

Peer Review 2

Gruppo GC33:

Andrea Torti, Cristiano Valtolina, Diego Viganò, Fabio Vokrri

1 UML Networking

La classe `serverApp` ha il solo compito di creare un nuovo `Server`, il cui compito è quello di accettare nuovi client in connessione. Inoltre, ogni giocatore avvia la classe `ClientApp` a cui è delegato il compito di creare un nuovo `Client`.

La classe `Client` istanzia una nuova `View` (`Tui` o `Gui`) e richiede una connessione al server; il server accetta la richiesta del client e lo inserisce all'interno di una `Lobby` di giocatori (dai due ai quattro), se esistente, altrimenti ne istanzia una nuova. La `Lobby` ha il compito di istanziare il `Model` (nel nostro caso la classe `Game`) e il `Controller` (`GameController`) per quella partita.

In fase di accettazione della connessione del client, il server ha il compito di istanziare anche un nuovo `ClientHandler` per ogni `Client` che richiede connessione, inserendolo nella lista dei `clientHandlers` della `lobby` in cui viene inserito. Ogni `ClientHandler` ha il compito di comunicare con il client assegnatogli, inviare di volta in volta la `modelView` aggiornata e ricevere i comandi dal client inoltrandoli alla `lobby`.

I metodi in evidenza sono stati volutamente ridotti a quelli necessari alla comunicazione di rete.

2 Protocollo di rete

L'idea alla base della nostra architettura client-server, come già si intuisce dall'uml, è quella di fare uso della `Java Reflector API` per recuperare a destinazione il tipo di classe degli oggetti transitati via rete. Viene fatto pesante uso del listener pattern tramite due classi implementate ad-hoc, che a loro volta utilizzano internamente `Java Reflector` per chiamare la `update` desiderata tra quelle in overloading.

Per ora tutta la parte di rete è realizzata tramite `Socket`; abbiamo intenzione di fare uso di `RMI` ove possibile, ma non abbiamo ancora deciso dove, esattamente.