Università degli Studi di Napoli "Parthenope"
Corso di laurea in Informatica (percorso Generale)

Corso di Programmazione 3 e Laboratorio

Gioco Forza 4 (IA)

Docente
Angelo Ciaramella

Candidato
Raffaele Talente
0124002658

Anno Accademico 2023–2024

Contents

1 Introduzione

L'obiettivo del progetto è la creazione di un programma per simulare il gioco da tavolo *Forza 4*. L'obiettivo del gioco è mettere in fila (orizzontale, verticale o diagonale) quattro pedine del proprio colore, facendo cadere le pedine in maniera verticale su di una colonna dentro una griglia rettangolare di dimensioni **6x7**.

2 Sintesi dei requisiti

2.1 Requisiti funzionali

- Il programma deve poter simulare una partita tra l'utente e un avversario finto le cui mosse sono effettuate dall'applicazione stessa sfruttando un particolare algoritmo selezionato opportunamente. Per semplicità, ci si riferirà a tale avversario finto semplicemente come "Computer".
- Le pedine sono rappresentate nella griglia da un colore, il rosso simboleggia una pedina controllata dall'utente, il giallo simboleggia una pedina controllata dal computer.
- L'utente deve poter inserire all'interno della griglia di gioco una propria pedina in una qualsiasi colonna libera, il programma deve rispondere inserendo una pedina controllata dal computer.
- Seguendo le regole del gioco classico, nel momento in cui quattro pedine dello stesso colore sono in fila in una qualsiasi direzione, il gioco termina e il controllore di quelle pedine vince la partita.
  Se la partita è stata vinta dall'utente, il risultato deve essere salvato.
- L'utente deve essere in grado di autenticarsi all'interno dell'applicazione inserendo nome e cognome.
  Dopo aver inserito tali dati, il programma deve registrarlo opportunamente, in modo che i risultati delle partite vengano memorizzati.
- L'utente deve poter scegliere una modalità di gioco, che simboleggia il modo in cui il computer inserisce le proprie pedine. Le modalità disponibili sono le seguenti:

  - **Difesa:** Il computer controlla il numero massimo di pedine in fila del giocatore, e inserisce una propria pedina per tentare di bloccare la sequenza.
  - **Attacco:** Il computer utilizza l'algoritmo **Minimax con Alpha-beta pruning**[3] per dare una valutazione alle mosse attualmente disponibili - dove per mossa si intende l'inserimento di una propria pedina all'interno di una colonna - e giocare la mossa con la valutazione più alta.
    L'algoritmo in questione verrà discusso nel dettaglio in una sezione successiva.
  - **Neutrale:** Il computer effettua una mossa scegliendo in maniera casuale tra la modalità Difesa e Attacco.

- Il risultato di una partita deve essere memorizzato dal programma, deve essere inoltre possibile per l'utente visualizzare una classifica degli utenti che hanno totalizzato i punteggi più altri, dove per punteggio si intende il numero di vittorie.
- Deve essere possibile per l'utente mettere in pausa una partita in un qualsiasi momento per riprenderla successivamente.

2.2 Requisiti non funzionali

- L'applicazione deve essere sviluppata utilizzando il linguaggio **Java** e attenendosi ai principi della programmazione **SOLID**.
- L'applicazione deve prevedere un'interfaccia grafica per rendere agevole l'interazione con l'utente.
- E' necessario utilizzare almeno due Design Pattern tra quelli studiati.
- I dati devono essere memorizzati tramite file o database.

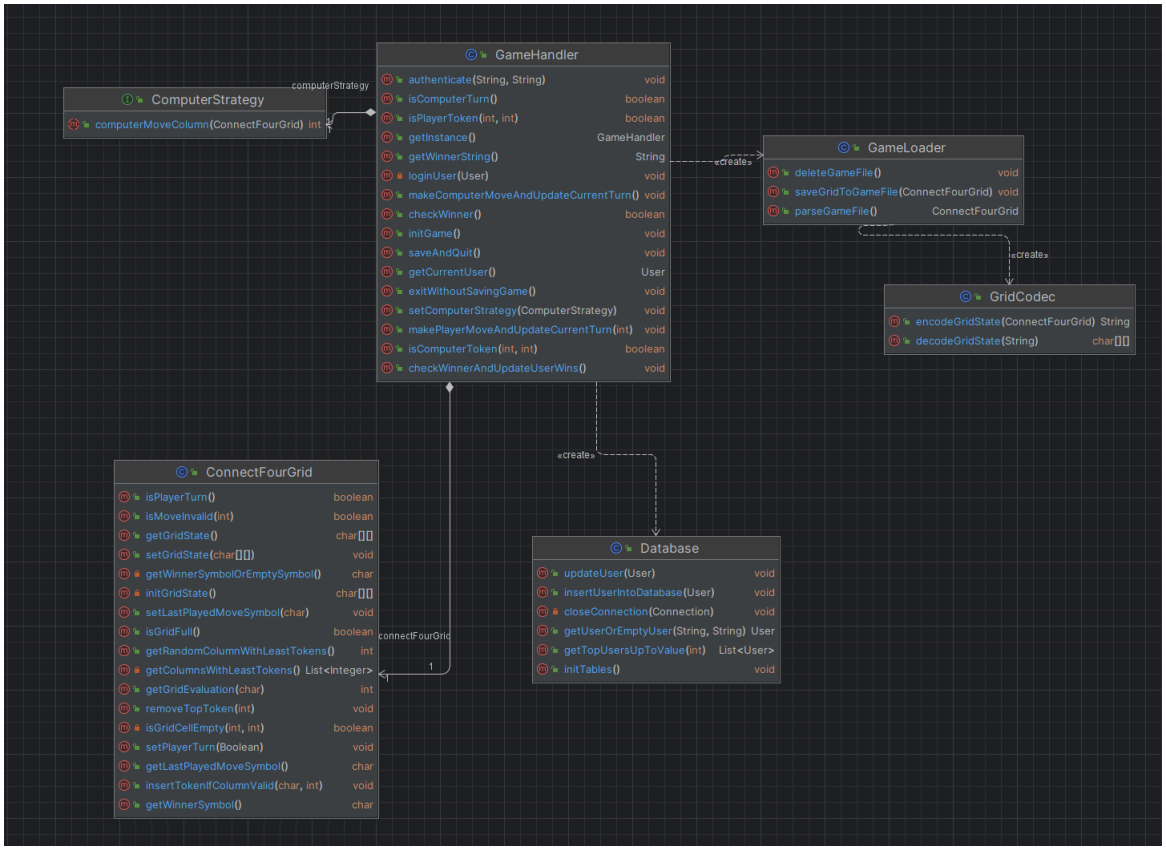

Figure 1: Diagramma UML della classe GameHandler

3 Il Game Handler

GameHandler è la *de facto* classe principale dell'applicazione.
La funzione della classe è quello di definire un insieme di metodi che permettano di gestire lo scorrimento della partita senza esporre l'oggetto griglia (definito dalla classe *ConnectFourGrid*).
Il Game Handler utilizza la classe ausiliaria *GameLoader* per delegare la gestione dei file, che a sua volta delega la codifica e decodifica della griglia alla classe *GridCodec*.
GridCodec permette di trasformare la rappresentazione della griglia di gioco nel programma, che utilizza un array di caratteri, in una stringa salvata su file, e di effettuare l'operazione inversa, al fine di permettere il salvataggio di una partita in corso e la successiva ripresa.
Il Game Handler mantiene una reference ad un oggetto che implementa l'interfaccia *ComputerStrategy* che permette di ottenere una mossa da parte del computer quando richiesto.
Infine, si interfaccia con la classe *Database* per aggiornare l'utente at-

tualmente autenticato.
GameHandler implementa il Design Pattern **Singleton**[1] in quanto vogliamo esista solo un Gamehandler, e che questi sia globalmente accessibile.

Si potrebbe pensare che questa classe violi il *Single Responsibility Principle*, tuttavia tutto ciò che fa di concreto è indirizzare il proseguimento della partita richiamando metodi di altre classi, le implementazioni vere e proprie sono incapsulate in tali classi e pertanto totalmente disaccoppiate.

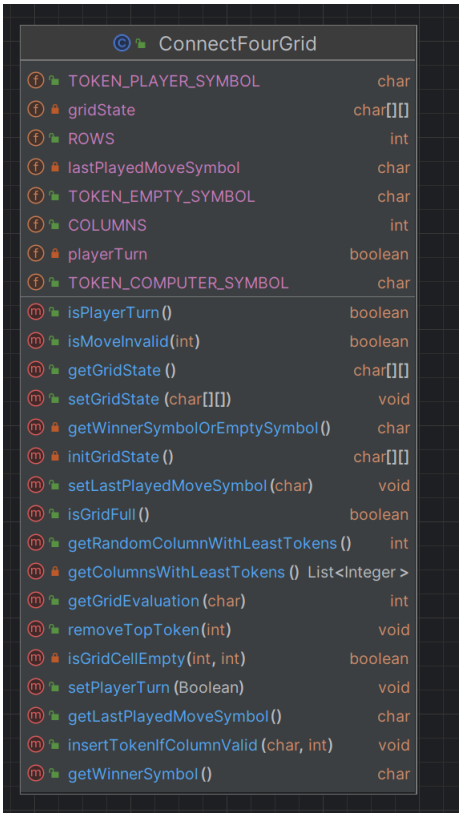

Figure 2: Diagramma UML della classe ConnectFourGrid

4 La griglia

La class *ConnectFourGrid* incapsula il comportamento della griglia rappresentata da un array bidimensionale di caratteri, ed espone metodi per intervenire direttamente sulle pedine presenti nella griglia e controllare la situazione all'interno di essa.

Il metodo insertTokenIfColumnValid controlla prima se la colonna scelta non è attualmente piena, e se possibile scorre la colonna fino a trovare lo spazio libero più in basso per inserire una pedina.

Il metodo getRandomColumnWithLeastTokens sfrutta la funzione ausiliaria *getColumnsWithLeastTokens* per ritornare una colonna casuale tra quelle che attualmente contengono il numero minore di pedine.

```
public void insertTokenIfColumnValid(char tokenSymbol,
                                     int column) {
    if (isMoveInvalid(column)) {
        return;
    }
    for (int i = 0; i < ROWS; i++) {
        if (i == ROWS - 1) {
            if (isGridCellEmpty(i, column)) {
                gridState[i][column] = tokenSymbol;
                this.lastPlayedMoveSymbol = tokenSymbol;
            }
        } else if (!isGridCellEmpty(i + 1, column) &&
                    isGridCellEmpty(i, column)) {
            gridState[i][column] = tokenSymbol;
            this.lastPlayedMoveSymbol = tokenSymbol;
        }
    }
}
```

Figure 3: Metodo utilizzato dalla griglia per inserire una pedina

```
public int getRandomColumnWithLeastTokens() {
    List<Integer> values = getColumnsWithLeastTokens();
    Random r = new Random();
    int index = r.nextInt(values.size());
    return values.get(index);
}
```

Figure 4: Metodo utilizzato per selezionare una colonna a caso tra quelle con meno pedine

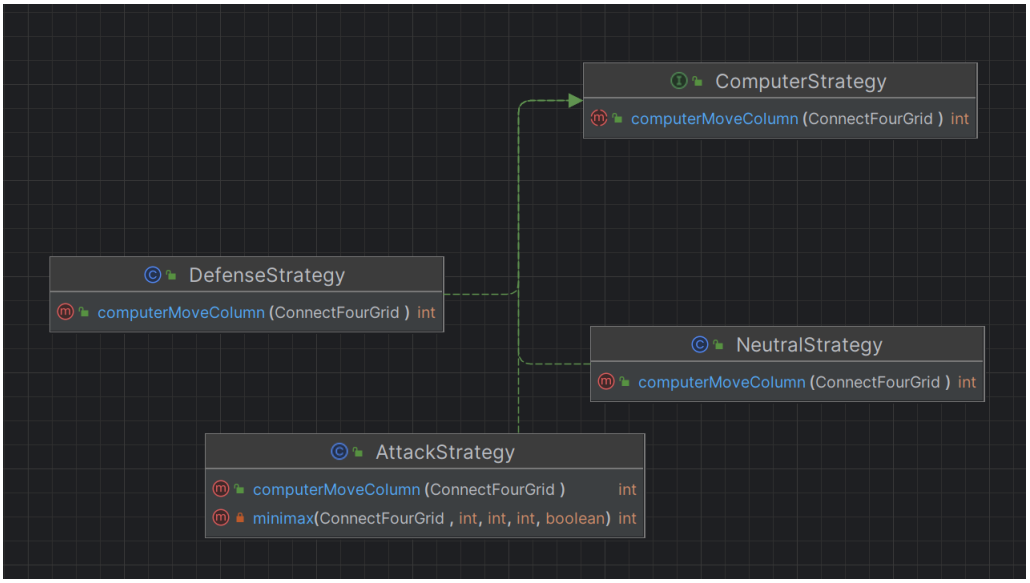

Figure 5: Diagramma UML dell'interfaccia ComputerStrategy

5 ComputerStrategy

L'interfaccia *ComputerStrategy* è utilizzata per implementare il Design Pattern **Strategy** [1], permettendo di incapsulare i diversi algoritmi utilizzati per la scelta da parte del computer di una mossa.
Strategy rende possibile cambiare l'algoritmo a run-time, e diventa estremamente facile aggiungere nuove modalità di gioco semplicemente creando nuove classi che implementano tale interfaccia. ComputerStrategy espone un unico metodo chiamato *computerMoveColumn* che ritorna la colonna dove inserire la pedina del computer.
L'algoritmo di scelta della colonna è così totalmente disaccoppiato dalla griglia e dal GameHandler.

5.1 AttackStrategy

La classe **AttackStrategy** utilizza l'algoritmo **Minimax con Alpha-beta pruning**[3] per scegliere la mossa. L'algoritmo viene implementato in questo modo:

- Si controllano tutte le colonne e vengono inserite in un array di interi dove l'indice nell'array indica l'indice della colonna stessa e il valore corrispondente a quell'indice indica la valutazione della

mossa stessa. A una mossa invalida viene associato il valore di *Integer.MIN_VALUE* in modo che non possa essere scelta.

- L'algoritmo di **minimax** viene utilizzato per dare una valutazione a ogni possibile mossa. (Continua sotto)
- Alla fine, le mosse con la valutazione più alta vengono inserite in un ArrayList. Se vi è più di una mossa migliore, ne viene selezionata una casualmente.

L'algoritmo **Minimax** è un algoritmo ricorsivo che si basa sull'idea di *minimizzare la massima perdita*. Applicato al gioco *Forza quattro* la massima perdita indica la perdita di una partita, ovvero quando l'avversario è in grado di concatenare quattro delle proprie pedine.
In generale, l'algoritmo funziona alternando i turni di inserimento delle pedine, e facendo finta di inserirle. Questo finto inserimento serve per valutare la situazione della griglia per controllare se vi è o meno un vincitore, e associare un punteggio alla mossa in questione.
L'alternanza dei turni permette a ogni ricorsione di valutare la miglior mossa dal punto di vista del turno corrente. Poiché l'algoritmo viene eseguito inizialmente dal computer, se la posizione della griglia indica una vittoria del giocatore, viene associato uno punteggio alla mossa molto basso, o più semplicemente negativo, viceversa una vittoria del computer associa alla mossa un punteggio maggiore.

La complessità dell'algoritmo Minimax è proporzionale al numero di nodi generati dall'albero di ricorsione, ovvero $O(b^d)$
dove b indica il numero di mosse possibili a ogni turno, e d la profondità della ricerca.
Tuttavia le prestazioni dell'algoritmo possono essere migliorate notevolmente applicando il cosidetto **Alpha-beta pruning**, che è una miglioria all'algoritmo base, in quanto evita di continuare a esplorare una mossa se almeno una sotto-mossa è stata trovata che renda quella mossa peggiore di un'altra già trovata. Per fare ciò l'algoritmo utilizza due valori chiamati *Alfa* e *beta* che mantengono rispettivamente il valore minimo che il giocatore "massimizzante" è assicurato di avere da una mossa, e il valore massimo che il giocatore "minimizzante" può ottenere da quella mossa.
La ricerca della mossa corrente viene fermata nel momento in cui

Profondità	Minimax	Alpha-beta
2	0.9 ms	0.9 ms
3	7.0 ms	2.5 ms
4	31.1 ms	8.1 ms
5	56.4 ms	19.8 ms
6	109.6 ms	31.7 ms
7	341.2 ms	44.1 ms
8	1.5 s	53.2 ms
9	10.4 s	100.0 ms
10	1 min	159.3 ms
11	7.4 min	387 ms
12		1.2 s
13		6.1 s
14		21.2 s
15		2 min
16		6.5 min

Figure 6: Tabella che indica i tempi di esecuzione in relazione alla profondità

$alpha \geq beta$.
Nel caso peggiore, l'Alpha-beta pruning ha le stesse prestazioni dell'algoritmo minimax classico, tuttavia nel caso medio le prestazioni si avvicinano di più a $O(\sqrt{b^d})$.
Per rendere l'idea, ho deciso di effettuare un piccolo "benchmark" dell'algoritmo minimax con e senza alpha-beta pruning, con tutti gli altri fattori lasciati immutati, i risultati sono visibili nella tabella 6.

L'implementazione della funzione di minimax è mostrata nella figura 7. Nell'applicazione ho deciso di optare per una ricerca base di profondità 10.

```
    private int minimax(ConnectFourGrid connectFourGrid, int depth,
                        int alpha, int beta, boolean isComputerTurn) {
    if(depth <= 0) {
        return 0;
    }
    int currentGridEvaluation = connectFourGrid.getGridEvaluation(
                                connectFourGrid.TOKEN_COMPUTER_SYMBOL);
    if(currentGridEvaluation == 1) {
        return depth;
    } else if(currentGridEvaluation == -1) {
        return -depth;
    } else if(connectFourGrid.isGridFull()) {
        return 0;
    }

    if(isComputerTurn) {
        int maxValue = Integer.MIN_VALUE;
        for(int j = 0; j < connectFourGrid.COLUMNS; j++) {
            if(connectFourGrid.isMoveInvalid(j)) { continue; }
            connectFourGrid.insertTokenIfColumnValid(
                            connectFourGrid.TOKEN_COMPUTER_SYMBOL, j);
            maxValue = Math.max(maxValue,
                        minimax(connectFourGrid, depth - 1,
                                        alpha, beta, false));
            connectFourGrid.removeTopToken(j);
            alpha = Math.max(alpha, maxValue);
            if(alpha >= beta) {
               break;
            }
        }
        return maxValue;
    } else {
        int minValue = Integer.MAX_VALUE;
        for(int j = 0; j < connectFourGrid.COLUMNS; j++) {
            if(connectFourGrid.isMoveInvalid(j)) { continue; }
            connectFourGrid.insertTokenIfColumnValid(
                            connectFourGrid.TOKEN_PLAYER_SYMBOL, j);
            minValue = Math.min(minValue,
                        minimax(connectFourGrid, depth - 1,
                                        alpha, beta, true));
            connectFourGrid.removeTopToken(j);
            beta = Math.min(beta, minValue);
            if(alpha >= beta) {
                break;
            }
        }
        return minValue;
    }
}
```

Figure 7: Algoritmo Minimax con Alpha-beta pruning

5.2 DefenseStrategy

La strategia *Difesa* cerca semplicemente di bloccare le pedine dell'utente, per fare ciò l'algoritmo controlla tutte le colonne, tutte le righe ed entrambe le direzioni diagonali (ascendente e discendente) e mantiene traccia del numero di pedine dell'utente in quella direzione all'interno di una *Hash Table* implementata come una *HashMap* di Java.
La hash table identifica ogni direzione con una stringa ottenuta concatenando il tipo di direzione (h - horizontal, v - vertical, ad - ascending diagonal, dd - descending diagonal) con il numero di colonna da cui è iniziato il contatore delle pedine.
In questo modo la HashMap contiene informazioni sul numero di pedine per ogni direzione, ma ogni direzione conserva implicitamente anche la colonna dove eventualmente inserire la pedina del computer.
La HashMap viene quindi ordinata per valori, e la chiave relativa al valore più alto viene riconvertita in un numero di colonna che viene ritornato.
Se ci sono più chiavi con lo stesso valore, viene ritornata la prima.
Se le chiave da ritornare è 0 o 1, viene invece ritornato il numero di una colonna a caso tra quelle contenenti meno pedine, utilizzando il metodo *getColumnsWithLeastTokens* della classe *ConnectFourGrid*.
L'implementazione della modalità difesa è visibile nel codice in figura 8.

```
    public int computerMoveColumn(ConnectFourGrid connectFourGrid) {
    char symbolToCount = connectFourGrid.TOKEN_PLAYER_SYMBOL;
    Map<String, Integer> playableColumnsWithMostConnectedTokens
                            = new HashMap<>();
    // horizontal
    for(int i = 0; i < connectFourGrid.ROWS; i++) {
        for(int j = 0; j < connectFourGrid.COLUMNS - 3; j++) {
            if(connectFourGrid.isMoveInvalid(j)) {
                continue;
            }
            int currentCount = 0;
            for(int k = 0; k <= 3; k++) {
                if(connectFourGrid.getGridState()[i][j + k]
                        == symbolToCount) {
                    currentCount++;
                }
            }
            playableColumnsWithMostConnectedTokens
                .put("h"+j, currentCount);
        }
    }
    // stessa operazione per le altre direzioni
    // ...
    // codice omesso per semplicità

    String highestKey = Collections.max(
            playableColumnsWithMostConnectedTokens.entrySet(),
            Map.Entry.comparingByValue()).getKey();

    int value = playableColumnsWithMostConnectedTokens
                    .get(highestKey);
    // regex per togliere i caratteri non-cifre
    int column = Integer.parseInt(highestKey
                        .replaceAll("[^\\d.]", ""));
    if(value == 0 ||
       value == 1 ||
        connectFourGrid.isMoveInvalid(column)) {
        return connectFourGrid.getRandomColumnWithLeastTokens();
    }
    return column;
}
```

Figure 8: Algoritmo che implementa la modalità difesa descritta

5.3 NeutralStrategy

La strategia *Neutrale* sceglie casualmente tra la strategia *Difesa* e la strategia *Attacco* per selezionare una mossa.
La scelta casuale è ottenuta attraverso il metodo *nextBoolean()* della classe *java.util.Random*.

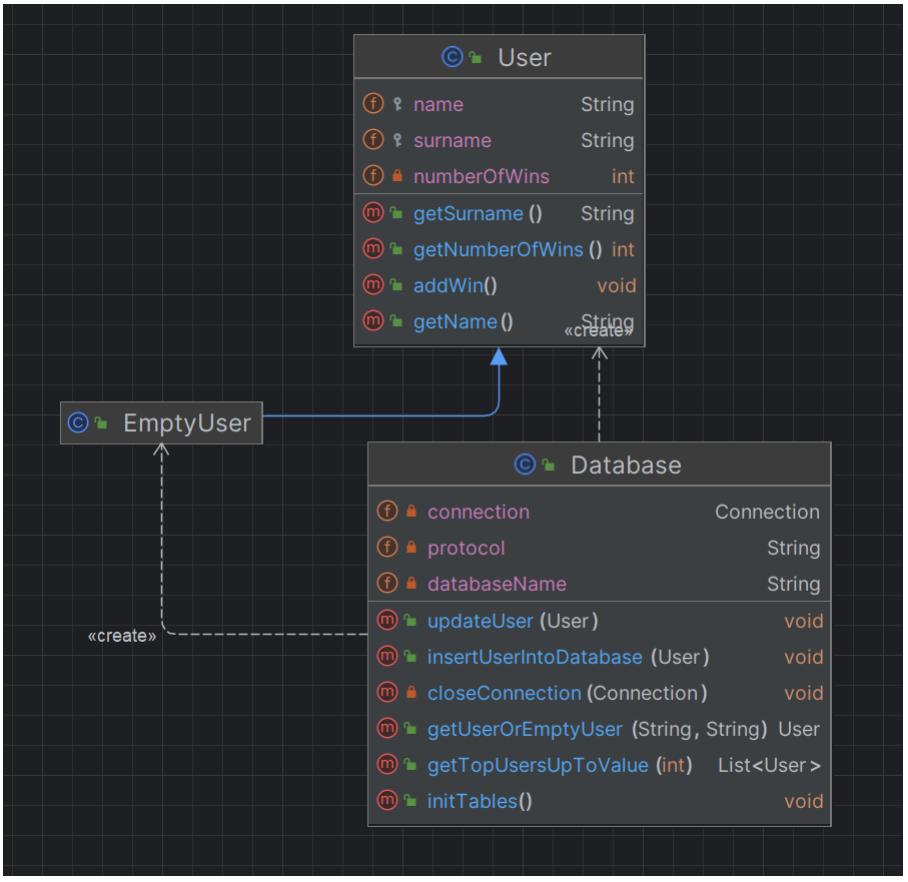

Figure 9: Diagramma UML della classe Database

6 Gestione del Database

La classe Database incapsula l'interazione con il Database SQLite, esponendo dei metodi che permettono di agire con la classe *User* traducendo le operazioni effettuate in operazioni sul database, e viceversa. Per esempio, la figura 10 mostra il metodo utilizzato per inserire un utente nella tabella utenti.

Tutte le operazioni sul database si compongono delle seguenti fasi:

- Creazione di una connessione tramite il metodo *DriverManager.getConnection(protocol + databaseName)* dove *protocol* e *databaseName* sono dei campi della classe Database che indicano rispettivamente il driver da utilizzare per accedere al database SQLite e il nome del file che contiene il database.
- Creazione ed esecuzione di uno statement SQL, solitamente un *PreparedStatement*.

```
public void insertUserIntoDatabase(User user) {
    try {
        connection = DriverManager
                     .getConnection(protocol + databaseName);
        String sql = "INSERT INTO users (name, surname, wins)"
                + " VALUES(?, ?, ?)";
        PreparedStatement statement = connection.prepareStatement(sql);
        statement.setString(1, user.getName());
        statement.setString(2, user.getSurname());
        statement.setInt(3, user.getNumberOfWins());
        statement.execute();
    } catch (SQLException e) {
        System.out.println(e.getMessage());
    } finally {
        closeConnection(connection);
    }
}
```

Figure 10: Metodo utilizzato per inserire un oggetto della classe Utente all'interno del database

- Chiusura della connessione, effettuata in questo caso attraverso il metodo ausiliario *closeConnection* che si occupa di effettuare il *try/catch* sull'operazione di close, utilizzato principalmente per leggibilità.

Infine, la classe EmptyUser serve per gestire il caso in cui un utente sia inesistente nel database, per evitare di ritornare *null*[2] dalla funzione che converte un utente nel database in un oggetto *User*.

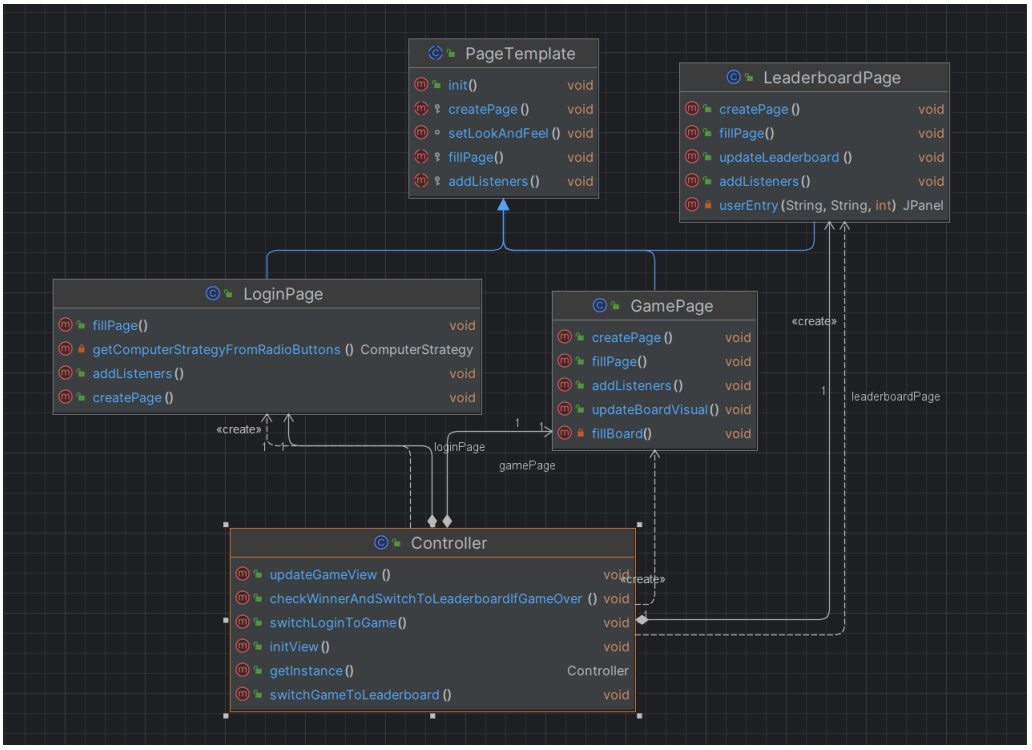

Figure 11: Diagramma UML delle classi Controller e PageTemplate

7 Interfaccia Grafica

7.1 Template Method

L'applicazione è stata sviluppata come programma standalone con supporto grafico utilizzando la libreria di Java **Swing**.
Ogni pagina dell'applicazione è creata utilizzando la classe astratta *PageTemplate*, che implementa il Design Pattern **Template Method**[1].
L'obiettivo di PageTemplate è quello di creare un unico metodo per inizializzare le pagine, in modo che la creazione delle pagine sia analoga.
Il metodo di template *init()* implementa la creazione delle pagine come segue:

- Il metodo astratto *createPage()* che include i metodi di Swing per la creazione della pagina.
- Il metodo **final** *setLookAndFeel()* che si occupa di impostare il look and feel di ogni pagina allo stesso modo, non deve quindi poter essere possibile per le sottoclassi fare un @Override di questo metodo.
- Il metodo *fillPage()*, che è un metodo di **hook**, ovvero non tutte

le classi sono obbligate a definirlo, e ha lo scopo di riempire la pagina con elementi dinamici, generati cioè in fase di esecuzione.

- Il metodo *addListeners()* dove le pagine possono andare a definire le operazioni da assegnare ai propri bottoni.

In questo caso il pattern Template Method è stato utilizzato con lo scopo di generalizzare il comportamento comune tra le classi rappresentanti le pagine dell'applicazione, e per controllare l'estensione delle sottoclassi attraverso il metodo di hook *fillPage()*.

```
public abstract class PageTemplate extends JFrame {
    public final void init() {
        createPage();
        fillPage();
        setLookAndFeel();
        addListeners();
        setLocationRelativeTo(null);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setResizable(false);
    }

    protected abstract void createPage();

    // metodo di hook
    protected void fillPage() {
    }

    // metodo final:
    // tutte le pagine devono avere lo stesso look and feel
    final void setLookAndFeel() {
        try {
            UIManager.setLookAndFeel(
                    "com.jtattoo.plaf.hifi.HiFiLookAndFeel
                                        ");
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }

    protected abstract void addListeners();
}
```

Figure 12: Implementazione di Template Method attraverso la classe astratta PageTemplate

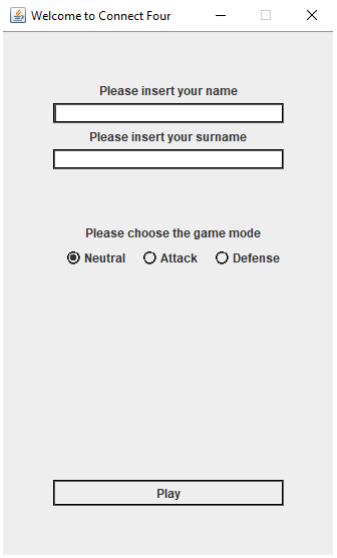

Figure 13: Pagina di Login

7.2 Controller

La classe *Controller* si occupa invece di gestire il flusso di operazioni relative all'interfaccia grafica.
Controller inizializza le diverse pagine che compongono l'applicazione richiamando i loro metodi template, dopodiché espone dei metodi per cambiare la pagina corrente, associando al cambiamento di pagine dei comportamento opportuni.
In questo modo Controller non espone le istanze concrete delle singole pagine, e pertanto permette di disaccoppiare la gestione della GUI e le transizioni tra le pagine dalla logica dell'applicazione.
Controller è implementato come **Singleton**[1], in quanto non avrebbe senso avere più gruppi di pagine attive, e deve essere facilmente accessibile dalle altre classi, per esempio dal GameHandler quando diventa necessario aggiornare la vista della griglia attraverso il metodo *updateGameView*.

7.3 Login Page

La pagina di Login permette al giocatore di inserire il proprio nome e cognome e di scegliere la modalità di gioco.
Si occupa di controllare che sia il campo nome che il campo cognome non siano vuoti, dopodiché quando l'utente preme il tasto *Play* invia

questi dati al GameHandler e comunica al Controller di cambiare pagina.

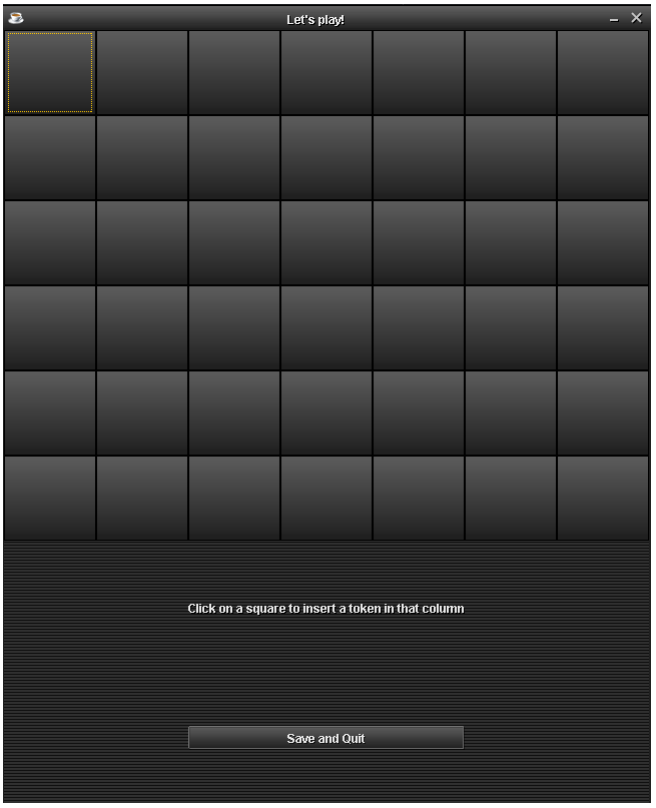

Figure 14: Pagina di gioco

7.4 Game Page

La classe GamePage gestisce la pagina di gioco, la griglia viene visualizzata come una griglia di bottoni delle stesse dimensioni.
Alla pressione di un bottone da parte dell'utente viene inserita una pedina in quella colonna, se vi sono quattro pedine in fila, viene richiamato il metodo apposito del Controller per cambiare pagina alla Leaderboard Page
In questa pagina è anche possibile utilizzare il bottone *Save and Quit* per salvare la partita, richiamando il metodo opportuno del GameHandler, che delega la codifica della partita in un file di testo al GameLoader.

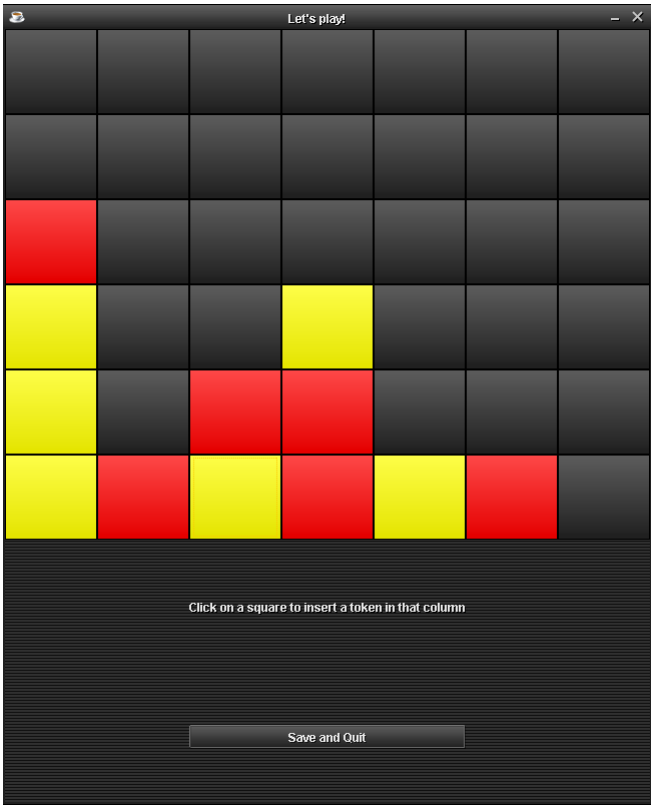

Figure 15: Pagina di gioco dopo aver inserito delle pedine

7.5 Leaderboard Page

Al termine della partita, la LeaderBoard page utilizza un metodo della classe Database per ottenere i primi dieci utenti nel database con il più alto numero di vittorie, e aggiorna opportunamente il contenuto della pagina.

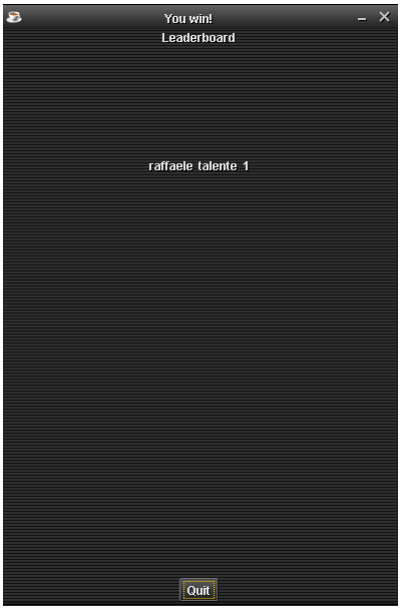

Figure 16: Pagina di Leaderboard

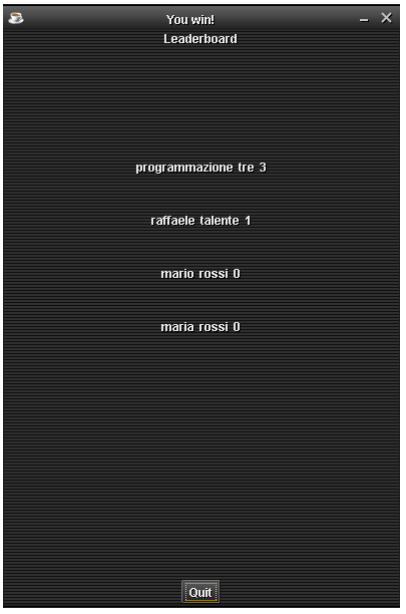

Figure 17: Pagina di Leaderboard con diversi utenti

8 Conclusioni

In questo progetto è stato sviluppato un programma in grado di simulare il gioco da tavolo *Forza quattro*.
L'utilizzo del Design Pattern **Strategy** ha permesso di disaccoppiare la logica centrale del funzionamento di una partita dalla modalità di gioco che definisce il comportamento del computer.
Grazie a questo pattern in futuro sarà possibile aggiungere nuove modalità di gioco in maniera estremamente facile, semplicemente implementando l'interfaccia *ComputerStrategy*, e senza dover modificare le modalità esistenti in alcun modo.
L'utilizzo del Design Pattern **Template Method** ha permesso di definire un algoritmo per standaridzzare le pagine dell'interfaccia grafica, grazie a questo pattern il programma può facilmente supportare nuove pagine della GUI senza dover modificare le pagine esistenti, inoltre se si dovesse decidere di cambiare il look and feel, basterebbe modificare soltanto il metodo all'interno della classe *PageTemplate* piuttosto che in ogni singola pagina.
L'utilizzo del Design Pattern **Singleton** ha permesso di definire un'unica istanza delle classi GameHandler e Controller, fornendo un punto di accesso globale ad esse.
Grazie a questo pattern diventa impossibile per altre classi istanziare inavvertitamente un secondo GameHandler o Controller, cosa che porterebbe soltanto ad errori.

References

[1] GoF (1994) *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley.

[2] Robert C. Martin (2009) *Clean Code: A Handbook of Agile Software Craftsmanship*, Prentice Hall.

[3] Wikipedia: *Alpha-beta pruning*
https://en.wikipedia.org/wiki/Alpha-beta_pruning