Dama Italiana

Progetto di Esperienze di Programmazione

Nicolò Tonci

Febbraio 2016

Indice

1 Introduzione

La relazione presentata deriva dal tentativo di implementare un gioco a due giocatori, la Dama Italiana, affrontando i temi di ricerca in spazi di stati sfociando cosi nell'intelligenza artificiale.
Tale progetto è stato scelto poiché il tema trattato è molto attuale e presenta numerosi aspetti interessanti (pratici e etici) argomento di dibattito tra scienziati (prevalentemente informatici) e filosofi.
Ci focalizzeremo sulle metodologie e scelte implementative di un agente intelligente (il giocatore) tralasciando aspetti puramente speculativi sul tema.

2 Regole ufficiali del gioco

Per implementare il gioco sono state seguite le regole ufficiali[1] e consistono nelle seguenti:

- La partita si svolge tra due giocatori;
- La damiera è composta da 64 caselle (32 chiare e 32 scure), disposta con le caselle scure d'angolo (cantone) alla destra dei giocatori;
- Le pedine sono 24: 12 bianche e 12 nere e si dispongono sulle caselle scure. All'inizio del gioco ogni giocatore posiziona le proprie pedine (o bianche o nere) nelle prime tre righe in basso nel proprio lato della damiera;
- Il gioco è sempre iniziato dal giocatore che conduce la partita con le pedine bianche;
- Le pedine muovono sempre in avanti di una casella sulle caselle scure e quando raggiungono la base avversaria diventano dame acquisendo la possibilità di muoversi in tutte le direzioni;
- La presa è obbligatoria: infatti, quando una pedina incontra una pedina di colore diverso, con una casella libera dietro, sulla stessa diagonale, è obbligata a prenderla (si dice anche catturarla o mangiarla). La pedina, dopo la prima presa, qualora si trovi nelle condizioni di poter nuovamente prendere, deve continuare a catturare pezzi;
- Le dame seguono anch'esse la regola precedente e hanno la capacità di prendere altre dame;
- Se in un determinato istante della partita si hanno più scelte per mangiare, vanno rispettate obbligatoriamente nell'ordine le seguenti priorità:

[1] Presenti nel sito della Federazione Italiana Dama

1. mangiare più pezzi possibili;
2. a parità di numero di pezzi tra mangiate che coinvolgono dama e pedina, mangia la dama;
3. la dama effettua la mossa che mangia il maggior numero di dame avversarie;
4. a parità di tutte le precedenti regole, la scelta è libera.

- Vince il giocatore che prende tutti i pezzi dell'avversario o che riesce a mettere quest'ultimo in condizione di non poter fare una mossa;
- La partita è dichiarata patta quando le posizioni dei pezzi sulla damiera si ripetono;

3 Intelligenza artificiale

In questo capitolo presentiamo la parte più interessante, nonché il cuore, del progetto ovvero quella di intelligenza artificiale dando per ora una descrizione sommaria del motore interno e approfondendo successivamente ciascun aspetto.
Il motore viene invocato ogni qual volta è il calcolatore a dover effettuare una mossa e vengono eseguite le seguenti operazioni:

1. Costruzione dell'albero di ricerca
2. Applicazione della funzione di valutazione ai nodi foglia
3. Esecuzione dell'algoritmo MinMax
4. Esecuzione della miglior mossa

3.1 Costruzione dell'albero di ricerca

I nodi di questo albero rappresentano possibili configurazioni della damiera. L'albero ha queste caratteristiche:

- la radice dell'albero rappresenta la configurazione attuale (quella effettiva al momento dell'invocazione del motore;
- alla radice vengono aggiunti tanti figli quante sono le mosse legali da parte del computer, salvando in essi la configurazione dopo ogni mossa;
- a questi ultimi vengono aggiunti tanti figli quante sono le mosse legali da parte dell'avversario, salvando in essi la configurazione dopo ogni mossa;
- si itera questo procedimento fino ad arrivare ad una configurazione di vittoria (nodi foglia);

- i nodi a profondità dispari rappresentano possibili mosse da parte del calcolatore (chi invoca la procedura);
- i nodi a profondità pari rappresentano possibili mosse da parte dell'avversario;

Tale procedimento è ideale in quanto la complessità aumenta in modo esponenziale in funzione della profondità dell'albero rendendo la computazione poco interattiva e molto esigente in quanto a risorse fisiche, per questo motivo viene fissata una profondità massima valutando i nodi fogli con una funziona che ne determina la bontà basata su criteri euristici legati a regole empiriche del gioco.

3.2 Funzione di valutazione

La funzione di valutazione è un elemento fondamentale per il motore di intelligenza artificiale, e come vedremo rappresenta anche la strategia di gioco.
Per valutare ogni configurazione futura (nodi foglia dell'albero), che potrà essere favorevole, sfavorevole o stabile, si costruisce una funzione euristica (1) il cui dominio è l'insieme delle possibili configurazioni della damiera e come codominio l'insieme dei numeri interi.

$$f : \mathbb{A} \to \mathbb{Z} \tag{1}$$

In particolare, l'insieme A è quello delle matrici 8x8, modo in cui rappresentiamo la damiera e verrà trattato in seguito in questo documento. L'output, appartenente all'insieme Z, va invece cosi interpretato (in riferimento al giocatore avente pedine nere):

$$f(A) \begin{cases} < 0, & \text{favorevole} \\ > 0, & \text{sfavorevole} \\ = 0, & \text{stabile} \end{cases}$$

La prima caratteristica delle funzioni di valutazione è quella di essere computazionalmente leggere nel calcolo, infatti se cosi non fossero sarebbe più proficuo aumentare l'altezza dell'albero di ricerca incrementando cosi l'attendibilità della scelta effettuata.
Inoltre tale funzione deve essere efficace cioè deve prevedere gli stati terminali dell'albero decisionale con un margine di errore limitato.
E' evidente che la funzione determina la strategia del gioco ed è quindi possibili costruire funzioni prettamente difensive che tendono a proteggere il proprio lato della damiera o aggressive privilegiando il mangiare ed andare nel più breve tempo possibile a dama.

3.2.1 Realizzazione della funzione

La realizzazione si basa su una tecnica spesso usata per la realizzazione di funzioni di valutazione, ovvero viene assegnato un peso ad ogni pezzo e un valore in base alla posizione occupata all'interno della damiera.
In particolare, in questa realizzazione, alle pedine viene assegnato un peso di 100 mentre alle dame di 200, il valore invece è in funzione del numero di riga in cui si trova e cresce in maniera esponenziale tanto più si va verso lo schieramento difensivo.
Dall'osservazione che i pezzi che si trovano lungo i bordi sono penalizzati dal fatto che possono muoversi solo da un lato (se questo non è bloccato) è stato aggiunto il fattore bordo preferendo cosi spostamenti verso posizioni centrali.
Formalmente la valutazione si calcola applicando la seguente formula:

$$valutazione+ = \begin{cases} 100(\#\text{PedineNere} - \#\text{PedineBianche}) \\ 200(\#\text{DameNere} - \#\text{DameBianche}) \\ \#\text{PedineNereBound} - \#\text{PedineBiancheBound} \\ \sum\limits_{\forall \text{ pedina nera}} i^2 \\ - \sum\limits_{\forall \text{ pedina bianca}} (7-i)^2 \end{cases}$$

Se l'output ottenuto 'e vicino allo zero, vuol dire che la configurazione valutata non porta ad uno stato vantaggioso per nessuno dei due giocatori. Se invece è tanto maggiore di zero, allora risulterà favorevole per il giocatore nero; altrimenti sarà vantaggiosa per il bianco.

3.3 Algoritmo MiniMax

Una volta creato l'albero di ricerca ed applicato la funzione di valutazione ai nodi foglia, il computer deve scegliere effettivamente quale mossa scegliere tra quelle possibili, ovvero quella che lo porterà in uno stato più favorevole. La scelta effettiva è il prodotto dell'algoritmo Minimax che nella teoria delle decisioni è un metodo per minimizzare la massima perdita possibile, sotto l'ipotesi in cui anche l'avversario stia giocando in maniera ottimale. L'algoritmo può essere descritto sinteticamente ed in maniera informale come segue:

- per ogni nodo di altezza dispari (MAX) si assegna ad esso il valore minimo dei nodi figli (MIN)
- per ogni nodo di altezza pari (MIN) si assegna ad esso il valore max dei nodi figli (MAX)
- si procede ricorsivamente in modalità depth-first fino a trovare il valore della radice che rappresenta la scelta da effettuare

Qui di seguito l'algoritmo presentato in pseudocodice:

```
function minimax(nodo, profondità)
    SE nodo è un nodo terminale OPPURE profondità = 0
        return il valore euristico del nodo
    SE l'avversario deve giocare
        a := +infinito
        PER OGNI figlio di nodo
            a := min(a, minimax(figlio, profondità-1))
    ALTRIMENTI dobbiamo giocare noi
        a := -infinito
        PER OGNI figlio di nodo
            a := max(a, minimax(figlio, profondità-1))
    return a
```

4 Implementazione

Il progetto è stato realizzato in Java, con particolare attenzione alla modularità del codice che ha permesso nel corso della realizzazione di aggiungere un interfaccia grafica al progetto andandosi ad aggiungere a quella testuale (console) di partenza. Tale scelta (GUI) è stata fatta per facilitare l'interazione con il gioco e per approfondire il framework grafico Swing.
Il codice è diviso in tre package:

1. **dama_italiana**: cuore dell'applicazione dove risiede il modulo dell'intelligenza artificiale;
2. **GUI**: interfaccia grafica del gioco;
3. **ConsoleInterface**: interfaccia testuale (console) del gioco;

4.1 Package dama_italiana

In questo package, come prima riferito, vi sono tutti gli algoritmi di intelligenza artificiale e della dama, nonché tutte le strutture dati su cui operano quest'ultimi. A questo package è possibile collegare qualsiasi interfaccia (Console o Grafica) data la sua modularità.
Andiamo a descrivere in dettaglio le classi che lo compongono.

4.1.1 Classe Dama

E' la classe principale del package ed è il punto di collegamento con le interfacce che intendono dialogare con il motore della dama. In essa è memorizzata la matrice che rappresenta la damiera e le costanti (intere) con cui rappresentiamo i pezzi nella matrice (si veda il codice per le definizioni). Il

costruttore si limita ad inizializzare la damiera secondo le regole descritte al capitolo 2. Analizziamo ora i metodi partendo dai più semplici:

- *GetMatrix*: restituisce la matrice della scacchiera;
- *CopyMAtrix*: effettua una deep-copy della matrice e ne restituisce il clone;
- *ExecutePCMove*: effettua la miglior mossa possibile invocando il motore di intelligenza artificiale, tale mossa è effettuata al giocatore passato per parametro (0: nero; 1: bianco);
- *Valute*: la funzione di valutazione usata dal motore di intelligenza artificiale;
- *RetriveMoves*: invoca il motore di gioco e restituisce un vettore di mosse legali del giocatore passato per parametro (0: nero; 1: bianco);
- *CheckMoveAndExecute*: Dati (come parametro) le mosse possibili e il vettore di coordinate che caratterizzano la mossa se questa è legale viene eseguita sulla damiera;

4.1.2 Classe MovesEngine

Questa è la classe che implementa il vero e proprio gioco della dama ovvero le dinamiche e tutte le relative regole del gioco. Particolare rilevanza ha il metodo che si occupa di calcolare tutte le mosse legali per un giocatore poiché queste saranno usate sia dal calcolatore per giocare sia per verificare che quella effettuata dal giocatore umano sia legittima. Vediamo ora i metodi uno ad uno:

- *CheckCoordinates*: restituisce true se le coordinate x e y passate per parametro sono valide (ovvero sono all'interno della damiera) false altrimenti;
- *MovesCompare*: funzione di comparazione tra mosse, essa serve per trovare la mossa con maggiore priorità, secondo le regole definite nel capitolo 2, nel caso in cui vi siano più mangiate disponibili;
- *CompareValue*: funzione ausiliaria alla *MovesCompare* per comparare l'ordine dei pezzi mangiati (codificati in stringhe);
- *RegisterMoves*: per ogni mossa calcolata dalla *CalculateMoves* si confronta la priorità con quella maggiore e se risulta maggiore o uguale viene registrata tra la mosse legali (nel caso di mossa con priorità maggiore vengono eliminate quelle precedenti);

- *CloneMAtrixAndUpdate*: esegue la mossa (parte di mossa) definita dalle coordinate di inizio e di fine e restituisce la matrice facendone prima una deep-copy;
- *CalculateMoves*: per ogni pezzo, del giocatore che deve muovere nel corrente turno, calcola l'albero delle possibili mosse ed invoca la *RegisterMoves* quando giunge ad una foglia, calcolandosi cosi tutti i percorsi possibili per ogni singola pedina;

Il metodo CalculateMoves è stato quello che ha destato più difficoltà nell'implementazione del progetto, si rimanda al codice adeguatamente commentato per la completa comprensione delle operazioni effettuate.

4.1.3 Classe Move

Move è una classe molto semplice che può essere considerata come un record contenente:

- *Sequence*: sequenza di Point che caratterizzano la mossa (ovvero tutti punti in cui passa un pezzo nel corso dell'intera mossa)
- *FinalMatrix*: matrice che rappresenta la damiera una volta effettuata una mossa

Vi è inoltre un metodo, *CheckForNewDame*, che controlla eventuali nuove dame dopo l'esecuzione di una mossa.

4.1.4 Classe DamaTree

Questa classe è implementa l'albero di ricerca usato per il modulo di intelligenza artificiale e l'algoritmo minimax. Si rimanda al capitolo 3 per la comprensione degli usi e elle proprietà di tale albero. Ci soffermiamo brevemente però sul metodo *ExecuteBestMove*, il quale scorre i figli della radice dell'albero per trovare la mossa che porta ad una situazione meno sfavorevole e la esegue, ritornando cosi la matrice che rappresenta, al solito, la damiera dopo aver effettuato la mossa e averne fatto una deep-copy.

4.2 Package ConsoleInterface

In questo package vi è l'interfaccia Console del gioco semplice e minimale. Di questo package fanno parte due classi *ConsoleGame* che andremo ad approfondire e *Main* che si limita a creare un nuovo ConsoleGame.

4.2.1 Classe ConsoleGame

E' la classe che implementa tutti i metodi di interazione per mezzo di terminale. Nella creazione viene stampato un menu in cui si invita l'utente a

selezionare la modalità di gioco e successivamente avviarla. Fanno parte di questa classe i seguenti metodi:

- *Run*: si entra nel game loop e, a seconda della modalità, si eseguono le azioni che caratterizzano una partita a dama. In particolare, se la modalità selezionata è CPU vs. CPU si invocherà il motore di calcolo della miglior mossa alternativamente con periodicità "bianco-nero" fino al termine della partita; se la modalità è, invece, Human vs CPU verrà invocato alternativamente il metodo *HumanMove*, il quale rappresenta la mossa dell'utente (bianco), e il motore di calcolo per l'esecuzione il quale rappresenta la mossa del calcolatore (nero);

- *Print*: stampa a schermo la damiera formattata opportunamente (si veda inoltre i metodi ausiliari per la formattazione che tralasciamo in questa documento);

- *HumanMove*: vengono calcolate le mosse legali tramite l'ausilio della classe *MovesEngine* e se una mossa data in input è tra quelle lecite viene eseguita, se la mossa risulta non lecita viene stampato un messaggio di errore e si invita l'utente ad inserire una nuova "mossa";

- *ParseInput*: effettua la conversione dei comandi (es. D4 F6) in coordinate intere per l'accesso alla matrice, esso restituisce cosi un vettore di Point che caratterizzano la mossa. Si invocano inoltre metodi di conversione ausiliari che tralasciamo ma di facile comprensione.;

4.3 Package GUI

Esso è il package utilizzato per realizzare la finestra di gioco, la damiera e l'avvio del programma. Per l'interazione con il contesto grafico è stato utilizzato il framework Swing. Molto brevemente andiamo a presentare le classi principali.

4.3.1 Classe Board

Questa classe realizza il controllo grafico della damiera. Si occupa di ridisegnare la configurazione della damiera ogni qualvolta viene terminata una mossa. E' simile alla classe *ConsoleGame* nei comportamenti ma utilizziamo gli eventi del mouse (al posto dell'input da tastiera codificato) e nelle stampa della damiera invochiamo il metodo *PaintComponent* della libreria Swing per disegnare sul controllo grafico.

4.3.2 Classe Form

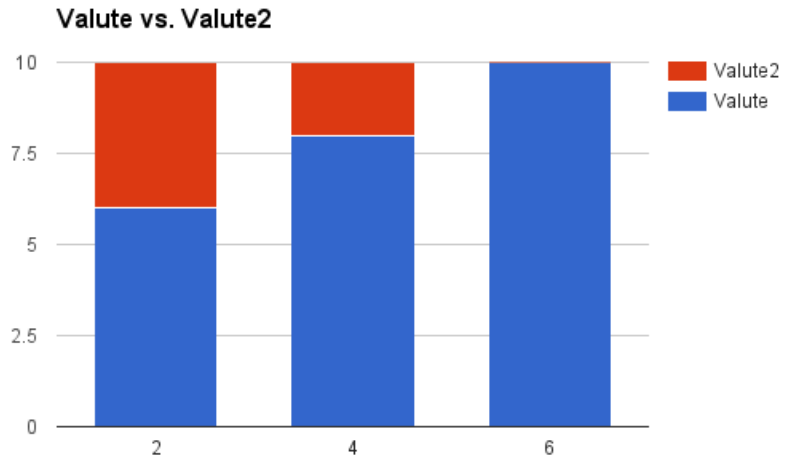
Implementa la finestra di gioco, la barra del menu per la scelta delle modalità di gioco e la status bar per la stampa di informazioni.

5 Test

5.1 Funzione di Valutazione

Nel corso del progetto sono state implementate diverse funzioni di valutazione e per determinare quella più efficace sono state sottoposte a test in cui venivano messe a confronto giocando 10 partite l'una contro l'altra (CPUvsCPU), modificando inoltre l'altezza dell'albero di ricerca. Riportiamo, sotto, un grafico del risultato ottenuto con le due più efficaci:

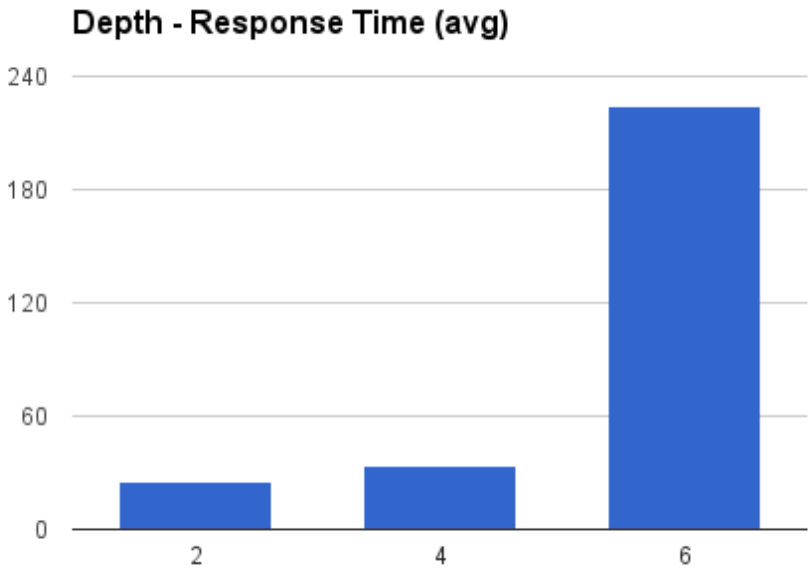
- **Valute**: presentata nella sezione 3.2.1;
- **Valute2**: simile alla Valute ma si predilige la difesa rispetto all'attacco;



Dai dati è evidente che la prima funzione (Valute) è nettamente superiore, soprattutto aumentando l'altezza dell'albero di ricerca, ed è per questo che è stata scelta.

5.2 Tempi di risposta

E' stata eseguita una valutazione sui tempi di risposta in funzione dell'altezza dell'albero di ricerca. Prevedibilmente queste due misure sono proporzionali, ovvero aumentando l'altezza dell'albero aumenta anche il tempo di risposta.
Qui di seguito i risultati di risposta media (in ms) variando l'altezza:



E' stato deciso di fissare a 6 l'altezza considerandolo il valore ottimale a seguito di numerose osservazioni che hanno dato frutto a queste conclusioni:

- < 6 : il calcolo risulta molto rapido, sebbene le scelte durante il gioco non sono sempre ottimali;
- > 6 : il calcolo non risulta molto performante dando luogo, a volte, a saturazioni della memoria impedendo il prosieguo del gioco;

6 Conclusioni

Questo progetto, per quanto impegnativo, mi ha reso cosciente di una realtà come quella dell'intelligenza artificiale che nel futuro si espanderà in numerosi contesti modificando la nostra quotidianità e l'interazione uomo-macchina. Gran parte del tempo impiegato nel progetto è stato dedicato allo studio e alla realizzazione del modulo di intelligenza artificiale, trascurando volutamente le interfacce di gioco che risultano piuttosto essenziali.
In conclusione, è stata un'esperienza formativa e alquanto interessante.

Allegati: Codice

Nelle prossime pagine viene listato il codice di cui si fa riferimento nell'intero documento.

/Users/nico/Documents/EP/Dama_v1/src/dama_italiana/Dama.java

```java
package dama_italiana;

import java.awt.Point;
import java.util.*;

public class Dama {

    /*
    Definizione delle pedine e delle dame
     */
    public final static int BLANK = 0;

    public final static int BLACK = 1;
    public final static int WHITE = 2;

    public final static int D_BLACK = 3;
    public final static int D_WHITE = 4;

    /*
    Altro
     */
    private static int MAX_DEPTH = 6;

    private int matrix[][];

    public Dama() {
        /*
        Inizializzo la matrix con l'opportuna disposizione delle pedine
         */
        this.matrix = new int[8][8];
       for(int i = 0; i<8; i++)
           for(int j = 0; j<8; j++)
               if ((i % 2) == (j % 2)) {
                   if (i<3)
                       this.matrix[i][j] = BLACK;
                   else if (i>4)
                       this.matrix[i][j] = WHITE;
                   else
                       this.matrix[i][j] = BLANK;
               } else
                   this.matrix[i][j] = BLANK;
    }

    public int[][] GetMatrix(){
        return this.matrix;
    }

    public int[][] CopyMatrix() {
        int [][] newM = new int[8][8];
        for(int i = 0; i < 8; i++){
            newM[i] = Arrays.copyOf(matrix[i],8);
        }
        return newM;
```

/Users/nico/Documents/EP/Dama_v1/src/dama_italiana/Dama.java

```
54      }
55
56      public Vector<Move> RetriveMoves(int color){
57          MovesEngine me = new MovesEngine(matrix, color);
58          return me.PossibleBestMooves;
59      }
60
61
62      public boolean CheckMoveAndExecute(Vector<Move> m, Vector<Point> p){
63          for(Move move : m)
64              if (move.Sequence.equals(p)) {
65                  for(int i = 0; i < 8; i++)
66                      this.matrix[i] = Arrays.copyOf(move.FinalMatrix[i],8);
67                  return true;
68              }
69          return false;
70      }
71      /*
72      turn  = 0 -> Muove il black
73      turn  = 1 -> Muove il white
74       */
75      public void ExecutePCMove(int turn) throws UnsupportedOperationException{
76          DamaTree dt = new DamaTree(this.matrix, MAX_DEPTH, turn);
77          this.matrix = dt.ExecuteBestMove();
78          dt = null;
79          System.gc();
80      }
81
82      /* Funzione di valutazione */
83      public static int Valute(int[][] matrix){
84          int score = 0;
85          for (int i = 0; i < 8; i++)
86              for (int j = 0; j < 8; j++)
87                  if (matrix[i][j] == WHITE) {
88                      score -= 100;
89                      score -= 1 * (7-i) * (7-i);
90                  }
91                  else if (matrix[i][j] == D_WHITE){
92                      score -= 200;
93
94                      if (i == 0 || i == 7)       //check  WHITE
95                          score += 10;
96
97                      if (j == 0 || j == 7)
98                          score += 10;
99                  }
100                 else if (matrix[i][j] == D_BLACK) {
101                     score += 200;
102
103                     if (i == 0 || i == 7)       //check BLACK
104                         score -= 10;
105
106                     if (j == 0 || j == 7)
```

/Users/nico/Documents/EP/Dama_v1/src/dama_italiana/Dama.java

```
107                         score -= 10;
108                 }
109                 else if (matrix[i][j] == BLACK) {
110                     score += 100;
111                     score += 1 * i * i;
112                 }
113 
114         score += (int)(Math.random() * 10);
115         return score;
116     }
117 
118 }
119 
120 
```

/Users/nico/Documents/EP/Dama_v1/src/dama_italiana/Move.java

```
 1 package dama_italiana;
 2 import java.awt.*;
 3 import java.util.Vector;
 4 public class Move{
 5     public Vector<Point> Sequence;
 6     public int[][] FinalMatrix;
 7     public Move(Vector<Point> seq, int[][] m){
 8         this.Sequence = seq;
 9         CheckForNewDame(m);
10         this.FinalMatrix = m;
11     }
12 
13     private static void CheckForNewDame(int[][] matrix) {
14         for(int j = 0; j < 8; j++) {
15             if (matrix[0][j] == Dama.WHITE) // check dame bianche
16                 matrix[0][j] = Dama.D_WHITE;
17 
18             if (matrix[7][j] == Dama.BLACK)
19                 matrix[7][j] = Dama.D_BLACK;
20         }
21     }
22 }
23 
```

/Users/nico/Documents/EP/Dama_v1/src/dama_italiana/DamaTree.java

```java
package dama_italiana;

import java.awt.*;
import java.util.Vector;
import java.util.Arrays;

public class DamaTree {
    /* massima profondità di costruzione dell'albero */


    public int score;
    public int[][] matrix;

    private Vector<DamaTree> sons = new Vector();
    private int turn;

    public DamaTree(int[][] matrix, int depth, int turn){
        this.matrix = matrix;
        this.turn = turn;
        if(depth == 0){
            score = Dama.Valute(matrix);
            return;
        }
        int color;

        if (turn == 0)
            color = depth % 2 == 0 ? 1 : 0; // 0 se è il turno del bianco; 1 se è turno del computer
        else
            color = depth % 2 == 0 ? 0 : 1;

        MovesEngine me = new MovesEngine(matrix, color);

        /* caso in cui il nodo sia una foglia */
       /* if (me.PossibleBestMooves.size() == 0) {
            score = Dama.Valute(matrix);
            return;
        }

        /* creazione albero di ricerca */
        for(Move m : me.PossibleBestMooves){
            sons.add(new DamaTree(m.FinalMatrix, depth-1, turn));
        }

        /* applicazione algoritmo MIN-MAX */
        if (color == 1){
            this.score = Integer.MIN_VALUE;
            // cerco il MAX e lo metto nel mio score
            for (DamaTree s : sons)
                if (s.score > this.score)
                    this.score = s.score;
            if (this.score == Integer.MIN_VALUE)
                this.score = Integer.MAX_VALUE;
```

/Users/nico/Documents/EP/Dama_v1/src/dama_italiana/DamaTree.java

```
53          }
54          else {
55              this.score = Integer.MAX_VALUE;
56              // cerco il MIN e lo metto nello score
57              for (DamaTree s : sons)
58                  if (s.score < this.score)
59                      this.score = s.score;
60              if (this.score == Integer.MAX_VALUE)
61                  this.score = Integer.MIN_VALUE;
62          }
63      }
64
65      public int[][] ExecuteBestMove() throws UnsupportedOperationException{
66          int matrix[][] = new int[8][8];
67
68          if (sons.size() == 0)
69              throw new UnsupportedOperationException("No moves available for PC! "+ (turn
    == 0 ? "WHITE" : "BLACK") +" Wins!");
70
71          for (DamaTree s : sons)
72              if (s.score == this.score){
73                  for(int i = 0; i < 8; i++)
74                      matrix[i] = Arrays.copyOf(s.matrix[i],8);
75
76                  return matrix;
77              }
78          return matrix;
79      }
80
81      /* DEBUG METOHD --- ignore*/
82      public static void PrintMoves(Vector<Move> m){
83          System.out.println("Serie di mosse: "+ m.size());
84          for(Move serie : m){
85              for(Point c : serie.Sequence){
86                  System.out.print(c.toString());
87              }
88              System.out.print("\n");
89          }
90
91      }
92
93  }
94
```

/Users/nico/Documents/EP/Dama_v1/src/dama_italiana/MovesEngine.java

```
 1 package dama_italiana;
 2
 3 import java.util.Arrays;
 4 import java.util.Vector;
 5 import java.awt.Point;
 6
 7 public class MovesEngine {
 8     /* if BLACK: 0, if WHITE: 1; */
 9     private int color;
10
11
12     /* Max Mooves */
13     private int MaxEaten = 0;
14     private String BestSeq = new String();
15     private int MaxType; // 0= pedina ; 1=Dama
16     public Vector<Move> PossibleBestMooves = new Vector();
17
18     /* End Max Mooves */
19
20
21     public MovesEngine(int [][] matrix, int color){
22         this.color = color;
23
24
25         for(int i = 0; i < 8; i++)
26             for(int j = 0; j < 8; j++)
27                 if (matrix[i][j] != Dama.BLANK && matrix[i][j] % 2 == this.color) {
28                     CalculateMoves(matrix, new Point(i,j), new Vector<Point>(), new
   String());
29                 }
30
31     }
32
33     private boolean CheckCoordinates(int x, int y){
34         return ((x > -1 && x < 8) && ((y > -1) && (y < 8)));
35     }
36
37     private int CompareValue(String s){
38        int occ1 = 0, occ2 = 0;
39        for(int i = 0; i< s.length(); i++){
40            if (s.charAt(i) == 'd')
41                occ1++;
42            if (this.BestSeq.charAt(i) == 'd')
43                occ2++;
44        }
45        return Integer.signum(occ1 - occ2);
46    }
47
48     private int MovesCompare(int type,  int eaten, String seq){
49         if (eaten == 0 && MaxEaten == 0)
50             return 0;
51         if (eaten == this.MaxEaten){
52             if (type == this.MaxType){
```

/Users/nico/Documents/EP/Dama_v1/src/dama_italiana/MovesEngine.java

```
53                  switch (CompareValue(seq)) {
54                      case 0:
55                          return seq.compareToIgnoreCase(this.BestSeq);
56                      case 1: return 1;
57                      case -1: return -1;
58                  }
59              }
60              else if(type > this.MaxType)
61                  return 1;
62              else
63                  return -1;
64          } else if (eaten > this.MaxEaten)
65              return 1;
66
67          return -1;
68      }
69
70      private void RegisterMoves(int[][] finalMatrix, Vector<Point> MCoord, boolean isDama
    , String seq) {
71          /* Registro le mosse solo se sono più convenienti oppure se sono esattamente
    equipotenziali */
72
73          switch (MovesCompare(isDama ? 1 : 0, seq.length(), seq)) {
74              case 1:  this.MaxEaten = seq.length();
75                  this.MaxType = isDama ? 1 : 0;
76                  this.BestSeq = seq;
77                  PossibleBestMooves.clear();
78                  PossibleBestMooves.add(new Move(MCoord, finalMatrix));
79                  break;
80              case 0: PossibleBestMooves.add(new Move(MCoord, finalMatrix));
81                  break;
82          }
83
84      }
85
86      private static int[][] CloneMatrixAndUpdate( int[][] m, Point source,  Point dest,
    Point eaten){
87          int [][] newM = new int[8][8];
88          for(int i = 0; i < 8; i++){
89              newM[i] = Arrays.copyOf(m[i],8);
90          }
91          newM[dest.x][dest.y] = newM[source.x][source.y];
92          if (eaten != null)
93              newM[eaten.x][eaten.y] = Dama.BLANK;
94          newM[source.x][source.y] = Dama.BLANK;
95          return newM;
96      }
97
98      private void CalculateMoves(int[][] matrix, Point c, Vector<Point> PrevC, String
    score){
99          boolean imDama = matrix[c.x][c.y] > 2;
100         boolean canMove = PrevC.size() == 0;
101         int dir = matrix[c.x][c.y] % 2 == 0 ? -1 : 1;
```

/Users/nico/Documents/EP/Dama_v1/src/dama_italiana/MovesEngine.java

```
102         int myType = matrix[c.x][c.y];
103         PrevC.add(c);
104 
105         boolean stop = true;
106 
107         /* vado a vedere la casella verso destra nella direzione opportuna */
108         if (CheckCoordinates(c.x + dir, c.y+1)){
109 
110             /* posso spostarmi sulla destra */
111             if (matrix[c.x + dir][c.y + 1] == Dama.BLANK && canMove){
112                 Vector<Point> currC = new Vector<Point>(PrevC);
113                 Point newC = new Point(c.x + dir, c.y + 1);
114                 currC.add(newC);
115                 RegisterMoves(CloneMatrixAndUpdate(matrix, c, newC, null), currC, imDama
    , score);
116                 stop = false;
117             }
118             /* se è occupata controllo se posso mangiare */
119             else if (matrix[c.x + dir][c.y + 1] % 2 != myType % 2 && CheckCoordinates(c.
    x+ 2*dir, c.y + 2) && matrix[c.x + 2*dir][c.y + 2] == Dama.BLANK && matrix[c.x + dir][c.
    y + 1] != Dama.BLANK){
120                 /* se quella è una dama e io sono una dama */
121                 if (matrix[c.x + dir][c.y + 1] > 2 && myType > 2){
122                     // MANGIO LA DAMA CON UNA DAMA
123                     Point newPos = new Point(c.x + 2*dir, c.y + 2);
124                     CalculateMoves(CloneMatrixAndUpdate(matrix, c, newPos, new Point(c.x
     + dir, c.y +1)), newPos, new Vector<Point>(PrevC), score.concat("d"));
125                     stop = false;
126                 }
127                 /* altrimenti se è una pedina */
128                 else if (matrix[c.x + dir][c.y + 1] < 3){
129                     // MANGIO UNA PEDINA
130                     Point newPos = new Point(c.x + 2*dir, c.y + 2);
131                     CalculateMoves(CloneMatrixAndUpdate(matrix, c, newPos, new Point(c.x
     + dir, c.y +1)), newPos, new Vector<Point>(PrevC), score.concat("p"));
132                     stop = false;
133                 }
134             }
135         }
136 
137         /* vado a vedere la casella verso sinistra nella direzione opportuna */
138         if (CheckCoordinates(c.x + dir, c.y - 1)){
139 
140             /* posso spostarmi sulla sinistra */
141             if (matrix[c.x + dir][c.y - 1] == Dama.BLANK && canMove){
142                 Vector<Point> currC = new Vector<Point>(PrevC);
143                 Point newC = new Point(c.x + dir, c.y - 1);
144                 currC.add(newC);
145                 RegisterMoves(CloneMatrixAndUpdate(matrix, c, newC, null), currC, imDama
    , score);
146                 stop = false;
147             }
148             /* se è occupata controllo se posso mangiare */
```

/Users/nico/Documents/EP/Dama_v1/src/dama_italiana/MovesEngine.java

```
149             else if (matrix[c.x + dir][c.y - 1] % 2 != myType % 2 && CheckCoordinates(c.
    x+ 2*dir, c.y - 2) && matrix[c.x + 2*dir][c.y - 2] == Dama.BLANK && matrix[c.x + dir][c.
    y - 1] != Dama.BLANK){
150                 /* se quella è una dama e io sono una dama */
151                 if (matrix[c.x + dir][c.y - 1] > 2 && myType > 2){
152                     // MANGIO LA DAMA CON UNA DAMA
153                     Point newPos = new Point(c.x + 2*dir, c.y - 2);
154                     CalculateMoves(CloneMatrixAndUpdate(matrix, c, newPos, new Point(c.x
     + dir, c.y -1)), newPos, new Vector<Point>(PrevC), score.concat("d"));
155                     stop = false;
156                 }
157                 /* altrimenti se è una pedina */
158                 else if (matrix[c.x + dir][c.y - 1] < 3){
159                     // MANGIO UNA PEDINA
160                     Point newPos = new Point(c.x + 2*dir, c.y - 2);
161                     CalculateMoves(CloneMatrixAndUpdate(matrix, c, newPos, new Point(c.x
     + dir, c.y -1)), newPos, new Vector<Point>(PrevC), score.concat("p"));
162                     stop = false;
163                 }
164             }
165         }
166 
167         /* se sono una dama posso guardare anche le mosse da poter fare nella direzione
    opposta alla mia */
168         if (imDama) {
169             /* vado a vedere la casella verso destra nella direzione opportuna */
170             if (CheckCoordinates(c.x - dir, c.y+1)){
171 
172             /* posso spostarmi sulla destra */
173                 if (matrix[c.x - dir][c.y + 1] == Dama.BLANK && canMove){
174                     Vector<Point> currC = new Vector<Point>(PrevC);
175                     Point newC = new Point(c.x - dir, c.y + 1);
176                     currC.add(newC);
177                     RegisterMoves(CloneMatrixAndUpdate(matrix, c, newC, null), currC,
    imDama, score);
178                     stop = false;
179                 }
180             /* se è occupata controllo se posso mangiare */
181                 else if (matrix[c.x - dir][c.y + 1] % 2 != myType % 2 &&
    CheckCoordinates(c.x- 2*dir, c.y + 2) && matrix[c.x - 2*dir][c.y + 2] == Dama.BLANK &&
    matrix[c.x - dir][c.y + 1] != Dama.BLANK){
182                     // MANGIO E IN BASE A COSA HO MANGIATO AGGIORNO LA STRINGA E LE
    EATEN
183                     Point newPos = new Point(c.x - 2*dir, c.y + 2);
184                     CalculateMoves(CloneMatrixAndUpdate(matrix, c, newPos, new Point(c.x
     - dir, c.y +1)), newPos, new Vector<Point>(PrevC), score.concat(matrix[c.x - dir][c.y +
     1] > 2 ? "d" : "p"));
185                     stop = false;
186                 }
187             }
188 
189             /* vado a vedere la casella verso sinistra nella direzione opportuna */
190             if (CheckCoordinates(c.x - dir, c.y - 1)){
```

/Users/nico/Documents/EP/Dama_v1/src/dama_italiana/MovesEngine.java

```
191
192             /* posso spostarmi sulla sinistra */
193                 if (matrix[c.x - dir][c.y - 1] == Dama.BLANK && canMove){
194                     Vector<Point> currC = new Vector<Point>(PrevC);
195                     Point newC = new Point(c.x - dir, c.y - 1);
196                     currC.add(newC);
197                     RegisterMoves(CloneMatrixAndUpdate(matrix, c, newC, null), currC,
    imDama, score);
198                     stop = false;
199                 }
200             /* se è occupata controllo se posso mangiare */
201                 else if (matrix[c.x - dir][c.y - 1] % 2 != myType % 2 &&
    CheckCoordinates(c.x- 2*dir, c.y - 2) && matrix[c.x - 2*dir][c.y - 2] == Dama.BLANK &&
    matrix[c.x - dir][c.y - 1] != Dama.BLANK){
202                     // MANGIO E IN BASE A COSA HO MANGIATO AGGIORNO LA STRINGA E LE
    EATEN
203                     Point newPos = new Point(c.x - 2*dir, c.y - 2);
204                     CalculateMoves(CloneMatrixAndUpdate(matrix, c, newPos, new Point(c.x
     - dir, c.y -1)), newPos, new Vector<Point>(PrevC), score.concat(matrix[c.x - dir][c.y -
     1] > 2 ? "d" : "p"));
205                     stop = false;
206                 }
207             }
208         }
209
210         /* se sono semplicemente rimasta ferma, ma con almeno una mossa (quindi una
    mangiata) registro la mia posizione */
211         if (stop && PrevC.size() > 1){
212             RegisterMoves(matrix, PrevC, imDama, score);
213         }
214
215     }
216 }
217
```

/Users/nico/Documents/EP/Dama_v1/src/ConsoleInterface/Main.java

```java
package ConsoleInterface;


public class Main {

    public static void main(String[] args) {
    // write your code here
        new ConsoleGame();



    }
}

```

/Users/nico/Documents/EP/Dama_v1/src/ConsoleInterface/ConsoleGame.java

```java
package ConsoleInterface;

import dama_italiana.Dama;
import dama_italiana.Move;

import java.awt.Point;
import java.util.Vector;

public class ConsoleGame {
    private Dama dama;
    private int Mode;

    public ConsoleGame(){
        this.dama = new Dama();
        System.out.println("Welecome in Dama Italiana game!");
        System.out.println("Please choose one game mode:");
        System.out.println(" 1: Human vs PC");
        System.out.println(" 2: PC vs PC");
        while(!CheckMode()){
           System.out.println("Bad input. Try again!");
        }
        Run();
    }

    public void Run() {
        Print();
        if (Mode == 1)
            while(true){
                try {
                    HumanMove();
                    dama.ExecutePCMove(0);
                    Print();
                } catch (UnsupportedOperationException e) {
                    System.out.println("\n\n\n" + e.toString().split(":")[1]);
                    return;
                }
            }
        else if (Mode == 2)
            while(true){
                try {
                    dama.ExecutePCMove(1);
                    dama.ExecutePCMove(0);
                    Print();
                } catch (UnsupportedOperationException e) {
                    System.out.println("\n\n\n" + e.toString().split(":")[1]);
                    return;
                }
            }
    }

    private boolean CheckMode(){
        String i = System.console().readLine();
        switch (Character.getNumericValue(i.charAt(0))) {
```

/Users/nico/Documents/EP/Dama_v1/src/ConsoleInterface/ConsoleGame.java

```
54              case 1: Mode = 1; return true;
55              case 2: Mode = 2; return true;
56              default: return false;
57          }
58      }
59
60      private void Print() {
61          int current_matrix[][] = dama.GetMatrix();
62          System.out.println("  ---------------------------------");
63          for(int i=0; i<8; i++) {
64              System.out.print("\033[1m"+ (8-i) +"\033[0m");
65              for (int j=0; j<7; j++) {
66                  System.out.print(" | " + StyleIt(current_matrix[i][j]));
67              }
68              System.out.print(" | " + StyleIt(current_matrix[i][7]) + " |\n");
69              System.out.println("  ---------------------------------");
70
71          }
72          System.out.println("\033[1m    A   B   C   D   E   F   G   H \033[0m");
73      }
74
75      private String StyleIt(int i){
76          switch (i){
77              case 0: return " ";
78              case 1: return "\u001B[34ma\u001B[0m";
79              case 2: return "\u001B[37ma\u001B[0m";
80              case 3: return "\u001B[34m@\u001B[0m";
81              case 4: return "\u001B[37m@\u001B[0m";
82              default: return "";
83          }
84      }
85
86      public void HumanMove() throws UnsupportedOperationException {
87          Vector<Move> LegalMoves = dama.RetriveMoves(0);
88
89          if (LegalMoves.size() == 0)
90              throw new UnsupportedOperationException("No moves available! BLACK Wins!");
91          //PrintMoves(LegalMoves);
92          System.out.println("It's your turn, type your move.");
93          Vector<Point> i_parsed = ParseInput();
94
95          while(!dama.CheckMoveAndExecute(LegalMoves, i_parsed)) {
96              System.out.println("Illegal moves");
97              dama_italiana.DamaTree.PrintMoves(LegalMoves);
98              i_parsed = ParseInput();
99          }
100
101         Print();
102         System.gc(); // Chiamo il Garbage Collector esplicitamente
103     }
104
105
106     private Vector<Point> ParseInput() {
```

/Users/nico/Documents/EP/Dama_v1/src/ConsoleInterface/ConsoleGame.java

```
107         String in = null;
108         while(true) {
109             try {
110                 in = System.console().readLine();
111 
112 
113                 Vector<Point> v = new Vector();
114                 String[] splitted = in.split(" ");
115 
116                 for (String s : splitted) {
117 
118                     int x = 8 - Character.getNumericValue(s.charAt(1));
119 
120                     CheckRow(x);
121 
122                     v.add(new Point(x, L2C(s.charAt(0))));
123                 }
124 
125                 return v;
126 
127             }
128             catch (IllegalArgumentException e) {
129                 System.out.println(e.toString().split(":")[1]);
130             }
131             catch (StringIndexOutOfBoundsException e){
132                 System.out.println("Bad input: check the format!!!");
133             }
134         }
135     }
136 
137     static private int L2C(char input) throws IllegalArgumentException {
138         switch (Character.toUpperCase(input)){
139             case 'A': return 0;
140             case 'B': return 1;
141             case 'C': return 2;
142             case 'D': return 3;
143             case 'E': return 4;
144             case 'F': return 5;
145             case 'G': return 6;
146             case 'H': return 7;
147             default: throw new IllegalArgumentException("ERROR: Possible wrong column
    character");
148         }
149     }
150 
151     static private void CheckRow(int x) throws IllegalArgumentException {
152         if (x < 0 || x > 7)
153             throw new IllegalArgumentException("ERROR: Possible wrong row number");
154     }
155 
156 
157 
158 }
```

/Users/nico/Documents/EP/Dama_v1/src/GUI/Form.java

```java
package GUI;

import javax.swing.*;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.Dimension;

public class Form extends JFrame {
    private int SQ_DIM = 80;
    public JLabel statusBar = new JLabel();
    private Board b = new Board(statusBar);

    public Form(String txt){
        super(txt);
        setLocation(300, 15);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(this.SQ_DIM * 8, this.SQ_DIM * 8 + 61);
        setResizable(false);


        /* creo il menu */

        JMenuBar menuBar = new JMenuBar();
        JMenu menu = new JMenu("Game");
        menuBar.add(menu);

        JMenuItem HvsC_m = new JMenuItem("Play Human vs CPU");
        HvsC_m.addMouseListener(new MouseAdapter() {
            @Override
            public void mousePressed(MouseEvent e) {
                b.Reset_Game();
                b.HUMANvsCPU = true;
            }
        });

        JMenuItem CvsC_m = new JMenuItem("Play CPU vs CPU");
        CvsC_m.addMouseListener(new MouseAdapter() {
            @Override
            public void mousePressed(MouseEvent e) {
                new Thread(() -> {
                    b.Reset_Game();
                    b.CPUvsCPU = true;
                    b.Play_CvsC();
                }).start();

            }
        });

        JMenuItem Restart_m = new JMenuItem("Restart Game");
        Restart_m.addMouseListener(new MouseAdapter() {
            @Override
            public void mousePressed(MouseEvent e) {
                b.Reset_Game();
```

/Users/nico/Documents/EP/Dama_v1/src/GUI/Form.java

```
54              }
55          });
56
57          menu.add(HvsC_m);
58          menu.add(CvsC_m);
59          menu.add(Restart_m);
60
61          this.setJMenuBar(menuBar);
62
63          this.getContentPane().add(b);
64
65          /* aggiungo la status Bar */
66          statusBar.setPreferredSize(new Dimension(this.SQ_DIM * 8, 16));
67          this.add(statusBar, java.awt.BorderLayout.SOUTH);
68      }
69  }
70
```

/Users/nico/Documents/EP/Dama_v1/src/GUI/Main.java

```java
package GUI;

public class Main {
    public static void main(String[] args)
    {
        GUI.Form f = new GUI.Form("DamaGame");
        f.setVisible(true);
    }
}

```

/Users/nico/Documents/EP/Dama_v1/src/GUI/Board.java

```java
package GUI;

import dama_italiana.Dama;
import dama_italiana.Move;

import javax.swing.*;
import java.awt.*;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.util.ArrayList;
import java.util.Vector;

public class Board extends JPanel {
    private static int SQ_DIM = 80;
    private static int IN_R = 20;
    private dama_italiana.Dama Dama;
    private int[][] m = new int[8][8];

    public boolean HUMANvsCPU = false;
    public boolean CPUvsCPU = false;

    /* Logic */
    private Vector<Move> LegalMoves = null;
    private Vector<Point> current_m_coo = new Vector<Point>();



    /* Graphics */
    private Color d_brown = new Color(106, 53, 18);
    private Color l_brown = new Color(255, 225, 184);
    private ArrayList<Rectangle> coordinates_list = new ArrayList<>();
    private Point Start;
    private Point End;
    private JLabel statusBar;



    public Board(JLabel sB){
        super(true);
        setBackground(l_brown);
        setDoubleBuffered(true);

        statusBar = sB;
        /* inizializzo la lista delle coordinate per le correlazioni */
        for (int row = 0; row < 8; row++)
            for (int col = 0; col < 8; col++)
                if ((col % 2) == (row % 2))
                {
                    coordinates_list.add(new Rectangle(col * 80, row * 80, 80, 80));
                }

        this.addMouseListener(new MouseAdapter() {
            public void mousePressed(MouseEvent e) {
```

/Users/nico/Documents/EP/Dama_v1/src/GUI/Board.java

```
54                  if (HUMANvsCPU){
55                      for(Rectangle r : coordinates_list)
56                          if(r.contains(e.getPoint())){
57                             Start = new Point(r.y / SQ_DIM, r.x / SQ_DIM);
58                              break;
59                          }
60                  }
61              }
62          });
63
64          this.addMouseListener(new MouseAdapter() {
65              public void mouseReleased(MouseEvent e) {
66                  if (HUMANvsCPU && (m[Start.x][Start.y] == Dama.WHITE || m[Start.x][Start
    .y] == Dama.D_WHITE))
67                      for (Rectangle r : coordinates_list)
68                          if (r.contains(e.getPoint())){
69                              End = new Point(r.y / SQ_DIM, r.x / SQ_DIM);
70                              moveWhite();
71                              break;
72                          }
73              }
74          });
75      }
76
77      private void moveWhite(){
78          if (Start.equals(End))
79              return;
80
81          if (current_m_coo.isEmpty()){
82              LegalMoves = Dama.RetriveMoves(0);
83              if (LegalMoves.size() == 0){
84                  JOptionPane.showMessageDialog(null,"No moves available! BLACK Wins!");
85                  Reset_Game();
86                  return;
87              }
88              current_m_coo.add(Start);
89              current_m_coo.add(End);
90          }
91          else {
92              current_m_coo.add(End);
93          }
94
95          if (Dama.CheckMoveAndExecute(LegalMoves,current_m_coo)){
96              m = Dama.CopyMatrix();
97              System.gc();
98              current_m_coo.clear();
99              this.repaint();
100             try {
101                 Dama.ExecutePCMove(0);
102             } catch (UnsupportedOperationException e) {
103                 JOptionPane.showMessageDialog(null,e.toString().split(":")[1]);
104                 Reset_Game();
105                 return;
```

/Users/nico/Documents/EP/Dama_v1/src/GUI/Board.java

```
106             }
107             m = Dama.CopyMatrix();
108             this.repaint();
109             System.gc();
110             CheckWins();
111             return;
112         }
113 
114         for(Move mo : LegalMoves)
115             if (Check(mo.Sequence, current_m_coo)){
116                 if (m[End.x][End.y] == Dama.BLANK)
117                 {
118                     m[End.x][End.y] = m[Start.x][Start.y];  //switch pedina
119                     m[Start.x][Start.y] = Dama.BLANK;
120 
121                     if (Math.abs(Start.x - End.x) != 1){
122                         m[Start.x + ((End.x - Start.x) / 2)][Start.y + ((End.y - Start.y
    )/2)] = Dama.BLANK;
123                     }
124 
125                 }
126 
127                 if (End.x == 0)
128                     m[End.x][End.y] = Dama.D_WHITE;
129                 this.repaint();
130                 return;
131             }
132 
133         if (current_m_coo.size() == 2){
134             current_m_coo.clear();
135         }
136 
137         // a questo punto la mossa era illegale
138         JOptionPane.showMessageDialog(null, "Move not allowed!");
139         return;
140 
141     }
142 
143     private boolean Check (Vector<Point> LegalM, Vector<Point> CurrM){
144         for (int i = 0; i < CurrM.size(); i++)
145             if (!LegalM.elementAt(i).equals(CurrM.elementAt(i)))
146                 return false;
147         return true;
148     }
149 
150     public void Play_CvsC(){
151         long Tempo1, Tempo2;
152         statusBar.setText("CPU vs CPU mode");
153         for (int i = 0; i < 200; i++){
154             try {
155                 Dama.ExecutePCMove(1);
156                 m = Dama.GetMatrix();
157                 this.repaint();
```

/Users/nico/Documents/EP/Dama_v1/src/GUI/Board.java

```
158
159                 Tempo1 = System.currentTimeMillis();
160                 Dama.ExecutePCMove(0);
161                 Tempo2 = System.currentTimeMillis();
162                 System.out.println((Tempo2-Tempo1)+"ms");
163                 m = Dama.GetMatrix();
164                 this.repaint();
165             } catch (UnsupportedOperationException e) {
166                 JOptionPane.showMessageDialog(null, e.toString().split(":")[1]);
167                 Reset_Game();
168                 return;
169             }
170         }
171         JOptionPane.showMessageDialog(null, "DRAWN GAME!");
172         Reset_Game();
173         return;
174     }
175
176     public void Reset_Game(){
177         this.Dama = new Dama();
178         this.m = Dama.CopyMatrix();
179         CPUvsCPU = HUMANvsCPU = false;
180         current_m_coo.clear();
181         LegalMoves = null;
182         this.paint(this.getGraphics());
183         statusBar.setText("Please select a mode in the menu");
184     }
185
186     private void CheckWins() {
187             int numBLACK = 0, numWHITE = 0;
188             for (int i = 0; i < 8; i++)
189                 for (int j = 0; j < 8; j++)
190                 {
191                     if (m[i][j] == Dama.WHITE || m[i][j] == Dama.D_WHITE)
192                         numWHITE++;
193                     else
194                     if (m[i][j] == Dama.BLACK || m[i][j] == Dama.D_BLACK)
195                         numBLACK++;
196                 }
197
198         if (numBLACK == 0){
199             JOptionPane.showMessageDialog(null, "No moves available for PC! WHITE Wins!"
    );
200             Reset_Game();
201         }
202
203         if (numWHITE == 0){
204             JOptionPane.showMessageDialog(null, "No moves available for PC! BLACK Wins!"
    );
205             Reset_Game();
206         }
207     }
208
```

/Users/nico/Documents/EP/Dama_v1/src/GUI/Board.java

```
209     public void paintComponent(Graphics g){
210         int x, y;
211         setDoubleBuffered(true);
212         Graphics2D g2 = (Graphics2D) g;
213         super.paintComponent(g2);
214         g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,RenderingHints.
    VALUE_ANTIALIAS_ON);
215 
216         /* disegno la scacchiera */
217         g2.setColor(d_brown);
218         for (int i = 0; i < 8; i++)
219             for (int j = 0; j < 8; j++)
220                 if((i % 2) == (j % 2))
221                     g2.fillRect(j * SQ_DIM, i * SQ_DIM, SQ_DIM, SQ_DIM);
222 
223         /* disegno le pedine */
224         for(int i = 0; i < 8; i++)
225             for(int j = 0; j < 8; j++)
226                 switch (m[i][j]){
227                     case 0:             break;
228                     case 1:             g2.setColor(Color.BLACK);
229                                         g2.fillOval(j * SQ_DIM + 5, i * SQ_DIM + 5,
    SQ_DIM - 10, SQ_DIM - 10);
230                                         break;
231                     case 2:             g2.setColor(new Color(224, 226, 213));
232                                         g2.fillOval(j * SQ_DIM + 5, i * SQ_DIM + 5,
    SQ_DIM - 10, SQ_DIM - 10);
233                                         break;
234                     case 3:             x = j * SQ_DIM;
235                                         y = i * SQ_DIM;
236                                         g2.setColor(Color.BLACK);
237                                         g2.fillOval(x + 5, y + 5, SQ_DIM - 10, SQ_DIM -
    10);
238                                         g2.setColor(Color.RED);
239                                         g2.fillOval(x + IN_R, y + IN_R, 2 * IN_R, 2 *
    IN_R);
240                                         break;
241                     case 4:             x = j * SQ_DIM;
242                                         y = i * SQ_DIM;
243                                         g2.setColor(new Color(224, 226, 213));
244                                         g2.fillOval(x + 5, y + 5, SQ_DIM - 10, SQ_DIM -
    10);
245                                         g2.setColor(Color.DARK_GRAY);
246                                         g2.fillOval(x + IN_R, y + IN_R, 2 * IN_R, 2 *
    IN_R);
247                                         break;
248                 }
249     }
250 
251 }
252 
```