Network and Controller

Peer-Review 2: UML

Simone Francesco, Rossi Raffaele, Scalmani Paolo, Zuccherino Giuseppe

Gruppo GC51

Valutazione del diagramma UML Network e Controller del gruppo GC41

Lati positivi

Controller

- La gestione delle partite multiple é senza dubbio una delle funzionalitá aggiuntive piú complesse da gestire, ció nonostante, l'idea di costruire la classe MainController come una sorta di "lobby", risulta una soluzione particolarmente efficace nella definizione di un binding, tramite HashMap, dei GameController istanziati e nella definizione di nuove istanze di quest'ultimo.
- L'utilizzo di una Queue per la ricezione, controllo ed esecuzione delle request lato server, con la conseguente gestione asincrona dei server (socket ed RMI), é un approccio pratico ed efficiente che permette di rendere "opaca" la visione del controller, il quale, infatti, non avrá necessitá di distinguere request provenienti da tecnologie di connessione diverse.
- La definizione di un attributo "IDGenerator", associato univocamente alle diverse istanze dei controller, permette in modo semplice di individuare senza ambiguitá la partita a cui si desidera unirsi.

RMI Server

- La differenziazione degli RMI server per la gestione del flusso di gioco pre e post inizio partita, rende certamente molto piú leggibile e modulare il codice. L'implementazione snella di entrambe le classi consente modifiche puntuali al codice senza un'integrazione piú ampia. Ció nonostante porta ad alcuni svantaggi che saranno elencati nei lati negativi a seguire.

Socket Server

- Così come per l'implementazione RMI, avere varie classi distinte che compiono una funzione ben specifica aiuta molto in termini di modularità del codice, leggibilità e implementazione di eventuali modifiche, che possono essere svolte sulla singola classe piuttosto che su parti più estese di codice. Intelligente l'implementazione di un client handler (pressochè un requisito, data la necessità di supportare più client) che favorisce una maggiore efficienza (in particolar modo per l'invocazione di metodi dalla rapida esecuzione, quali possono essere appunto quelli di un flusso di gioco - calcolo del punteggio, aggiunta/rimozione da una struttura dati...), dal momento che ogni handler è in grado di essere istanziato in thread distinti per l'esecuzione dello stesso, oltre che gestire la comunicazione diretta con il socket e gli IO stream.

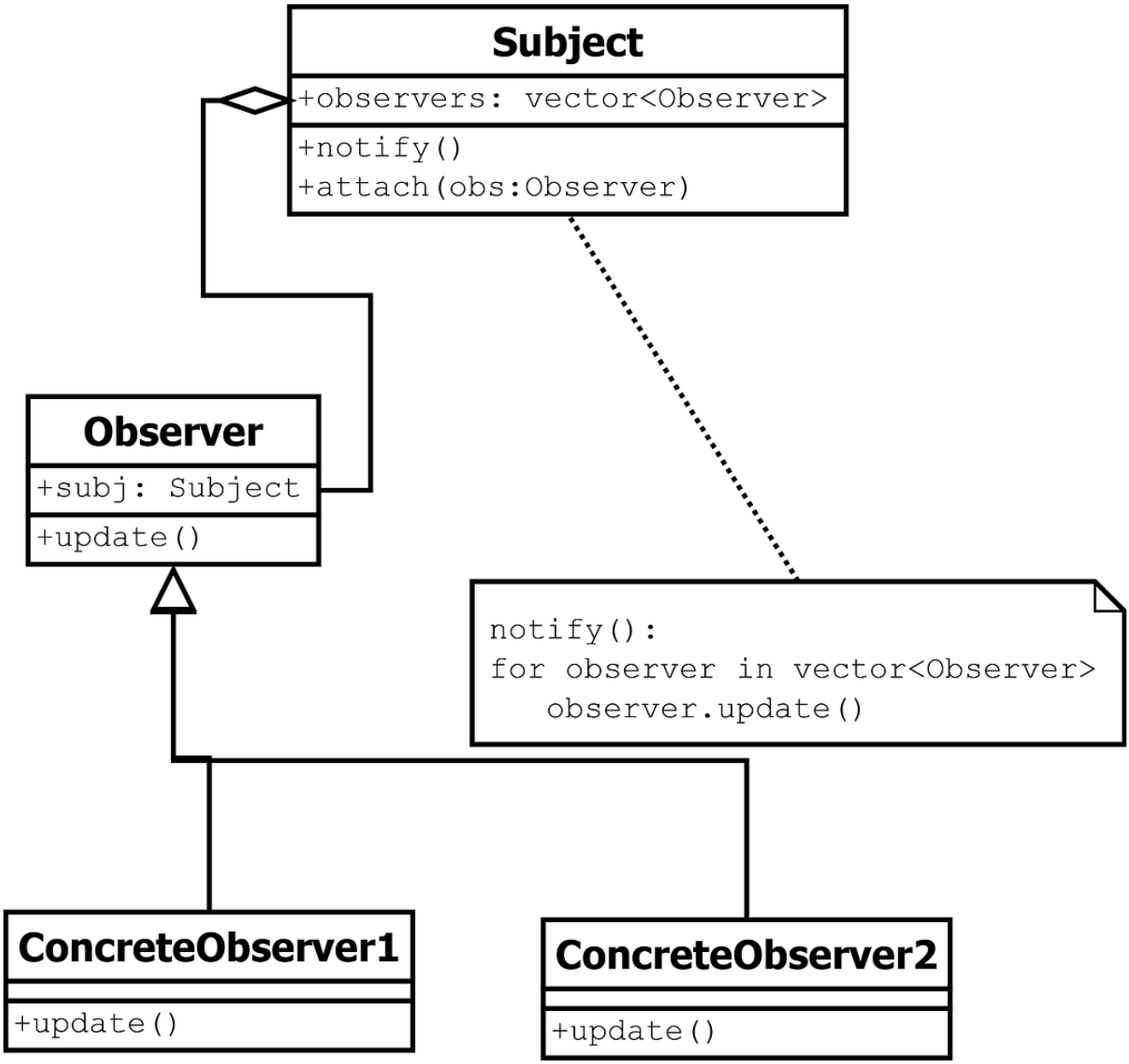
Lati negativi

Controller

- La classe GameController sembra avere il compito di notificare le View in caso di modifiche del Model, ma questo approccio sembrerebbe rompere il pattern MVC, in quanto, da documentazione Java:

> **Model View Controller (MVC) Pattern:**
>
> - the **Model:** represents data and the rules that govern access to and updates of this data (it can also have business logic).
> - the **View:** displays the data and also takes input from user.
> - the **Controller:** acts as an interface between the Model and the View parts and it intercepts all the incoming requests.

Infatti il compito di notificare le View delle avvenute modifiche del Model, dovrebbe essere demandato al Model stesso, con degli opportuni Observer (con il pattern ad essi associato) ad esso collegati, i quali, notificati indistinguibilmente dal Model della modifica, prendono da esso gli attributi modificati e li notificano alle View. Il Controller, al contrario, dovrebbe soltanto occuparsi di gestire le richieste, eseguirle, gestire le varie ed eventuali eccezioni, passarle al Model, il quale seguirà nuovamente il pattern notificando gli Observer.



RMI Server

- La differenziazione di un'architettura RMI, giá notevolmente povera di verbosità implementative, porta a due svantaggi principali:
  - L'avvio manuale di due server distinti ed il conseguente doppio bind di registro per ogni client

- La definizione di una doppia interfaccia che gestisce metodi differenti potrebbe portare i due server, dovendo per ogni esecuzione creare Thread concorrenti e non sincronizzabili tra i due server, ad inviare richieste difficilmente ordinabili, rendendo complesso capire quale metodo verrá chiamato per primo e spedito al controller per l'esecuzione. L'utilizzo di due code distinte per i diversi tipi di messaggi risulta essere poco efficiente, in quanto, per la definizione di quali metodi chiamare e quando, potrebbe bastare l'enumerazione dello stato del gioco (come nel Model).

Socket Server

- La modularità potrebbe rivelarsi un'arma a doppio taglio: l'utilizzo di più classi distinte che agiscono da proxy potrebbe risultare macchinoso e conflittuale, dal momento che tutte e tre le classi implementano le rispettive interfacce e un proprio output stream. Riteniamo che distinguere così nettamente questi flussi di output, oltre che separare Game e Main server non sia necessario all'interno della connessione TCP, poichè facilmente risolvibile tramite l'utilizzo di un Network Manager (vedi confronto a seguire). Questa distinzione potrebbe creare ambiguità tra le informazioni scambiate, oltre che aumentare la complessità della rete.

Confronto tra le architetture

Controller

Per rispettare a pieno il pattern MVC, il nostro gruppo ha optato per delle classi observer collegate al Model che si occupano di ottenere i cambiamenti, di notificarli alle view attraverso uno strato di rete diverso da quello per i messaggi di modifica del Model. Inoltre il controller, pur occupandosi della gestione delle eccezioni, utilizza una classe "TaskFailed" (nome non definitivo) le cui istanziazioni vengono conservate, grazie ad una struttura dati opportuna, nel Model, il quale, in caso di aggiunta di un errore, notifica gli observer dell'avvenuto cambiamento, conseguentemente un observer specifico si adopererá per notificare la/le View interessata/e all'errore.

RMI Server

Il nostro gruppo ha scelto di utilizzare una classe RMIServer che si occupi unicamente di ricevere e spedire i messaggi, senza distinzione di tipo, ricevuti dalla View (grazie all'input dell'utente) ad un Network Manager (conservando una Queue di messaggi), al quale sará demandato il compito di invocare il metodo giusto del Controller, il quale modifica il Model che notifica le View dei cambiamenti avvenuti.
Inoltre una classe Main si occupa di avviare in una volta sola tutti i server necessari al corretto funzionamento del gioco.

Socket Server

Il funzionamento del Socket Server è parallelo a quello di RMI: sostanzialmente il rapporto tra client e server consiste solamente nello scambio di messaggi, contenenti informazioni sui metodi che il cliente desidera invocare, che saranno poi ricevuti dal Network Manager, univoco per entrambe le connessioni. In questo modo la complessità della rete si concentrerà all'interno del Manager e nella modalità di scambio di informazioni (es. la serializzazione/deserializzazione di un'informazione per l'invio e l'analisi di essa stessa). Tutto ciò va a beneficio del pattern MVC, dal momento in cui vi è un modo univoco di comunicare con il controller (riducendo quindi le ambiguità) e di conseguenza modificare il model.