Peer Review 2: Network

Enrico Vento, Federico Valentino, Giacomo Verdicchio

GC61

Valutazione UML (rete e controller) del gruppo GC06

3/4/2022

1. Lati Positivi

1. *Apprezzato l'utilizzo degli Events*

   Secondo noi degna di nota la scelta di utilizzare questo tipo di pattern in sostituzione al più "problematico" *Observer-Observable* pattern. Dalla documentazione si vede che la scelta verrà ben supportata ed implementata.

2. *Utilizzo di Light Model*

   Utilizzare un Model "leggero" consente di tenere sempre aggiornati i client senza mandare numerose informazioni che, al fine della visualizzazione e dell'utente finale, risulterebbero inutili.

3. *Gestione Character effects*

   La gestione degli effetti dei personaggi nel controller, sfruttando classi astratte parenti per funzioni standard, poi specializzate in sottoclassi selezionate tramite *strategy*, è di sicuro originale.
   Risulta inoltre espandibile in caso si debbano inserire altre carte che comportino la modifica di quelle stesse funzionalità standard.

2. Lati Negativi

1. *Controller Troppo Centralizzato*

   Un po' come già notato nel model, la classe principale del *serverController* è ricolma di metodi che coprono le funzioni più svariate. Sarebbe meglio optare per un maggiore distribuzione della logica nelle classi pertinenti (che tra l'altro esistono) per favorire leggibilità e facilità di *debugging*.

   A questo si aggiunge anche un doveroso lavoro di "limatura" di alcuni gruppi di metodi, come quelli atti a gestire le liste che tengono conto dei turni dei vari giocatori: tre metodi diversi che si occupano di determinare ordini di turno sembrano troppi.

2. *Riferimenti a Tipi di Dato Inesistenti*

   In *serverController* viene indicato come attributo un dato di tipo *LobbyState*, che però non sembra essere definito da nessuna parte; a meno di un errore dalla nostra parte, siamo sicuri che si tratti di un placeholder o di una dimenticanza, ci limitiamo semplicemente a segnalarlo.

3. *Oggetti del Model nel Controller*

   Una criticità da segnalare, secondo noi, è il salvataggio di alcuni oggetti del model direttamente nel controller. È accettabile la lista degli *AvailableWizards*, però, forse, piuttosto che salvare come attributi direttamente liste di *Player*, si poteva optare per l'utilizzo di nickname o ID per poi operare su oggetti del model solo all'interno di specifiche funzioni, salvandoli dunque solo localmente.
   Stessa cosa per il *CurrentPlayer*.
   Questo aiuterebbe a meglio separare gli ambiti di Model e Controller.

3. Conclusioni e Confronto Architetture

A livello di rete e networking ci sembra che ci siano fondamenta solide per un lavoro di qualità; di pregio, come già detto, le decisioni di utilizzare un *LightModel* e di evitare l'*Observer* pattern.
Contrariamente, la nostra decisione è stata quella di implementare comunque un *Observer-Observable* pattern, ridefinendolo in parte per non utilizzare quello *deprecated*.
Prenderemo sicuramente in considerazione l'idea di implementare un *LightModel*.

Dall'esempio del funzionamento dell'*EventManager* potrebbe sorgere il sospetto che i messaggi scambiati siano composti unicamente da stringhe: se così fosse si genererebbe un traffico di rete piuttosto importante, ma confidiamo che fosse solo una scelta "semplice" per fare un esempio comprensibile del funzionamento del protocollo.

Le maggiori perplessità derivano dalla realizzazione del controller e, retroattivamente, del model. Riteniamo che un approccio così "centralizzato", seppure semplice da gestire in un progetto di dimensioni piccole, non rappresenti una buona abitudine nel caso poi ci si trovi a lavorare su progetti più grandi; avere la logica distribuita in classi di servizio, o direttamente nelle classi a cui essa si riferisce, aiuta nel debugging e rimane un approccio più futuribile, rendendo più semplice affrontare un'eventuale richiesta riguardante la scalabilità.