Daniele Romanini *Federico Seri*

Politecnico di Milano
Sede di Cremona

-
Ingegneria Informatica

Prova finale di Ingegneria del Software

Anno Accademico 2014/2015

Invasion

Scopo del progetto era quello di realizzare un'applicazione multithread client – server in Java.

Prima della fase di implementazione, vi è stata una fase di progettazione dell'applicazione su carta. Abbiamo realizzato schemi UML che rappresentavano le classi che avrebbero fatto parte dell'applicazione.
Congiuntamente si è provato a separare la progettazione in macro-blocchi: Server, Client e Comunicazione, dovendo essere essi il più possibile indipendenti tra loro.

Il Server permette agli utenti di connettersi e giocare tra loro. Contenendo tutta la logica di gioco è stata la parte più complessa del progetto.

Il Server ha una classe principale (chiamata "Server"), di cui esiste una sola istanza per ogni sessione (è stato utilizzato il pattern Singleton). Essa è la sola ad avere un riferimento al Database, in cui sono salvate le informazioni riguardanti gli utenti registrati e le partite giocate, oltre alla mappa del gioco, che viene estratta all'inizio di ogni nuova partita. Si è scelto di salvare la mappa sul Database poiché, in questo modo, un eventuale modifica ai confini o ai territori può essere più semplice.
Ogni utente, una volta connesso, può avere diversi stati: LIMBO, INTAVOLO, INPARTITA. Il Limbo è lo stato nel quale il giocatore viene informato di qualsiasi modifica ai tavoli (ovvero le partite in attesa di essere avviate). In questo stato l'utente può vedere tutti i tavoli che sono ancora disponibili e il numero di utenti connessi ad essi. Queste informazioni vengono aggiornate in tempo reale, per permettere un'esperienza di gioco migliore. Una volta che un utente decide di entrare in un tavolo, potrà vedere i giocatori connessi ad esso (di cui potrà richiedere le relative informazioni) e decidere se restare in attesa dell'inizio della partita o tornare nel Limbo. Il primo giocatore ancora attivo che si è connesso a un tavolo in ordine cronologico sul tavolo può avviare la partita.
Nella classe "Server" vi è quindi una lista di utenti connessi, una lista di partite in corso e una lista di Tavoli Aperti. Ogni partita e ogni tavolo hanno un id identificativo univoco e si gestiscono in maniera indipendente, a parte per quanto riguarda l'aggiunta di un giocatore a un tavolo (richiesta ricevuta da "Server"), l'avvio e la chiusura di una partita, in cui è coinvolto il database, in quanto vengono salvate le informazioni di partecipazione ad una partita e di un'eventuale vittoria degli utenti.

La classe “GiocatoreConnesso” è la classe rappresentante il client lato Server (vi è quindi un’istanza per ogni utente collegato). Essa è l’unica che, per conto del client, comunica con la logica del Server e si occupa di inviare (tramite apposite interfacce di comunicazione) comunicazioni da Server a Client.

Anche il Client ha un’unica classe “Ambasciatore” che fa da tramite tra la Comunicazione, la logica del client e quindi, l’interfaccia grafica.
Esiste un’unica istanza per ogni sessione di gioco per singolo utente della classe Ambasciatore, ed esso ha un riferimento a una classe Limbo, Tavolo o Partita (esclusive). Queste sono astrazioni lato Client dello stato dell’utente.

Il modello è quindi assolutamente simmetrico: sia lato Client che lato Server vi è una sola classe rappresentante un giocatore connesso dalle quali passa tutto il flusso di comunicazione in ingresso e in uscita riguardante un singolo utente.
che raccoglie le richieste che arrivano dall’altro lato e si preoccupa di smaltirle. Queste classi (GiocatoreConnesso e Ambasciatore) permettono un disaccoppiamento ancora maggiore.

La comunicazione è assolutamente indipendente dal resto della logica. Un giocatore può scegliere in fase di login quale protocollo di connessione utilizzare (Socket o RMI). E’ stato utilizato il pattern Strategy: un’apposita classe (SetCommunicationStrategy) si occuperà di istanziare le classi di comunicazioni corrispondenti, e passare il loro riferimento a Ambasciatore (con visibilità di interfaccia, Client2Server). La gestione nel resto della sessione di gioco è del tutto trasparente all’utente e indipendente dal tipo di comunicazione scelto.
Questo approccio permette un’estendibilità del programma: potrebbe essere implementato un ulteriore protocollo di comunicazione senza cambiare assolutamente nulla nella logica del server o del client. Lo sviluppo della comunicazione RMI e Socket è stato infatti assolutamente indipendente.

Il progetto è basato sul modello MVC (Model-View-Controller), che fa uso di interfacce per disaccoppiare la grafica dal resto dell'implementazione.
La grafica (sia client che server) comunica con le altre classi tramite interfacce (distinguibili dal prefisso "Controller-"): esse sono implementate dalle classi di logica, che a loro volta hanno un riferimento alla GUI con visibilità di interfaccia ("View-"), implementate appunto dalle classi di grafica. Esse vengono utilizzate per aggiornare in tempo reale le informazioni visualizzate dall'utente (tavoli, giocatori, aggiornamenti riguardanti la partita in corso).
La grafica Client è più complessa di quella Server. Le interfacce ControllerPartita, ControllerTavolo e ControllerLimbo (implementate rispettivamente da Partita, Tavolo e Limbo) estendono l'interfaccia ControllerAccountEStatistiche, implementato dalla classe astratta AccountEStatistiche. L'utente, infatti, in qualsiasi stato si trovi, può sempre accedere a funzionalità come: cambio della password, logout, richiesta della classifica generale di gioco.

Quando un tavolo è pronto per essere avviato e ne viene fatta richiesta, viene avviata una partita, un thread indipendente che si occupa di gestire tutte le fasi di gioco.
Il metodo *run* inizialmente si occupa di comunicare a tutti i giocatori l'inizio della partita e successivamente si occupa di inizializzare la partita e gestire i turni.
Nell'inizializzazione della partita viene richiesto a tutti di scegliere il colore e posizionare le armate sui territori assegnati casualmente.

Al momento del turno di ogni giocatore e alla fine di ogni mossa di questo viene controllato che l'utente sia ancora connesso o non abbia abbandonato la partita. In caso contrario viene comunicato agli altri utenti dell'abbandono e il giocatore non riceverà alcun punto.
Per effettuare questo controllo verifichiamo che il riferimento ancora attivo nella classe Giocatore Connesso corrisponda a quella in corso.
Il giocatore che abbandona una partita viene riportato nel Limbo dove potrà effettuare una nuova partita mentre la vecchia continuerà senza interruzioni fino a che non rimarrà un solo giocatore, il vincitore.

Nella fase iniziale i giocatori hanno un tempo prestabilito per posizionare le proprie armate, così come all'inizio di ogni turno.
Il timer è visualizzato graficamente per indicazione all'utente ed è gestito dalla partita tramite una classe di supporto "PosizionamentoArmate" sulla quale la partita si mette in wait() per al massimo il tempo prestabilito in attesa che i comandi di posizionamento delle armate (ricevuti dal server sotto forma di stringhe) siano completi. In tal caso la partita viene risvegliata prima dello scadere del tempo tramite notifyAll().
Una strategia simile è stata usata anche per gestire la fase di scelta del colore.

Durante il proprio turno il giocatore può decidere di attaccare, spostare le armate o passare il turno.
Lo spostamento delle armate è consentito solo se i due territori appartengono dello stesso giocatore e raggiungibili tramite territori confinanti propri. Per verificare che i due territori siano raggiungibili si è usato un algoritmo di raggiungibilità per la visita di un DAG (Directed Acyclic Graph).

Al momento della ricezione di un comando da parte di un giocatore la partita dopo aver fatto gli opportuni controlli di attacco consentito, lancia i dadi di attaco e difesa, calcolando i risultati finali.
Essi vengono comunicati ugualmente a tutti i giocatori ma vengono visualizzati in modi differenti a seconda del ruolo del giocatore durante l'attaco per una migliore user experience.

Il socket comunica solamente tramite stringhe, che vengono scambiate tra le due classi del Server e del client, rispettivamente LettoreServer e LettoreClient che sono thread.
Esistono diversi tipi di messaggio: i messaggi di comando che vengono inviati e successivamente attendono una risposta dal client (simulano un chiamata di una funzione che attende una risposta). Chi invia il comando resta in attesa di un valore di ritorno in un wait, quando necessario.
Poi ci sono i comandi di risposta o eccezione che rispettivamente sbloccano il thread in attesa o lanciano un eccezione in caso di problemi.
I messaggi vengono interpretati tramite una classe di appoggio chiamata: “CreareMessaggio” che si occupa di creare gli appositi messaggi, comunicandoli in modo sicuro senza rischio di injection e di interpretarli estraendo dalla stringa passata.

L'RMI scambia oggetti condivisi senza limitazioni. Le classi di comunicazione implementano oltre alle interfacce Client2Server e Server2Client, implementate anche dal socket, anche le interfacce remote ServerComunicazioneInterfaccia e ClientComunicazioneInterfaccia.
Ad esempio il client per chiamare un metodo nel server implementato nel server, chiama un ulteriore metodo nella classe server comunicazione con la visibilità di interfaccia remota.
Lato server esiste un oggetto ServerComunicazione che implementa anche l'interfaccia Runnable per ogni client. Esso è in grado di accorgersi immediatamente della disconnessione di un client in quanto nel run chiama un metodo fittizio nella classe client comunicazione; se esso lancia una RemoteException, il client viene visto come disconnesso e viene gestito nella maniera opportuna.
Anche il client si accorge di un eventuale errore di connessione da parte server quando questo fa una richiesta al server e viene riportato alla schermata di login senzafar terminare il programma.
Stessa cosa viene fatta via socket poiché entrambi i LettoriSocket sono in continua lettura e catturano una eventuale eccezione di connesione.

Si allegano inoltre alcune classi di test, grafici UML di alto livello e tutta la documentazione Javadoc.