Universitá degli studi di Napoli Uniparthenope

Dipartimento di Scienze e Tecnologie

Progetto di Programmazione 3

Docenti:

Prof. Angelo Ciaramella

Prof. Raffaele Montella

Studente:

Giovanni D'Angelo

0124/1369

Anno Accademico 2019/2020

Contents

Chapter 1

Descrizione della Traccia

Traccia Originale

Si vuole sviluppare un'applicazione per la simulazione di un gioco *di carta e matita* noto come Tris. L'applicazione permette di identificare un giocatore tramite il suo *nome* e *cognome*. Il secondo giocatore è il computer. Si considera una griglia quadrata di 3 x 3 caselle. A turno, i giocatori scelgono una cella vuota e vi inseriscono il proprio simbolo (di solito un giocatore ha come simbolo una "X" e l'avversario un cerchio "O"). Vince il giocatore che riesce a disporre tre dei propri simboli in linea retta orizzontale, verticale o diagonale. Se la griglia viene riempita senza che nessuno dei giocatori sia riuscito a completare una linea retta di tre simboli, il gioco finisce in parità. Il computer ha la possibilità di scegliere tra due strategie: *difesa* e *attacco*. L'utente sceglie inizialmente un valore K legato alla probabilità di scelta nella strategia. Nella *difesa* il computer applica, nell'ordine, queste scelte

- nel $K\%$ dei casi inserisce il proprio simbolo in una posizione ricavata dall'albero minimax che permette di "intercettare" il tris del giocatore. Nel caso non sia possibile, il simbolo viene inserito in una delle possibili posizioni descritte dall'albero minimax.
- nel (100-K)% dei casi inserisce il proprio simbolo in una posizione casuale.

Nella strategia attacco il computer applica queste scelte

- nel $K\%$ dei casi inserisce il proprio simbolo in una posizione ricavata dall'albero minimax che permette di "completare" un tris. Nel caso non sia possibile, il simbolo viene inserito in una delle possibili posizioni descritte dall'albero minimax.
- nel (100-K)% dei casi inserisce il proprio simbolo in una posizione casuale.

Sviluppare un'applicazione per il gioco del Tris che permette la visualizzazione della scacchiera e delle mosse dei giocatori. Le condizioni finali di vincita sono verificate automaticamente dal computer. Permettere inoltre il salvataggio di una partita e il ripristino della stessa in un momento successivo. All'inizio e alla fine di ogni partita viene visualizzata la classifica delle partite vinte dai diversi giocatori.

Revisione della traccia

In fase di progettazione è stato fatto noto che esiste un problema inerente la scelta della strategia da parte del computer. il valore K va a identificare quale sarà la scelta della mossa (se usare il minimax o meno) ma non della strategia (attacco o difesa). Inoltre, l'algoritmo minimax, che sarà visto nel dettaglio nel capitolo successivo, ha la funzione di scegliere la mossa che minimizza la massima perdita, ignorando di fatto se questa mossa possa intercettare o attaccare. E' stata dunque fatta la seguente modifica nella scelta della stratgia del computer, eliminando il concetto di attacco e difesa:

L'utente sceglie inizialmente un valore K legato alla probabilità di scelta nella strategia. il computer applica queste scelte:

- nel $K\%$ dei casi inserisce il proprio simbolo in una posizione ricavata dall'albero minimax che massimizzi la probabilità di vincere.
- nel (100-K)% dei casi inserisce il proprio simbolo in una posizione casuale.

Al fine di rendere l'applicativo più intuitivo, facile da utilizzare (e divertente, NDR), è stata modificata la scelta della $K\%$. Infatti l'utente potrà scegliere 4 ordini di difficoltà che corrispondono a dei valori di k fissi. Essi sono:
Facile = 25%, Medio = 50%, Difficile = 75%, Impossibile = 100%.

Chapter 2

Scelte progettuali

Saranno descritte ora le scelte progettuali fatte a priori in fase di progettazione. Si è utilizzato per l'implementazione come richiesto il linguaggio Java ed in particolare la sua versione **Java 8**, per poter utilizzarne il potenziale introdotto da questa versione, come interfacce funzionali e funzioni lambda.
Per l'implementazione dell'interfaccia grafica si è scelto di utilizzare il framework Java **Swing** per la sua semplicità nel gestire alcune operazioni grafiche. Inoltre, è stato fatto uso del Java GUI Designer **WindowBuilder**, offerto dall'IDE **Eclipse**, per disegnare interfacce grafiche complesse in modo semplice.
Si è fatto uso dei **file** per il salvataggio delle partite degli utenti e di un database basato su **MariaDB** come DBMS e sintassi **MySQL**.
Nella realizzazione di questo progetto sono stati utilizzati un totale di un totale di sei pattern riconosciuti dalla **Gang of Four**.

- Chain of Resposability
- Factory Method
- Observer
- Singleton
- State
- Strategy

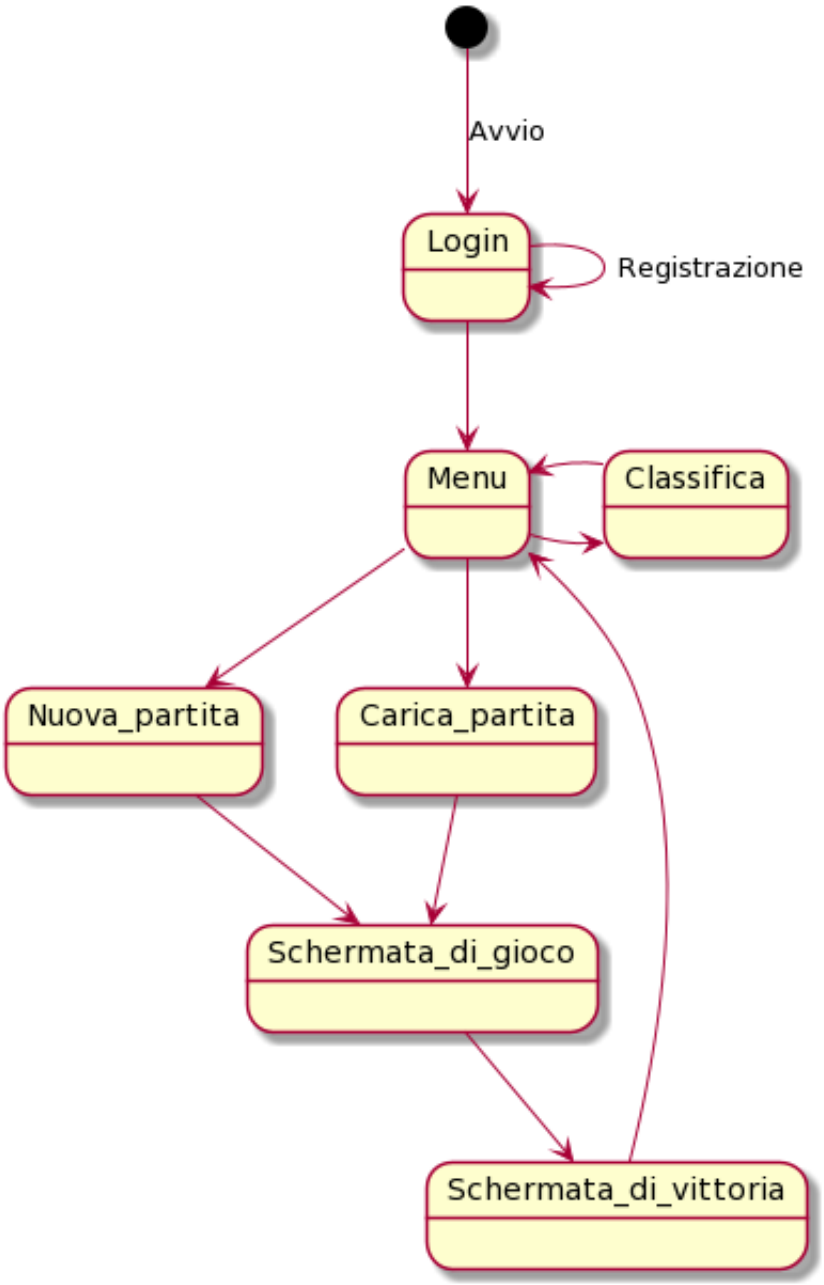
E' stato inoltre utilizzato un pattern non riconosciuto dalla Gang of Four:

- Data Access Object (DAO)

Chapter 3

Diagramma degli stati

Sarà mostrato di seguito il diagramma degli stati dell'applicativo



L'utente dopo aver avviato l'applicativo può eseguire il *Login* se possiede già un account presente nel DB, o in alternativa registrarsi nella stessa schermata.
Una volta effettuato il login, viene avviato il *Menù* principale. All'interno del menù è possibile visualizzare la *Classifica*, contenente le partite migliori fatte dagli utenti registrati, *Avviare* una nuova partita o *Caricare* una precedentemente salvata.
Prima di avviare la partita verrà mostrata la schermata di scelta della difficoltà e del simbolo che si vuole utilizzare in gioco.
Che si inizi o si carichi una partita, viene avviata la *Schermata di gioco*, dove, oltre a poter effettuare le mosse, l'utente può in qualunque momento salvare la partita.
Al termine di una partita, verrà mostrata una *schermata di vittoria*, dove verrà decretato il vincitore, nella quale non è possibile effettuare operazioni se non la chiusura della stessa schermata, la quale riporterà l'utente al menù di gioco.

Chapter 4

Descrizione dell'algoritmo

Come richiesto dalla traccia, è stato implementato un algoritmo di intelligenza artificiale, chiamato **Minimax**, per poter descrivere una delle due strategie effettuabili dal computer in fase di gioco.
Il **Minimax**, nella teoria delle decisioni, è un metodo per minimizzare la massima perdita possibile. Fu scoperto nella teoria dei giochi in caso di gioco a somma zero con due giocatori, sia nel caso di giochi a turni sia di mosse simultanee, e fu successivamente esteso a giochi più complessi e al supporto decisionale in presenza di incertezza.
L'idea di base è questa:
Se il giocatore **A** può vincere con una sola mossa, la mossa migliore è quella vincente.
Se il giocatore **B** sa che una data mossa porterà **A** a poter vincere con la sua prossima mossa, mentre un'altra lo porterà a pareggiare, la migliore mossa del giocatore **B** è quella che lo porterà alla patta.
Verso la fine del gioco è facile capire quali sono le mosse migliori; l'algoritmo Minimax trova la mossa migliore in un dato momento cercandola a partire dalla fine del gioco e risalendo verso la situazione corrente. Ad ogni passo l'algoritmo assume che il giocatore **A** cerchi di massimizzare le sue probabilità di vincere, mentre **B** cerchi di minimizzare le probabilità di vittoria di **A**, massimizzando le proprie chance di vittoria.
Nei giochi a turni il principio del minimax assume la forma di algoritmo minimax, cioè un algoritmo ricorsivo per la ricerca della mossa migliore in una determinata situazione.
É costituito da una funzione di valutazione che misura la bontà di un determinato stato nel gioco e indica quanto è desiderabile per il dato giocatore raggiungere quella posizione; il giocatore fa poi la mossa che minimizza il valore del miglior stato raggiungibile dall'altro giocatore. Quindi l'algoritmo assegna un valore ad ogni mossa, proporzionale a quanto essa diminuisce il valore della posizione per l'altro giocatore.
Il valore che viene assegnato ad ogni stato viene valutato secondo una fun-

zione che cambia in base al gioco (Nel nostro caso è stato scelto un valore arbitrario pari a 10 in caso di vittoria, -10 in caso di sconfitta, 0 in caso di pareggio).
Nella pratica, l'algoritmo di Minimax si occupa di esplorare tutto l'albero delle decisioni del gioco, dove ogni nodo corrisponde ad una mossa legale ed i figli dei nodi sono le mosse legali possibili dopo aver scelto la mossa selezionata. Ogni nodo ha un valore che dipende dal turno in cui viene eseguita la mossa (se è il giocatore massimizzante o minimizzante) e dalle conseguenze di quella mossa (se porterà alla vittoria/perdita/pareggio). Ad esempio, il nodo i-esimo assumerà il valore del figlio con il valore più alto tra tutti i figli nel caso sia il turno del giocatore massimizzante oppure il figlio con il valore più basso nel caso sia il turno del giocatore minimizzante.
La complessità computazionale è NP-Completa, redendolo di fatto inutilizzabile in giochi non banali, come ad esempio gli scacchi.
La complessità di tempo è O(b^m) e la complessità di spazio O(bm), dove b è il numero di mosse legali e m è la massima profondità dell'albero di decisioni.
Di seguito uno pseudocodice dell'algoritmo:

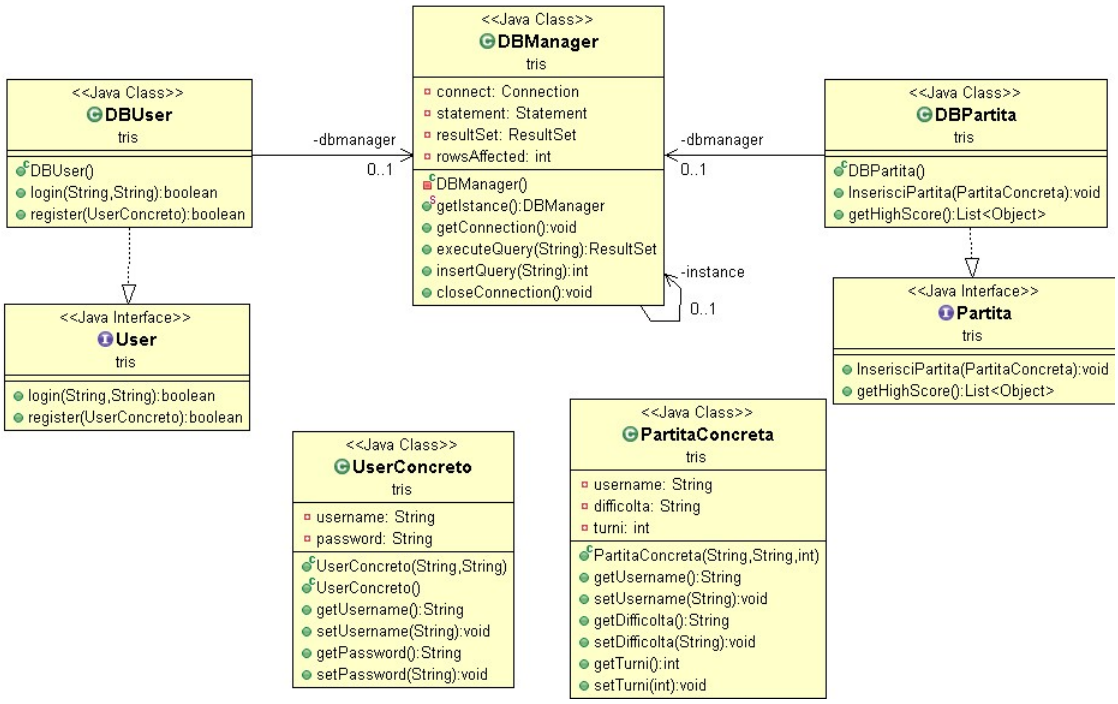
function Minimax ($Nodo, Profondita$);
Input : $Nodo$,$Profondita$
Output: a
if *Nodo è un nodo terminale OPPURE profondità = 0* **then**
  return il valore euristico del nodo;
end
if *l'avversario deve giocare* **then**
  $a := +\infty$
  for *Ogni figlio di nodo* **do**
    $a := min(a, minimax(figlio, profondita - 1))$
  end
else
  $a := -\infty$
  for *Ogni figlio di nodo* **do**
    $a := max(a, minimax(figlio, profondita - 1))$
  end
end

Chapter 5

Struttura del programma

In questo capitolo verranno analizzate la classi, raggruppate in base alle loro funzioni, descritte dal loro diagramma UML.

5.1 Database

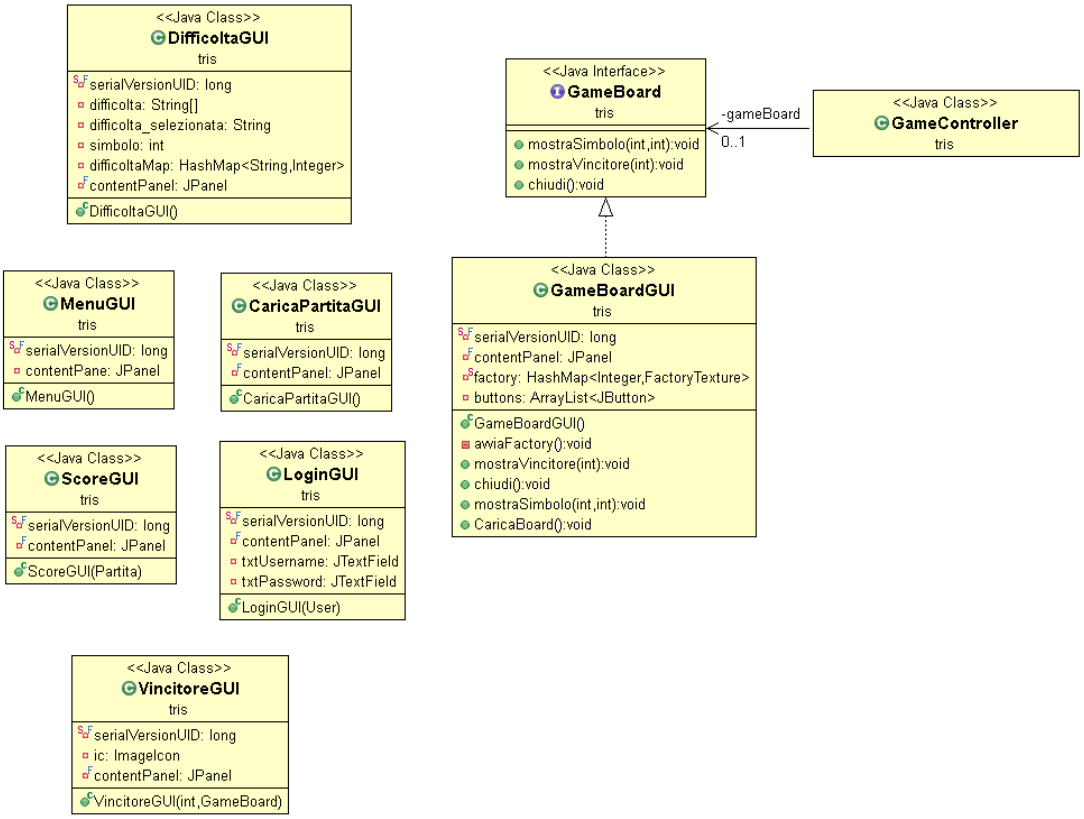


In questa sezione sono raggruppate le classi che interagiscono, direttamente o indirettamente con il Database.
É stato implementato il pattern *DAO*, che regola l'accesso ai dati di un modello, come il Database, per mezzo di un interfaccia astratta, separandola dalla sua implementazione. Nel nostro caso, Le interfacce **User** e **Partita** (che descrivono le operazioni sulle rispettive tabelle nel RDBMS) vengono

implementate dalle classi **DBUser** e **DBPartita**.
La classe **DBManager**, implementata tramite pattern *Singleton*, è l'unico punto di accesso effettivo verso il Database, rendendosi responsabile di avviare/chiudere la connessione e portatore di richieste per query di inserimento e modifica.
Sia la classe **DBUser** che **DBPartita** comunicano con il **DBManager** per poter eseguire le loro funzioni.
Le classi **UserConcreto** e **PartitaConcreta** sono classi ausiliarie che rappresentano tuple delle rispettive tabelle nel DB, utili al fine di trattarle come fossero oggetti, in modo da semplificarne operazioni come l'inserimento.

5.2 Interfaccia Grafica

In questa sezione sono raggruppate tutte le interfacce grafiche utilizzate e la relazione tra loro.

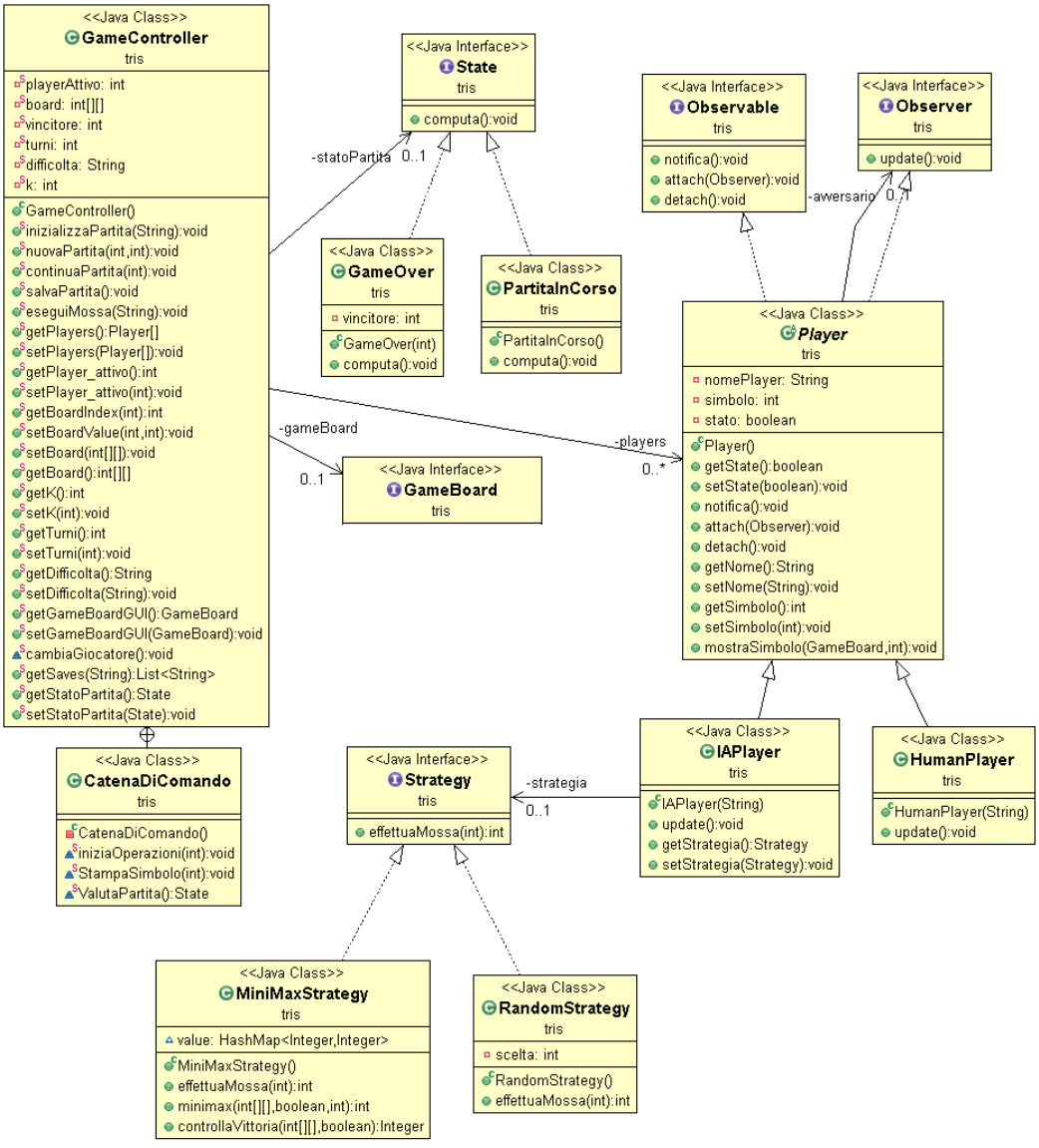


Queste classi rappresentano le interfacce grafiche utilizzate nel programma. L'ordine con cui sono visualizzate sono state descritte dal diagramma degli stati nel capitolo 4.
Si occupano di comunicare con le classi di controllo per poter eseguire le richieste dell'Utente con cui interagiscono.
textbfLoginGUI si occupa della gestione dell'accesso e registrazione da parte di un utente. **MenuGUI** si avvia dopo aver effettuato il login e si occupa di avviare **ScoreGUI**, che permette la visualizzazione dell'HighScore, **CaricaPartitaGUI**, che si occupa di caricare una partita già salvata in precedenza, e **DifficoltaGUI**, che si occupa di avviare una nuova partita, instanziando una **GameBoardGUI** con le impostazioni di difficoltà selezionate. La classe **VincitoreGUI** viene instanziata da **GameBoardGUI** quando la partita è terminata.
Viene fatto notare che la classe GameBoardGUI eredita l'interfaccia **GameBoard** la quale racchiude funzioni utilizzate dal **GameController**, classe

che ne gestisce le partite.

5.3 Logica di Gioco

In questa sezione sono raggruppate le classi che gestiscono la logica di gioco



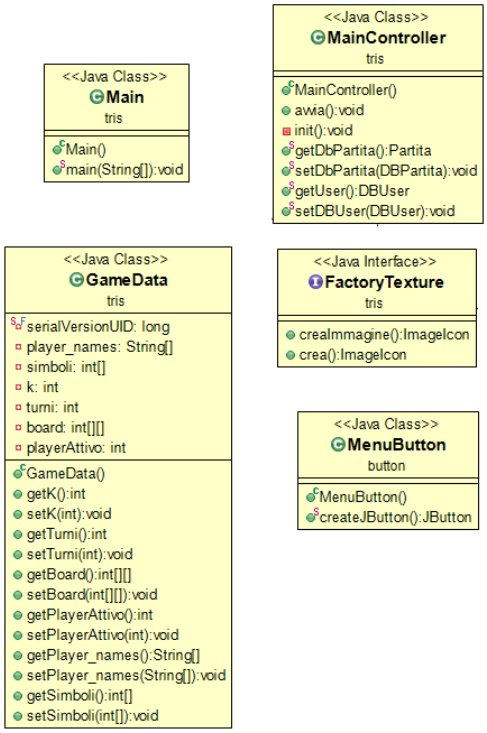
La classe **GameController** è la classe principale che si occupa di gestire le seguenti dinamiche di gioco:

- Avviare una nuova partita.
- Salvataggio e caricamento di una partita, attraverso l'utilizzo della classe d'appoggio **GameData** che sarà analizzata nel paragrafo successivo
- Eseguire una mossa di un **Player** durante una partita, attraverso la sua **CatenaDiComando**, mostrandola nella **GameBoard** associata ad essa.
- Tiene traccia dello stato della partita, attraverso il pattern *State*, che sarà analizzato nel capitolo 6.

La classe **Player** Rappresenta uno dei giocatori durante una partita. Il giocatore implementa il pattern *Observer* e può rappresentare o un giocatore reale o un giocatore controllato dall'Intelligenza artificiale. Quest'ultimo, sceglie la mossa da effettuare in base alla *Strategy* selezionata.

5.4 Classi Ausiliari

In questa sezione verranno visualizzato le classi ausiliari ed il loro ruolo all'interno del programma.



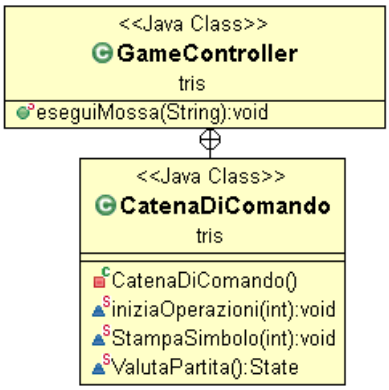
Il **Main** è il punto di inizio del programma e si occupa solamente di avviare il **MainController**. Quest'ultimo, nella sua procedura *init*, è il responsabile dell'instanziamento del **DBManager**, **DBUser** e **DBPartita**, inoltre ne contiene i riferimenti per eventuali esecuzioni di query. Si occupa anche di avviare la schermata di **LoginGUI**.
GameData è una classe che contiene le informazioni di una partita in corso. Implementa l'interfaccia *Serializable*, che ne consente la lettura e scrittura su File da parte del **GameController**.
FactoryTexture è un'interfaccia funzionale utilizzata per generare a runtime le immagini rappresentanti i simboli dei giocatori sulla scacchiera. Viene utilizzata in **GameBoardGUI**.
MenuButton è una *factory* generata da Java utilizzata dal **MenuGUI** che permette di generare pulsanti dello stesso tipo, modificandone solo il testo.[1]

[1] Questa classe è stata generata automaticamente dal plugin WindowBuilder e, nonostante utilizzi il pattern Factory method (non me ne prendo il merito NDR), di conseguenza non verrà analizzata successivamente.

Chapter 6

Pattern Utilizzati

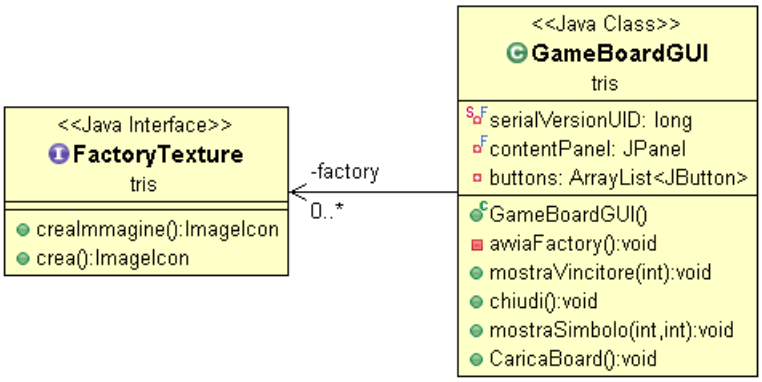
6.1 Chain of Responsability



Il pattern è stato implementato in modo diverso rispetto al modello originale della *Gang of Four*, ma la logica ne resta invariata, diminuendone però la quantità di codice. Secondo l'originale modello UML, avremmo necessitato di una classe differente per ognuno degli anelli della catena, mentre in questo caso basta un metodo per ciascuno di essi (I metodi privati della classe **CatenaDiComando**).
Il pattern permette di separare gli oggetti che invocano richieste, dagli oggetti che le gestiscono, dando ad ognuno la possibilità di gestire queste richieste. Viene utilizzato il termine catena perché di fatto la richiesta viene inviata e "segue la catena" di oggetti, passando da uno all'altro, finché non trova quello che la gestisce.
Nel progetto viene implementata una variante, denominata Catena di Comando, che assicura che un singolo oggetto faccia eseguire più tasks.
Nel nostro caso l'azione *eseguiMossa()* darà l'avvio alla **CatenaDiComando** che eseguirà le operazioni necessarie (Stampare il simbolo sulla **GameBoardGUI**, valutare lo stato della partita e, nel caso, decretarne il vinci-

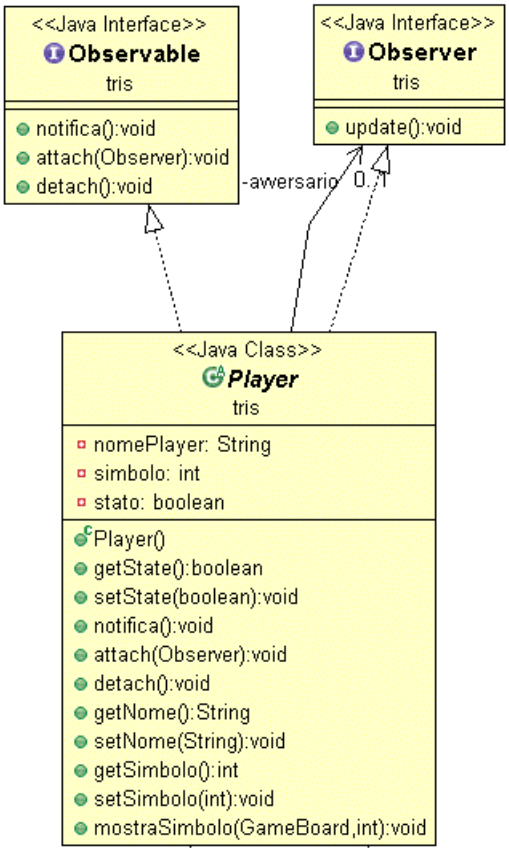
tore) in modo atomico.

6.2 Factory Method



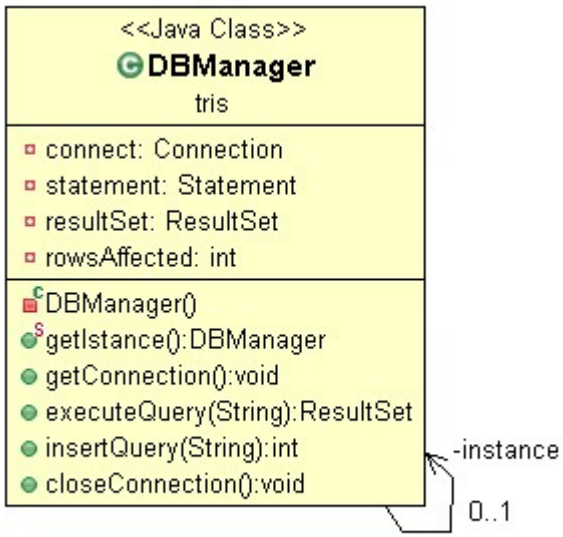
Questo pattern è stato utilizzato in quanto permette la creazione di un oggetto precludendo il suo riuso senza una significativa duplicazione di codice. Nuovamente, l'UML del pattern è diverso da quello originariamente concepito dalla *Gang Of Four* in quanto vengono sfruttate le potenzialità concesse dalle API di Java 8, senza modificarne la logica. Vengono usate infatti le interfacce funzionali le quali sono interfacce che hanno un solo metodo astratto (ma possono anche avere altri metodi di default ma uno solo astratto). Grazie a questa funzionalità, è stato possibile implementare il Factory Pattern utilizzando una HashMap che ha per chiave l'ID del simbolo da creare e per valore la creazione dello stesso (implementato per mezzo di un'espressione lambda). Ciò semplifica la gerarchia delle classi, permettendo inoltre future estensioni con poche righe (l'inserimento di una nuova coppia nella HashMap sopra citata)

6.3 Observer



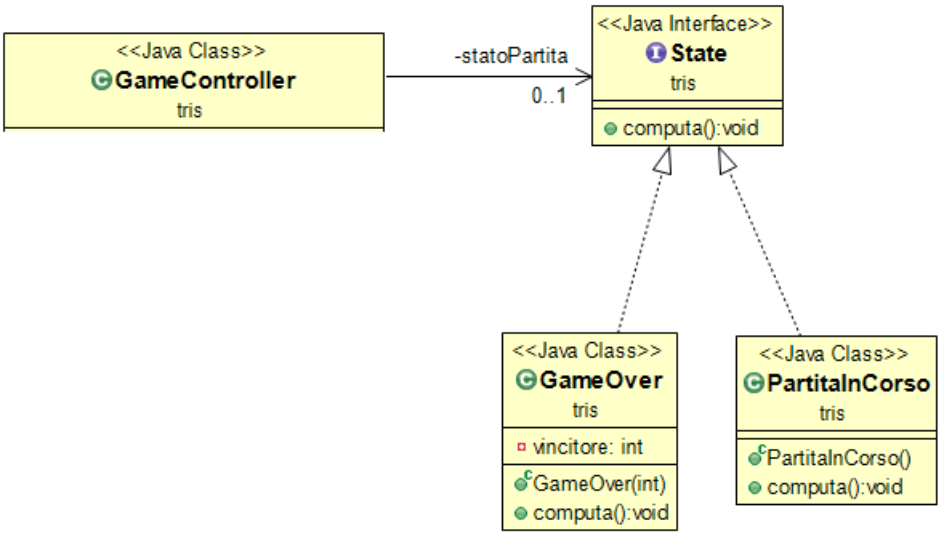
Il pattern Observer consente ad un oggetto (**Observable**) di mantenere una lista dei suoi dipendenti (**Observer**) e li notifica automaticamente ogni volta che cambia il suo stato, chiamando un loro metodo.
E' stato utilizzato per simulare la sincronizzazione di un gioco da tavolo a turni per due giocatori, dove un giocatore attende il suo turno finchè l'avversario non ha terminato il proprio. il **Player** assume infatti il ruolo di oggetto osservato e osservatore, ereditandone le interfacce. Prima di una partita, il Player setta il suo avversario, un altro oggetto di tipo Player, attraverso il metodo *attach()*. In ogni istante della partita, un giocatore avrà uno *stato*, un valore booleano che indica se sia il suo turno o meno. Quando il giocatore avrà passato il suo turno verrà settato il suo stato a *false* il quale manderà una *notifica()* all'avversario, richiamandone il suo *update()*, avvisando che è il suo turno.

6.4 Singleton



Il pattern Singleton ha lo scopo di garantire che una ed una sola istanza di una determinata classe e di fornire un punto di accesso globale a tale istanza. È stato utilizzato per implementare il **DBManager**, unico punto di accesso per effettuare query al database.

6.5 State



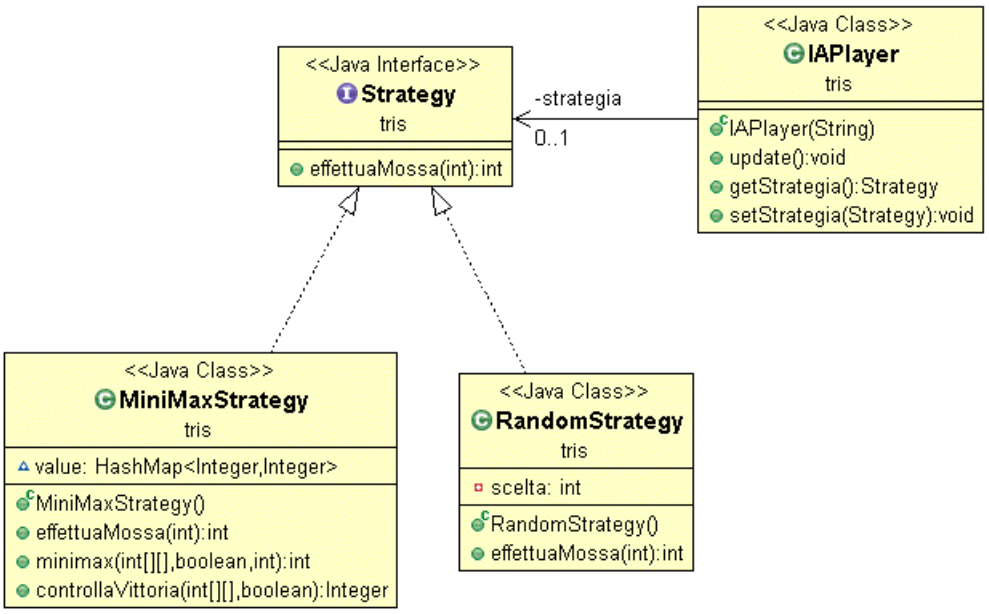
L'obiettivo di questo pattern è di risolvere due problemi:

- Un oggetto deve essere in grado di cambiare il suo comportamento quando cambia il suo stato interno
- Un comportamento che dipende da uno stato deve essere definito in modo indipendente, tale da non modificare il comportamento di stati esistenti.

Il pattern viene utilizzato in quanto il **GameController** (che ne rappresenta il contesto) non implementa direttamente i comportamenti che deve avere una partita ma fa riferimento alla sua interfaccia per compiere azioni che dipendono dal suo stato (attraverso il metodo *computa()*, che rende il **GameController** indipendente da come sono stati implementati i comportamenti che dipendono dallo stato della partita.[1]

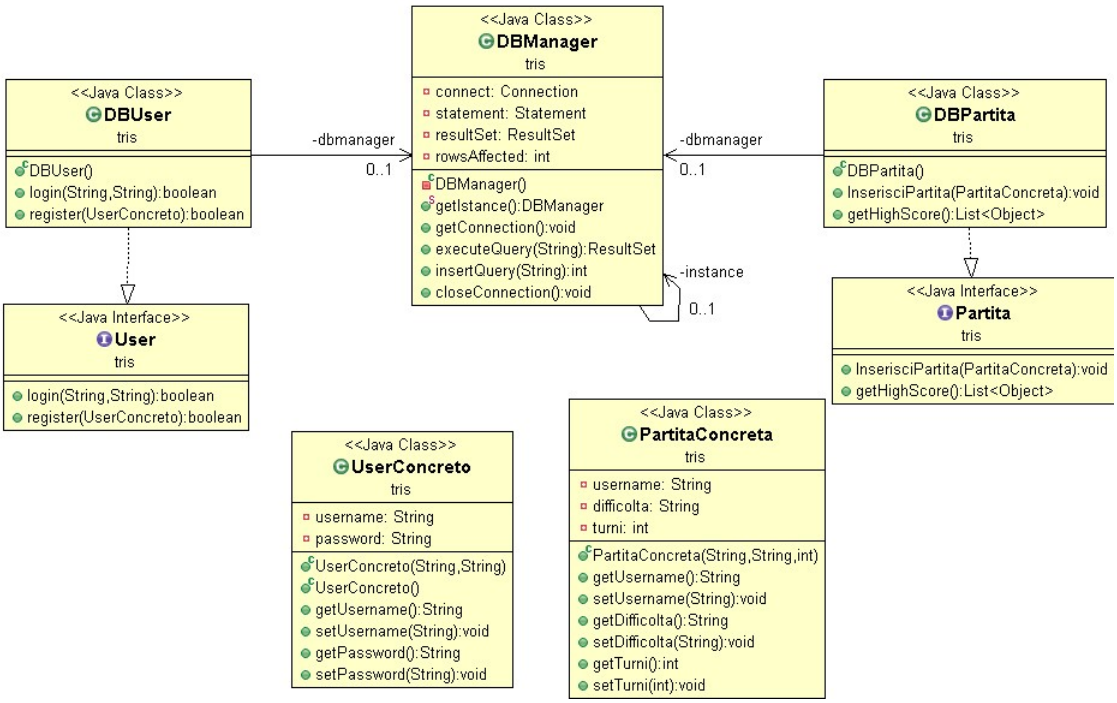
[1]Nonostante sia il GameController a conoscere lo stato della partita, è in realtà la classe **CatenaDiMontaggio**, la sua classe classe nidificata, che si occupa di richiamare la funzione **State**.*computa()*.

6.6 Strategy



L'obiettivo di questo pattern è isolare un algoritmo all'interno di un oggetto, in maniera tale da risultare utile in quelle situazioni dove sia necessario modificare dinamicamente gli algoritmi utilizzati da un'applicazione. Il motivo di utilizzo è abbastanza intuitivo, ad ogni istante è necessario che, in base alla percentuale **K** , l' **IAPlayer** sceglie uno dei due algoritmi (**Minimax** o **Random**) per eseguire la mossa sulla scacchiera.
L'utilizzo di questo pattern rende possibile l'aggiunta di altri algoritmi molto semplice, data la loro intercambiabilità.

6.7 DAO



Questo pattern, non presente tra i 23 pattern della *Gang Of Four*, fornisce una interfaccia astratta che consente di separare le business dell'application processing layer dalle operazioni di accesso ai dati del data management layer. Piuttosto che interfacciarsi direttamente ad un eventuale RDBMS, il client utilizza le interfacce fornite dal **DAO**, che nasconde completamente i dettagli dell'interazione con la sorgente dei dati.
L'interfaccia esposta dal DAO al client non cambia quando l'implementazione dell'origine dati sottostante cambia, e questo consente al DAO di adattarsi a diversi schemi di archiviazione senza dover modificare nullasugli altri layer. In sostanza, il DAO funge da adapter tra il componente della business logic e l'origine dati.
Il pattern è stato utilizzato, quindi, per fornire un'interfaccia per accedere ai dati contenuti nella base di dati.

Chapter 7

Conclusioni e Ringraziamenti

Verranno qui descritte le considerazioni relative all'elaborato ed i ringraziamenti.

7.1 Considerazioni

La realizzazione di questo progetto ha fortificato le conoscenze apprese durante il corso dei vari pattern, facendone capire il loro potenziale reale.
Lo studio e l'implementazione dell'algoritmo Minimax mi ha mostrato un mondo in continua evoluzione, quello dell'intelligenza artificiale che, per quanto i costi computazionali (di questo algoritmo in particolare) siano elevati possono creare delle meraviglie tecnologiche, come un computer virtualmente invincibile.

7.2 Ringraziamenti

Ringrazio Alan Mycroft e Mario Fusco per il refactoring del pattern Chain of Responsibility e del factory method, autori del libro **Java 8 in Action**.
Ringrazio i miei amici e la mia ragazza per il loro supporto morale.