Peer-Review 2: Connection protocol UML

Davide Tonsi, Mattia Zambetti, Leonardo Vaia

Gruppo GC35

Valutazione del diagramma UML delle classi del gruppo GC45.

Lati positivi

L'UML proposto dal gruppo GC45 ha diversi lati positivi:

- Ottimo utilizzo del pattern strategy nello sviluppo delle classi di comunicazione tra client e server creando una interfaccia comune e sfruttando l'overriding (riferimento alle classi ToClientMessage e ToServerMessage); con questa struttura è possibile aggiungere, modificare o eliminare i tipi di messaggi scambiati tra client e server in futuro (riadattabilità del codice e robustezza), senza dover aggiornare nulla nel resto del codice
- Ottima gestione delle carte personaggio, sfruttando lo stesso pattern discusso al punto precedente; grazie a questa scelta di progettazione diventa molto semplice aggiungere nuovi tipi di carte personaggio in seguito a possibili sviluppi futuri
- Classe server molto chiara e completa, ottima idea la gestione dei nicknames all'interno di esso, verificando se esistono giocatori con un nickname identico
- Ottima anche l'idea di creare una classe che si occupa di gestire la comunicazione tra client e server (riferimento a PlayerHandler): esso favorisce il riutilizzo, infatti, la classe è utilizzabile sia per la versione CLI che per la versione GUI del programma, senza dover riscrivere la stessa cosa due volte. Inoltre permette la separazione tra pattern MVC e parte riguardante i socket e la connessione.

Lati negativi

Cercando di essere il più critici possibile abbiamo incontrato alcune lacune nell'UML revisionato:

- L'utilizzo della classe GameDelta è pericoloso: così facendo si crea una stretta dipendenza tra quest'ultima classe e la classe Game. Di conseguenza potrebbero verificarsi diversi bug dati dal fatto che il modello potrebbe non essere aggiornato correttamente in ogni punto. Per evitarlo si può utilizzare, ad esempio, una sorta di pattern Adapter: in questo contesto GameDelta potrebbe essere sostituita da un'interfaccia GameInterface, la quale sarebbe implementata dalla classe Game e possiederebbe solo i metodi puri (che quindi non modificano lo stato del modello), potendo così inviare al client il modello stesso, e rendendo visibili solo i metodi presenti nell'interfaccia.
- Non è mai una buona cosa fornire ad un metodo un oggetto mutabile, soprattutto nel caso in cui il parametro sia il Game (si fa riferimento al passaggio di un oggetto di tipo ExpertGame nel metodo play della classe CharacterCard), si consiglia l'utilizzo di un'interfaccia intermedia, simile ad un Adapter (approfondito nel confronto tra architetture)
- Il controller possiede alcuni metodi che potrebbero essere incapsulati all'interno della classe ToServerMessage: ricevendo un messaggio da un client, a seconda del suo tipo il controller deve chiamare sempre gli stessi metodi. Si può quindi creare un metodo con stessa signature all'interno di ogni messaggio, il quale riceve come parametro il modello e ne modifica lo stato; così facendo la classe diventa molto più snella .

Confronto tra le architetture

Innanzitutto, anche nel nostro progetto abbiamo optato per l'utilizzo del pattern strategy nella gestione dei messaggi tra client e server(riferimento al punto 1 dei lati positivi), infatti, ciò rende il controller molto snello di struttura e il programma molto più robusto rispetto alle modifiche future; a differenza vostra, abbiamo deciso di gestire il tutto con un unico tipo di classe, chiamata Choice, la quale ha una duplice funzione: lato server viene inserita nel modello come indicazione della fase di gioco; lato client invece, viene riempita con i dati forniti dall'utente corrente. Così facendo il client conosce già quale tipo di dati deve inserire l'utente e può svolgere il compito di proxy, si evita quindi di fare richieste ulteriori al server.
Un possibile consiglio è sviluppare allo stesso modo e con lo stesso pattern (strategy) le eccezioni; nel nostro software esse possiedono un metodo comune, il quale si occupa di domandare di nuovo all'utente la stessa Choice se errata, stampare lo stack a video e inserire un messaggio di errore nel modello; oppure, nel caso in cui l'Exception determini la fine del gioco, essa gestisce le varie tipologie di fine delle partite sfruttando il binding dinamico e l'ereditarietà (chiaramente questa cosa non è verificabile attraverso l'UML parlando di eccezioni, ma per essere di aiuto il più possibile abbiamo deciso di aggiungere anche questo nella revisione).

Per poter mostrare lo stato del gioco al di fuori del modello, noi abbiamo utilizzato un'apposita interfaccia, come detto nel punto 1 dei Lati negativi; essa favorisce la riscrittura del codice senza dover tener aggiornate ulteriori classi, ma solo la classe Game. Nel nostro progetto si è utilizzato lo stesso approccio (riferimento a pattern Adapter) anche nella realizzazione delle carte in modalità esperti, così facendo si può dare ad un metodo (play nel vostro caso) il modello stesso in versione di "mutabilità controllata" (abbiamo notato che, per questo, siete stati costretti a rendere protetti o privati la maggior parte dei metodi, tuttavia non ci è chiaro fino in fondo l'utilizzo di essi al di fuori della classe stessa).

Per quanto riguarda le carte personaggio il diagramma UML potrebbe essere ulteriormente raffinato, aggiungendo una classe padre comune a tutte le carte che possiedono degli studenti (comodo per possibili sviluppi futuri del gioco): nel nostro caso ad esempio, all'interno del mercante, della principessa e del giullare, è presente solo il costruttore, il quale si occupa di inizializzare gli attributi (di cui la maggior parte sono identici all'interno delle tre classi). Inoltre, nella nostra architettura abbiamo diviso in due la fase in cui si gioca una carta personaggio: la prima consiste nella scelta di una delle tre carte presenti nel game, la seconda invece è la scelta degli studenti da spostare. Per questo esiste un metodo play (comune a tutte, contenuto all'interno del padre) che semplicemente notifica all'utente, aggiornando al suo interno l'apposita Choice contenuta nel Game, e permettendo l'azione relativa alla carta stessa (nel vostro caso sarebbe una classe figlia di ToClientMessage).

Per quanto riguarda il controller anche noi possedevamo una classe simile, contenente in particolare il metodo startMatch. Tuttavia, nel nostro caso, abbiamo deciso di rimuoverlo, creando un ciclo implicito nel programma: il Match inizia notificando ai client la prima versione del modello, contenente, oltre ai dati sulla partita, la prima scelta riguardante il colore e lo stregone che desidera ogni giocatore. Il Controller, invece, si occupa di chiamare un unico metodo contenuto nelle Choice (noi lo abbiamo chiamato manageUpdate), invocandolo all'interno del metodo update, il quale modifica il modello a seconda della scelta ricevuta(overriding). Infine, è il Game a notificare il suo nuovo stato, contenente anche la nuova fase di gioco oppure la vecchia (in caso si sia verificato un errore di inserimento).