Contents

1 Documento di descrizione UML - Model e ServerController

Gruppo IS-AM41: D'Alessio Edoardo, De Ciechi Samuele, Deidier Simone, Ermacora Iacopo.

1.1 Introduzione

Ciao!
Siamo il gruppo AM41 ed innanzitutto vi ringraziamo per il tempo speso sulla revisione del nostro progetto.
Il diagramma UML non dovrebbe richiedere troppe spiegazioni in quanto abbiamo cercato il più possibile di assegnare dei nomi auto esplicativi alle classi, metodi e attributi. Tuttavia ci sono alcuni aspetti che potrebbero creare confusione, ad esempio l'utilizzo di diversi *design pattern*, dunque alleghiamo questa breve descrizione che cerca di spiegare più specificamente anche alcuni dei ragionamenti che ci hanno portati ad ipotizzare questa implementazione.

1.2 Model

A livello generale, abbiamo cercato di **dividere la complessità il più possibile fra le classi**. In questo modo è l'unione di tutte le classi a permettere il corretto funzionamento del gioco e non un'unica classe che svolge tutto. Questo si vede, per esempio, nella scelta della modalità di implementazione delle carte obiettivo (*più in dettaglio in seguito*) dove il metodo che calcola i punti generati dalla carte, all'interno del programma, è appunto delegato alla classe Card stessa e non alla classe Game o Player.

1.2.1 Classe *Game* e *GameState*

La classe **Game utilizza lo *"State Pattern"***, per aiutarci nella gestione delle diverse fasi del gioco: * ServerInitState è lo stato che si ha quando il gioco è stanziato all'inizio, quando nessun player è ancora connesso. Da qui si potrà poi

passare o allo stato WaitingForSavedGame oppure WaitingForPlayers. La scelta relativa a quale dei due stati passare viene fatta dipendentemente da che il primo utente che si connette alla partita possegga un nickname *"nuovo"* per il server (*WaitingForPlayers*) oppure che questo nickname sia presente in una partita salvata da una sessione precedente (*dovuto alla nostra scelta di implementare la funzionalità avanzata di persistenza - WaitingForSavedGame*). * In seguito alla connessione del primo player si passa allo state WaitingForPlayers, aspettando il numero di giocatori indicato dal primo giocatore connesso. Già in questa fase avverrà la maggior parte del setup degli oggetti di gioco. * Finalmente si arriverà in RunningGameState dove comunque le funzioni verranno reimplementate (*per esempio addPlayer() in modo tale da non consentire nuove connessioni da giocatori che sarebbero in sovrannumero*). * WaitingForSavedGame è lo stato che si ottiene nel caso in cui si connetta come primo player un player appartenente ad una partita salvata in precedenza. Questo stato è necessario che venga distinto da WaitingForPlayers in quanto i requisiti per connettersi alla partita sono differenti. Per quanto riguarda la parte di caricamento del salvataggio non abbiamo ancora pensato alle classi che ci serviranno, pensiamo sarà una delle future aggiunte quando avremo la maggior parte del codice funzionante.

1.2.2 Carte obiettivo personali e comuni

Per quanto riguarda le carte obiettivo personale e obiettivo comune abbiamo deciso di **utilizzare il pattern *"Abstract Factory"*** per implementarle. In questo modo le carte avranno una funzione unica per ognuna di esse, con parametro in ingresso la Shelf del giocatore, che verifica se l'obiettivo a loro corrispondente viene o meno soddisfatto.
Se viene completato un obiettivo comune, verrà assegnato un token al giocatore corrispondente, stessa cosa avviene con il giocatore che termina per primo la partita. Questi token servono per il conteggio dei punti (*quello di fine partita anche per il check sull'ultimo giro di turni prima di finire la partita*).

1.2.3 *Singleton*

Un altro aspetto che può sorprendere a prima vista sono alcuni metodi costruttori indicati come privati: questo è dovuto alla scelta di usare il design pattern *"Singleton"*, che permette di essere sicuri che l'**oggetto in questione possa essere istanziato una sola volta**. Abbiamo scelto di seguire questa strada nella implementaziome di Game (*questo anche in quanto abbiamo scelto di non implementare la funzionalità avanzata che permette partite multiple sullo stesso server*), EndGameToken e la ItemsBag.

1.2.4 Shelf e Board

Per la Shelf e la Board usiamo delle matrici di Item e ciò che risulta interessante secondo noi è l'implementazione di Board.
La Board del gioco reale non è una semplice matrice 9x9 e per questo è presente un'altra matrice, una **bitMask, utilizzata per mappare quali caselle della**

Board potranno essere occupate dagli Item e quali no.
Utilizziamo dunque una Board astratta che in base al numero di giocatori selezionato dal primo giocatore istanzia o una TwoPlayersBoard, una ThreePlayersBoard o una FourPlayersBoard, che presentano una bitMask specifica in base al numero di giocatori. La bitMask è necessariamente dipendente da questo numero, in quanto alcune caselle della board sono occupabili da item solamente in partite con un certo numero di giocatori.

1.3 ServerController

Riguardo il controller lato server, la nostra scelta è quella di voler implementare **sia RMI che TCP**. La classe GameController conterrà tutti i metodi per controllare e modificare il model in base ai messaggi che arriveranno dai client, tutti i metodi saranno quelli invocati tramite RMI infatti. Per TCP invece abbiamo in mente di suddividere la rete sui tre livelli (*fisico, serialize/deserialize e TCP*), per questo abbiamo la classe SocketManager che si occuperà di ricevere tutti i messaggi dalla rete, la classe SerializeDeserialize che si occuperà di tradurre i bytestream che verrà inoltrato da SocketManger in messaggi TCP, ed infine la classe TCPMessageController che si occuperà di mappare tutti i messaggi TCP inoltrati da SerializeDeserialize in chiamate a metodi della classe GameController.

1.4 Ringraziamenti

Vi ringraziamo nuovamente per il tempo dedicatoci, non esitate a contattarci per ulteriori chiarimenti.
Edoardo, Samuele, Simone e Iacopo - Gruppo AM41