Peer-Review 2: Network Protocol

Bergagna Matteo, Brun Antony, Chen Jie, Chiroli Lorenzo – Gruppo 32

Valutazione del diagramma UML delle classi e dei sequence diagram del gruppo 41.

Lati positivi

- Implementabile: Non si riscontrano impedimenti che possano prevenire il funzionamento dell'implementazione Java di quest'idea. I componenti "Controller" e "Network" sono ben distinti e il loro ruolo è chiaro. Tutte le convenzioni tipiche sull'UML sono rispettate. L'insieme di messaggi (considerando sia le richieste Client che le risposte Server) è tale da trasportare tutte le informazioni necessarie per un corretto funzionamento della partita.
- Comprensibile: Il flusso dei messaggi dal client, qui visto "atomicamente", al server è comprensibile e ben delineato come anche il percorso inverso server-client per le risposte. Ottima la scelta di suddividere le classi in packages per facilitare la lettura dell'UML.
- Uso di Design Pattern: Ottimo l'utilizzo del pattern "Stategy" per l'implementazione e la gestione dei messaggi; corretto uso dei pattern "Listener" e "Observer" per la gestione delle code di messaggi.
- Separazione Client/Server: La separazione tra Client e Server è netta e ben delineata; non ci sono mai circostanze in cui è il Server a iniziare una conversazione senza prima una richiesta da parte del Client. Ottima l'idea di utilizzare JSON come formato di interscambio di messaggi.

Lati negativi

- Livello di Dettaglio: L'UML proposto è descritto molto "ad alto livello" trascurando molte scelte implementative che invece sarebbe utile trasmettere al lettore. Esempio di ciò è la classe "VirtualView" che è lasciata incompleta, senza né metodi né attributi, ma che sarebbe corretto spiegare all'utente dell'UML la sua funzione. Inoltre, nei diagrammi sequenziali tutti i metodi invocati non dettagliano né i parametri necessari né tantomeno gli eventuali ritorni.
- Thread: Nell'UML non sono marcate le classi "runnable" che devono venir eseguite in maniera asincrona: non è chiaro quanti thread vengano creati o quale sia la loro funzione.
- RMI/Socket: Nell'UML non viene definito come l'implementazione RMI e quella Socket differiscano tra loro; al lettore viene proposta una generica classe "ServerSocket" ma non viene fatta mai menzione di un possibile "duale" per RMI. Non è chiaro se il gruppo intenda implementare entrambe le modalità di comunicazione o solo quella mediante Socket TCP.
- Funzionalità avanzate: L'eventuale implementazione di funzionalità avanzate, specie quelle di resilienza alle disconnessioni e partite multiple, implicherebbe una sostanziale re-

progettazione dell'UML: in questo senso il class diagram proposto risulta poco flessibile. Non è chiaro quali (o se) funzionalità avanzate il gruppo intenda implementare.

- Entry Point: Non è chiaro quale sia l'entry point che faccia effettivamente partire il Server che serve come "radice" da cui poi costruire l'albero delle classi.
- Messaggi e istanceOf: Nel diagramma delle classi proposto tutti i messaggi vengono creati a partire da una singola classe astratta "Message"; l'utilizzo di più classi/interfacce (ad esempio una per i messaggi da Client a Server e una per le risposte Server-Client) avrebbe ridotto, ma non azzerato, il numero di occorrenze di instanceOf() da impiegare. Il problema che sorge è dovuto al fatto che al metodo execute nei messaggi non viene passato un parametro come, ad esempio, il GameController, che implica l'impossibilità per i messaggi di richiamare il metodo appropriato sul controller (non avendone l'istanza). Se per risolvere questo problema nel metodo execute il programma tentasse di ottenere l'istanza di gameController (cosa possibile se il GameController implementasse un pattern simil-singleton) si incorrerebbe a un altro problema: se invii un messaggio dal server al client che però è del tipo di quelli che il server riceve (es: CardPlacedMessage) ci sarebbero grossi problemi a runtime sul client.

Confronto tra architetture

- A livello generale l'idea nostra e quella del gruppo revisionato sono identiche: utilizzare RMI/Socket TCP per trasportare "messaggi" per scambiare tra Client e Server tutte le necessarie informazioni di gioco. Noi abbiamo scelto di implementare entrambi gli stack di rete.
- Sia noi che il gruppo revisionato implementiamo i messaggi sulla base del pattern "Strategy" in modo da compartimentare il più possibile gli effetti che questi devono avere sul Model o sulla View (sempre interfacciandosi mediante il controller) astraendo la "game logic" dalla parte relativa al "networking".
- Mentre il gruppo revisionato utilizza il formato JSON per la serializzazione dei messaggi noi abbiamo preferito demandare questo compito alla libreria di serializzazione build-in in Java. L'utilizzo di JSON offre si una maggiore interoperabilità e leggibilità dei dati, ma non lo abbiamo ritenuto necessario sviluppando sia Client che Server in linguaggio Java, evitando un inutile aumento di complessità dovuto al dover implementare manualmente le operazioni di deserializzazione dei messaggi per costruire istanze di messaggi.
- Il nostro network stack è stato pensato fin dal principio con l'idea di permettere l'utilizzo indistinto sia di RMI sia di Socket TCP sempre contemplando l'idea di rendere il tutto resiliente alle disconnessioni: una delle funzionalità avanzate che intendiamo implementare.
- Oltre alla feature sopracitata abbiamo intenzione di inserire nel gioco una chat e la possibilità di fare più partite in contemporanea utilizzando il medesimo server. Per fare ciò il Controller è stato concepito con la possibilità di accedere a un "ChatModel" per la gestione dei messaggi e disponiamo di un "GamesManager" per gestire i vari "GameController": uno per ogni partita.

- Noi, come da regolamento, iniziamo la partita immediatamente quando viene raggiunto il numero disegnato di giocatori; il gruppo revisionato ha invece deciso di demandare all'utente "host" il compito di far partire la partita. La differenza è minima ed è una scelta totalmente accettabile.