Wordle 3.0

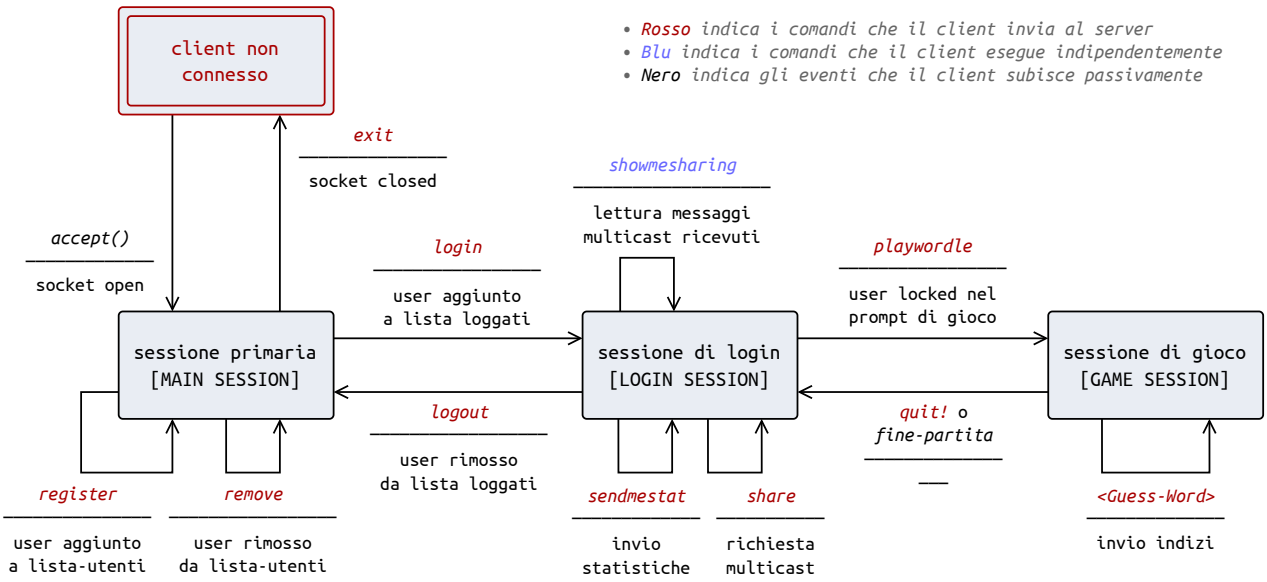
***Matteo Giorgi* 517183**

> *Il progetto consiste nella implementazione di Wordle, un gioco di parole web-based sviluppato da Josh Wardle nel 2021, acquistato poi dal New York Times a fine 2022.*
>
> *Ogni 24h il gioco estrae casualmente dal proprio dizionario una Secret-Word di 5 lettere che il giocatore deve indovinare proponendo una Guessed-Word per ciascuno dei 6 tentativi massimi consentiti. Ad ogni tentativo, Wordle risponderà con indizi utili riguardo le lettere che compongono la Guessed-Word così da aiutare il giocatore a indovinare la Secret-Word giornaliera.*
>
> *Questa implementazione consiste in una versione semplificata del gioco, che conserva la logica di base dell'originale ma apporta modifiche su alcune funzionalità come la condivisione social dei risultati (realizzata qui con un gruppo multicast), e l'assenza di una interfaccia grafica (sostituita da una semplice Command-Line UI).*
>
> *La presente relazione è corredata da documentazione JavaDoc secondo quanto specificato nelle Tecnical-Resources Oracle e affiancata dai sorgenti Java su GitHub.*

Wordle 3.0 usa una classica struttura client-server. Il server legge il proprio file di configurazione e si occupa di caricare in memoria l'elenco degli utenti, l'elenco delle parole (dizionario) e rimanere in attesa di connessioni su una welcome-socket (`ServerSocket`) appositamente allocata su una porta predefinita nel file di configurazione. Agganciato un client, il server lancerà dunque un nuovo Runnable (`Game`) con cui verranno soddisfatte le richieste, per poi rimettersi in attesa di una nuova connessione. Il client invece, dopo la lettura del proprio file di configurazione, ha l'unico scopo di connettersi al server con una socket (`Socket`) e inviare comandi sottoforma di *lines* (stringhe terminanti con il carattere di line-break).

Struttura del Progetto

Prima di entrare nelle specifiche dell'implementazione ecco qua sotto l'ASF che illustra i possibili stati di un client nelle varie fasi di gioco (si consideri ovviamente che client e server abbiano superato la fase iniziale di setup).



Modelli principali

- `Word` e `WordList` gestiscono le parole del gioco: la classe `Word` rappresenta una singola parola, mentre `WordList` il dizionario usato per estrarre ciclicamente la *Secret-Word* e controllare la validità delle *Guessed-Words* inserite dall'utente in partita.
- `User` e `UserList` in modo analogo alle classi precedenti, rappresentano rispettivamente il singolo utente e l'insieme degli utenti registrati sul server del gioco.

Funzionalità del server

- `ServerSetup` e `ServerMain` sono le classi che descrivono le proprietà e identificano il punto di ingresso principale del server.
- `Game` è la classe responsabile dell'interazione con ciascun client, secondo i quattro stati specificati nell'ASF.
- `MulticastSender` è responsabile della lettura dei messaggi di condivisione e dell'invio degli stessi sul canale multicast.

Funzionalità del client

- `ClientSetup` e `ClientMain` analogamente alle loro controparti, rappresentano le proprietà e il punto di ingresso del client.
- `MulticastReceiver` è responsabile della ricezione delle notifiche inviate dal `MulticastSender` sul canale multicast.

`Word`

Classe che rappresenta la *Guess-Word* che i giocatori devono indovinare; è implementata usando due variabili private. Di seguito la struttura di base.

- *`currentWord`*: `String` che identifica la *Secret-Word* corrente.
- *`userSet`*: `Set<String>` che contiene i nomi degli utenti che hanno già giocato la *Secret-Word* corrente.

Come illustrato nell'AST, un utente che avesse già giocato la *Secret-Word* corrente e chiedesse di iniziare una nuova partita, rimarrebbe nella sessione di login, in attesa della *Secret-Word* successiva.

Il costruttore della classe prende come parametro una `String` che rappresenta la parola da indovinare; non è consentito creare una parola *null*, causerebbe il lancio di una `IllegalArgumentException`.

Oltre ai metodi *`getWord`*, *`containsUser`* e *`addUser`*, la classe contiene anche *`getMask`*, necessario a ciascun `Game` per fornire informazioni al client sulla correttezza della *Guessed-Word* inserita, sottoforma di una maschera di caratteri speciali (`X`, `+`, `?`) come da specifica.

`WordList`

Classe che rappresenta il vocabolario utilizzato da *Wordle* per estrarre la *Secret-Word* e controllare la validità delle *Guessed-Word* inserite dagli utenti durante una partita. Di seguito la struttura base.

- *`wordVocabulary`*: `List<String>` che contiene le parole del vocabolario.
- *`currentWord`*: istanza della classe `Word` che rappresenta la parola attualmente selezionata come *Secret-Word*.
- *`wordExtractor`*: `ScheduledExecutorService` che estrae dal vocabolario ogni nuova *Secret-Word*.
- *`extractWord`*: `Runnable` che rappresenta il task di estrazione casuale di una nuova *Secret-Word* dal vocabolario, eseguito da `wordExtractor` ogni `wordTimer` secondi (tempo specificato nel file di configurazione del server).

Si ricorda che il vocabolario fornito contiene solamente stringhe di 10 caratteri e i tentativi consentiti per indovinare la *Guessed-Word* sono fissati ad un massimo di 12, come da specifica.

User

Classe che rappresenta l'utente come una mappa chiave-valore che ne definisce le seguenti proprietà.

- *`user`*: nome dell'utente (String).
- *`password`*: password per il login dell'utente (int).
- *`giocate`*: numero di partite giocate dall'utente (int).
- *`vinte`*: numero di partite vinte dall'utente (int).
- *`streaklast`*: lunghezza della serie di vittorie più recente dell'utente (int).
- *`streakmax`*: lunghezza massima della serie di vittorie ottenuta dall'utente (int).
- *`guessd`*: distribuzione che indica i tentativi impiegati dall'utente per arrivare alla soluzione nelle partite giocate (List).

Per evitare di esporre i metodi di modifica della mappa, invece di estendere una delle implementazioni di `Map`, è stata usata la variabile privata `Map<String, Object>` *`user`* che identifica l'utente con le sue proprietà.

Questa scelta permette di avere una struttura sicura e flessibile, facilmente modificabile al termine di ogni partita con l'apposito metodo *`update`*, che il relativo `Game` userà per aggiornare i dati dell'utente.

Oltre ai metodi getter con i quali recuperare i sette valori della mappa, la classe contiene anche il metodo *`copy`* che permette di creare una copia profonda dell'utente.

La classe ha un unico costruttore che permette di creare un nuovo utente partendo da una `Map<String, Object>`. Non è consentito creare un oggetto della classe `User` usando una mappa *null*, incompleta o contenente associazioni chiave-valore di tipo sbagliato: questi casi causerebbero il lancio di una `NullPointerException` o di una `IllegalArgumentException`.

UserList

Classe che implementa l'elenco degli utenti registrati e contiene due strutture dati principali.

- *`userRegistrati`*: `Map<String, User>` contenente gli utenti registrati al gioco, nella quale il nome utente rappresenta la chiave e l'oggetto `User` il valore associato. L'associazione è biunivoca: il nome identifica univocamemte l'utente registrato.
- *`userLoggati`*: `Set<String>` contenente i nomi degli utenti attualmente loggati al gioco.

La classe utilizza la libreria *Gson* per gestire la serializzazione e deserializzazione degli utenti in formato *JSON*, permettendo di caricare/salvare l'elenco degli utenti da/verso un file `.json` il cui path è specificato nel file di configurazione del server.

ServerSetup

Classe che fornisce al server tutte le informazioni necessarie per la sua configurazione iniziale. Estende la classe `Properties` e viene usata da `ServerMain` per leggere il file di configurazione all'avvio, memorizzando le varie proprietà.

1. *`PORT`*: numero di porta su cui il server si mette in ascolto (`int`).
2. *`PATH_VOCABULARY`*: path del file che contiene il vocabolario (dizionario) del gioco (`String`).
3. *`PATH_JSON`*: path del file *JSON* che contiene i dati degli utenti (`String`).
4. *`WORD_TIMER`*: tempo in secondi che intercorre tra due estrazioni della *Secret-Word* (`int`).

5. *MULTICAST_GROUP_ADDRESS*: indirizzo del gruppo multicast utilizzato per la condivisione dei risultati (`String`).

ServerMain

Classe che contiene il punto di ingresso (`main`) ed è responsabile dell'inizializzazione, dell'attesa di connessioni e della comunicazione con i client. Le variabili principali usate dalla classe sono le seguenti.

- *PATH_CONF*: `String` che rappresenta il path del file di configurazione del server.
- *serverProperties*: istanza di `ServerSetup` che memorizza le proprietà del server.
- *listaUtenti*: istanza di `UserList` che memorizza gli utenti registrati al gioco.
- *listaParole*: istanza di `WordList` che memorizza le parole del gioco.
- *welcomeSocket*: `ServerSocket` utilizzata per accettare connessioni dai client.
- *socket*: `Socket` di connessione per la comunicazione tra `Game` e client.
- *threadPool*: `ExecutorService` che gestisce i client connessi.
- *multicastListener*: `Thread` che legge le notifiche inviate dai `Game` e le inoltra sul canale multicast.
- *shutdownHook*: task atto allo spegnimento di *Wordle*, vedi Shutdown-Hook.

L'esecuzione del `main` può essere schematizzata in tre fasi.

1. *Creazione strutture dati*: all'avvio, il server crea l'elenco degli utenti (`UserList`) e il dizionario (`WordList`).
2. *Attesa connessioni*: il server rimane in attesa di connessioni dai client; ad ogni connessione ricevuta, lancerà una istanza di `Game` che si occuperà di gestire l'interazione con il client.
3. *Gestione socket e thread-pools*: *ServerMain* esegue anche la chiusura delle socket e dei thread-pools, usando il `Runnable` *shutdownHook* che garantisce tutte le operazioni indispensabili per una terminazione appropriata del server.

Shutdown-Hook

Nello specifico, `ServerMain` usa il `Runnable` *shutdownHook* per creare un nuovo `Thread` e registrarlo come *shutdown-hook* usando il `Runtime` della applicazione *Java* corrente.

```
Runtime.getRuntime().addShutdownHook(new Thread(shutdownHook));
```

In caso di una interruzione (`^C` da tastiera o *system-shutdown*), la *JVM* eseguità lo *shutdown-hook* come prima istruzione della procedura di spegnimento. *shutdownHook*, a sua volta, lancerà i seguenti comandi per chiudere *welcomeSocket*, *threadPool*, *multicastListener* e *wordExtractor* di *listaParole* appositamente controllati dai relativi `try-catch`.

```
welcomeSocket.close();
...
threadPool.shutdown();
multicastListener.interrupt();
listaParole.getSheduler().shutdown();
```

In coda allo *shutdown-hook* viene eseguito un blocco *try-catch* per salvare i dati relativi agli utenti registrati in `lib/USERS.json`.

```
try {
    listaUtenti.setRegistrati(serverProperties.getPathJSON());
} catch (IOException e) { ...
```

Game

`Runnable` che gestisce l'interazione tra il server e un singolo client, organizzando la comunicazione in tre sessioni (come da AST).

1. `[MAIN SESSION]`: fase iniziale subito successiva alla *`accept()`*, in cui il client può lanciare i seguenti comandi.
   - *`exit`*: disconnettersi e uscire dal programma.
   - *`register`*: registrarsi al gioco.
   - *`remove`*: cancellarsi dal gioco.
   - *`login`*: autenticarsi al gioco e passare alla sessione di login.
2. `[LOGIN SESSION]`: fase successiva all'esecuzione del comando *`login`*, in cui l'utente può lanciare i seguenti comandi.
   - *`playwordle`*: entrare nella sessione di gioco e iniziare una nuova partita.
   - *`sendmestat`*: richiedere le proprie statistiche di gioco.
   - *`share`*: condividere le proprie statistiche sul canale multicast.
   - *`showmesharing`*: visualizzare le statistiche condivise dagli altri utenti. Questo comando non è fornito dalla classe `Game`, ma eseguito direttamente dal client. Per questo motivo non è incluso nella documentazione *JavaDoc* allegata.
   - *`logout`*: effettuare il logout e tornare alla sessione principale.
3. `[GAME SESSION]`: fase successiva all'esecuzione del comando *`playwordle`*, in cui l'utente può lanciare i seguenti comandi.
   - *`quit!`*: interrompere la partita in corso e tornare alla sessione di login.
   - *`<tentativo>`*: nuova *Guessed-Word*.

I comandi documentati delle specifiche non sono quindi universalmente disponibili ma limitati al contesto della singola sessione. La classe `Game` gestisce l'interazione con l'utente: lo guida attraverso le diverse fasi del gioco fornendo, per ogni fase, un sottoinsieme di funzionalità basate sullo stato corrente (come da AST).

MulticastSender

`Runnable` che gestisce l'invio di notifiche tramite il gruppo multicast. Le principali variabili sono le seguenti.

- *`multicastGroupPort`*: `int` che rappresenta la porta del gruppo multicast.
- *`multicastGroupAddress`*: `String` che rappresenta l'indirizzo del gruppo multicast.
- *`queue`*: coda utilizzata per conservare temporaneamente le notifiche inviate dal server.

Per ogni notifica accodata su *`queue`* da un `Game`, il `MulticastSender` riceverà una *`notify()`*, eseguità una *`queue.poll()`* e si occuperà di inviare una notifica sul canale multicast. Svuotata la coda, il `Runnable` tornerà in attesa di altre notifiche con una *wait()*

ClientSetup

Classe analoga a `ServerSetup`, fornisce al client tutte le informazioni necessarie per la sua configurazione iniziale. Estende la classe `Properties` e viene usata da `ClientMain` per leggere il file di configurazione all'avvio, memorizzando le varie proprietà.

1. *`HOSTNAME`*: nome del server al quale il client deve connettersi (`String`).
2. *`PORT`*: numero di porta del server (`int`).
3. *`MULTICAST_GROUP_ADDRESS`*: indirizzo del gruppo multicast (`String`).
4. *`MULTICAST_GROUP_PORT`*: numero di porta del gruppo multicast (`int`).

ClientMain

Classe che contiene il punto di ingresso (`main`) ed è responsabile per l'inizializzazione e la comunicazione con il server. Di seguito le principali variabili.

- *PATH_CONF*: `String` che rappresenta il path del file di configurazione del client.
- *clientProperties*: istanza di `ClientSetup` che memorizza le proprietà del client.
- *multicastListener*: `Thread` che rimane in ricezione delle notifiche pubbliche sul canale multicast.
- *multicastReceiver*: istanza di `MulticastReceiver` che memorizza le notifiche ricevute.
- *logStatus*: `boolean` che indica lo stato del log (*false* se non-loggato, *true* se loggato).

La particolarità di `ClientMain` sta nel fatto che non mantiene traccia dello stato del client: invia il comando da eseguire sul server, stampa la risposta e, in base alla regex presente in testa a ciascuna *line* letta, decide il comando da eseguire.

MulticastReceiver

`Runnable` che gestisce la ricezione delle notifiche tramite il gruppo multicast. La classe estende `ConcurrentLinkedQueue<String>` fungendo da coda thread-safe delle notifiche ricevute. Le principali variabili sono le seguenti.

- *multicastGroupPort*: numero di porta del gruppo multicast.
- *multicastGroupAddress*: indirizzo del gruppo multicast.
- *userName*: nome dell'utente che ha inviato la richiesta di condivisione sul canale multicast.

Tramite la variabile *multicastReceiver* di `ClientMain`, le notifiche dal gruppo multicast vengono intercettate e messe in coda. Questo meccanismo assicura la ricezione in tempo reale e l'elaborazione asincrona delle notifiche. ai client un ricevimento delle notifiche in tempo reale e una loro elaborazione asincrona.

PDF e corrispondente pagina HTML
sono stati creati con vim-auxilium