Scarabeo

Malucelli Nicolò
nicolo.malucelli@studio.unibo.it

June 2023

Sommario

Il progetto prevede lo sviluppo di una versione online del noto gioco da tavolo “Scarabeo”. In questo gioco, da 2 a 4 utenti competono per la vittoria finale, componendo parole sulla scacchiera, utilizzando ciascuno le pedine in proprio possesso. Ad ogni parola formata corrisponde un punteggio ed a fine partita, il giocatore che ha totalizzato più punti è il vincitore. La soluzione proposta deve possedere tutte le proprietà che ci si aspetta di trovare in un qualsiasi sistema di questo tipo, soprattutto scalabilità e consistenza.

Indice

Capitolo 1

Analisi

L'obiettivo del progetto è quello di realizzare una versione distribuita del famoso gioco da tavolo "Scarabeo"[1] che permetta la presenza di molteplici partite in contemporanea.

1.1 Requisiti funzionali

Di seguito sono elencati i requisiti funzionali dell'applicazione sottoforma di domande e risposte:

- *Come è possibile entrare in una lobby?*

  All'avvio dell'applicazione, ogni utente può creare una nuova lobby oppure unirsi ad una lobby già esistente, indicando il numero della stanza. Una lobby può contenere al più 4 giocatori, pertanto non è possibile unirsi ad una lobby già al completo.

- *Ci sono limitazioni riguardanti la scelta dei nomi?*

  In una stessa lobby non possono essere presenti due giocatori aventi lo stesso nome; tuttavia, uno stesso nome può essere riutilizzato in differenti lobby.

- *Cosa accade se prima che inizi la partita, l'utente che ha creato la lobby crasha o chiude l'applicazione?*

  Se l'utente che ha creato la lobby crasha o chiude l'applicazione, tutti gli altri utenti appartenenti a quella lobby sono riportati alla schermata iniziale.

- *Cosa accade invece se è un altro giocatore a lasciare la lobby?*

  Se un giocatore non creatore della lobby crasha o chiude l'applicazione prima che la partita abbia inizio, quest'ultimo viene rimosso dalla stanza ma la lobby non cessa di esistere.

[1] `https://it.wikipedia.org/wiki/Scarabeo_(gioco)`

- *Chi può avviare una partita?*

  Una partita può essere avviata solamente da colui che ha creato la lobby e solamente quando il numero di partecipanti è maggiore di due. Da quando la partita inizia, non è più possibile per nuovi giocatori unirsi a quella lobby.

- *Cosa accade quando la partita viene avviata?*

  Quando la partita inizia, ad ogni giocatore sono distribuite 8 pedine. Il giocatore di turno può posizionare le proprie pedine sul tabellone e proporre la propria soluzione agli altri giocatori, oppure terminare il proprio turno senza proporre nessuna parola.

- *Quando una disposizione di pedine è considerata valida?*

  In Scarabeo, le parole possono essere scritte verticalmente (dall'alto verso il basso) o orizzontalmente (da sinistra verso destra). Perchè una disposizione di pedine sia considerata valida, tutte le pedine devono contribuire alla formazione di una stessa parola (orizzontalmente o verticalmente). È inoltre necessario che la soluzione proposta intersechi od estenda le parole già presenti sulla scacchiera in almeno un punto. Nel caso in cui sulla scacchiera non siano ancora presenti pedine, la soluzione deve includere il centro della scacchiera.

- *Quali sono le parole ammesse?*

  Il sistema non implementa alcun meccanismo automatico per la verifica della correttezza delle parole; per cui, un giocatore può proporre anche "parole" non esistenti, a patto che la disposizione delle pedine soddisfi le regole del gioco. Spetta quindi agli altri giocatore segnalare l'eventuale incorrettezza delle parole formate.

  Quando un giocatore propone una soluzione, agli altri giocatori sono mostrate le parole formate. A questo punto i giocatori hanno un breve lasso di tempo per segnalare la soluzione proposta. Nel caso in cui metà o più dei giocatori abbia segnalato la soluzione, questa viene annullata ed il turno torna al giocatore che l'aveva proposta.

- *Come può essere usato lo Scarabeo?*

  Lo scarabeo è una pedina jolly che può essere usato al posto di una qualsiasi lettera. Quando l'utente propone una soluzione contenente uno o più scarabei, gli viene chiesto di indicare la lettera rappresenta da ciascuno di essi.

  All'inizio di ogni turno, se il giocatore di turno possiede tra le proprie pedine la lettera che lo scarabeo rappresenta, prende lo scarabeo. Questa sostituzione è effettuata in automatico e non richiede l'intervento del giocatore.

- *Quanto dura un turno?*

  Ogni turno dura al più 90 secondi. Se allo scadere del tempo, il giocatore di turno non ha ancora proposto una soluzione valida, il turno termina e il giocatore di turno totalizza 0 punti.

Quando il giocatore di turno è in attesa dell'accettazione (o segnalazione) della parola, il timer non scorre.

- *Come sono calcolati i punteggi?*

  Quando il giocatore termina il proprio turno e la soluzione è considerata valida dagli altri giocatori, si procede con l'assegnazione del punteggio. Per ogni parola formata è calcolato il rispettivo punteggio, ottenuto sommando tra loro i punteggi di ciascuna lettera, eventualmente moltiplicati per i moltiplicatori di lettera. Il punteggio totale è ottenuto sommando il punteggio di ciascuna parola, eventualmente moltiplicato per eventuali moltiplicatori di parola.

  Al punteggio così ottenuto sono sommati eventuali bonus:

  - 10 punti se sono state usate 6 lettere
  - 30 punti se sono state usate 7 lettere
  - 50 punti se sono state usate 8 lettere
  - 100 punti se è stata formata la parola "scarabeo"

  Al bonus sono sommati altri 10 punti nel caso in cui siano state utilizzate almeno 6 pedine e tra esse non sia presente lo scarabeo.

  Quando il turno termina, al giocatore sono date tante pedine quante ne ha utilizzate, se ancora disponibili.

- *Quando termina una partita? Cosa succede quando la partita termina?*

  Una partita termina quando un giocatore termina tutte le proprie pedine e non può più pescarne di nuove in quanto finite.

  Quando una partita termina, ad ogni giocatore vengono assegnati i bonus ed i malus di fine partita:

  - al vincitore sono assegnati tanti punti pari alla somma di tutte le lettere possedute dagli altri giocatori (lo scarabeo vale 30 punti)
  - ad ogni altro giocatore sono sottratti tanti punti pari alla somma di tutte le lettere da lui possedute (lo scarabeo vale 30 punti)

  È inoltre mostrata la classifica finale e ai giocatori è data la possibilità di tornare alla schermata iniziale.

- *Cosa succede se un giocatore crasha o chiude l'applicazione a partita in corso?*

  Se un giocatore abbandona la partita per un qualsiasi motivo, esso è rimosso dalla partita e quindi dalla rotazione dei turni. Nel caso in cui il giocatore fosse il giocatore di turno, il turno termina ed il gioco passa al giocatore successivo.

  Qualora in una partita rimanga un solo giocatore, la partita termina.

- *Cosa succede se il server crasha?*

  Se il server crasha, indipendentemente da quando ciò accade, a tutti i giocatori connessi in quel momento è mostrata una finestra d'errore e l'applicazione viene chiusa.

1.2 Requisiti non funzionali

Ai requisiti funzionali sopra descritti, si aggiungono una serie di requisiti non funzionali che l'applicazione deve possedere:

- L'applicazione deve essere semplice da utilizzare e l'interfaccia grafica deve essere intuitiva.
- Alla fine di ogni turno, tutti gli utenti devono avere una stessa visione della scacchiera. Casi di inconsistenza non sono pertanto tollerati.
- L'architettura scelta deve permettere all'applicazione di essere scalabile: più partite devono poter essere giocate contemporaneamente e senza che ognuna interferisca in alcun modo con l'altra.
- Casi di disconnessione improvvisa da parte degli utenti devono essere gestiti nella maniera più consona.

1.3 Descrizione degli scenari

Di seguito sono descritti gli scenari in cui gli utenti si possono trovare a seconda della fase di gioco.

1.3.1 Scenario pre-partita

Avviata l'applicazione, un utente può scegliere se creare una nuova lobby o se unirsi ad una lobby già esistente inserendo il codice della lobby. In entrambi i casi l'utente deve inserire lo username (univoco in una lobby).

L'utente che ha creato la lobby può poi decidere di avviare la partita se il numero di giocatori presenti lo permette.

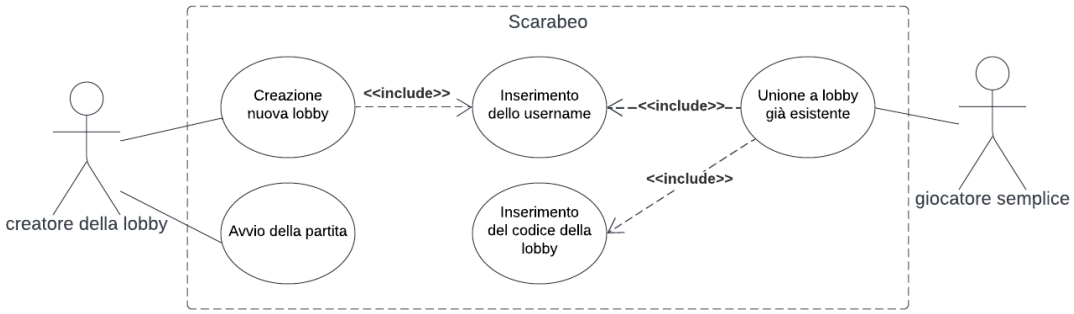


Figura 1.1: Use case diagram pre-partita

1.3.2 Scenario durante la partita

Una volta iniziata la partita, e quindi distribuite le pedine, il giocatore di turno può cominciare a posizionare le proprie pedine sulla scacchiera. Entro la fine del turno è sempre possibile rimuovere le pedine posizionate durante il turno corrente cliccando la corrispondente cella della scacchiera.

Il giocatore di turno può quindi scegliere di proporre una propria soluzione oppure di passare il turno senza posizionare alcuna nuova pedina.

La soluzione proposta può essere segnalata dagli altri giocatori nel caso in cui ci siano obiezioni sulla correttezza della parola. Se una soluzione è segnalata da almeno la metà degli avversari, questa non è considerata valida ed al giocatore di turno è richiesto di proporne un'altra, sempre rispettando i limiti di tempo.

Se allo scadere del tempo, il giocatore non ha trovato una soluzione valida, totalizza zero punti ed il turno passa al giocatore successivo.

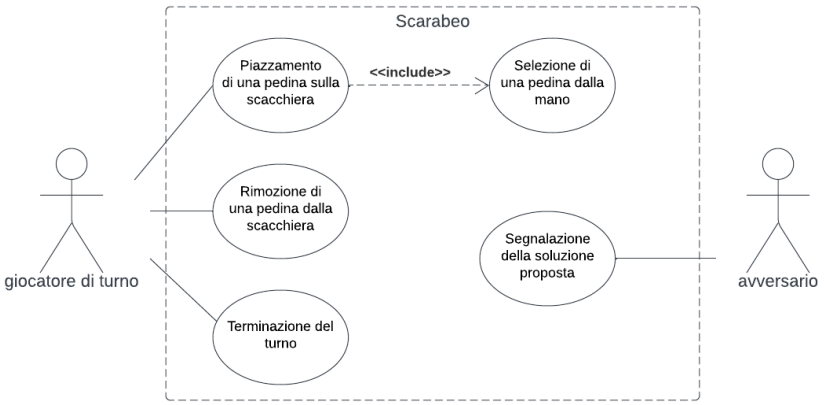


Figura 1.2: Use case diagram a partita in corso

1.3.3 Scenario post-partita

A fine partita, viene mostrata la classifica finale ed i giocatori possono tornare alla schermata iniziale.

Figura 1.3: Use case diagram a partita terminata

1.4 Politiche di autovalutazione

Il collaudo dell'applicazione sarà svolto tramite una serie di test automatici realizzati attraverso il framework JUnit. I test verificheranno che le interazioni tra i vari client ed il server avvengano come previsto e copriranno tutti gli aspetti rilevanti dell'applicazione:

- connettività
- disconnessioni
- corretta gestione delle lobby di gioco
- corretta gestione della partita

Capitolo 2

Design

2.1 Struttura dell'applicazione

Durante la fase di progettazione sono state prese in considerazione due diverse architetture: *client-server* e *peer-to-peer*. Quest'ultima tuttavia è stata presto scartata in quanto non la migliore per questo tipo di applicazioni per motivi legati alla sicurezza.

Si è quindi scelto di implementare un'architettura del tipo *client-server*. Come descritto più nel dettaglio in seguito, la logica di gioco è gestita quasi interamente dal *server*: le uniche operazioni che il *client* svolge, sono operazioni a livello locale che non impattano sugli altri giocatori fintanto che esse non vengono validate dal *server*.

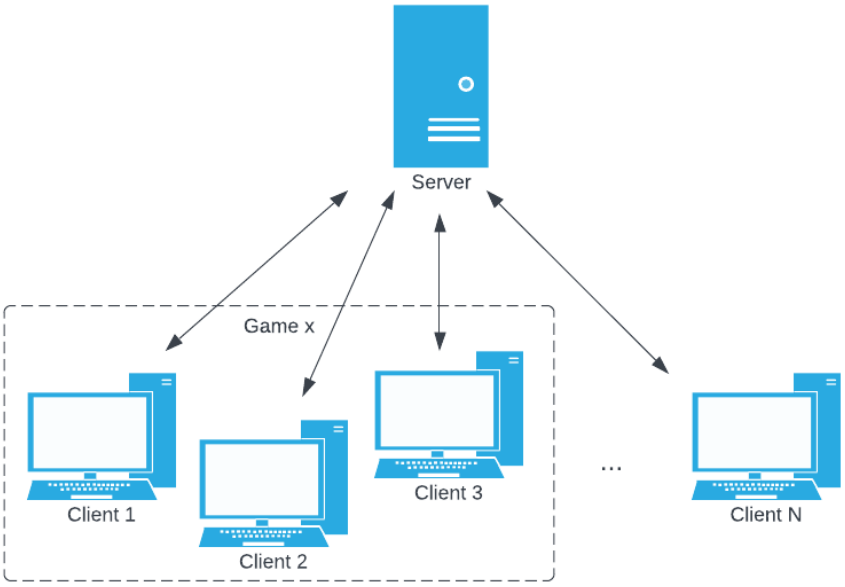


Figura 2.1: Architettura dell'applicazione

La comunicazione tra *client* e *server* avviene attraverso una **connessione TCP**: il *server* mantiene un *socket* per ogni *client* che richiede di connettersi. Questa implementazione consente una comunicazione bidirezionale tra *client* e *server*:

- i *client* inviano **richieste** al *server*;
- il *server* elabora la richiesta del *client* ed invia un messaggio di **risposta**;
- talvolta il *server* invia messaggi al *client* senza che essi abbiano fatto richiesta esplicite (**notifiche**). Le notifiche sono utilizzate sia per gestire aspetti grafici, come per esempio l'aggiornamento della lista dei giocatori, ma anche per gestire aspetti core, come per esempio notificare un giocatore che è iniziato il suo turno.

2.1.1 Struttura del Client

Il *client* è stato organizzato seguendo il **pattern MVC**, come mostrato in figura 2.2. Nonostante la ridotta complessità del *client*, si è scelta questa architettura in quanto permette di suddividere con chiarezza le responsabilità delle varie parti:

- la **view** si occupa degli aspetti prettamente grafici,
- il **model** gestisce gli aspetti di connessione e comunicazione con il *server*,
- il **controller** collega *view* e *model* aggiungendo un livello di astrazione.

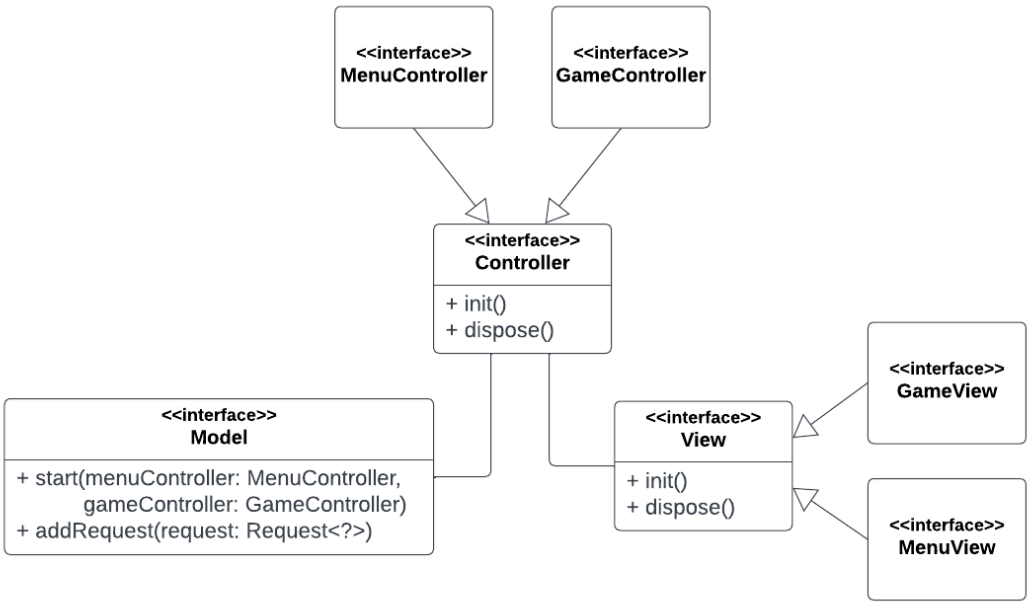


Figura 2.2: Architettura del client

Architettura della View

L'interfaccia grafica è stata realizzata attraverso il framework *Swing*[1] di Java. Poichè non elemento centrale del progetto, non è stato dato peso troppo elevato alla grafica vera e propria dell'applicazione; tuttavia, la soluzione adottata risulta essere piuttosto flessibile grazie al modo in cui è stata strutturata, e consente facilmente di migliorare l'aspetto dell'applicazione.

Entrambe le *view* (`GameViewImpl` e `MenuViewImpl`) estendono la classe astratta `AbstractView`, la quale implementa una serie di funzionalità utili come: mostrare messaggi di informazione o di errore attraverso un *Toast*, mostrare o chiudere la finestra.

Il comportamento di ogni *view* è gestito dal relativo *controller* (`GameControllerImpl` e `MenuControllerImpl`). Quando l'applicazione è avviata, la *view* visibile è la `MenuView`, la quale permette all'utente di creare una nuova lobby o di unirsi ad una esistente.

La `GameView` viene invece visualizzata nel momento in cui inizia la partita. Quest'ultima è costituita da 3 `JPanel`: il principale contiene la scacchiera di gioco, il secondo le pedine possedute dall'utente, mentre il terzo permette di visualizzare la classifica real-time e contiene, a seconda della fase di gioco, alcuni pulsanti di controllo, come per esempio quello per segnalare una parola errata.

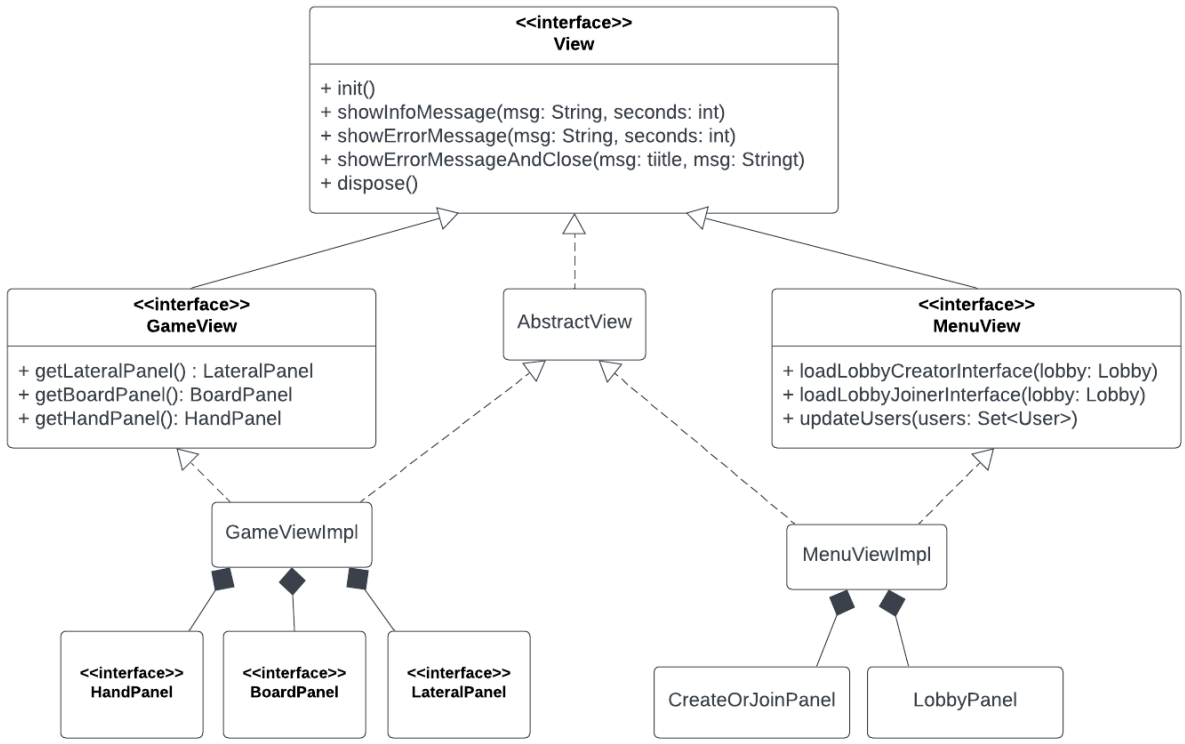


Figura 2.3: Organizzazione della view

[1] `https://it.wikipedia.org/wiki/Swing_(Java)`

Gestione della connessione

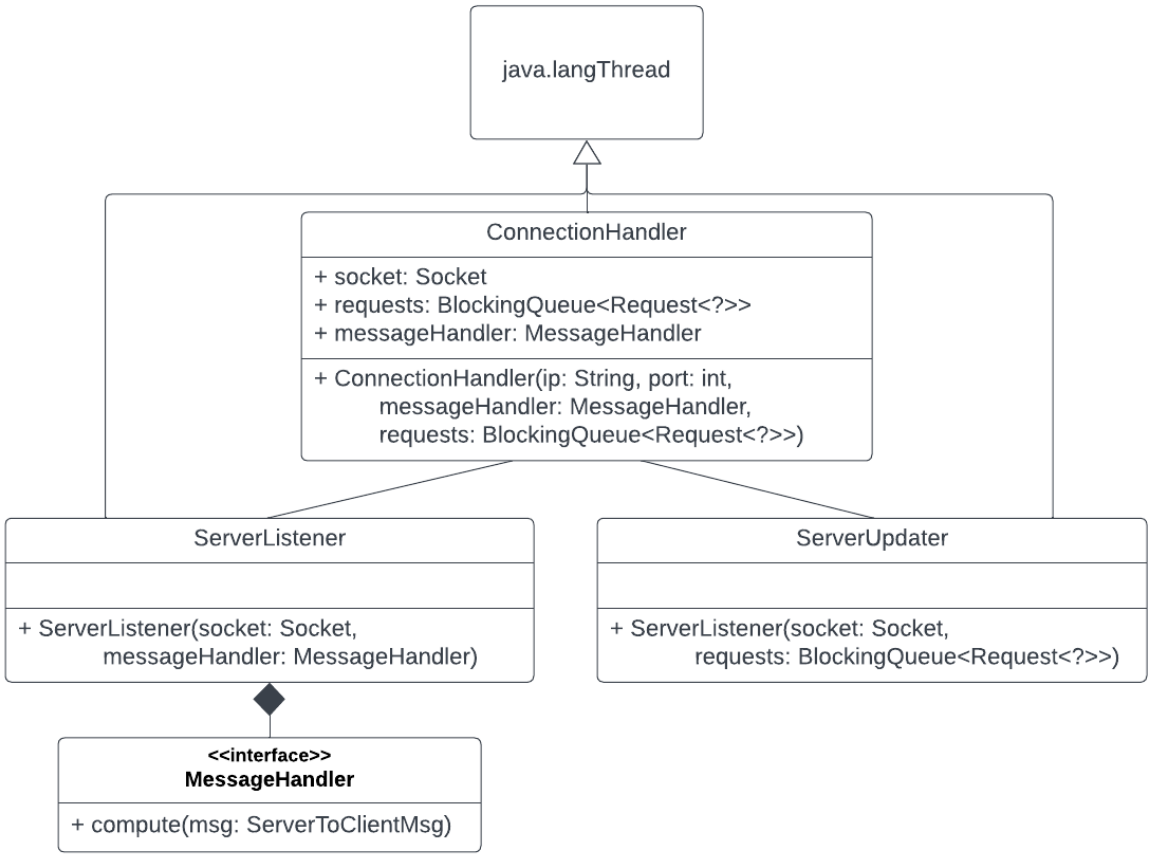


Figura 2.4: architettura di gestione della connessione

Come già accennato, gran parte della complessità risiede nel *server*; pertanto, il *model* del *client* ha le sole funzionalità di gestire la connessione e la comunicazione con quest'ultimo.

Quando viene invocato il metodo `start` del *model*, è creato un nuovo `ConnectionHandler`. Dopo aver istanziato la connessione con l'indirizzo e la porta specificate nel costruttore, il `ConnectionHandler` procede avviando l'esecuzione di un `ServerListener` e di un `ServerUpdater` e termina la propria esecuzione.

Qualora debba verificarsi un errore durante la creazione della connessione, per esempio se il *server* dovesse risultare irraggiungibile, la mancata connessione è notificata all'utente attraverso una `MessageDialog` e l'applicazione viene in seguito terminata.

I due stream del *socket* (input e output) sono quindi gestiti da due diversi flussi d'esecuzione; questo permette al *client* di inviare un messaggio al *server* e allo stesso tempo di rimanere in ascolto di eventuali messaggi in ingresso.

Il `ServerUpdater` è l'entità che si occupa di inviare richieste al *server*. Le richieste sono prelevate dalla coda bloccante `requests`, serializzate attraverso un serializzatore

GSON[2], e inviate al *server*. Ad inserire nuove richieste nella coda delle richieste sono il `MenuController` ed il `GameController`, attraverso il metodo `addRequest` del *model*. Gestendo le richieste in questo modo, si è sempre sicuri che il corretto ordine temporale sarà rispettato.

Il `ServerListener` si pone invece in ascolto di messaggi provenienti dal *server* e per ogni nuovo messaggio ricevuto, invoca il metodo `compute` del `MessageHandler`.

Il `MessageHandler` è l'elemento centrale di questo *model*: esso si occupa a tutti gli effetti di reagire ai messaggi provenienti dal *server*, siano essi risposte a richieste effettuate in precedenza o notifiche.

Nell'eventualità in cui la connessione con il *server* dovesse interrompersi, tale eccezione è catturata dal `ServerListener`, il quale procede notificando il `MessageHandler` con una `ServerCrashedNotification`. A questo punto il `MessageHandler` procede notificando l'utente e chiudendo l'applicazione, in maniera analoga a ciò che succede quando non si riesce ad instaurare la connessione.

2.1.2 Struttura del server

Descrizione del model

Il diagramma delle classi mostrato in figura 2.5 riassume la struttura del *model*, indicando per ogni entità solamente alcuni metodi, questo per evitare di rendere lo schema troppo pesante da leggere.

L'elemento centrale di questa architettura è l'entità `Game`. Ogni `Game` mantiene tutte le informazioni utili per la gestione di una singola partita, dal momento in cui viene creata la lobby, fino a quando la partita non termina.

Poichè all'interno di una partita è necessario gestire una grande varietà di aspetti differenti, ogni oggetto `Game` si compone di una serie di altri oggetti a cui sono delegati determinati compiti:

- Il `TilesDistributor` è l'entità che si occupa della distribuzione delle pedine. Al suo interno, il `TilesDistributor` mantiene una lista di tutte le pedine ancora disponibili (ovvero non distribuite) in quella partita. Attraverso i metodi `getTiles` e `getStartingTiles` è possibile ottenere un insieme di pedine tra quelle ancora disponibili. Nello specifico, il metodo `getStartingTiles` permette di ottenere un insieme di pedine tra le quali appaiono almeno una vocale ed una consonante. Al contrario, utilizzando il metodo `getTiles` non si applica nessun vincolo sull'output.

- Il `PlayersManager` è l'entità che mantiene la lista dei giocatori (`Player`) presenti all'interno della partita. Attraverso il `PlayerManager` è possibile aggiungere e rimuovere giocatori, ottenere informazioni utili come ad esempio chi è il giocatore di turno o sapere se un determinato username è disponibile, ma anche inviare messaggi in broadcast ad un set di giocatori. Questa funzionalità è utilizzata per notificare un

[2] `https://github.com/google/gson`

certo evento a tutti i giocatori non di turno, come per esempio l'inizio della fase di report.

- Come già descritto nei requisiti, una funzionalità chiave di questa applicazione consiste nel poter segnalare una soluzione proposta da un giocatore. Questo aspetto di gioco è gestito dal `ReportHandler`. Attraverso il metodo `waitReportPhaseResult` è possibile avviare la fase di segnalazione, mentre attraverso il metodo `report` è possibile effettuare una segnalazione. Entrambi i metodi prendono in ingresso il *socket* corrisponente a chi ha inviato il messaggio, questo permette di far sì che la fase di report possa essere avviata soltanto dal giocatore di turno e che un giocatore non segnali una stessa soluzione più di una volta.

- Infine, la `Board` si occupa di mantenere memorizzate le pedine posizionate sulla scacchiera, permettendo inoltre di effettuare altre operazioni, quali verificare la validità di una certa soluzione in termini di dispozione delle pedine e calcolare il punteggio di una certa soluzione.

I diversi `Game` esistenti in un certo istante, sono tutti gestiti dal `GameManager`. All'interno del *server* è presente una sola istanza di questa classe. Attraverso questa entità è possibile creare e cancellare nuove partite. È inoltre possibile ottenere un riferimento della partita nella quale si trova un certo utente o a partire da un determinato identificatore. Gli identificatori delle partite sono generati dall'`IdGenerator` che garantisce la loro univocità.

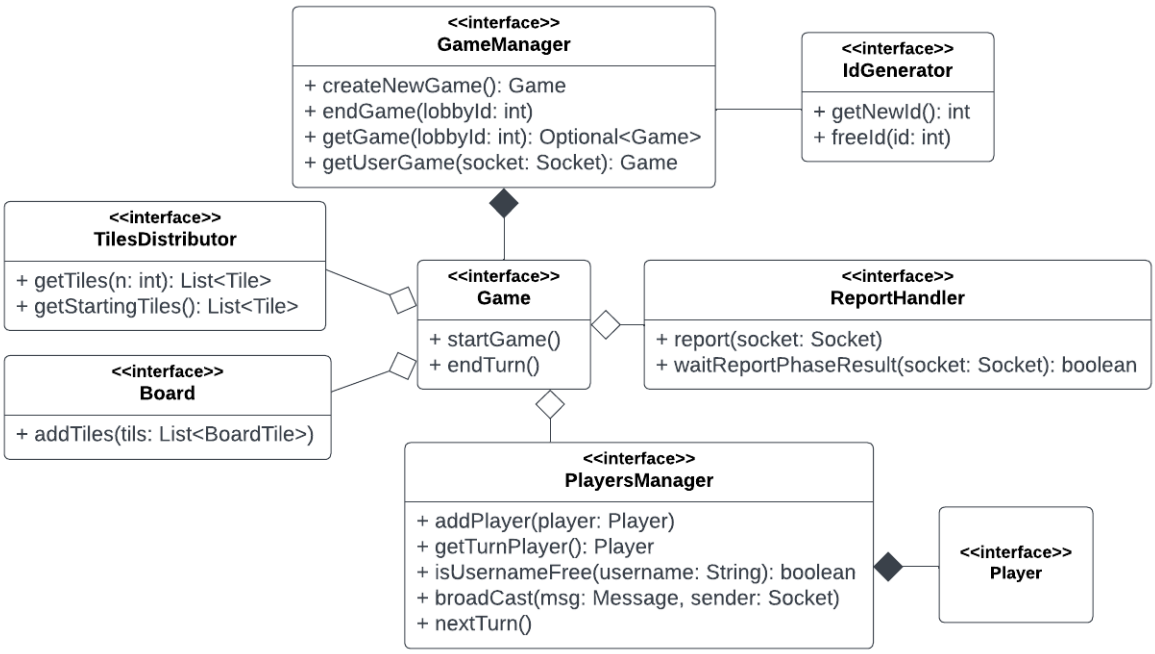


Figura 2.5: Model del server

Gestione delle connessioni

Lato *server*, le connessioni sono gestite in maniera molto simile a quanto avviene nei *client*. Quando viene invocato il metodo `start` dello `ScarabeoServer`, è istanziato un nuovo `ServerSocket`, utilizzando la porta indicata nel costruttore.

A questo punto il *server* si pone in ascolto di nuove connessioni attraverso il metodo `accept` del `ServerSocket`. La gestione di ogni connessione così creata è delegata al `ClientHandler`, il quale crea a sua volta un `ClientListener` ed un `ClientUpdater`, che rispettivamente operano in maniera analoga al `ServerListener` e al `ServerUpdater` descritti in precedenza: il `ClientListener` è in ascolto di nuove richieste che saranno inserite in una coda di richieste, mentre il `ClientUpdater` preleva ed elabora i messaggi ed invia le risposte/notifiche al *client*.

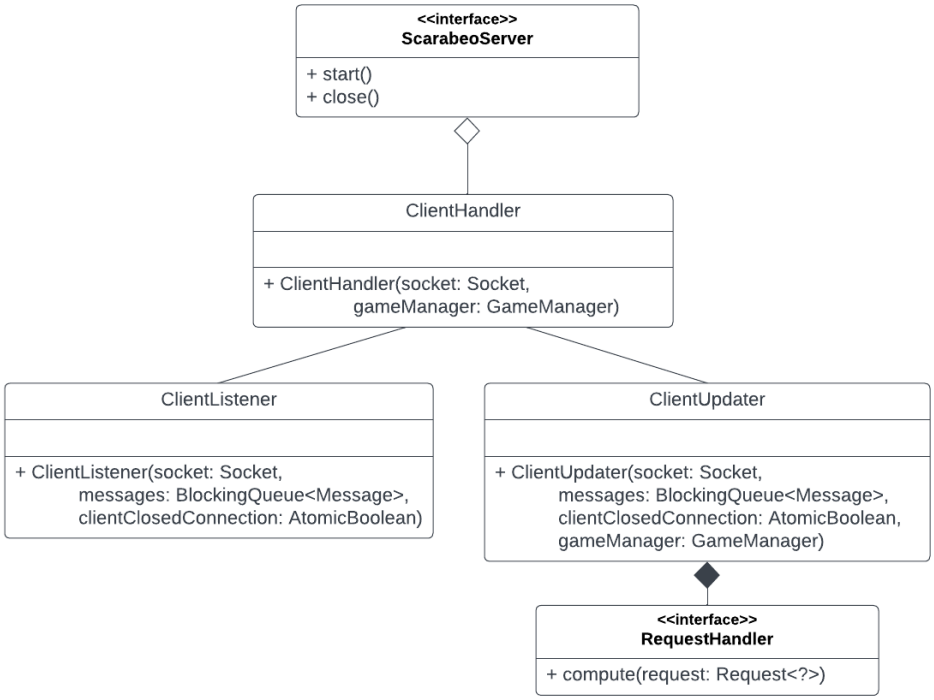


Figura 2.6: Gestione delle connessioni lato Server

2.1.3 Struttura dei messaggi

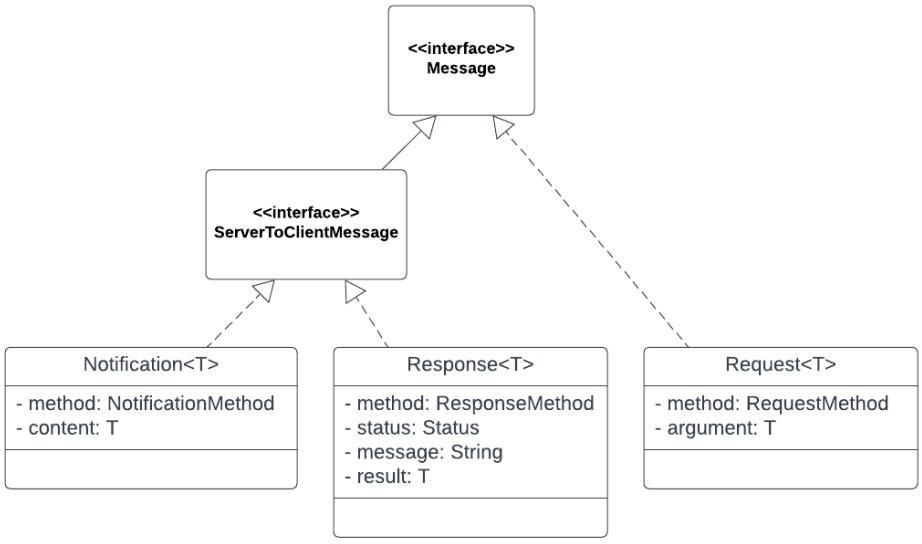


Figura 2.7: Gerarchia dei messaggi

All'interno dell'applicazione possono essere distinte tre diverse tipologie di messaggi, ciascuna delle quali implementa l'interfaccia `Message`. Tutti i messaggi contengono un campo `method`, il cui tipo varia a seconda del tipo di messaggio. Questo campo è fondamentale per la deserializzazione del messaggio stesso.

- Le richieste (`Request`) sono semplici messaggi che viaggiano dal *client* al *server*. Esempi di richieste sono la `CreateLobbyRequest` e la `JoinLobbyRequest`, rispettivamente utilizzate per richiedere la creazione di una nuova lobby e l'unione ad una lobby esistente.

- Le risposte (`Response`) sono messaggi che viaggiano da *server* a *client*. Ogni `Response` è sempre scatenata da una `Request`. Oltre ad indicare metodo e risultato, le risposte contengono uno *status*, utile per capire se la richiesta è andata a buon fine, ed un messaggio testuale. Le prima citate `CreateJoinRequest` e la `JoinLobbyRequest`, dopo essere state elaborate dal server, producono come risposta rispettivamente una `CreateJoinResponse` e una `JoinLobbyResponse`.

- come le risposte, anche le notifiche (`Notification`) sono messaggi che viaggiano da *server* a *client*, ma a differenza di quest'ultime non sono originati da una richiesta ma sono inviati dal *server* al verificarsi di determinati eventi, come ad esempio l'aggiunta di un nuovo giocatore alla lobby (`UserJoinOrLeftLobbyNotification`) o l'inizio di una partita (`GameStartNotification`).

2.2 Comportamento dell'applicazione

Il diagramma degli stati di figura 2.8 mostra il comportamento dell'applicazione dal punto di vista di un utente.

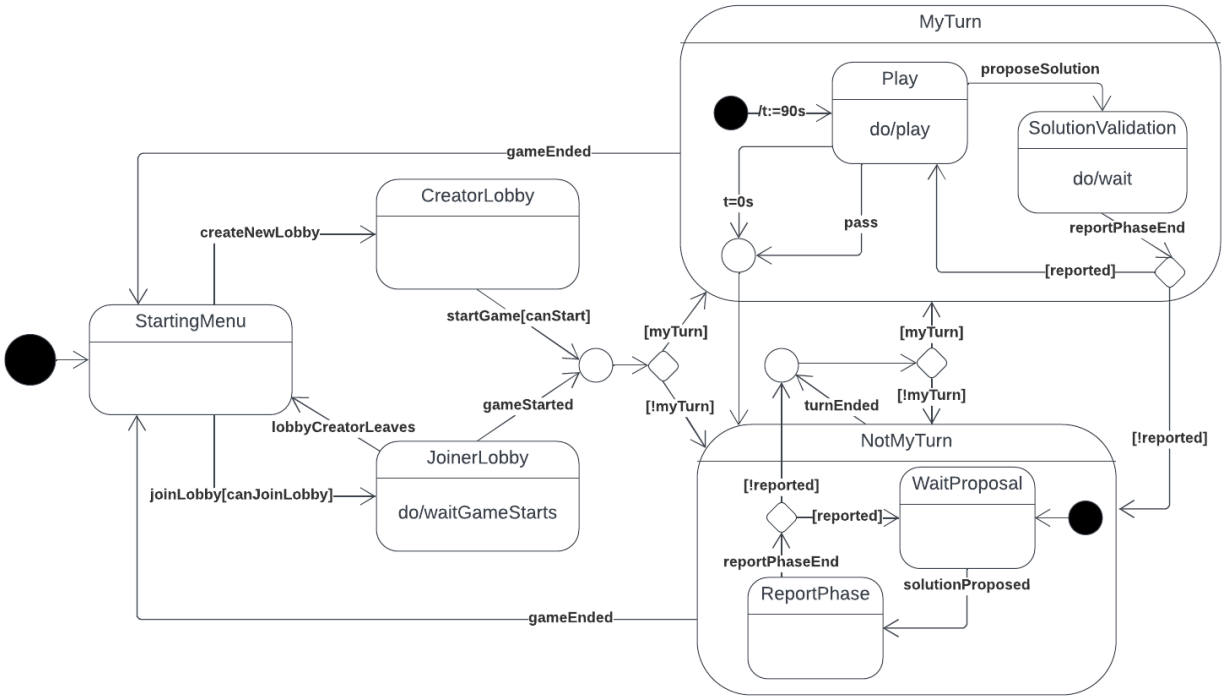


Figura 2.8: Comportamento dell'applicazione dal punto di vista dell'utente

Quando l'applicazione viene avviata, l'utente si trova nel menù iniziale. Qui l'utente può scegliere di creare una nuova lobby o unirsi ad una lobby già esistente, a patto che siano soddisfatte le condizioni descritte in precedenza (username libero, lobby non piena e partita non cominciata).

I giocatori che si sono uniti ad una lobby attendono a questo punto l'inizio della partita che può essere dato solamente dal creatore della lobby e soltanto se sono presenti almeno due giocatori. Nel caso in cui l'utente che ha creato la lobby dovesse lasciare il gioco prima di avviare la partita, gli altri partecipanti della lobby sono riportati al menù iniziale.

Una volta avviata la partita è deciso l'ordine di gioco. Il giocatore di turno può piazzare e rimuovere le pedine sulla scacchiera, mentre i giocatori non di turno attendono una proposta di soluzione. Quando il giocatore di turno propone una soluzione, essa è sottoposta alla validazione degli altri giocatori che possono decidere di segnalarla qualora non considerino valide una o più parole.

Terminata la fase di segnalazione, se la proposta ha passato la validazione degli altri giocatori, il turno termina; altrimenti, il giocatore di turno è chiamato a riproporre una nuova soluzione.

Il giocatore di turno ha 90 secondi per proporre una soluzione valida. Allo scadere dei 90 secondi, il giocatore è costretto a passare il turno.

Gli utenti possono liberamente lasciare l'applicazione in qualsiasi stato di gioco essi si trovino. La gestione della disconnessione è gestita diversamente da stato a stato:

- se il giocatore di turno lascia l'applicazione, il turno termina
- se un giocatore non di turno lascia l'applicazione, è semplicemente rimosso dalla lista dei giocatori
- nel caso dovesse rimanere un solo giocatore, la partita termina

2.3 Interazioni

Il diagramma delle interazioni di figura 2.9 mostra cosa accade quando un utente richiede la creazione di una lobby ed un altro richiede di unirsi alla medesima lobby.

Per ogni istanza attiva del *client*, il *server* crea un `ClientListener`, un `ClientUpdater` e un `RequestHandler`, che si occupano di gestire la connessione con lo specifico *client*.

- Il `ClientListener` riceve le richieste e le inoltra al `ClientUpdater`
- Il `ClientUpdater` risolve le richieste attraverso il `RequestHandler` ed invia le risposte al *client*

La risoluzione di una richiesta proveniente da un certo *client* può generare una notifica che sarà inviata agli altri utenti che si trovano nella stessa partita. Questo avviene inserendo la notifica nei `ClientUpdater` dei rispettivi *client*. Come si può notare, quando il `RequestHandler` del *client c2* gestisce la richiesta di *join*, oltre a creare un messaggio di risposta, produce anche una notifica che inserisce nel `ClientUpdtater` del *client c1*. Così facendo, il *client c1* è a conoscenza del fatto che un nuovo giocatore si è unito alla lobby in cui si trova e può reagire di conseguenza aggiornando la lista dei giocatori.

Per facilità di lettura, il diagramma in figura riporta alcune semplificazioni lato *client*. Ogni *client* dovrebbe essere infatti costituito da 3 entità: un `ServerUpdater`, un `ServerListener` ed un `MessageHandler` (in corrispondenza alle entità lato server ClientUpdater, ClientListener e RequestHandler).

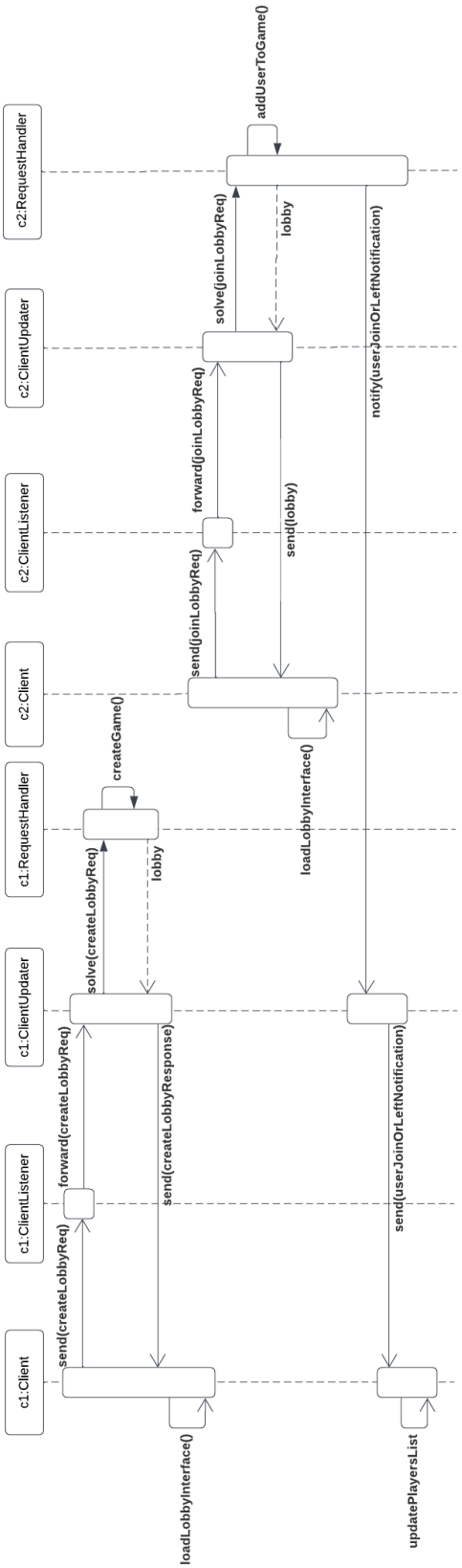


Figura 2.9: Rappresentazione semplificata di creazione e unione a una lobby

Capitolo 3

Dettagli implementativi

Il sistema è stato implementato sottoforma di progetto multi-modulo. I 4 moduli principali sono:

- `client`
- `server`
- `common`
- `presentation`

A questi moduli va infine sommato un quinto modulo (`testSystem`), utilizzato per testare l'interazione tra le varie componenti del sistema.

L'intero sistema è stato documentato attraverso l'uso di *javadoc*. La documentazione così prodotta è disponibile e navigabile al seguente link `https://nicolomalucelli.github.io/ScarabeoJavaDoc/`

3.1 Client

Il *client* è stato implementato seguendo il pattern *MVC*. Per la parte di *view* è stata realizzata un'interfaccia grafica attraverso l'utilizzo del framework *Swing*[1]. Il *model* è molto semplice in quanto si occupa unicamente di stabilire e gestire la connessione con il *server*, mentre il *controller* permette di aggiornare la *view* in base ai messaggi provenienti dal *server*.

3.2 Server

Il *server* è invece costituito da un'applicazione a linea di comando attraverso la quale è possibile visualizzare tutti i messaggi ricevuti ed inviati dal *server* verso i vari *client*.

[1] `https://en.wikipedia.org/wiki/Swing_(Java)`

All'interno del *server* è presente il vero e proprio *model* del sistema. Le informazioni mantenute localmente dai vari *client* sono infatti soltanto informazioni ausiliarie ed il *server* è l'unica fonte di verità all'interno del sistema. Questa strategia permette un maggiore livello di controllo sulle operazioni effettuate dagli utenti, eliminando la possibilità di operazioni illegali o non autorizzate.

3.3 Common

Il modulo *common* contiene una serie di elementi utili sia al *client* che al *server*, come ad esempio la porta di default sulla quale è in ascolto il *server*.

3.4 Presentation

Il modulo *presentation* contiene la definizione di tutti i possibili messaggi: richieste, risposte e notifiche ed i rispettivi deserializzatori.

Anche il modulo *presentation*, così come il modulo *common*, è quindi utilizzato sia dal *client* che dal *server*.

Capitolo 4

Validazione

La fase di verifica si è composta sia di **test manuali**, atti a verificare la correttezza della componente grafica, sia di **test automatici** realizzati attraverso il framework *Junit*[1].

I test automatici (figura 4.1) verificano sia la correttezza delle singole componenti, sia la corretta interazione tra le diverse componenti del sistema.

Nello specifico i test di interazioni verificano i seguenti aspetti:

- corretta connessione dei *client* al *server*
- corretta gestione della lobby
- corretta gestione dell'inizio della partita
- corretta gestione della partita
- corretta gestione della disconnessione da parte dei *client* durante le diverse fasi di gioco

Ulteriori aspetti, come il crash del *server* sono stati verificati manualmente.

[1] `https://junit.org/junit5/`

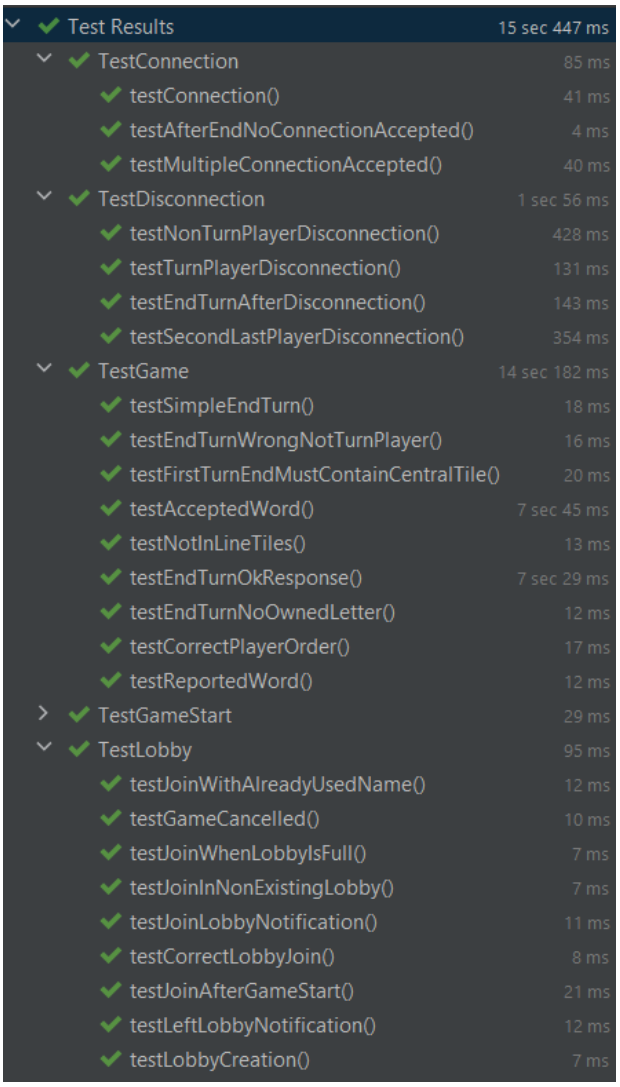


Figura 4.1: test Junit

Capitolo 5

Deployment

Essendo il progetto un applicativo *Java*, l'unico prerequisito per poterlo eseguire è che la macchina che lo esegue sia dotata di una *JVM*.

Collocandosi con il terminale nella cartella del progetto, per eseguire il *server* è sufficiente lanciare il comando:

```
./gradlew server:run
```

Il *server* così lanciato accetterà connessioni sulla porta 9821. Qalora questa porta sia già occupata o si voglia per un qualsiasi altro motivo modificare questo valore, è possibile farlo specificando il numero di porta come argomento del comando, come nell'esempio che segue:

```
./gradlew server:run --args='port'
```

Una volta avviata un'istanza del *server*, è possibile lanciare i vari *client*. Ancora una volta, il comando è:

```
./gradlew client:run
```

Di default, il *client* tenterà di connettersi alla porta 9821 del *localhost*; tuttavia, è possibile specificare sia l'indirizzo che la porta verso cui effettuare la connessione attraverso il comando:

```
./gradlew client:run --args='address port'
```

o anche solo l'indirizzo (in questo caso sarà utilizzata la porta di default 9821):

```
./gradlew client:run --args='address'
```

Capitolo 6

Esempi d'uso

Dopo aver avviato un'istanza del *server* è possibile eseguire i *client*. Nel caso in cui non sia possibile connettersi al *server*, è mostrato il messaggio in figura 6.10 ed alla sua chiusura l'applicazione è terminata.

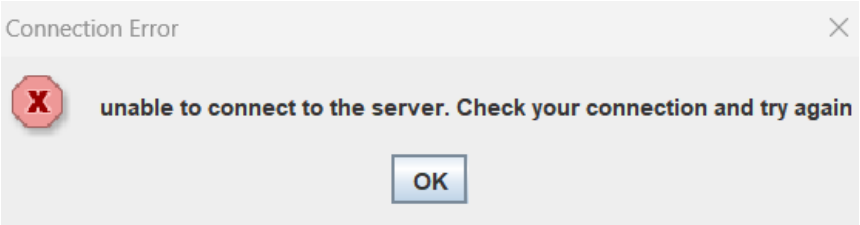


Figura 6.1: Notifica della mancata connessione al *server*

Se invece la connessione riesce, l'utente può decidere se creare una nuova lobby o se unirsi ad una già esistente (figura 6.2).

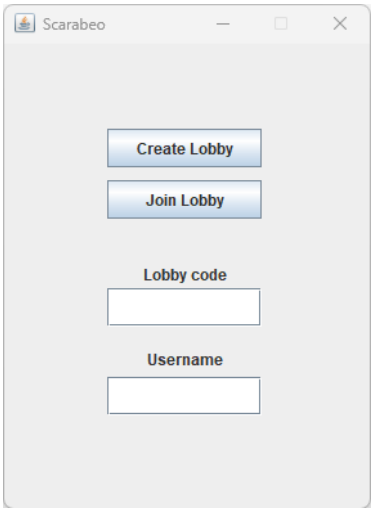
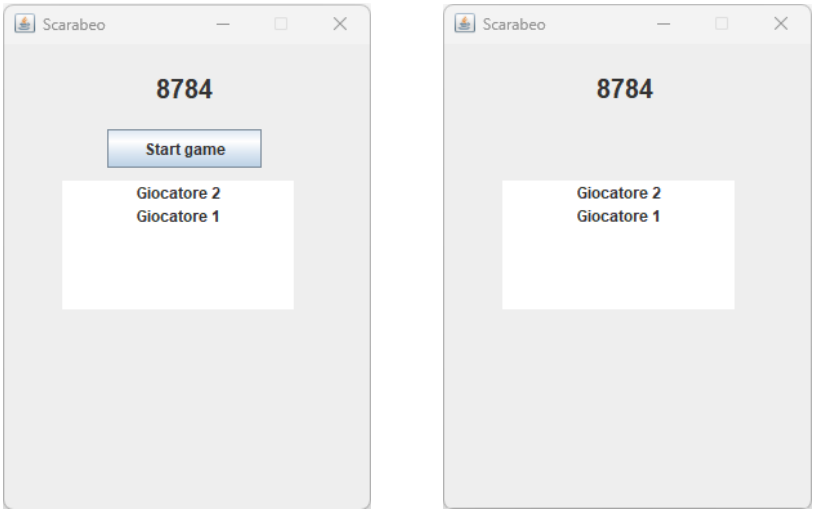


Figura 6.2: Schermata iniziale

Se il giocatore sceglie di creare la lobby è portato alla schermata in figura 6.3 e può comunicare il codice della lobby ad i propri amici. In questa fase, il creatore non può ancora avviare la partita in quanto il numero di giocatori non è sufficiente.

Figura 6.3: Schermata del creatore della lobby

Non appena un altro giocatore si unirà alla lobby, la lista dei giocatori sarà aggiornata ed il pulsante "*Start Game*" diventerà cliccabile (figura 6.4a). Il giocatore che si è unito alla lobby non può avviare la partita, infatti la sua schermata non contiene il pulsante "*Start Game*" (figura 6.4b). Se il creatore della lobby lascia l'applicazione prima di avere avviato la partita, i giocatori in lobby sono riportati alla schermata iniziale (figura 6.2).

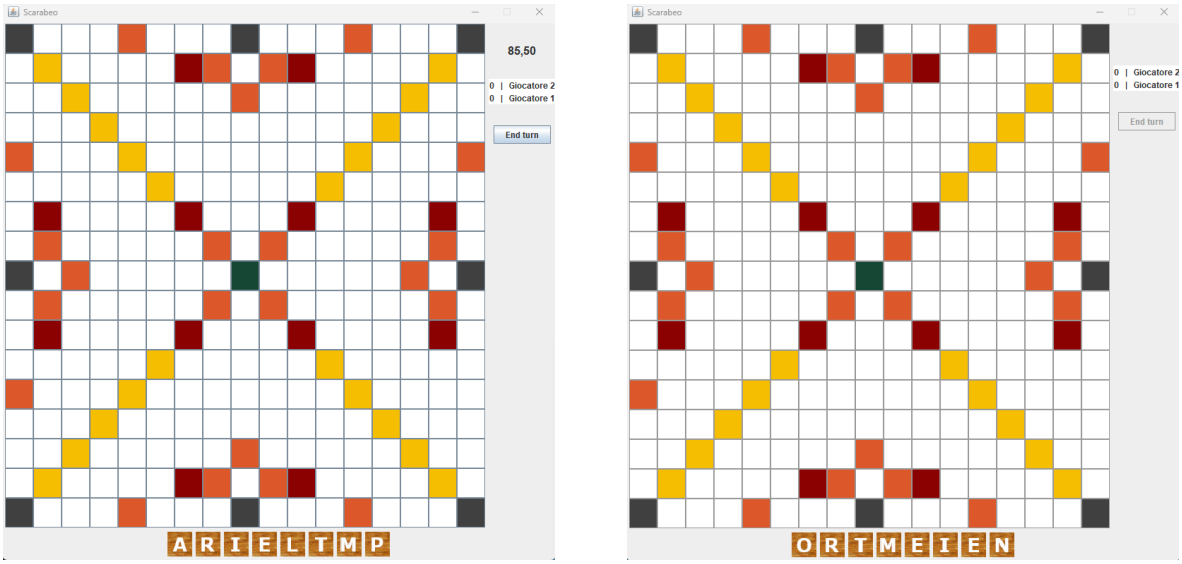


(a) (b)

Figura 6.4: Schermate di lobby a confronto: a sinistra il creatore, a destra l'utente che si è unito

Una volta iniziata la partita, il giocatore di turno ha tempo fino allo scadere del timer per proporre una soluzione valida (figura 6.5a). Il giocatore di turno può:

- posizionare una pedina in una posizione libera sulla scacchiera cliccando prima sulla pedina da posizionare dalla propria mano e poi sulla cella libera della scacchiera;
- rimuovere una delle pedine che ha piazzato nel turno corrente cliccandoci sopra mentre nessuna pedina della propria mano è selezionata;
- sostituire una delle pedine che ha piazzato nel turno corrente cliccandoci sopra mentre una pedina della propria mano è selezionata;
- proporre una soluzione, cliccando il pulsante "*End turn*" dopo aver posizionato le pedine;
- passare il turno, cliccando il pulsante "*End turn*" senza che nessuna nuova pedina sia presente sulla scacchiera



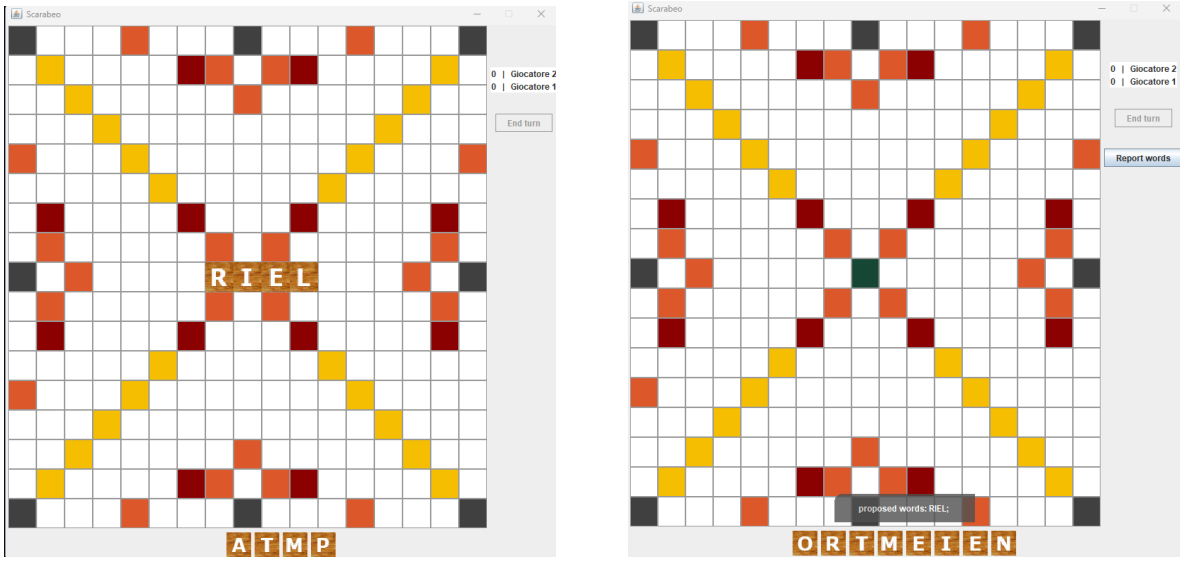
(a) Giocatore di turno

(b) Giocatore non di turno

Figura 6.5: Partita iniziata

Fin quando il giocatore di turno non termina il proprio turno, i giocatori non di turno possono soltanto attendere (figura 6.6a).

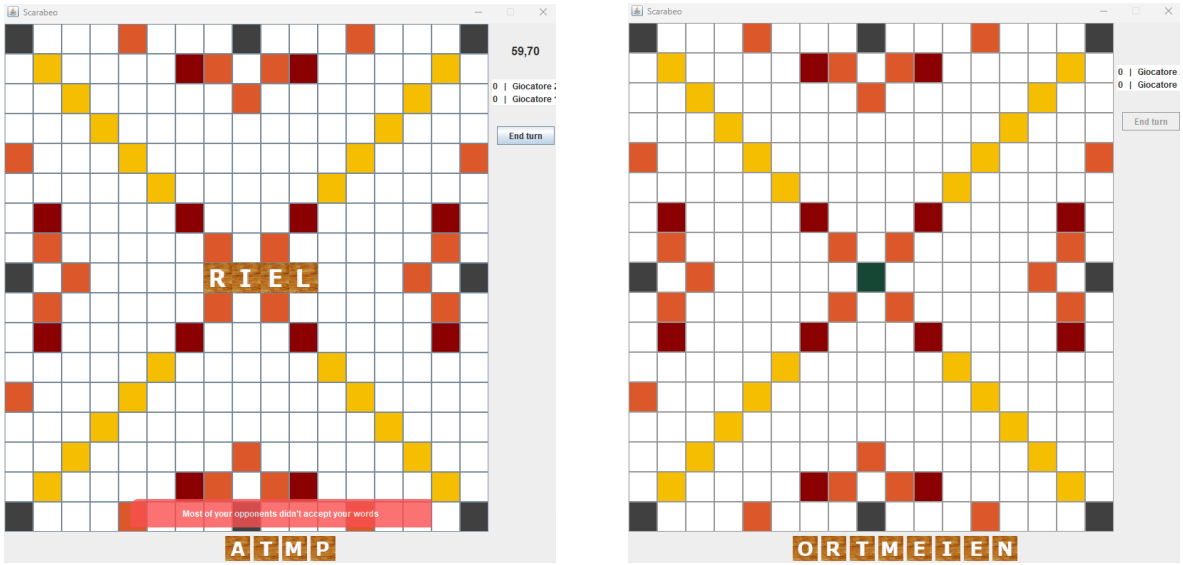
Non appena il giocatore di turno propone una soluzione (figura 6.6a), i giocatori non di turno possono decidere di segnalare la parola attraverso l'apposito pulsante "*Report words*" o in alternative non fare nulla. Le parole formate sono mostrate agli altri giocatori attraverso un *Toast* (figura 6.6b).



(a) Giocatore di turno (b) Giocatore non di turno

Figura 6.6: Proposta di una soluzione

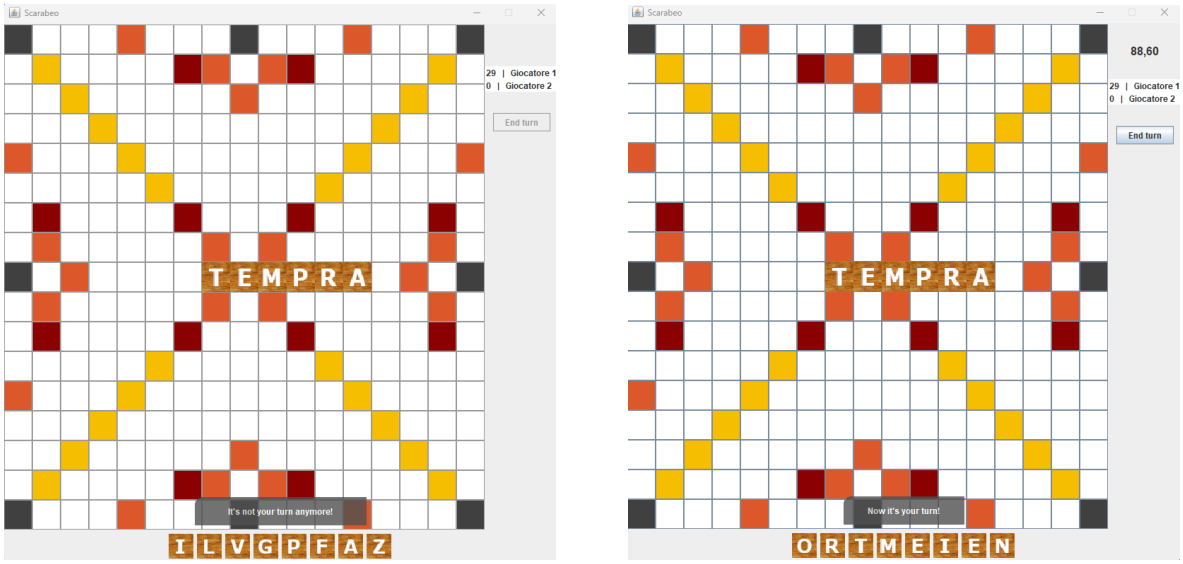
Se la soluzione non passa la validazione degli altri giocatori, al giocatore di turno è mostrata una notifica sottoforma di *Toast* (figura 6.7a), il tempo riprende a scorrere ed il giocatore può riproporre un'altra soluzione o anche decidere di passare il turno (rimuovendo prima tutte le pedine precedentemente posizionate sulla scacchiera).



(a) Giocatore di turno (b) Giocatore non di turno

Figura 6.7: Rifiuto di una soluzione

Se invece la soluzione è approvata dalla maggioranza degli avversari, il turno passa al giocatore successivo, la classifica è aggiornata, e agli altri giocatori è mostrata la schacchiera aggiornata (figure 6.8a e 6.8b). Inoltre, al giocatore di turno sono assegnate tante pedine quante ne ha utilizzate per comporre la soluzione precedente.

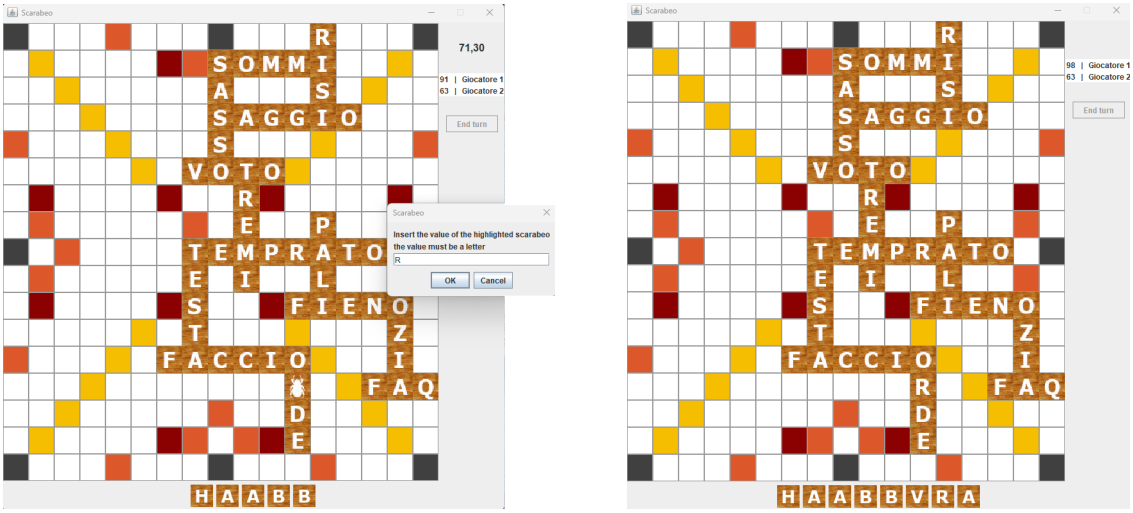


(a) Giocatore di turno

(b) Giocatore non di turno

Figura 6.8: Approvazione di una soluzione

Nel caso in cui la soluzione proposta contenga uno scarabeo, dopo il click sul pulsante di fine turno, all'utente è chiesto di assegnare il valore allo scarabeo (figura 6.9a). Una volta che una soluzione contenente lo scarabeo è stata approvata, lo scarabeo è sostituito sulla scacchiera con la pedina corrispondente. Non appena un giocatore di turno possiederà la pedina corrispondente ad uno scarabeo, questa sarà automaticamente sostituita senza che il giocatore debba fare nulla.



(a) prima dell'approvazione (b) dopo l'approvazione

Figura 6.9: Soluzione proposta contente uno scarabeo

Quando la partita termina, gli utenti ancora in gioco possono tornare al menù iniziale cliccando sul pulsante "*Exit*" che compare sulla parte destra dell'interfaccia.

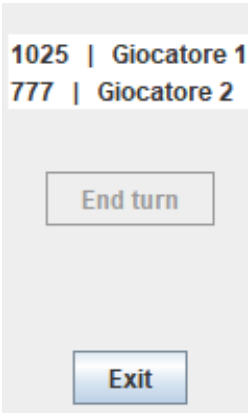


Figura 6.10: Pulsante di ritorno al menù iniziale

Capitolo 7

Conclusioni

Il risultato ottenuto è una versione *online*, *scalabile* e *consistente* del gioco da tavolo *Scarabeo*, che soddisfa tutti i requisiti prefissati nelle sezioni precedenti.

7.1 Sviluppi futuri

Possibili sviluppi futuri possono riguardare:

- aggiungere un sistema di registrazione e login che permetta ai giocatori di creare il proprio account e mantenere quindi lo storico di tutte le partite giocate. Questo permetterebbe inoltre l'aggiunta di altre funzionalità, come la possibilità di realizzare classifiche globali o aggiungere le amicizie.
- un miglioramento dell'aspetto grafico. Essendo la grafica una parte marginale di questo progetto è stata fatta senza darle troppo peso.

7.2 Che cosa ho imparato

Questo progetto mi ha permesso di sperimentare in prima persona lo sviluppo di un sistema distribuito sotto ogni suo aspetto: dalla fase di design e progettazione, fino alla fase di implementazione e testing, permettendomi di interfacciarmi con problemi e situazioni in cui finora non mi ero ancora imbattuto come ad esempio la gestione di disconnessioni improvvise da parte degli utenti.

Questo progetto mi ha inoltre permesso di approfondire lo sviluppo di applicazioni attraverso l'uso di *socket* e comprendere quindi la realtà dietro ad applicazioni di questo tipo.