Peer-Review 2: Rete

Francesco Tarantino, Carlo Prestifilippo, Davide Vinci, Alessandro Scibilia

Gruppo 2

Valutazione dell'implementazione di rete del gruppo 42.

Lati positivi

Durante la revisione del diagramma delle classi UML abbiamo particolarmente apprezzato la scelta modellistica di suddividere quest'ultimo in due parti: una dedicata al modello ed al controller, ed un'altra esclusivamente per la rete. Tale rappresentazione è risultata molto efficace nel presentare nel suo insieme le connessioni più importanti della struttura del progetto da revisionare.

Inoltre, dal diagramma di rete fornito, l'implementazione del protocollo di rete appare conforme alle buone norme illustrateci relativamente al pattern MVC distribuito, garantendo la corretta distribuzione della computazione e permettendo la gestione efficiente degli errori.

Lati negativi

Il Sequence Diagram revisionato è riferito alla sola comunicazione tramite *Socket* e non a quella *RMI*, di cui però è facile intuire il flusso di chiamate.

Inoltre, dal diagramma, si evincerebbe che la classe ClientImpl comunichi, tramite ServerStub, direttamente con la classe ServerImpl senza che ClientSkeleton faccia da intermediario. Allo stesso modo, ServerImpl sembra inviare dati tramite ClientSkeleton direttamente a ClientImpl senza passare per il ServerStub. Se ciò non fosse un semplice errore di modellazione del diagramma, le comunicazioni dirette tra Stub e Client, e tra Skeleton e Server violerebbero il pattern MVC distribuito, non rendendo trasparente l'utilizzo del protocollo *Socket*: ClientSkeleton e ServerStub dovrebbero essere le uniche classi delegate alla comunicazione sul canale *Socket*.

È parso inopportuno l'invio diretto, tramite le classi di *ModelView*, di alcune parti del modello, rese per tale motivo serializzabili. Per rispettare il modello MVC distribuito ogni qual volta si debba inviare degli attributi al client, bisognerebbe creare una copia di questi ed inviarli in rete.

Sarebbe opportuno differenziare le chiamate alle funzioni per la creazione di una nuova partita o l'unione ad una partita già esistente utilizzando il *frame* Alt definito dallo standard UML.

È risultato poco chiaro il ruolo della classe PreGameController che, stando al diagramma, invierebbe notifiche direttamente alla UI senza passare per il protocollo di comunicazione.

Infine, le classi AppClientRMI e AppClientSocket potrebbero essere unificate per permettere la scelta a runtime del protocollo da utilizzare da parte dell'utente, come da specifica.

Confronto tra le architetture

La differenza principale tra le due architetture di rete è la gestione dello scambio di messaggi tra le classi ServerImpl e ClientImpl. Nello specifico, nella nostra architettura *RMI* il server non chiama un metodo *update* sul client (e viceversa) avente attributo l'evento di interesse, ma vengono chiamate diverse funzioni ognuna con uno specifico comportamento. Nel protocollo *Socket* viene invece inviato, prima dell'effettiva chiamata ai metodi corrispondenti, il valore di una enumerazione che specifica il metodo da chiamare e in seguito tutti gli attributi necessari.

Ulteriore differenza è l'implementazione da parte del gruppo revisionato del pattern *Observer/Observable* generico ad eventi. Nella nostra architettura è stato invece utilizzato il pattern *Listener* per la comunicazione tra le varie classi, abbandonando l'utilizzo e la specifica di eventi specifici, così da renderne meno verboso il controllo e avere maggiore flessibilità. Ciò si è rivelato necessario in quanto Java non permette l'ereditarietà multipla.