Relazione progetto Ingegneria del Software: applicazione per giocare a scacchi.

Bernini Sara, Senoner Jay

Anno Accademico
2022-2023

Indice

1 Introduzione

1.1 Introduzione al problema

L'obiettivo del progetto è la realizzazione di un applicativo in linguaggio Java che permetta a due utenti di giocare una partita a scacchi in locale, e scaricare un file di testo contenente le mosse effettuate durante la partita espresse in formato PGN ("Portable Game Notation"). L'utente potrà interagire con il sistema mediante l'uso di una semplice GUI contenente una barra delle azioni e una scacchiera, rappresentata come una matrice di bottoni contenenti le immagini dei pezzi del gioco degli scacchi. Quando l'utente sceglie di iniziare una nuova partita, verrà mostrata a video tramite la GUI una scacchiera contenete i pezzi collocati nella posizione iniziale. Per effettuare una mossa all'interno della partita l'utente dovrà cliccare sul pezzo che desidera muovere: facendo ciò le caselle della scacchiera dove è possibile spostare il pezzo selezionato diventeranno grigie. Una volta selezionata una casella grigia, allora verrà eseguita la mossa, e il pezzo verrà spostato nella nuova posizione. Il software che calcola le possibili mosse che può compiere un pezzo, tiene in considerazione tra le mosse che può eseguire un pezzo in base alla sua tipologia e alle regole degli scacchi, le mosse effettive che può eseguire evitando di porre il proprio re sotto scacco. Il software inoltre è in grado di rilevare quando il re si trova in stato di scacco e limitare le mosse dei vari pezzi alle sole che impediscano questo stato. Se tali mosse non esistono, quindi lo scacco è inevitabile, allora passiamo nello stato di scacco matto e la partita termina con la sconfitta del giocatore che lo ha subito. In qualsiasi momento della partita l'utente avrà la possibilità di scaricare il file contenete il pgn premendo il bottone con etichetta "Download PGN".

1.2 Elenco delle funzionalità richieste

Qui riassumiamo le funzionalità che il software dovrà realizzare:

- **Iniziare una nuova partita**: L'utente può iniziare una nuova partita premendo sul bottone "New game" all'interno della barra delle azioni. Così facendo la scacchiera verrà ripristinata allo stato iniziale, e sarà possibile giocare una nuova partita.

- **Scaricare il file PGN**: L'utente può scaricare un file contenente le mosse della partita descritte in formato PGN premendo sul bottone "Download PGN" all'interno della barra delle azioni.

- **Visualizzare le mosse legali**: L'utente può visualizzare le mosse legali(secondo le regole del gioco degli scacchi) di un determinato pezzo premendo sull'icona del pezzo stesso. Un utente può visualizzare le mosse legali dei soli pezzi del giocatore di turno. Le case che rappresentano una valida destinazione per il pezzo selezionato verranno evidenziate di grigio scuro.

- **Effettuare una mossa**: L'utente può effettuare una mossa all'interno della partita premendo su una casa che è precedentemente stata evidenziata in grigio scuro, ovvero una casa che rappresenta una destinazione valida per il pezzo precedentemente selezionato.

2 Diagrammi UML

2.1 Diagramma dei casi d'uso

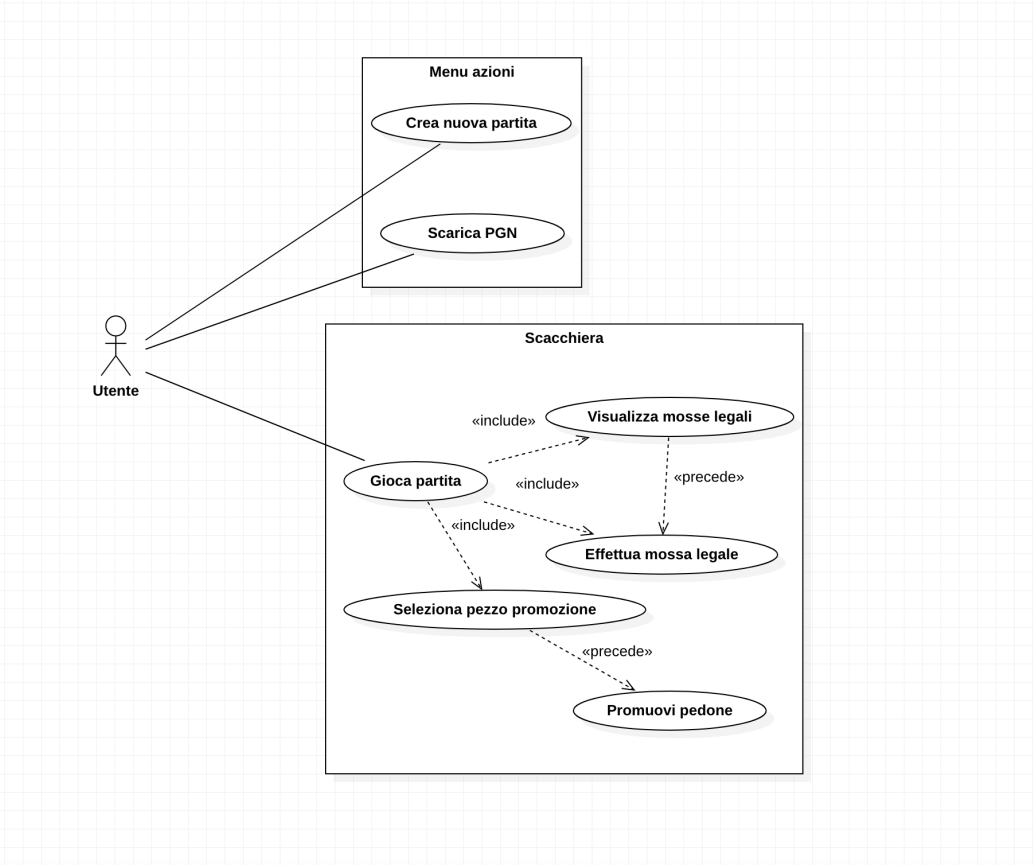

Figura 1: Diagramma dei casi d'uso

2.2 Diagramma delle classi

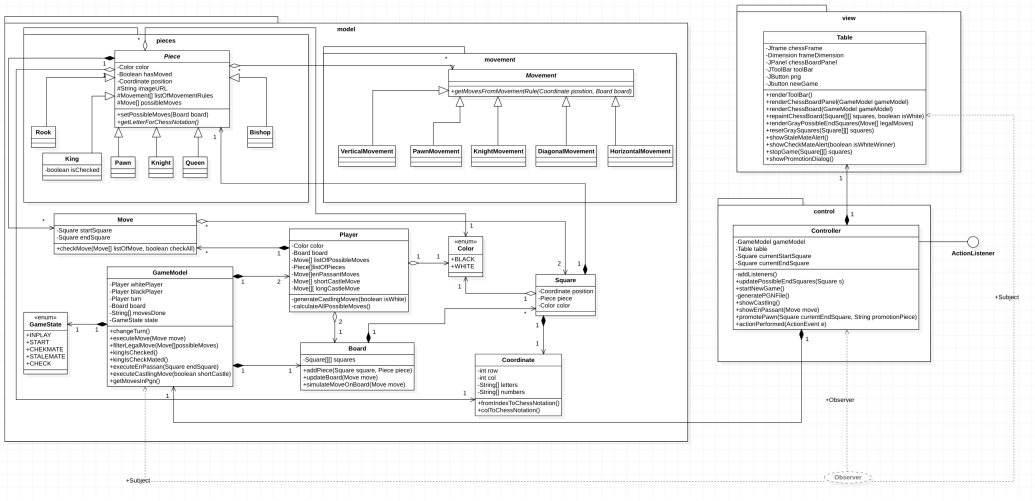

Figura 2: Diagramma delle classi

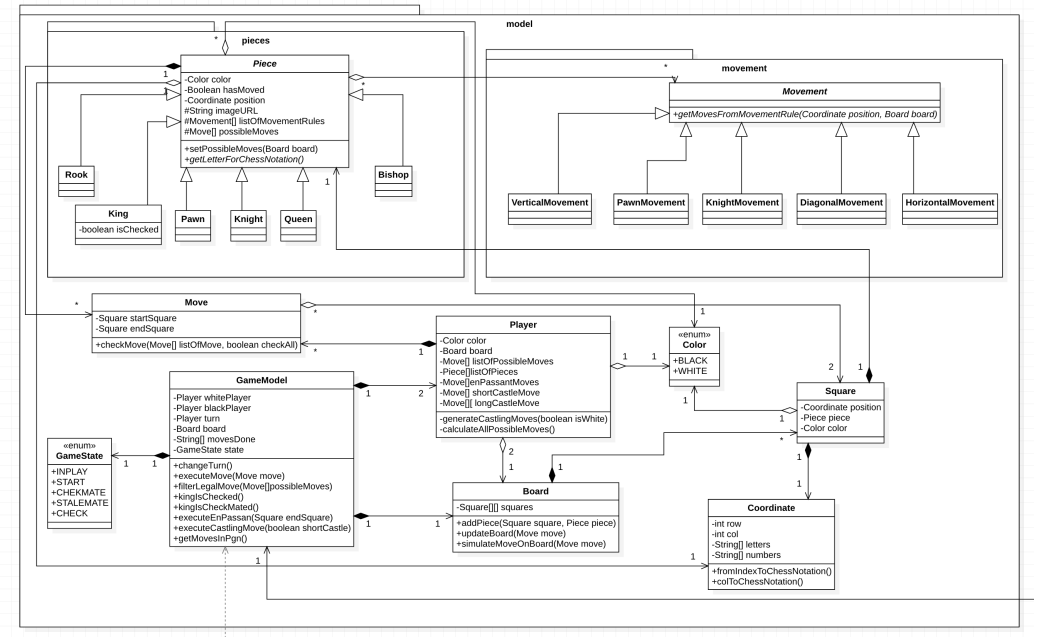

Figura 3: Package Model

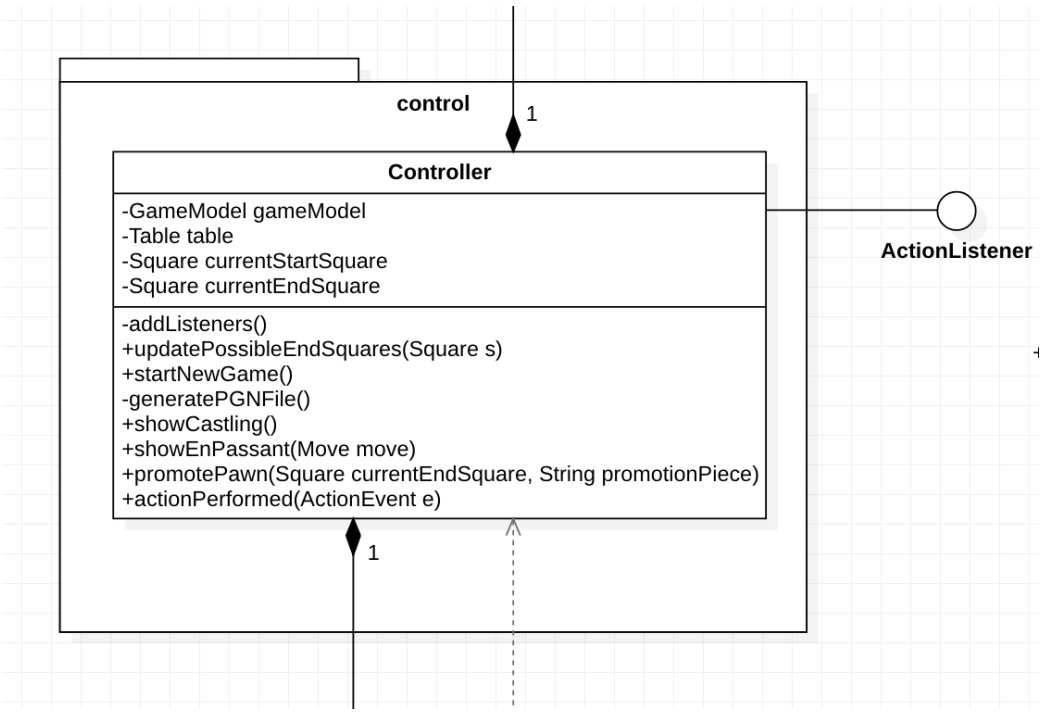

Figura 4: Package Control

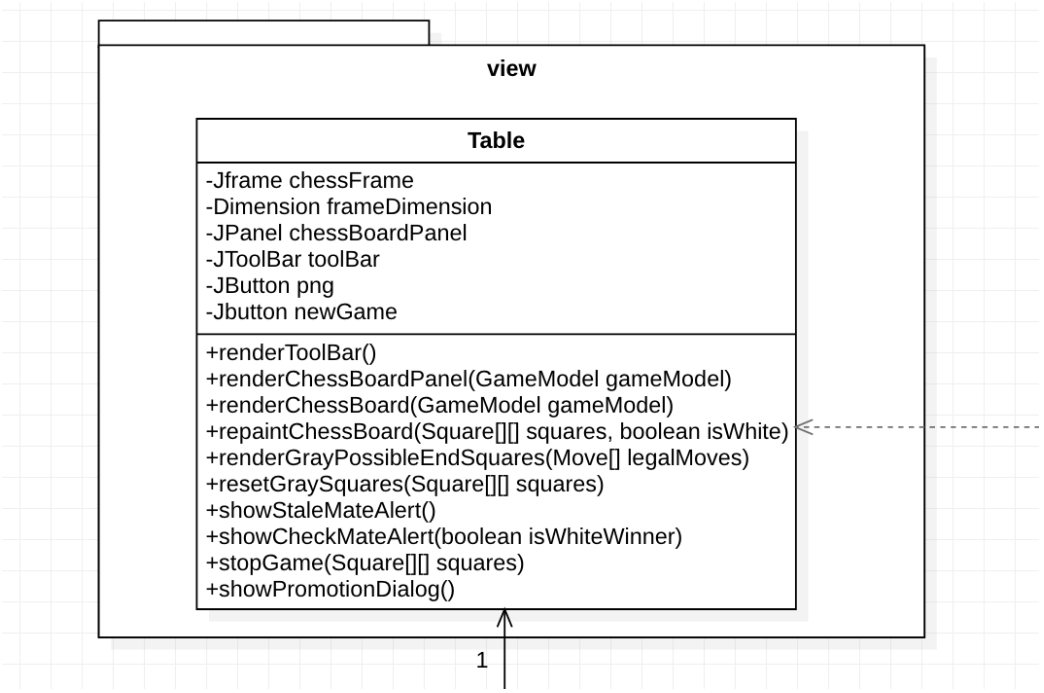

Figura 5: Package View

2.3 Diagrammi di sequenza

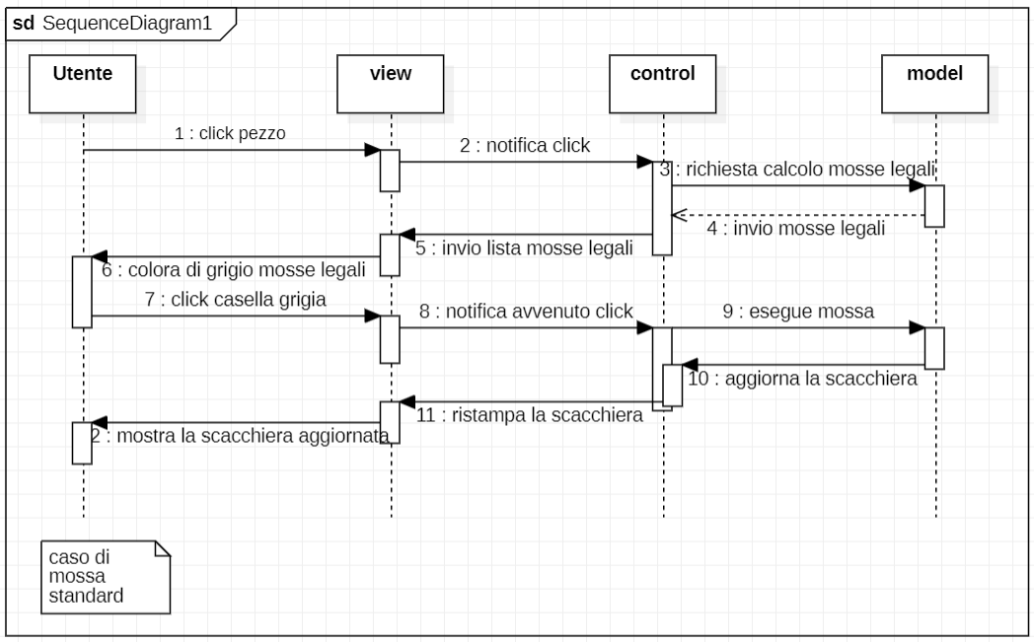

Figura 6: Mossa standard

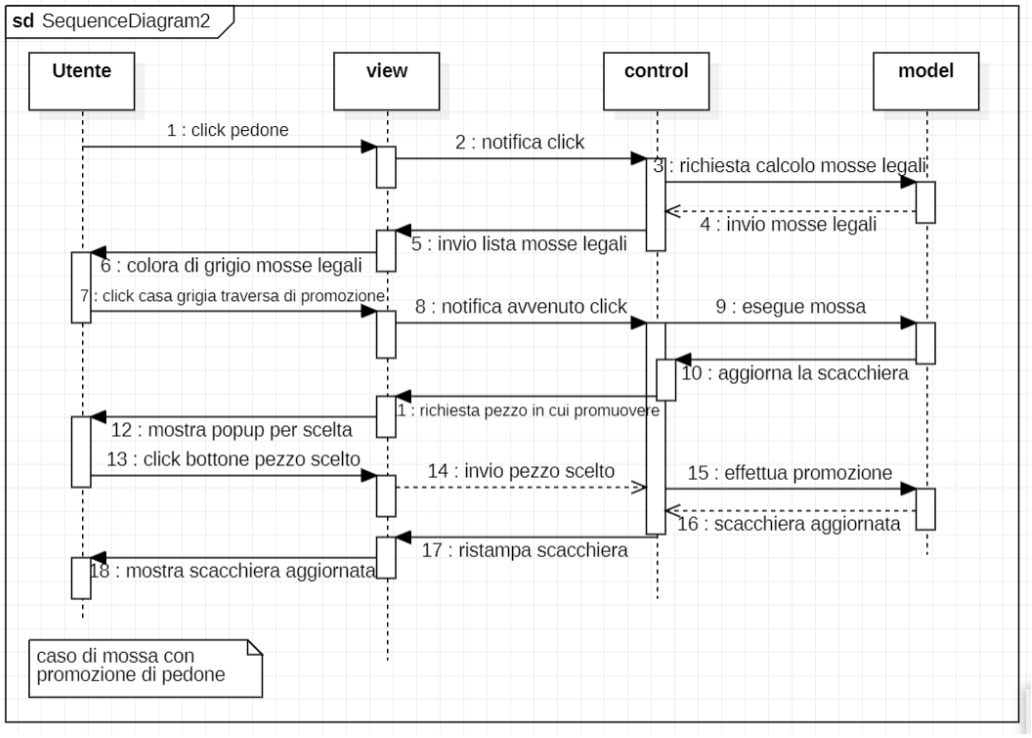

Figura 7: Promozione

3 Analisi e descrizione dell'implementazione

3.1 Scelta del pattern architetturale

In fase di progettazione dell'applicativo è stato scelto come pattern architetturale il pattern **MVC**(**M**odel **V**iew **C**ontroller), che permette di disaccoppiare totalmente la rappresentazione del modello di dominio (**Model**) dall'interfaccia utente(**View**) mediante un modulo di controllo delle interazioni(**Controller**). Nel caso pratico dell'implementazione dell'applicativo descritto in sezione 1.1 si ha che:

- Il modulo **Model** contiene le classi atte alla rappresentazione di una partita a scacchi tra due giocatori: conterrà quindi classi che rappresentano la scacchiera, i giocatori, le mosse, lo stato di gioco, i pezzi e le loro specifiche regole di movimento

- Il modulo **View** contiene le classi necessarie alla visualizzazione su video della scacchiera e all'interazione con essa.
- Il modulo **Control** contiene le classi il cui scopo è realizzare l'interazione tra i moduli Model e View: in particolare si occupa di interpretare gli input dell'utente sulla View e associarli a modifiche specifiche del Model, e di tradurre le eventuali modifiche del Model in cambiamenti grafici nella View

3.2 Model

Il package Model è il package contenente l'insieme di tutte le classi utili a rappresentare la logica dello svolgimento di una partita di scacchi convenzionale. Poichè la logica dello svolgimento di una partita di scacchi risulta essere un dominio abbastanza complesso, è stato scelto di dividere le classi all'interno del Model in diversi package che ne riunissero in qualche modo le funzionalità principali. Vediamo adesso in dettaglio il contenuto di tali packages:

3.2.1 Movement

Il package Movement contiene le classi il cui obiettivo è la rappresentazione e l'implementazione delle diverse regole di movimento che i vari tipi di pezzi presenti sulla scacchiera devono rispettare. Complessivamente, il package Movement contiene le classi:

- **Movement**
- **VerticalMovement**
- **HorizontalMovement**
- **DiagonalMovement**
- **KnightMovement**
- **PawnMovement**

Le classi all'interno del package Movement non si occupano di determinare se una mossa è **legale**, ovvero se una determinata mossa non fa si che il re del giocatore che sta muovendo sia attaccato dall'avversario(questo

verrà controllato nella classe GameModel), ma controllano la **possibilità** delle mosse, cioè si focalizzano esclusivamente sul rappresentare i movimenti possibili di un determinato pezzo in una determinata posizione in base alle regole di gioco del pezzo che sto muovendo. In particolare il package Movement contiene una classe astratta **Movement** che ospita un metodo astratto *getMoveFromMovementRule()* la cui firma è presentata nel listato 1 a riga 7.

Listing 1: Classe Movement

```
1
2  package com.company.model.movement;
3  ...
4
5  public abstract class Movement {
6
7      public abstract ArrayList<Move>
           getMovesFromMovementRule(Coordinate startPosition,
           Board chessBoard);
8  }
```

Tale metodo verrà poi sovraccaricato dalle classi concrete che implementano le regole di movimento specifiche, e lo scopo nella specifica classe sarà il determinare l'insieme delle mosse possibili (rispetto al tipo di movimento rappresentato dalla classe concreta) di un pezzo in una determinata coordinata, data la posizione della scacchiera. Nel listato 2 viene mostrato come la classe VerticalMovement effettua il sovraccaricamento del metodo astratto della classe Movement per dedurre tutte le possibili mosse verticali di un dato pezzo in una determinata posizione.

Listing 2: Classe VerticalMovement

```
10 public class VerticalMovement extends Movement {
11     public boolean checkAll;
12     public VerticalMovement(boolean checkAll){
13         this.checkAll = checkAll;
14     }
15     @Override
16     public ArrayList<Move> getMovesFromMovementRule(
           Coordinate startPosition, Board chessBoard) {
17         ArrayList<Move> listOfMove = new ArrayList<>();
18         Square startSquare = chessBoard.getSquare(
               startPosition);
```

```
19
20          boolean stop = false;
21          int i = startPosition.getRow();
22          int col = startPosition.getCol();
23          while (!stop) {
24              i++;
25              Move move = new Move(startSquare, chessBoard.
                    getSquare(i, col));
26              stop = move.checkMove(listOfMove, checkAll);
27          }
28          stop = false;
29          i = startPosition.getRow();
30          while (!stop) {
31              i--;
32              Move move = new Move(startSquare, chessBoard.
                    getSquare(i, col));
33              stop = move.checkMove(listOfMove, checkAll);
34          }
35          return listOfMove;
36      }
37  }
```

3.2.2 Pieces

Il package Pieces contiene le classi il cui obiettivo è rappresentare i diversi tipi di pezzi del gioco degli scacchi. Complessivamente il package Pieces contiene le seguenti classi:

- **Piece**
- **Bishop**
- **Queen**
- **Knight**
- **Rook**
- **King**
- **Pawn**

All'interno di questo package, la classe astratta Piece rappresenta le caratteristiche di un generico pezzo: in particolare per rendere più semplice il calcolo di tutte le mosse possibili di un generico pezzo in una determinata posizione, viene specificato un attributo *listOfMovementRules*, che contiene la lista di tutte le regole di movimento del pezzo in questione. Per ottenere le mosse possibili in una determinata posizione, è presente un metodo(presentato nella riga 43 del listato 3) che itera su tutte le regole di movimento e aggiunge ad un attributo *listOfPossibleMoves* tutte le mosse possibili dedotte dalla specifica regola di movimento, considerando la posizione relativa del pezzo rispetto agli altri pezzi sulla scacchiera. Il metodo astratto della classe Piece, mostrato a riga 49 del listato 3, ha la funzione di ottenere la lettera corretta associata ad una specifica tipologia di pezzo nella notazione algebrica utilizzata per rappresentare le mosse nel formato PGN

Listing 3: Frammento classe Piece

```
38 public abstract class Piece implements Cloneable {
39 ...
40 protected ArrayList<Movement> listOfMovementRules;
41 protected ArrayList<Move> possibleMoves;
42 ...
43 public void setPossibleMoves(final Board board){
44         Coordinate startPosition = getPosition();
45         possibleMoves.clear();
46         for(Movement movementRule: listOfMovementRules)
47             possibleMoves.addAll(movementRule.
                   getMovesFromMovementRule(startPosition,
                   board));
48     }
49 public abstract String getRightLetterForChessNotation();
50
51
52 }
```

Di seguito viene mostrato un esempio di una classe concreta che estende la classe astratta Piece, e di conseguenza implementa il metodo a riga 49 del listato 3 per il pezzo in questione. Inoltre dall'esempio mostrato nel listato 4 si può vedere come l'estensione concreta della classe astratta Piece attribuisca al pezzo specifico le regole di movimento associate alla determinata tipologia di pezzo all'interno del costruttore.

Listing 4: Frammento classe Queen

```
54 public class Queen extends Piece{
55     public Queen(Color color) {
56         super(color);
57         if( color == Color.WHITE)
58             imageURL="image/WQ.gif";
59         else
60             imageURL="image/BQ.gif";
61         listOfMovementRules.add(new DiagonalMovement(true))
               ;
62         listOfMovementRules.add(new HorizontalMovement(true
               ));
63         listOfMovementRules.add(new VerticalMovement(true))
               ;
64     }
65     ...
66     @Override
67     public String getRightLetterForChessNotation() {
68         return "Q";
69     }
70 }
```

3.2.3 Classe GameModel

All'interno del package Model si trovano una serie di classi che non sono contenute in ulteriori package, e che risultano fondamentali per la corretta rappresentazione di una partita di scacchi. Di seguito viene data una breve descrizione delle funzionalità che tali classi offrono :

- **Coordinate**: Classe il cui scopo è rappresentare una valida coordinata bidimensionale all'interno della scacchiera, contiene metodi per tradurre la coordinata numerica nella sua corretta notazione algebrica
- **Square**:Classe che implementa una casa della scacchiera, rappresentata da una coordinata valida e un eventuale pezzo contenuto, fornisce metodi per impostare il pezzo contenuto
- **Board**: Classe che rappresenta la scacchiera come matrice 8x8 di oggetti Square. Nel costruttore viene inizializzata la scacchiera in posizione

iniziale, con i pezzi nelle corrette posizioni. Fornisce metodi per aggiornare la scacchiera dopo che una mossa è stata compiuta, e metodi per simulare una mossa(tali metodi risultano molto utili per il calcolo delle mosse illegali)

- **Move**: Classe che rappresenta una mossa all'interno di una partita mediante due oggetti Square(lo Square iniziale e lo Square finale). Fornisce metodi per controllare che una mossa sia possibile e per tradurre una mossa in notazione algebrica

- **Player**: Classe che rappresenta un giocatore all'interno di una partita. Contiene membri che danno informazioni sulla posizione, sui pezzi posseduti dal giocatore e sull'insieme delle sue mosse possibili,dati i suoi pezzi. Fornisce metodi per generare le mosse dell'arrocco(viste come mosse speciali, dato che compongono due mosse in una) e calcolare l'insieme delle mosse possibili in un dato momento della partita(Non si verifica la legalità di tali mosse)

Di seguito viene mostrato un frammento di codice della classe Move, che implementa il metodo per tradurre una mossa in notazione algebrica, necessario per poter offrire all'utente la possibilità di ottenere il file in formato PGN della partita giocata.

Listing 5: Frammento classe Move

```
72 public class Move {
73     private final Square startSquare;
74     private Square endSquare;
75     ...
76     //Metodo che data una mossa legale(quindi gia
          controllata a priori) ne determina l'espressione in
          notazione algebrica
77     // DA CHIAMARE PRIMA CHE VENGA FATTO L'UPDATE DELLA
          SCACCHIERA, altrimenti non torna.
78     // Per torre e cavallo scrive anche la colonna di
          appartenenza del pezzo, al fine di evitare ambiguita
           sulla mossa effettuata
79     public String getMoveInChessNotation(){
80
81         String moveInChessNotation;
```

```
82          String letter = startSquare.getPiece().
                getRightLetterForChessNotation();
83
84            if( letter.equals("R") || letter.equals("N"))
85                moveInChessNotation = letter +
86                        startSquare.getPosition().
                            colToChessNotation();
87            else moveInChessNotation = letter;
88            if(endSquare.isOccupied()) {
89                if(letter.equals(""))
90                    moveInChessNotation = letter +
91                            startSquare.getPosition().
                                colToChessNotation();
92                moveInChessNotation = moveInChessNotation.
                    concat("x" + endSquare.getPosition().
                    fromIndexToChessNotation());
93            }
94            else moveInChessNotation = moveInChessNotation.
                    concat(endSquare.getPosition().
                    fromIndexToChessNotation());
95          return moveInChessNotation;
96      }
97  }
```

Tutte le funzionalità che tali classi offrono vengono utilizzate all'interno della classe **GameModel**, il cui scopo ultimo è quello di rappresentare un intera partita di scacchi. Difatti, la classe **GameModel** contiene come membri:

- Due oggetti Player *whitePlayer* e *blackPlayer*, necessari per rappresentare il bianco e il nero
- Un oggetto Player *turn*, che rappresenta il giocatore di turno
- Un oggetto Board *board*, che rappresenta la scacchiera sulla quale i Player svolgono la loro partita
- Una lista di stringhe *movesDone*, dove durante il corso della partita vengono memorizzate, in notazione algebrica, le mosse effettuate dai Player.

- Un oggetto GameState *state*, necessario per rappresentare lo stato corrente della partita(GameState è un enumerazione definita all'interno della classe GameModel, contentente gli stati *START*, *INPLAY*, *CHECK*, *CHECKMATE*, *STALEMATE*)

Per quanto riguarda i metodi, la classe **GameModel** offre metodi necessari allo svolgimento della partita, alla determinazione dello stato corrente e al filtraggio delle mosse legali dall'insieme delle mosse possibili di un determinato giocatore. Ricordiamo infatti che, nel gioco degli scacchi, le mosse legali sono un sottoinsieme proprio delle mosse possibili, e sono tutte e sole le mosse possibili che non pongono il proprio re sotto scacco, o che impediscono che lo scacco al re sia presente anche alla mossa successiva . Nel listato 6 viene mostrato un frammento di codice della classe GameModel contenente i metodi necessari a determinare le mosse legali dato un insieme di mosse possibili, e per la scrittura corretta del file in formato PGN. In aggiunta ai metodi presentati nel listato 6 sono presenti anche i metodi per eseguire mosse legali,cambio del turno e metodi per la determinazione dello stallo e dello scacco matto. Inoltre è stato scelto di codificare nel gameModel le mosse "speciali", ovvero mosse che esulano dai movimenti standard dei pezzi, ovvero le mosse dell'arrocco lungo/corto e la cattura en passant.

Listing 6: Frammento classe GameModel

```
99  enum GameState{START, INPLAY, CHECK, CHECKMATE, STALEMATE}
100 public class GameModel{
101     private final Player whitePlayer;
102     private final Player blackPlayer;
103     private Player turn;
104     private final Board board;
105     private ArrayList<String> movesDone;
106     private GameState state;
107     ...
108     public ArrayList<Move> filterLegalMoves(ArrayList<Move>
            possibleMoves){
109         ArrayList<Move> illegalMovement= new ArrayList<>();
110         for(Move move: possibleMoves) {
111             //try movement
112             Piece pieceToReinsert = this.getBoard().
                   simulateMoveOnBoard(move);
113             if(pieceToReinsert != null){
```

```
114                 if(pieceToReinsert.getColor()==Color.WHITE)
                        {
115                     whitePlayer.getListOfPieces().remove(
                            pieceToReinsert);
116                 }
117                 else{
118                     blackPlayer.getListOfPieces().remove(
                            pieceToReinsert);
119                 }
120             }
121             if(kingIsChecked()){
122                 illegalMovement.add(move);
123             }
124
125             if(pieceToReinsert != null){
126                 if(pieceToReinsert.getColor()==Color.WHITE)
                        {
127                     whitePlayer.getListOfPieces().add(
                            pieceToReinsert);
128                 }
129                 else{
130                     blackPlayer.getListOfPieces().add(
                            pieceToReinsert);
131                 }
132             }
133             this.getBoard().reverseSimulatedMove(move,
                    pieceToReinsert);
134         }
135         for(Move m : illegalMovement) {
136             possibleMoves.remove(m);
137         }
138         return possibleMoves;
139     }
140
141     //Calcolo tutte le mosse possibili del player
            avversario per determinare se esiste una mossa che
            attacca direttamente il re del player corrente
142     public boolean kingIsChecked(){
143         if (this.turn.isWhite()) {
144             this.blackPlayer.calculateAllPossibleMoves();
```

```
145             for (Move move : this.blackPlayer.
                    getListOfPossibleMoves()) {
146                 if (move.getEndSquare().getPiece() != null
                        && move.getEndSquare().getPiece().
                        getClass() == King.class) {
147                     state = GameState.CHECK;
148                     return true;
149                 }
150             }
151         }else{
152             this.whitePlayer.calculateAllPossibleMoves();
153             for (Move move : this.getWhitePlayer().
                    getListOfPossibleMoves()) {
154                 if(move.getEndSquare().getPiece() != null)
155                 if (move.getEndSquare().getPiece().getClass
                        () == King.class) {
156                     state = GameState.CHECK;
157                     return true;
158                 }
159             }
160         }
161         state= GameState.INPLAY;
162         return false;
163     }
164
165     public ArrayList<String> getMovesInPgn(){
166         ArrayList<String> pgn = new ArrayList<>();
167         int moveNumber = 1;
168         if(movesDone.size()%2 ==0)
169         for(int i = 0; i < movesDone.size(); i+=2){
170             pgn.add(moveNumber + ". " + movesDone.get(i) +
                    " " +movesDone.get(i+1));
171             moveNumber++;
172         }
173         else{
174             for(int i = 0; i < (movesDone.size()-1); i+=2){
175                 pgn.add(moveNumber + ". " + movesDone.get(i
                        ) + " " +movesDone.get(i+1));
176                 moveNumber++;
177             }
```

```
178                 pgn.add( (moveNumber) + ". "+ movesDone.get(
                        movesDone.size()-1)+ " ");
179             }
180             return pgn;
181         }
182 }
```

3.3 View

Il package view contiene un unica classe, denominata **Table**, il cui scopo è fornire un interfaccia grafica che permetta all'utente di visualizzare la posizione della scacchiera e interagire con essa, andando quindi a modificare la posizione effettuando mosse all'interno della partita. In ogni momento della partita l'utente, attraverso i tasti presenti nella ToolBar, etichettati "New Game" e "Download PGN", può decidere di iniziare una nuova partita o scaricare il file in formato PGN(Portable Game Notation) della partita corrente. L'interazione con l'utente al fine di effettuare una mossa è costituita dai seguenti passaggi:

1. L'utente effettua un click sul pezzo che intende muovere, selezionandolo.
2. Viene chiamato un metodo della classe Table che colora di grigio scuro tutte e sole le case contenenti destinazioni legali per il pezzo precedentemente selezionato
3. L'utente sceglie tra le opzioni proposte la casa dove intende muovere il pezzo selezionato

O, in alternativa:

3 L'utente seleziona un altro pezzo che intende muovere, poi torna al passo 2.

Inoltre, per le regole degli scacchi quando un pedone bianco(nero) raggiunge la ottava(prima) traversa, tale pedone guadagna il diritto di promozione, ovvero può trasformarsi in un pezzo qualsiasi tra Donna, Alfiere, Torre e Cavallo. Per gestire la scelta, la View predispone un metodo per invocare una finestra nella quale è possibile scegliere in quale pezzo si desidera promuovere il pedone. Infine, quando la partita si conclude per scacco matto o per stallo, viene chiamato un metodo della View che mostra a video un Alert contenente

un messaggio di conclusione della partita. Nel listato 7 viene mostrata la parte della classe Table che implementa i metodi descritti. In aggiunta ai metodi presentati, sono anche presenti i metodi per effettuare il rendering della scacchiera in posizione iniziale e dei vari componenti del *chessFrame*

Listing 7: Frammento classe Table

```
183 public class Table {
184     private final JFrame chessFrame;
185     private final Dimension frameDimension = new Dimension
            (1000, 1000);
186     private static final String COLS = "ABCDEFGH";
187     private JPanel chessBoardPanel;
188     private JToolBar toolBar;
189     private final JButton pgn;
190     private final JButton newGame;
191     ...
192     public void repaintChessBoard(Square[][] squares,
            Boolean isWhite){
193          for(Square[] ss : squares){
194              for(Square s: ss){
195                  String image = "";
196                  if(s.getPiece() != null) {
197                      image = s.getPiece().getImageURL();
198                      s.setEnabled((!isWhite || s.getPiece().
                             getColor() != com.company.model.
                             Color.BLACK) &&
199                                   (isWhite || s.getPiece().
                                          getColor() != com.company.
                                          model.Color.WHITE));
200                      s.setDisabledIcon(new ImageIcon(image))
                             ; // Aggiunto per far si che l'icona
                              rimanga invariata disabilitando/
                             abilitando un bottone contenente un
                             pezzo
201                  }else{
202                      s.setDisabledIcon(null);    //Se una
                             casa non ha pezzo imposto la sua
                             icona da disabilitato come null
203                      s.setEnabled(false);
204
```

```
205                 }
206                 s.setIcon(new ImageIcon(image));
207             }
208         }
209     }
210
211     public void renderGrayPossibleEndSquares(ArrayList<Move
        > legalMoves) {
212             for(Move move : legalMoves){
213                 move.getEndSquare().setBackground(Color.
                        DARK_GRAY);
214                 move.getEndSquare().setEnabled(true);
215             }
216     }
217     public void resetGraySquares(Square[][]
        chessBoardSquares){
218         for(Square[] ss : chessBoardSquares) {
219             for (Square s : ss) {
220                 if (s.getBackground() == Color.DARK_GRAY) {
221                     if (s.getColor() == com.company.model.
                            Color.WHITE) {
222                         s.setBackground(Color.WHITE);
223                     } else {
224                         s.setBackground(Color.BLACK);
225                     }
226                 }
227             }
228         }
229     }
230
231     public void showStaleMateAlert() {
232         JOptionPane.showMessageDialog(chessFrame, "
                STALEMATE!");
233     }
234     public void showCheckMateAlert(Boolean isWhiteWinner) {
235         String text= "CHECKMATE!";
236         if(isWhiteWinner){
237             text= text + " White player wins!";
238         }
239         else{
```

```
240             text = text + " Black player wins!";
241         }
242         JOptionPane.showMessageDialog(chessFrame, text);
243     }
244 
245     public String showPromotionDialog() {
246         String[] options = {"Queen", "Knight", "Bishop", "
                Rook"};
247         int x = JOptionPane.showOptionDialog(null, "Select
                the piece you want to promote into",
248                 "Promotion!",
249                 JOptionPane.DEFAULT_OPTION, JOptionPane.
                    INFORMATION_MESSAGE, null, options,
                    options[0]);
250         return options[x];
251     }
```

3.3.1 Presentazione GUI

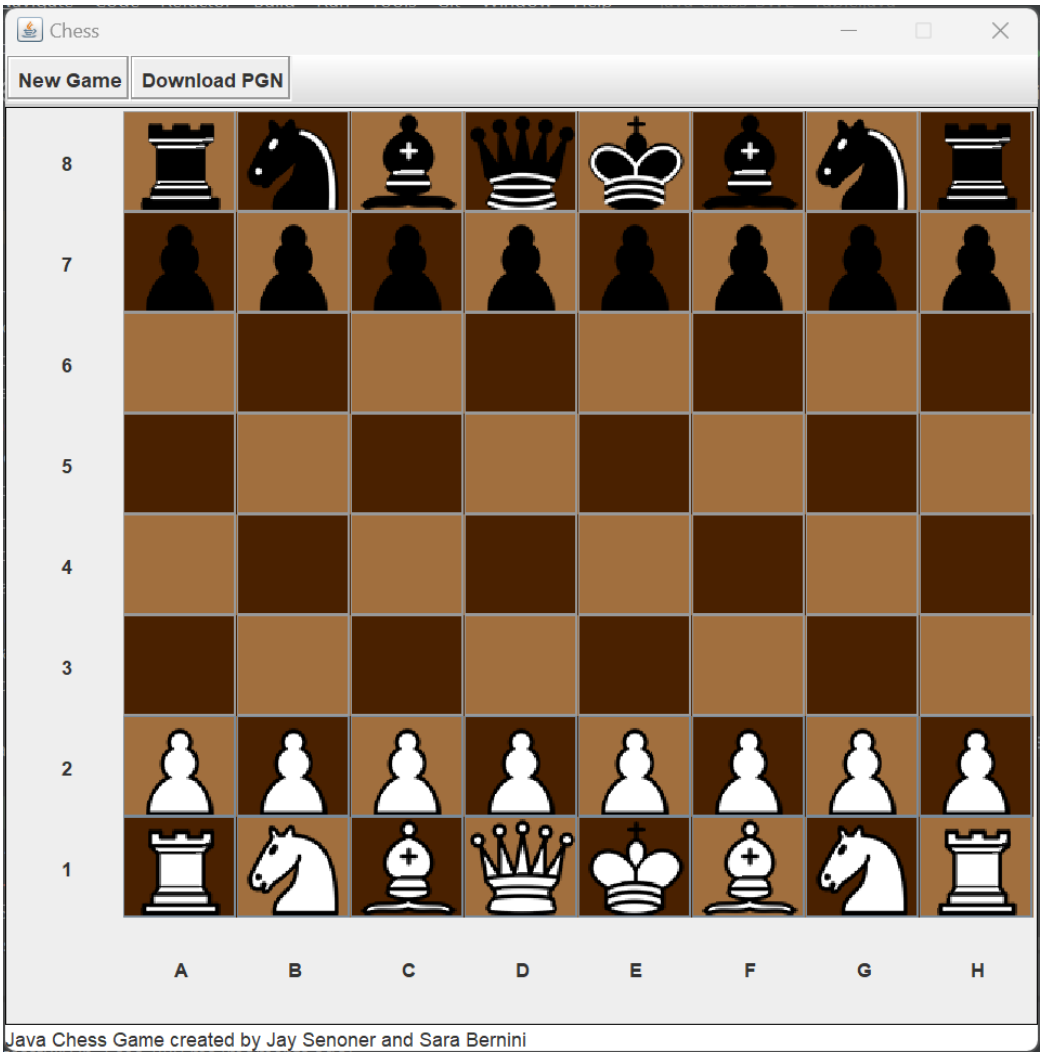

Figura 8: Interfaccia grafica

3.4 Control

Il package Control contiene un unica classe **Controller**, il cui obiettivo è realizzare tre funzioni principali:

1. Gestire gli input dell'utente ricevuti dai componenti della Table, e utilizzare tali input per apportare le modifiche opportune al GameModel. Tali modifiche verranno apportate mediante chiamate a metodi specifici del GameModel. Ad esempio, dopo che un utente clicca su un pezzo, manifestando l'intenzione di muoverlo, il Controller invocherà i metodi del gameModel necessari a determinare l'insieme delle mosse legali di tale pezzo.

2. Gestire i cambiamenti del GameModel al fine di renderli visualizzabili sulla View. Riconducendoci all'esempio precedente, dopo che un utente ha premuto su un pezzo, e tutte le mosse legali di quel pezzo sono state conseguentemente determinate, il Controller invocherà metodi specifici della Table per rendere visibili le case che rappresentano le destinazioni legali per tale pezzo.

3. Gestire le richieste dell'utente di effettuare una delle operazioni presenti sulla ToolBar, ovvero la possibilità di creare una nuova partita e di scaricare il file PGN della partita corrente

   Per realizzare tali funzioni, nella classe Controller saranno presenti un attributo Table e un attributo GameModel, che rappresentano rispettivamente la GUI e la partita correnti. Nel momento in cui l'utente preme sul bottone etichettato "New Game", due nuovi oggetti GameModel e Table verranno creati ed inizializzati. Quando l'utente preme sul bottone "Download PGN", un nuovo file denominato *"chess_game.pgn"* contenente l'elenco di mosse della partita corrente in notazione algebrica verrà creato all'interno della directory del programma. La classe Controller contiene anche metodi per la gestione della promozione: quando uno dei due giocatori riesce a portare un pedone nella rispettiva traversa di promozione, il Controller chiamerà il metodo della Table per mostrare all'utente una finestra tramite la quale egli potrà selezionare il pezzo in cui intende promuovere il pedone. Tale scelta verrà registrata, e in seguito in base alla scelta fatta verranno apportate le opportune modifiche al GameModel. Infine, il Controller presenta metodi per la gestione delle "mosse speciali", ovvero delle mosse dell'arrocco e della presa al varco(che in gergo scacchistico viene comunemente chiamata "Cattura *En passant*")

Listing 8: Classe Controller

```
253     package com.company.control;
254 import com.company.model.GameModel;
255 import com.company.model.Move;
256 import com.company.model.Player;
257 import com.company.model.Square;
258 import com.company.model.pieces.*;
259 import com.company.view.Table;
260 import javax.swing.*;
261 import java.awt.*;
262 import java.awt.event.ActionEvent;
263 import java.awt.event.ActionListener;
264 import java.awt.event.WindowEvent;
265 import java.io.File;
266 import java.io.FileWriter;
267 import java.io.IOException;
268 import java.util.ArrayList;
269
270 public class Controller implements ActionListener {
271     private GameModel gameModel;
272     private Table table;
273     private Square[][] squares;
274     private Square currentStartSquare;
275     private Square currentEndSquare;
276
277
278
279     public Controller() {
280         super();
281         gameModel = new GameModel();
282         table = new Table();
283         squares = gameModel.getBoard().squares;
284         table.initializeView(gameModel);
285         //Setup listeners
286         addListeners();
287         gameModel.getWhitePlayer().
                calculateAllPossibleMoves();
288         gameModel.getBlackPlayer().
                calculateAllPossibleMoves();
289     }
290
```

```
291     private void addListeners(){
292         table.getPgn().addActionListener(this);
293         table.getNewGame().addActionListener(this);
294         for(Square[] row: squares)
295         {
296             for(Square square: row)
297             {
298                 square.addActionListener(this);
299             }
300         }
301     }
302 
303     public void updatePossibleEndSquares(Square s) {
304         if(s.getPiece() != null){
305             table.resetGraySquares(gameModel.getBoard
                    ().getSquares());
306             table.renderGrayPossibleEndSquares(
                    gameModel.filterLegalMoves(s.getPiece()
                    .getPossibleMoves()));
307         }
308     }
309     private void startNewGame(){
310         gameModel = new GameModel();
311         JFrame frame = table.getChessFrame();
312         frame.dispatchEvent(new WindowEvent(frame,
                WindowEvent.WINDOW_CLOSING));
313 
314         table = new Table();
315         squares = gameModel.getBoard().squares;
316         table.initializeView(gameModel);
317         //Setup listeners
318         addListeners();
319         gameModel.getWhitePlayer().
                calculateAllPossibleMoves();
320         gameModel.getBlackPlayer().
                calculateAllPossibleMoves();
321     }
322 
323     private void generatePGNFile(){
324         ArrayList<String> movesInPGN = gameModel.
```

```
            getMovesInPgn();
325         try {
326             File pgnFile = new File("chess_game.pgn");
327             if(pgnFile.createNewFile()){
328                 System.out.println("File pgn creato
                        con successo");
329             }
330             else{
331                 System.out.println("File gi
                        esistente, il file verr
                        sovrascritto");
332             }
333             FileWriter writer = new FileWriter("
                    chess_game.pgn");
334             for(String move: movesInPGN)
335             {
336                 writer.write(move + " ");
337             }
338             writer.close();
339         }
340         catch(IOException e){
341             System.out.println("Errore nella creazione
                    del file");
342             e.printStackTrace();
343         }
344     }
345     //metodo che controlla se l'arrocco    possibile e
            in caso affermativo coloro di grigio la casa
            in cui si sposter
346     // il re in base al tipo di arrocco
347     public void showCastling(){
348         if(gameModel.isShortCastlingLegal()){
349             Square shortCastleSquare = gameModel.
                    getTurn().getShortCastleMove().get(0).
                    getEndSquare();
350             shortCastleSquare.setName("
                    short_castle_square");
351             shortCastleSquare.setBackground(Color.
                    DARK_GRAY);
352             shortCastleSquare.setEnabled(true);
```

```
353             }
354             if(gameModel.isLongCastlingLegal()){
355                 Square longCastleSquare = gameModel.
                        getTurn().getLongCastleMove().get(0).
                        getEndSquare();
356                 longCastleSquare.setName("
                        long_castle_square");
357                 longCastleSquare.setBackground(Color.
                        DARK_GRAY);
358                 longCastleSquare.setEnabled(true);
359             }
360         }
361         @Override
362         public void actionPerformed(ActionEvent e) {
363             Object source = e.getSource();
364 
365             //Controllo se    stato premuto un quadrato
                    nella scacchiera
366             if (source.getClass() == Square.class) {
367 
368                 if (((Square) source).getBackground() !=
                        Color.DARK_GRAY &&((Square) source).
                        getPiece() != null) {
369                     currentStartSquare = (Square) source;
370                     updatePossibleEndSquares(
                            currentStartSquare);
371                     //Se    stato selezionato il re,
                            controllo che l'arrocco sia
                            possibile
372                     if(((Square) source).getPiece().
                            getClass() == King.class){
373                         showCastling();
374                     }
375                     ArrayList<Move> enPassantMoves =
                            gameModel.getTurn().
                            getEnPassantMove();
376                     if(!enPassantMoves.isEmpty()) {
377                         enPassantMoves = gameModel.
                                checkLegacyEnPassant(gameModel.
                                getTurn().getEnPassantMove());
```

```
                    if (enPassantMoves.size()>0 &&
                        source == enPassantMoves.get(0)
                        .getStartSquare()) {
                         showEnPassant(enPassantMoves.
                            get(0));
                    }
                    if (enPassantMoves.size()==2 &&
                        source == enPassantMoves.get(1)
                        .getStartSquare()) {
                         showEnPassant(enPassantMoves.
                            get(1));
                    }
                }
            }

            else if (((Square) source).getBackground()
                == Color.DARK_GRAY) {
                if(((Square) source).getName().equals(
                    "short_castle_square"))
                {
                    gameModel.executeCastlingMove(true
                        );
                }
                else if(((Square) source).getName().
                    equals("long_castle_square")){
                    gameModel.executeCastlingMove(
                        false);
                }
                else {
                    currentEndSquare = (Square) source
                        ;
                    Move moveToExecute = new Move(
                        currentStartSquare,
                        currentEndSquare);
                    gameModel.executeMove(
                        moveToExecute);
                    if(currentEndSquare.getPiece().
                        getClass() == Pawn.class &&
                            (currentEndSquare.
                                getPosition().getRow()
```

```
                                == 0 ||currentEndSquare
                                .getPosition().getRow()
                                == 7)){
401                         //Mostro OptionDialog per
                                selezionare il pezzo
402                         String promotionPiece = table.
                                showPromotionDialog();
403                         promotePawn(currentEndSquare,
                                promotionPiece);
404
405                     }
406                 }
407                 if(((Square) source).getName().equals(
                        "en_passant_square")){
408                     gameModel.executeEnPassant((Square
                            ) source);
409                 }
410                 table.repaintChessBoard(gameModel.
                        getBoard().getSquares(), gameModel.
                        getTurn().isWhite());
411                 table.resetGraySquares(gameModel.
                        getBoard().getSquares());
412
413                 //contrtollo se il giocatore ha subito
                        scacco matto, in caso arresto il
                        gioco e stampo un alert
414                 if(gameModel.kingIsCheckMated()){
415                     table.stopGame(gameModel.getBoard
                            ().getSquares());
416                     table.showCheckMateAlert(gameModel
                            .getTurn().isWhite());
417                 }
418
419                 //contrtollo se siamo in stato di
                        stallo, in caso arresto il gioco e
                        stampo un alert
420                 if(gameModel.isStaleMate()){
421                     table.showStaleMateAlert();
422                     table.stopGame(gameModel.getBoard
                            ().getSquares());
```

```
423                 }
424              }
425          }
426
427          //Controllo se    stato premuto un bottone
                della ToolBar
428          if(source.getClass() == JButton.class){
429             if( ((JButton) source).getText().equals("
                   New Game")){
430
431                 startNewGame();
432             }
433             else if( ((JButton) source).getText().
                   equals("Download PGN")){
434                 System.out.println(gameModel.
                       getMovesInPgn());
435                 generatePGNFile();
436             }
437          }
438      }
439
440      private void showEnPassant(Move move) {
441          Square enPassantSquare= move.getEndSquare();
442          enPassantSquare.setName("en_passant_square");
443          enPassantSquare.setBackground(Color.DARK_GRAY)
                ;
444          enPassantSquare.setEnabled(true);
445      }
446
447      private void promotePawn(Square currentEndSquare,
           String promotionPiece ) {
448          Player playerToPromote;
449          ArrayList<String> movesDone = gameModel.
                getMovesDone();
450          String lastMove = movesDone.get(movesDone.size
                ()-1);
451          if(gameModel.getTurn().isWhite()){
452              playerToPromote = gameModel.getBlackPlayer
                    ();
453          }else{
```

```
454                 playerToPromote = gameModel.getWhitePlayer
                        ();
455             }
456             playerToPromote.getListOfPieces().remove(
                    currentEndSquare.getPiece());
457             gameModel.getBoard().removePiece(
                    currentEndSquare);
458
459             Piece newPiece = null;
460             switch (promotionPiece) {
461                 case "Queen" -> {
462                     newPiece = new Queen(playerToPromote.
                            getKing().getColor());
463                     movesDone.set(movesDone.size()-1,
                            lastMove + "=Q");
464                 }
465                 case "Knight" -> {
466                     newPiece = new Knight(playerToPromote.
                            getKing().getColor());
467                     movesDone.set(movesDone.size()-1,
                            lastMove + "=N");
468                 }
469                 case "Bishop" -> {
470                     newPiece = new Bishop(playerToPromote.
                            getKing().getColor());
471                     movesDone.set(movesDone.size()-1,
                            lastMove + "=B");
472                 }
473                 case "Rook" -> {
474                     newPiece = new Rook(playerToPromote.
                            getKing().getColor());
475                     movesDone.set(movesDone.size()-1,
                            lastMove + "=R");
476                 }
477             }
478             gameModel.getBoard().addPiece(currentEndSquare
                    , newPiece);
479             playerToPromote.getListOfPieces().add(newPiece
                    );
480
```

```
481             gameModel.getBlackPlayer().
                    calculateAllPossibleMoves();
482             gameModel.getWhitePlayer().
                    calculateAllPossibleMoves();
483
484         }
485     }
```

4 Possibili sviluppi futuri

In questo paragrafo conclusivo vengono esposti una serie di possibili funzionalità e caratteristiche aggiuntive per l'applicativo da noi realizzato. Fondamentalmente, tutte le funzionalità necessarie a giocare una partita di scacchi sono già state implementate, pertanto i possibili sviluppi futuri si concentrano sull'arricchimento dell'applicazione con nuove funzionalità.

1. **Gioco multiplayer online**: Nella maggior parte delle applicazioni moderne, in particolare in quelle che realizzano un qualche tipo di gioco, è presente la possibilità di connettersi alla rete per giocare con altri giocatori. Le applicazioni di maggior successo per giocare a scacchi, come *Lichess*, *Chess.com* e *Chesstempo.com* offrono questa possibilità.

2. **Giocare contro il Computer**:Sempre facendo riferimento alle applicazioni sopra citate, in esse è quasi sempre possibili giocare una partita contro bot di diversa difficoltà. Una possibile aggiunta al nostro applicativo consisterebbe nell'utilizzare l'Algoritmo minimax per realizzare una semplice intelligenza artificiale contro cui giocare.

3. **Aggiunta delle impostazioni temporali**: Nelle partite di scacchi, e in particolare in quelle dei tornei ufficiali, è sempre presente un limite temporale entro il quale i giocatori devono restare, pena la sconfitta per tempo. Una possibile aggiunta al nostro progetto potrebbe consistere nell'aggiunta di una funzione alla ToolBar che permetta di settare un timer per entrambi i giocatori, con eventuale incremento temporale costante per mossa effettuata

4. **Regole di patta e patta per accordo**: Poichè la nostra applicazione è stata pensata fin dall'inizio come un applicazione che permettesse a

due giocatori di disputare una partita in locale, è stato volutamente omesso il concetto di patta per accordo(poichè essendo in locale, i due giocatori possono accordarsi verbalmente). Inoltre, sempre per le stesse motivazioni, è stato omesso lo sviluppo della Regola delle 50 mosse, della patta per ripetizione e della partita morta(partita nella quale non è matematicamente possibile vincere per entrambi i giocatori, ad esempio Re e cavallo contro Re, Re e alfiere contro Re, Re contro Re), mentre invece è stata inclusa la patta per stallo.