Alessio Perugini mat. 562087 corso B

WordQuizzle

Indice

1 Introduzione

E' stato deciso di realizzare il sistema tenendo a mente la semplicità e la chiarezza del codice. Evitando, ove possibile, di utilizzare ciò che non è stato visto a lezione.
Questa semplicità non ha voluto escludere la performance tenuta a pari importanza della semplicità. L'utilizzo frequente di hashmap e *concurrenthashmap* garantiscono buone prestazioni in lettura e scrittura. L'impiego di FileOutputStream e StreamBuferrizatti migliorano le performance di i/o in particolar modo nelle interazioni con il dizionario ed il file json. Ulteriore documentazione è presente nel progetto con relativi commenti.

2 Architettura

2.1 Server

Gestisce i client che si vogliono collegare con un selector. L'utilizzo di un selector è la scelta più efficiente per gestire l'ingresso dei client. Realizzare un *threadpool* che gestisce un thread associato ad ogni connessione, sarebbe stato troppo dispendioso con l'aumentare dei client connessi.
La gestione dei dati degli utenti, una volta aperto il server è caricato in memoria le informazioni contenute nel file json, viene completamente svolta in memoria per aumentare l'efficienza ed evitare troppi cicli di lettura scrittura i/o che produrrebbero dei rallentamenti non indifferente. Il salvataggio di eventuali nuovi dati viene effettuato alla chiusura del server.
E' possibile realizzare un salvataggio automatico ogni X secondi per prevenire una perdita totale dei dati. Nel caso in cui il server rimanga aperto per molte ore, con una grande quantità di informazioni da salvare, se si verificasse un crash improvviso andrebbero tutti persi. Per gestire nel modo migliore dei modi la *fault-tollerance*, sarebbe necessario creare un sistema journal, cosa molto complessa ma disponibile se si utilizzano i più moderni dbms.

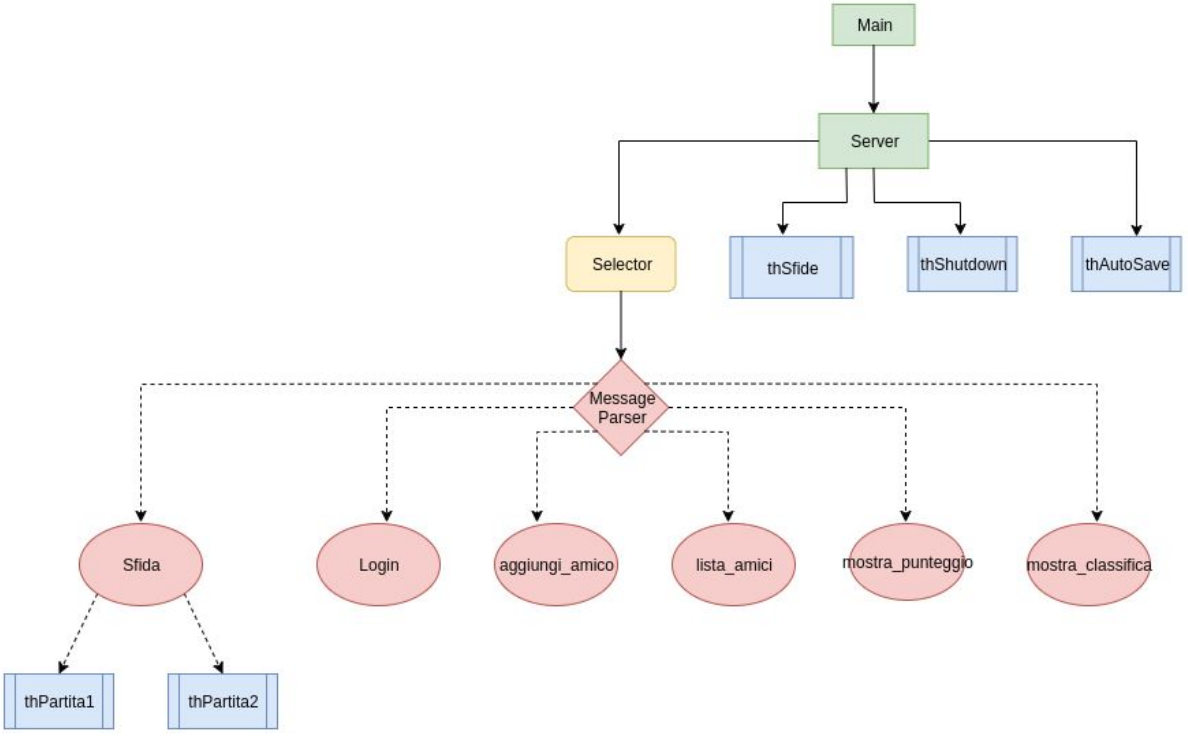
Per la gestione delle sfide è stato realizzato 2 classi di supporto *Sfida* e *Partita*. *Sfida* mantiene traccia delle 2 partite degli utenti. La *partita* tiene traccia dei punteggi che effettua quell'utente e delle interazione che ha con il sistema di verifica della traduzione. Entrambi le partite sono dei *thread* gestiti da un *threadpool* delle partite così posso liberare il selector a future richieste da parte di altri client.
Un ultimo *thread* di verifica della fine della sfida è spawnato all'inizio del server e fa *pooling* sulla struttura dati verificando se una partita è terminata dopo la scadenza del minuto. Questa non è la migliore scelta in quanto è un'attesa attiva ed effettua controlli (anche se poco dispendiosi) che potrebbero essere evitati utilizzando sistemi di altro tipo: come l'impiego di un thread timer.
Il server *RMI*, gestito dal relativo thread, rimane sempre in ascolto sulla porta *50000*.

2.2 Client

Realizzato con interfaccia console, ha 1 solo thread che fa da server udp per accettare o rifiutare le richieste di sfida. La comunicazione tra il thread udp e mainclass avviene mediante una struttura sincronizzata chiamata *RichiestaSfida*. Sfrutta il paradigma domanda-risposta e questo lo rende: semplice nella lettura e nell'implementazione, ma un po' limitante nelle features ed il server va realizzato di conseguenza. Nel caso in cui volessi "rompere" il paradigma rischio di bloccare a vita il client su una read che non arriverà mai. Per esempio, se volessi comunicare l'esito del vincitore in un secondo momento, per consentire al giocatore che termina per primo la partita di effettuare nuove nuove interazioni con il server, si verificherebbe il problema sopra descritto. Un modo per evitare questo sarebbe di realizzare un thread costantemente in lettura e realizzare una coda di comandi e risposte da inviare e da leggere.

3 Schema dei thread e concorrenza

3.1 Server



Thread:

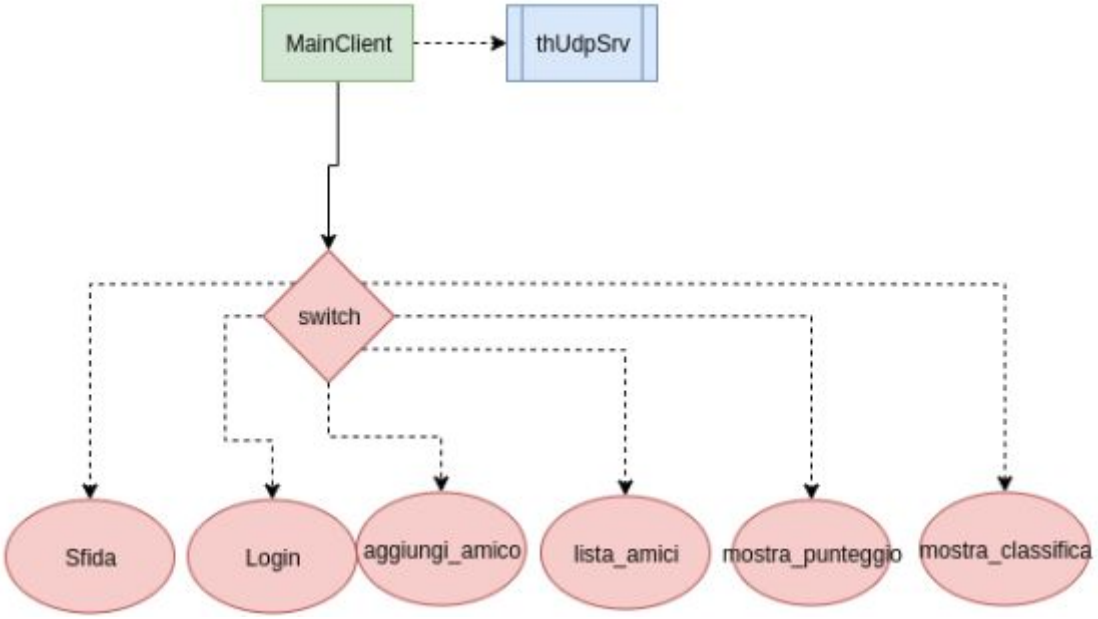
- **thRMI**: gestisce la richieste di registrazione
- **thSfida**: gestisce la fine delle sfide tra utenti
- **thShutdown**: gestisce la chiusura soft del server, aspettando eventuali sfide in corso ed una volta terminate salva su file json le strutture dati in memoria
- **thAutoSave**: se abilitato gestisce il salvataggio automatico della struttura dati in memoria su file json
- **thPartita1** e **thPartita2**: sono 2thread che vengono spawnati per gestire le sfide di user1 e user2 gestiti dal FixedThreadPool

La gestione della concorrenza viene effettuata sulle strutture dati principali dove posso concorrere più thread in lettura e scrittura e sono:

- **ListaUtenti**
- **Utente**
- **Parita**
- **ListaSfide**

ListaUtenti e *ListaSfide* sono classi realizzati con il singleton-pattern per garantire che in tutto il programma sia in esecuzione 1 sola istanza di questi. Consente di realizzare un codice più pulito e elimina la probabilità di creare nuove istanze non necessarie di oggetti.
Ogni funzione presente dentro la *ListaUtenti* e *ListaSfide* sono synchronized poichè il thread del selector attivo con i thread della partita e della sfida potrebbero modificare punteggi, stati della connessione dell'utente ed informazioni critiche in determinati punti del programma.
Per la classe *Utente* le funzioni collegate alla ConcurrentHashMap garantiscono una mutua esclusione e la variabile *connesso* è di tipo AtomicBoolean. Utilizzare monitor o altre strutture di sincronizzazione su una sola variabile sarebbero molto più dispendiose ed una peggiore scalabilità.
Per la classe *Partita* la sola variabile *finita* segue lo stesso ragionamento precedente in questo caso l'unico thread che potrebbe creare problemi è quello che fa pooling sulla lista delle sfide.

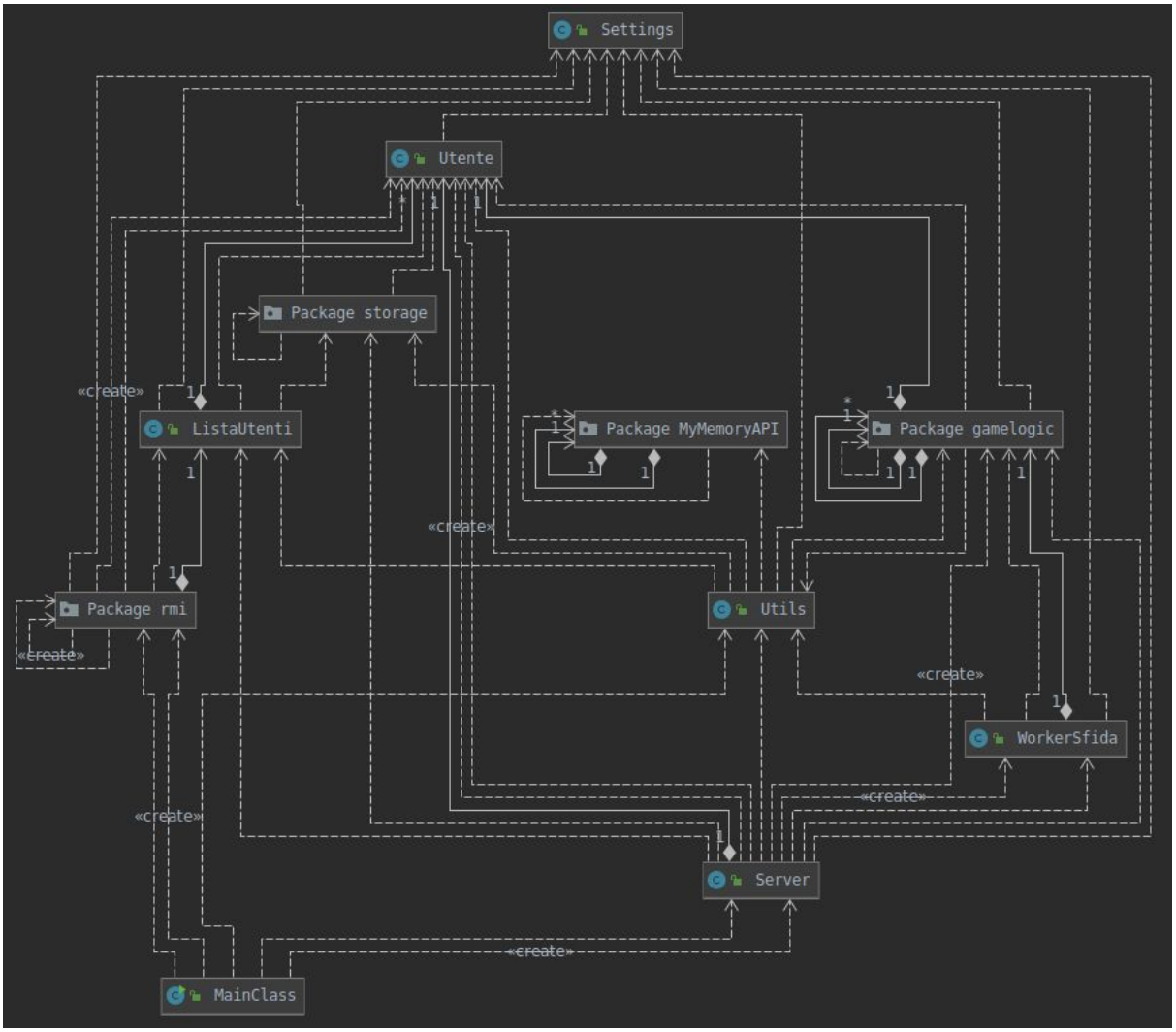
3.2 Client



E' presente il solo thread di gestione del server udp.
La struttura dati più importanti è rappresentata dalla classe *RichiestaSfida*.
Svolge un ruolo chiave per gestire la risposta da parte della console, a fronte delle sfide che arrivano al server udp. Anch'essa realizzata con il singleton-pattern contiene *sfidaToAnser* variabile AtomicBoolean che viene letta a seguito di un input da console. Se è true l'input da console sarà rediretto verso la funzione *challangeRequest,* che sarà in attesa di un input valido di tipo (si/no) fino alla scadenza di un timer, rappresentato dalla variabile *dataScadenzaRichiesta*. Se l'utente risponde con si/no o scade il timer tramite la funzione *setRispostaSfida* cambio lo stato di *sfidaAnswered, che sta su UdpListener,* sbloccando il thread che manderà l'esito della risposta allo sfidante tramite udp. Un modo migliore sarebbe stato quello di creare una ReentrantLock e mettere il thread in wait e realizzare una notify da parte della console.

4 Classi

4.1 Server



(diagramma delle classi e package utilizzati dal server)

- **ListaUtenti**: tiene traccia di tutti gli utenti registrati e connessi
- **MainClass**: spawna il server rmi e tcp
- **Server**: gestisce le richieste tcp
- **Settings**: classe per configurare alcuni parametri di gioco e non.
- **Utente**: gestisce tutti i dati essenziali dell'utente e della sua gestione in memoria
- **Utils**: classe di utility (invio richieste http, costruzione query, log server…)
- **WorkerSfida**: gestisce la terminazione delle sfide

gamelogic:

- **DefaultWords**: enum nel caso in cui non esiste il file del dizionario prendo queste
- **Dizionario**: gestisce le operazioni del dizionario come lettura e generazione parole
- **ListaSfide**: tiene traccia di tutte le sfide presenti

- **Partita**: gestisce il gioco tra client e server
- **Sfida**: tiene traccia delle partite associate alla sfida dei 2 utenti

MyMemoryAPI: contiene tutte le classi che ricostruiscono gli oggetti del json che restituisce l'api rest delle traduzioni. Sono state generate automaticamente[1]

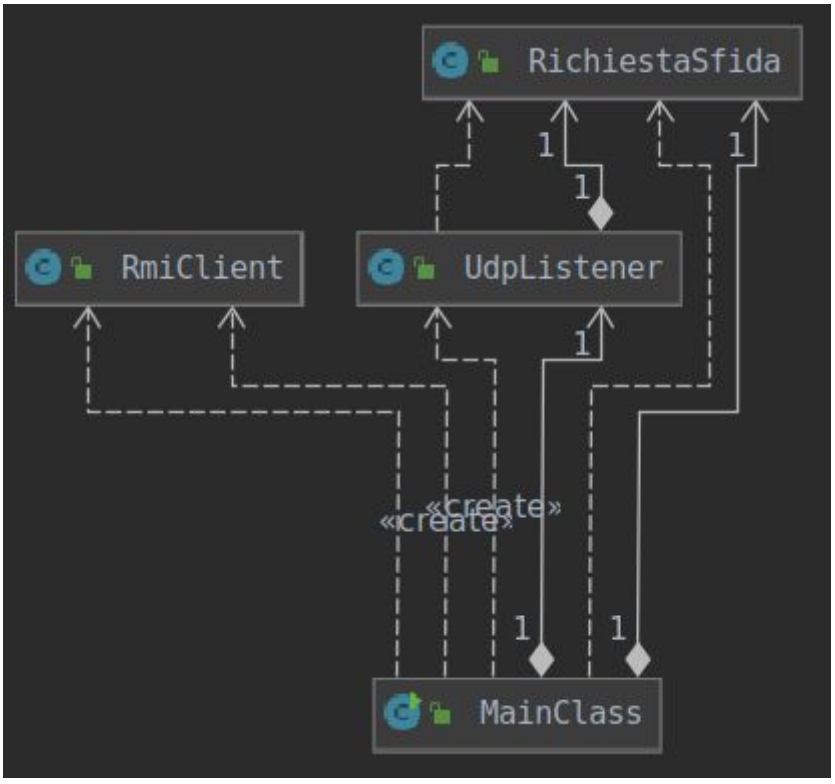
rmi:

- **Registrazione**: usato dal client per registrare l'utente
- **RegistrazioneServer**: servizio rmi del server per consentire la registrazione

storage:

- Storage: serve per gestire lettura e scrittura del json su file o in memoria.

4.2 Client



(Diagramma delle classi)

- **RmiClient**: gestisce la fase di registrazione
- **MainClass**: gestisce il main del client e l'interazione con il server tcp
- **UdpListner**: utilizzato per il server udp che accetta o rifiuta le sfide
- **RichiestaSfida**: sincronizza *thUdpListner* e MainClass sulla risposta delle sfide

5 Istruzioni compilazione ed esecuzione

Il progetto è stato realizzato con maven così come la relativa gestione delle dependency (Jackson per il json). Viene utilizzato java 11.

[1] https://app.quicktype.io/

5.1 Maven

Per installare le dependency:

mvn install

Per compilare:

mvn compile

Per creare gli eseguibili:

mvn package

Oppure con 1 comando solo:

mvn verify

5.2 Docker

Scarica l'immagine docker e viene avviata, risolve ed installa tutte le dependency, inizia la compilazione e crea i file jar.

[2]Linux:

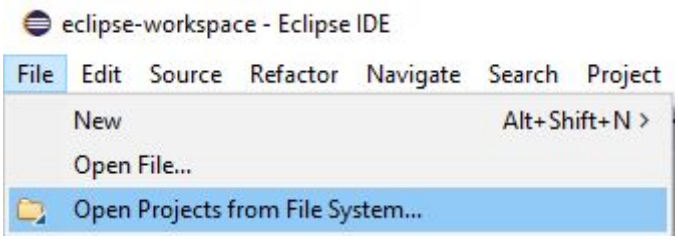
docker run -it --rm --name my-maven-project -v "$(pwd)":/usr/src/mymaven -w /usr/src/mymaven maven:3.6.3-jdk-11-slim mvn verify

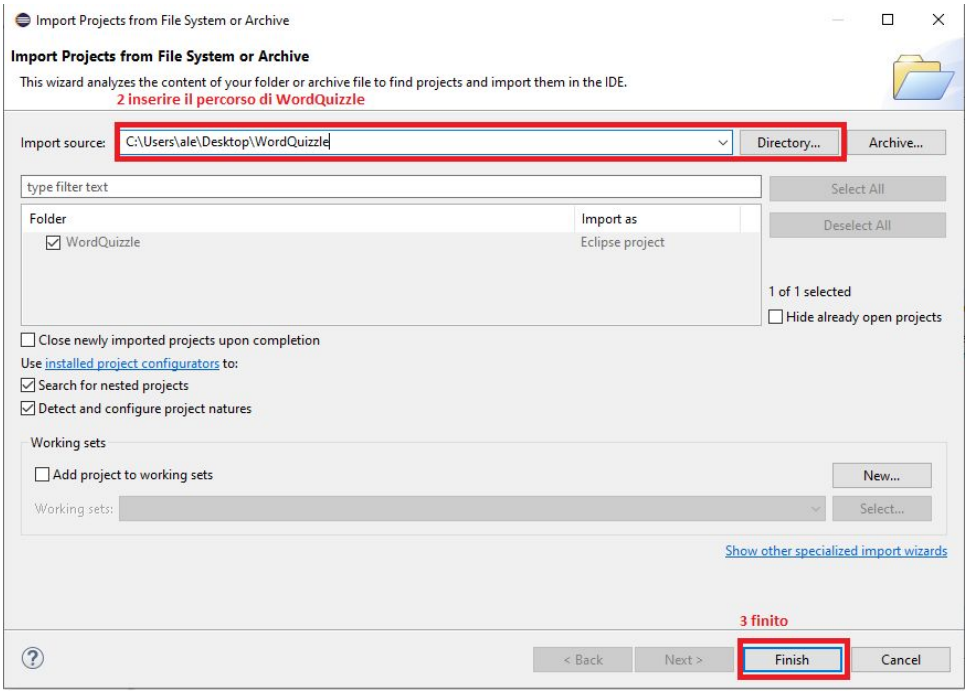
5.3 Eclipse

Assicurarsi di avere java11 installato.

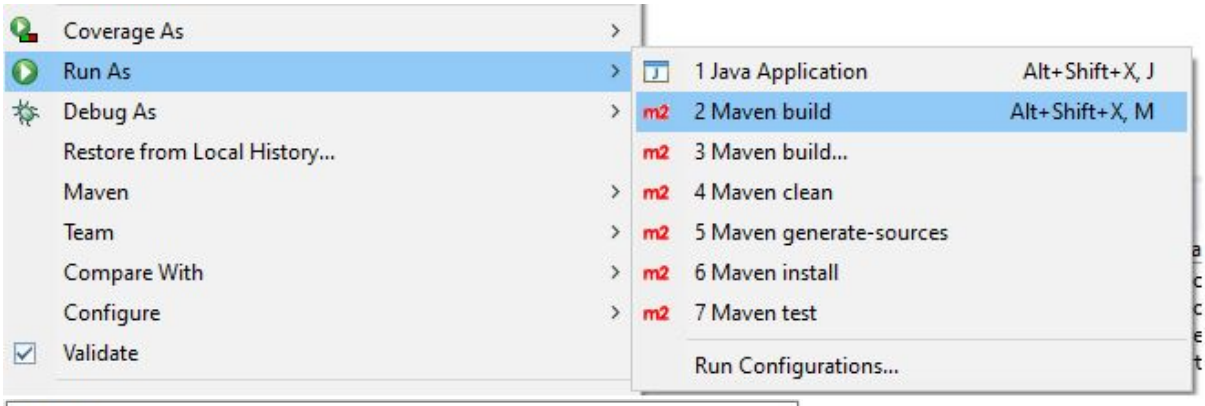
Aprire eclipse e recarsi sulla toolbar.

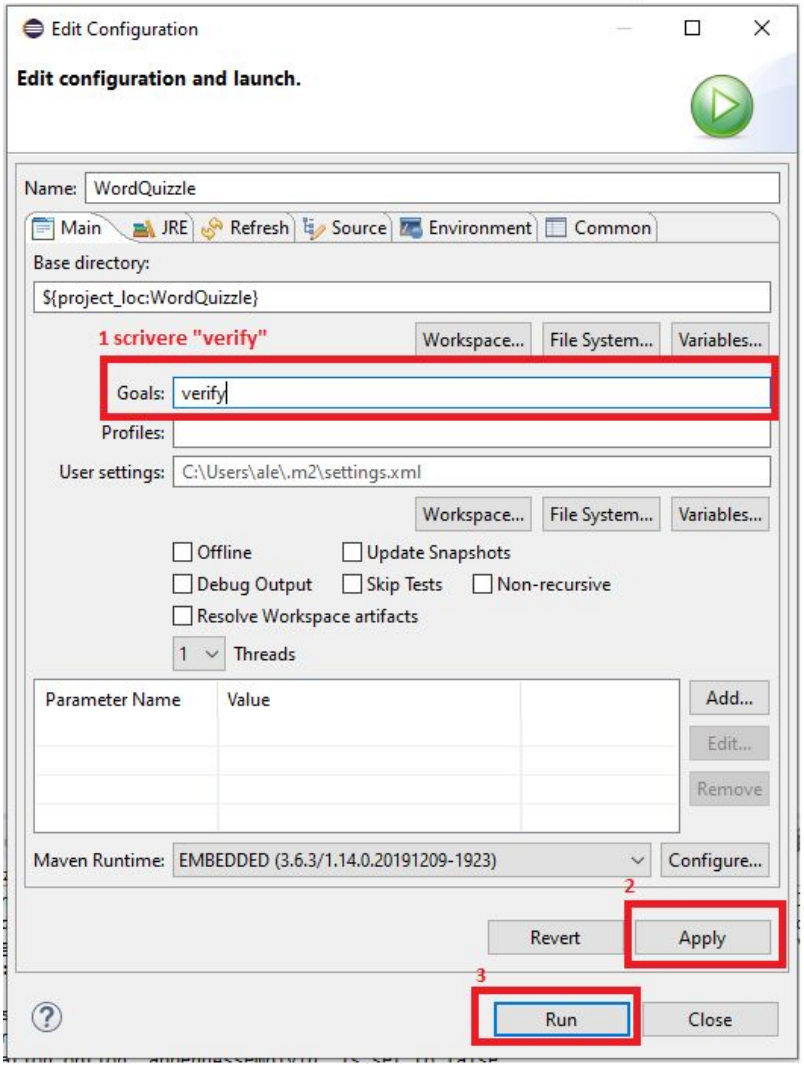
File -> **Open Projects From File Systems**… -> **import source** *(inserire la directory di WordQuizzle)* -> **finish**





Tramite eclipse si fa click destro sul nome del progetto e sul menù contestuale andare in:
Run As -> **Maven build**
Si è la prima volta che si builda si apre la finestra delle configurazione:
Goals: ***verify*** -> **apply** -> **run**
Dentro il campo di testo “Goals” scrivere: “**verify**”





In questo modo maven inizierà ad installare le dependency, compilare e generare i jar del client e del server.

Esecuzione

I file jar si trovano nella cartella **"./WordQuizzle/target"**
Client: "./WordQuizzle/target/**WordQuizzle-client.jar"**
Server: "./WordQuizzle/target/**WordQuizle-server.jar"**

Server: ***java -jar WordQuizle-server.jar***
Client1: ***java -jar WordQuizzle-client.jar {PORTA_UDP}***
Client2: ***java -jar WordQuizzle-client.jar {PORTA_UDP + 1}***
Client3: ***java -jar WordQuizzle-client.jar {PORTA_UDP + 2}***
ClientN: ***java -jar WordQuizzle-client.jar {PORTA_UDP + N}***
Per aprire più client sullo stesso pc passare come primo argomento il numero di **PORTA_UDP** che deve essere necessariamente > **50002** e da lì in poi aprire ogni altro client aumentando il numero di porta sempre maggiore di 50002.
N.B: Per il primo client si può omettere il parametro in quanto lo mette di default a 50002

Esempio pratico

Server: ***java -jar WordQuizle-server.jar***
Client1: ***java -jar WordQuizzle-client.jar 50002***
Client2: ***java -jar WordQuizzle-client.jar 50003***
Client3: ***java -jar WordQuizzle-client.jar 50004***

Sintassi comandi client

Digitare **--help** per un riepilogo di tutti i comandi possibili
registra_utente <**nickUtente** > <**password** > registra l' utente.
login <**nickUtente** > <**password** > effettua il login logout effettua il logout.
logout effettua il logout.
aggiungi_amico <**nickAmico**> crea relazione di amicizia con nickAmico lista_amici mostra la lista dei propri amici.
lista_amici mostra la lista dei propri amici.
sfida <**nickAmico** > richiesta di una sfida a nickAmico.
mostra_punteggio mostra il punteggio dell'utente.
mostra_classifica mostra una classifica degli amici dell'utente (incluso l'utente stesso)
quit per uscire.

6 Fonti

Logo client: http://www.patorjk.com/software/taag/#p=display&f=3D%20Diagonal&t=WordQuizzle
Idee funzioni http:
https://github.com/eugenp/tutorials/tree/master/core-java-modules/core-java-networking-2
Dizionario (sono state eliminate alcune parole):
https://github.com/napolux/paroleitaliane/blob/master/paroleitaliane/1000_parole_italiane_comuni.txt