings.java*).

Il sistema è progettato per operare in locale pertanto le porte dei client successivi al primo, il quale occuperà quella di default, devono essere specificate da linea di comando.

2. Server

La gestione dei client che vogliono accedere al servizio viene effettuata tramite un Selector (NIO), questa scelta risulta più efficiente rispetto all'implementazione multithreaded.

2.1 Server.classi



- **User:** mantiene dati utenti e quelli necessari alla comunicazione con il server
- **RegisteredUsers:** mantiene la lista degli utenti registrati e metodi per la gestione di quest'ultima
- **Server:** gestisce richieste TCP dei client tramite Selector ed un Parser per il riconoscimento dei comandi
- **Main:** fa partire server TCP e RMI

UTILS

- **JSONUtils:** Utils per lettura, creazione e salvataggio file e oggetti JSON
- **TimestampUtils:** Utils per setting e interrogazione dei timeout (partite e richiesta sfida)
- **Dictionary:** Utils per il caricamento delle parole dal file *words.txt* e per reperire le parole random per una sfida
- **ConfigurationSettings :** contiene settings di default

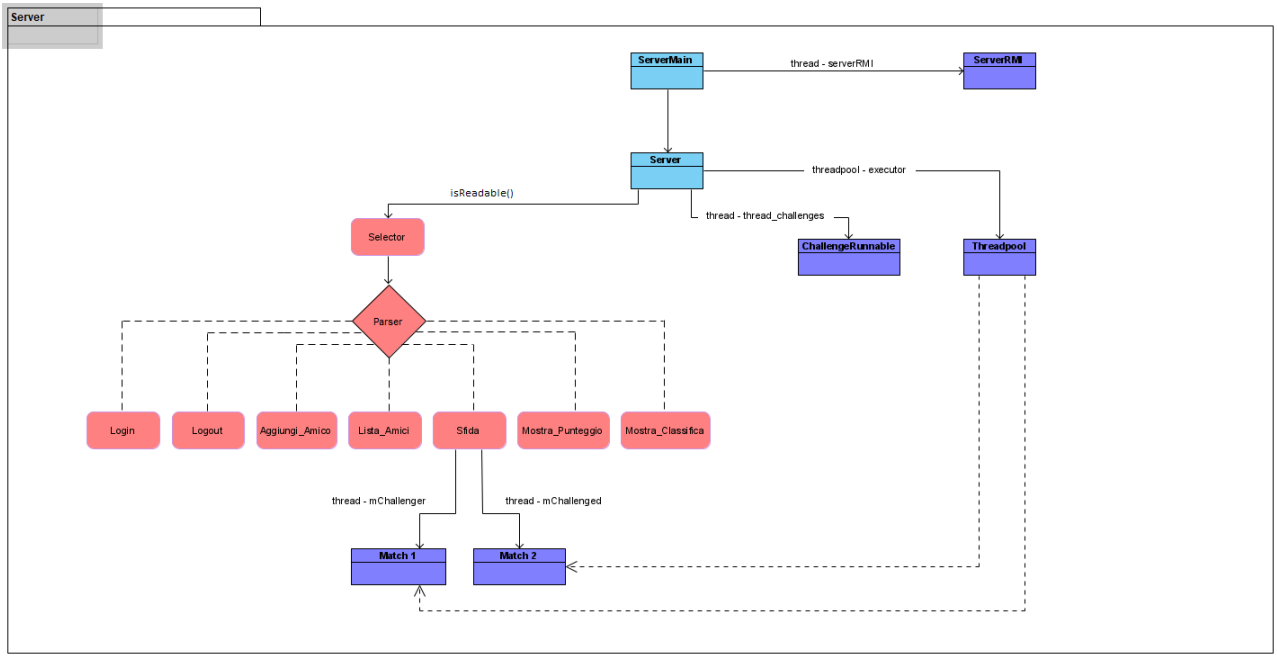
CHALLENGE

- **Match (thread):** *vedere SERVER.THREADS*
- **Challenge:** gestisce la sfida tra due utenti ed invia i essaggi di vittoria, sconfitta o pareggio
- **ChallengesList:** mantiene la lista delle sfide in esecuzione
- **ChallengeRunnable (thread):** *vedere SERVER.THREADS*

RMI

- **IRegistrationService:** interfaccia dell'oggetto remoto RMI
- **RegistrationService**: contiene il metodo per la registrazione ed effettua il salvataggio su file JSON
- **ServerRMI (thread):** *vedere SERVER.THREADS*

2.2 Server.threads



- **ServerRMI:** gestisce richieste registrazione
- **ChallengeRunnable:** gestisce le sfide in esecuzione e loro terminazione
- **Match sfidante:** gestisce la partita dell'utente sfidante e invia il messaggio con le sue statistiche
- **Match sfidato:** gestisce la partita dell'utente sfidato e invia il messaggio con le sue statistiche

2.3 Server.concorrenza

Le classi in cui deve essere gestita la concorrenza sono le seguenti

- **ChallengesList**
- **RegisteredUsers**
- **User**
- **Match**

Le due liste *ChallengesList* e *RegisteredUsers*, rispettivamente di sfide e utenti, sono state realizzate con il singleton-pattern in modo che sia garantita, in tutto il programma, una sola istanza in esecuzione delle due classi. In questo modo viene eliminata la possibilità di creare istanze di tali oggetti non necessarie.

I metodi accessibili da più thread,sia *Match* che *ChallengeRunnable*, sono stati resi synchronized, mentre per gestire la concorrenza di alcuni valori booleani è stato utilizzato il tipo AtomicBoolean, necessario nei casi in cui i valori possono essere controllati e modificati da più thread.

2.4 Strutture Dati

Sono state utilizzate delle *ConcurrentHashMap* per migliorare le performance dove era necessario garantire l'accesso concorrente di thread differenti:

- **User:** per la lista amici
- **RegisteredUsers:** per la lista di utenti registrati
- **ChallengesList:** per la lista di sfide in esecuzione

Sono state invece utilizzate delle *ArrayList* per la memorizzazione delle parole utilizzabili, di quelle che vengono scelta casualmente per una sfida e delle relative traduzioni. La corrispondenza parola-traduzione viene mantenuta grazie all'indice dell'*ArrayList*.

- **Dictionary:** caricamento parole dal file *words.txt*
- **Challenge:** parole e traduzioni per la sfida
- **Match:** parole e traduzioni del match (le stesse della Challenge corrispondente)

2.5 Gestione Sfida

Per la gestione della sfida sono state realizzate quattro (4) classi: *ChallengeRunnable.java*, *ChallengesList.java*, *Challenge.java* e *Match.java*:

- **Match (thread):** mantiene le informazioni della singola partita associata ad un utente e gestisce la verifica della risposta data dall'utente rispetto alla corretta traduzione. Alla conclusione della sfida o timeout invia le statistiche delle partite all'utente.
- **Challenge:** mantiene le informazioni relative alla sfida, nonché le istanze dei due *Match*. Effettua il caricamento delle parole da tradurre e reperisce le traduzioni.
- **ChallengesList:** lista di oggetti Challenge, contiene metodi utili per l'aggiunta di nuove sfide e la rimozione di quelle concluse o scadute.
- **ChallengeRunnable (thread):** effettua pooling sulla lista di sfide in esecuzione (*ChallengesList*) ed invia i messaggi di vittoria e sconfitta agli utenti.

2.6 Gestione parole e traduzioni

Dictionary.java è un singleton in cui vengono caricate le parole utilizzabili per il servizio, mantenute nel file *words.txt*, e che offre anche un metodo *getWords(),* poi utilizzato in *Challenge*, per permutare l'insieme delle parole prima di ogni sfida e per reperire un numero default (5) di queste ultime.

In *Challenge.java* troviamo invece il metodo che si occupa dell'interazione con l'API e quindi di reperire le traduzioni. Il metodo *getTranslation (String word)* richiede la traduzione di una singola parola che viene poi memorizzata nell'ArrayList contenente le traduzioni.

2.7 Salvataggio dati

Le informazioni degli utenti sono rese persistenti tramite il salvataggio, su file JSON (*WQusers.json*), di tutte le modifiche che vengono apportate:

- **Registrazione utente**
- **Aggiunta amici**
- **Modifiche ai punteggi dovute alle sfide**

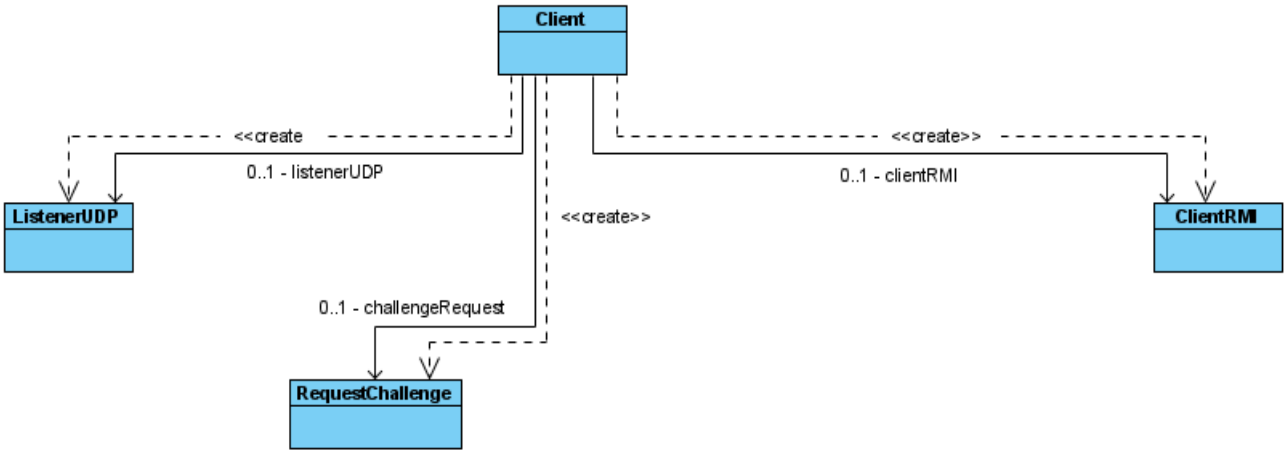
In questo modo viene minimizzata la possibilità di perdita dati dovuta ad un crash improvviso a discapito delle performance visto che viene eseguito un salvataggio dopo ogni singola modifica.

Il salvataggio viene eseguito dal metodo *JSONUtils.saveJSONFile()* in cui viene utilizzata la libreria *Json-simple.*

3. Client

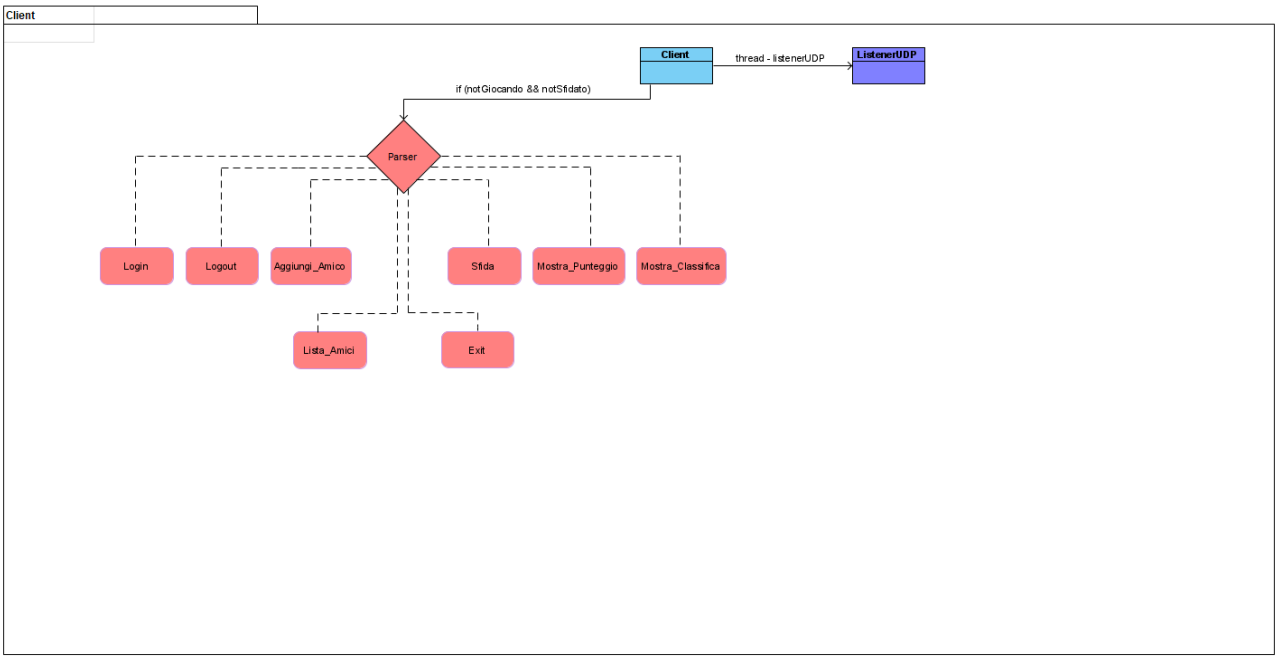
Per il Client si è optato per l'utilizzo di una semplice interfaccia da console, invece che quello di una GUI, che offre all'utente che vuole accedere al servizio un comodo comando *'—help'* per la visualizzazione di tutte le operazioni disponibili. Il Client informa l'utente della disponibilità di tale comando tramite un messaggio di benvenuto stampato a schermo al momento dell'accesso al Client.

3.1 Client.classi



- **Client:** main del client che gestisce l'interazione con il server, utilizza un Parser per il riconoscimento dei comandi e un thread per gestire richieste di sfida
- **ListenerUDP:** *vedere SERVER.THREADS*
- **RequestChallenge:** gestisce l'invio di una richiesta di sfida ad un altro utente
- **ClientRMI:** gestisce la registrazione lato client

3.2 Client.threads



È presente un unico thread che resta in ascolto su porta UDP per eventuali richieste di sfida (*ListenerUDP.java*), questo comunica con il main del Client tramite una classe ausiliaria (*ChallengeRequest*.java).

3.3 Gestione richiesta sfida

Nel caso in cui l'utente risponda (*si/no*) oppure scada il timer, il metodo *setChallengeReply(String reply)* della classe *ListenerUDP.java* imposta la risposta, vuota se il timer è scaduto, modifica il *boolean toAnswer* in modo tale che il thread che manderà l'esito della risposta allo sfidante venga sbloccato.

4. Istruzioni esecuzione

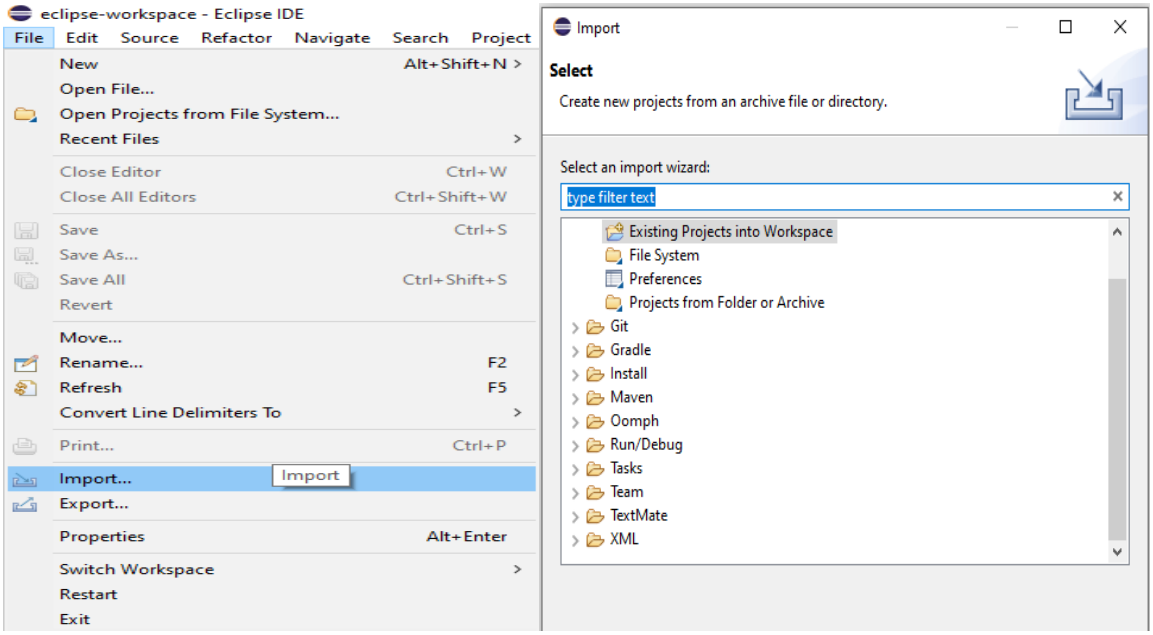
Comandi

- **Login <nickUtente> <password>:** effettua il login
- **Logout:** effettua il logout
- **Aggiungi_amico <nickAmico>:** crea relazione di amicizia
- **Lista_amici:** restituisce la lista dei propri amici
- **Sfida <nickAmico>:** richiede una sfida ad un amico
- **Mostra_Punteggio:** restituisce il proprio punteggio
- **Mostra_Classifica:** restituisce la classifica della propria lista amici
- **Exit:** effettua logout e chiude il servizio client

Eclipse

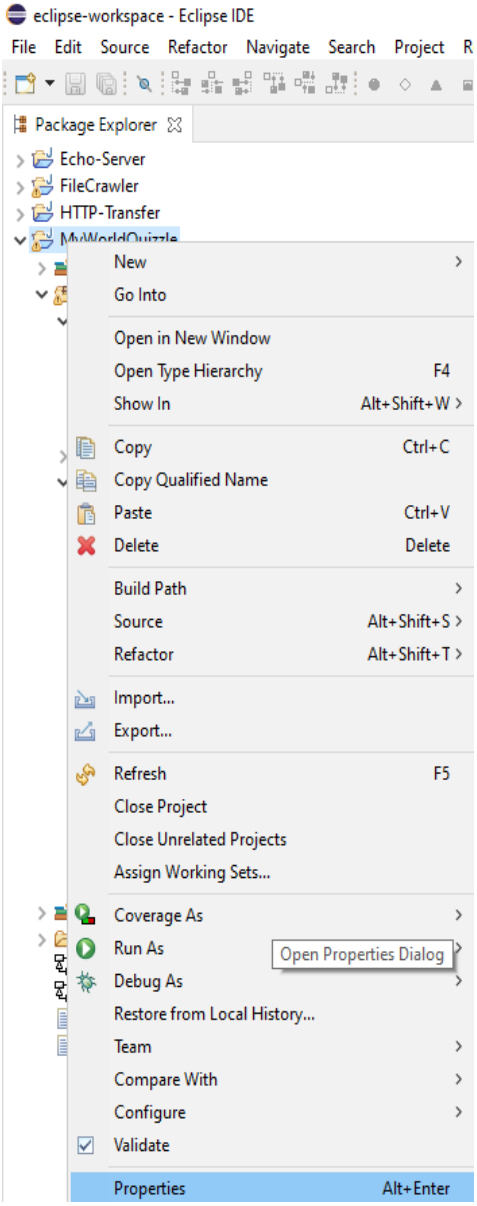
Il progetto è stato realizzato su Esclipse (Java 11).

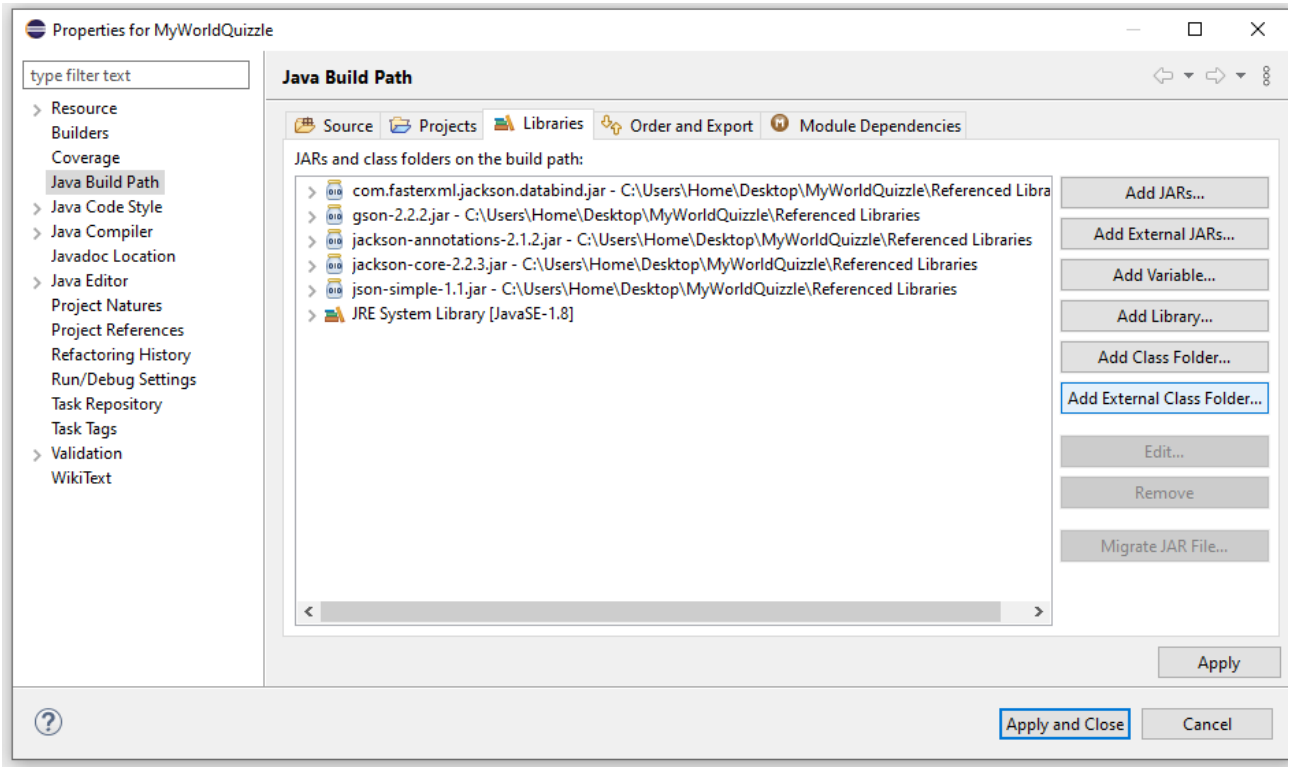
Aprire Eclipse → File → Import → Existing Project into workspace → Selezionare il File System del progetto



Dopo che il progetto sarà aggiunto al workspace, nel caso in cui il percorso delle librerie esterne non sia riconosciuto automaticamente, sarà necessario importale manualmente dalla cartella Referenced Libraries, pertanto con Eclipse ancora aperto:

Right-click sul progetto → Properties → Java Build Path → Add External External Class Folder





Il progetto è pronto per essere eseguito, per far partire il server è sufficiente selezionare il *package Server* e cliccare su *Run*

Per quanto riguarda il Client invece, tenuto conto del fatto che il progetto è realizzato in local, sarà necessario specificare la porta per i client successivi al primo (quella di default sarà già occupata):

Tasto destro sul progetto → Run As → Run Configurations… → Arguments → specificare portaUDP > 15002 (default) → Run