Relazione “Mastermind” - Joel Sina Mat.100780

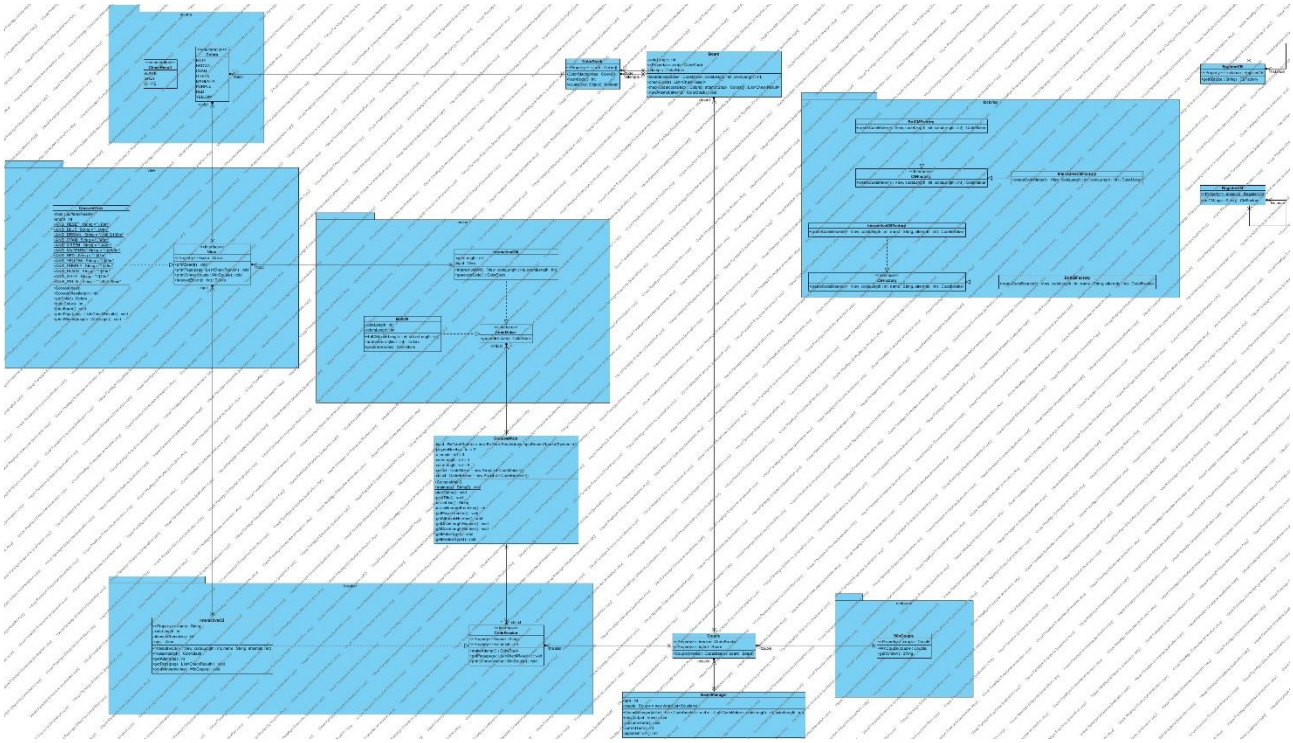
Premessa:

Il progetto è partito col presupposto che ogni elemento creato all’interno del progetto avrebbe avuto una specifica funzione, così modifiche e/o altre implementazioni sarebbero molto più semplici.

Abbiamo dei pacchetti divisi in “gruppi” per individuare le diverse entità, nel caso di Mastermind potremmo avere il CodeMaker e il CodeBreaker, ovvero colui che crea il codice da decrittare e colui che fa dei tentativi per individuare il codice.

Architettura:

L’architettura di questo progetto può essere veduta come l’uso del pattern MVC. Infatti, i due modelli più importanti che possiamo notare sono il CodeMaker e il CodeBreaker. La View è implementata semplicemente utilizzando la console o il terminale per input e output. Il controller può essere notato nel fatto di aver fatto i vari controlli dei turni e il fatto stesso di giocare in classi differenti e che non c’entrano con i modelli e le views.



Il progetto inizia da ConsoleMain, questa classe ha la responsabilità di ricevere tramite la console le informazioni per costruire le proprietà della partita. Infatti, si

chiedono varie proprietà per costruire la tavola dei colori e la lunghezza, insieme alla scelta dei giocatori e il nome da dare.

Queste informazioni vengono ulteriormente passate al GameManager che è il core dell'applicazione.

Per facilitare l'estensibilità dei giocatori, sono stati creati dei Factory per creare il giocatore in base alla proprietà ricevuta da ConsoleMain.

Precisazioni:

Sono implementati di default 2 CodeMaker: uno di tipo interattivo e l'altro di tipo randomico. Il tipo interattivo fa scegliere tramite la console la sequenza da decifrare.

È stato implementato però soltanto 1 CodeMaker: la scelta è andata su quello di tipo interattivo in quanto avere un giocatore randomico che prova ad azzeccare il codice aveva poco senso. Si poteva implementare l'algoritmo delle 5 supposizioni ideato da Knuth, ma ha prevalso l'idea di avere comunque una persona fisica che prova ad individuare il codice.

A differenza dal codice da decifrare che può essere creato in modo randomico per fare per esempio una modalità single player, non vedevo la necessità di avere un bot che provava a decifrare la sequenza.

È stato utilizzato anche l'aiuto delle Registry per avere meno codice da modificare in caso di aggiunta di nuovi CodeBreaker o CodeMaker

Il programma prevede dei parametri di default come la lunghezza del codice oppure la quantità di colori da utilizzare, in caso non si vogliano modificare. L'unica cosa che si deve inserire è il nome del giocatore, questo nome alla fine è utilizzato per far capire chi ha vinto.

Detto questo possiamo soffermarci al concetto di "Couple" o coppia. Ad ogni giocatore viene associata una tavola da decifrare. Si ha appunto una relazione uno a uno, nel caso di più giocatori non possiamo confonderci con le tavole da decifrare in quanto all'interno della Couple salviamo entrambi. Basta ripescare una couple dalla Lista e possiamo dire al giocatore di fare il tentativo, di dire alla tavola di controllarlo e di ritornare un risultato in base al controllo fatto.

Funzionalità nuove:

Anche se l'utilizzo delle nuove funzionalità di Java come Streams e Lambda expressions non erano obbligatorie e gli stessi risultati potevano essere ricavati in modi un po' più lunghi e con variabili intermediari, perché reinventare la ruota e non

utilizzare i tools di Java. Gli streams sono stati utilizzati per semplificare in fatto di velocità di scrittura di codice di stampa delle informazioni.

Molto più pratiche e veloci non dovendo fare un blocco ***for*** tradizionale.

Oltre alla stampa sono state usate anche per vedere se un colore nel codice da decifrare è stato messo oppure no, invece di scorrere tutto l'array con un ***Array.streams*** e il metodo *anyMatch* si poteva avere subito il booleano del risultato.

Estendibilità:

Il progetto nello stato attuale si può estendere da ogni lato. La classe "principale" per il comportamento del gioco è la GameManager. Una volta passate i parametri a quella classe si può fare qualsiasi estensione.

Per esempio, si può implementare una UI con JavaFX tranquillamente, provvedendo di implementare i metodi dell'interfaccia **View** e richiamando GameManager con i dovuti parametri. Inoltre, si potrebbe fare pure un porting per Android, anche in quel caso basterebbe fare qualche piccola modifica, ma anche quelle modifiche sarebbero legate al fatto della View e non delle capacità di Android in per sé.

Si potrebbe implementare anche una sorta di "multiplayer" tramite la rete. Avere un server che accetta più client. Aprire un socket ed associare al socket un thread che fa solo quel lavoro, stando in attesa per qualcuno di connettersi. Una volta che una persona si connette si crea un nuovo Thread tra server e client fino a quando non finisce la partita oppure il client non vuole sconnettersi.

Junit & Gradle:

È stato implementato il sistema Gradle per il building del progetto. Gradle ha all'interno pure Junit per i Test Case, la versione di Junit è la 4, ma con le dovute modifiche al file ***build.gradle*** si può installare l'ultima versione. La cosa bella di Gradle è che se serve una dependency, basta aggiungerla e ci penserà lui a fare il download, installazione ecc.

I test case usati sono pochi. Non ho creato test per i setter e getter in quanto sono dei semplici metodi e non serve fare dei test case per quelli. Siccome non si possono fare test per i metodi privati, ma solo di quelli pubblici, ho cercato di fare i più possibili su quelli pubblici e non modificare la visibilità dei metodi solo per i test, ma forse era quello che avrei dovuto fare!