Peer-Review 2: UML di Rete

Beniamino Perri, Nicolò Ravasio, Paolo Vanotti, Davide Roccuzzo
GC47

23 giugno 2024

Valutazione del diagramma UML delle classi del gruppo GC37.

1 Lati positivi

Nell'analisi del diagramma UML e dei relativi sequence diagram scritti dal gruppo 37 sono stati riscontrati numerosi aspetti positivi. In particolare, facendo un breakdown degli aspetti positivi divisi per sezione:

- **Class diagram**
  - utilizzo di un'apposita classe *PlayerBean* per la rappresentazione del giocatore; anche se è un aspetto legato principalmente allo sviluppo della parte UI del gioco, riteniamo che sia una buona strategia e che possa semplificare anche l'implementazione della logica di rete;
  - presenza di un launcher apposito per gestire il setup preliminare del gioco, specie per quanto riguarda la scelta della tecnologia di rete da parte dell'utente;
  - implementazione corretta del design pattern Observer/Observable (con l'aggiunta del pattern mediator, in quanto l'Observable è una classe concreta), oltre che un intelligente utilizzo del polimorfismo per evitare la duplicazione del codice;
  - in generale, ordine presente nell'architettura del sistema, che rende di facile comprensione il funzionamento e le interazioni fra i moduli;

- **Sequence diagram**
  - Buon livello di dettaglio presente nei due sequence diagram. La comunicazione tra le parti è rappresentata a livello basso, mettendo bene in evidenza i metodi chiamati e le classi in gioco; ciononostante, il diagramma risulta comprensibile anche per avere una visione d'insieme più ad alto livello dei meccanismi descritti.

2 Lati negativi

Sono stati individuati i seguenti lati negativi:

- **Class diagram**
  - Gestione della chat di gioco: sebbene siano presenti delle classi apposite nel Model, non è chiaro come debba funzionare a livello di rete.
- **Sequence diagram**
  - Per quanto eloquenti, i sequence diagram presenti descrivono solamente la procedura di login alla partita, omettendo la rappresentazione delle altre dinamiche di gioco. Sarebbe opportuno rappresentare, anche a livello più alto, il funzionamento dei restanti meccanismi di gioco;
  - come detto precedentemente, neppure nel sequence diagram è menzionato il funzionamento della chat;
  - un utilizzo così massiccio delle classi Message può risultare scomodo nella comunicazione tra il Model e il Server RMI: magari, si potrebbero aggiungere metodi specifici all'Observable/Observer per gestire le varie fasi della comunicazione, usando le classi Message solo come "traduzione" per il Socket.

3 Confronto tra le architetture

L'architettura implementata dal nostro gruppo ha alcune differenze rispetto a quella del gruppo 37. In entrambi i casi si è optato per implementare entrambe le tecnologie di rete (socket/RMI) e di sviluppare un launcher per la

gestione delle fasi iniziali di setup.

Il gruppo 37, a differenza del nostro, ha optato per un'architettura thin client, come è intuibile dalla descrizione dei messaggi socket: le comunicazioni relative alle carte si occupano infatti di trasmettere tutto il contenuto della carta, rendendo pesante la comunicazione client-server ma snellendo l'applicativo del client. Anche la presenza di messaggi per gestire azioni che in un applicativo thick client sarebbero svolgibili localmente (e.g. FlipRequestMessage) confermano tale scelta.

Il gruppo in analisi utilizza il design pattern Template per l'implementazione del server: quest'ultimo è descritto da una superclasse, estesa poi da due sottoclassi che si occupano di gesitre le connessioni RMI e socket. Riteniamo che tale scelta di sviluppo, non presente nel nostro codice, sia interessante, in quanto permette di portare la duplicazione del codice ai minimi termini. Tale scelta sarà pertanto adottata anche nel nostro codice.

Un'altra differenza tra le scelte fatte dai due gruppi sta nella rappresentazione dei messaggi: se il nostro gruppo aveva optato per far comunicare client/server socket mediante stringhe JSON, il gruppo 37 ha invece scelto di scrivere classi apposite Message (con anche una *enumeration* per tenere traccia dei possibili messaggi scambiabili tra client e server) per generare le stringhe, oppure per essere inviate con RMI.

Infine, la logica utilizzata per il pattern Observer/Observable rimane piuttosto astratta. Sono infatti presenti metodi "generici" quali Notify/NotifyAll, mentre nel nostro caso le interfacce Observer/Observable contengono metodi direttamente riconducibili al flusso di gioco (e.g. showWinner, updatePoints, updateDecks).