18 czerwca 2013

Artur Buchwald
Wojciech Przybylski
Marcin Wcisło
I11-1

Sprawozdanie z laboratorium sztucznej inteligencji

Informatyka, sem. VI

I. Opis zadania

Przydzielona została nam implementacja afrykańskiej gry Dara. Jest to gra logiczna, która polega na ułożeniu tak zwanych trójek kamieni w rzędzie i blokowaniu przeciwnika czyli uniemożliwieniu mu ustawienie trójki kamieni w rzędzie. Wygrywa ten który zablokuje przeciwnika, lub zabierze mu wszystkie kamienie.

Gra stworzona przez nas ma przejrzysty interfejs, który jest bardzo intuicyjny i pozwala na wprowadzanie danych muszką. W pierwszej fazie gry gracz ustawia 12 kamieni na planszy 5x6 na przemian z komputerem, po ustawieniu 12 kamieni rozpoczyna się druga faza gry. Gdy gracz zablokuje kamienie rozłożone przez komputer zostanie poinformowany przez system iż został zwycięzcom pojedynku, w odwrotnym przypadku system poinformuje go iż przegrał pojedynek.

System gry kontroluje cały pojedynek, nie można oszukiwać, trzeba przestrzegać zasady gry, które zostały zaimplementowane w systemie. Po każdym złym ruchu system informuje gracza o nie możliwym ruchu podając przyczynę. Gdy gracz gra pierwszy raz może skorzystać z systemu pomocy i poznać zasady gry.

II. Założenia realizacyjne.

1. Algorytmy wykorzystane do stworzenia aplikacji:

Algorytm *MIN

Dane: b – obiekt klasy ***Board***, d – aktualna głębokość w drzewie decyzyjnym.
Wynik: m – obiekt klasy ***Move***, ruch po którym stan planszy został oceniony najniżej.
Metoda:

1. Sprawdź czy stan gry obiektu b nie jest stanem końcowym lub d jest większe od ustawionej wcześniej wartości. Jeżeli tak to pobierz ostatni ruch, ustal dla niego wartość funkcji użyteczności zwróć go.
2. Utwórz nowy obiekt ***Move*** na podstawie obiektu b.
3. Dla każdego możliwego stanu gry pochodzącego od b sprawdź wartość funkcji użyteczności dla ostatniego ruchu korzystając z funkcji ***MAX*** dla głębokości d + 1. Do zmiennej value przyporządkuj ruch o większej wartości funkcji użyteczności.

Złożoność: $O(2^n)$.

Implementacja:

```
private static Move minValue(Board board, int depth)
{
   if (board.isEnd() || depth >= MAX_DEPTH)
   {
      Move move = board.getLastMove();
      if (move == null)
      {
         move = new Move((Board)board.DeepClone(), BoardSide.MAX);
      }
      move.setRating(UtilityRating.getRating(board));
      return move;
   }
   Move value = new Move(board, BoardSide.MIN);
```

```
    foreach (Board possibleBoard in board.getPossibleBoards())
    {
        Move currentMove = maxValue(possibleBoard, depth + 1);
        value = currentMove.getRating() <= value.getRating() ? currentMove : value;
    }
    return value;
}
```

Algorytm *MAX

Dane: b – obiekt klasy ***Board***, d – aktualna głębokość w drzewie decyzyjnym.
Wynik: m – obiekt klasy ***Move***, ruch po którym stan planszy został oceniony najniżej.
Metoda:

1. Sprawdź czy stan gry obiektu b nie jest stanem końcowym lub d jest większe od ustawionej wcześniej wartości. Jeżeli tak to pobierz ostatni ruch, ustal dla niego wartość funkcji użyteczności zwróć go.
2. Utwórz nowy obiekt ***Move*** na podstawie obiektu b.
3. Dla każdego możliwego stanu gry pochodzącego od b sprawdź wartość funkcji użyteczności dla ostatniego ruchu korzystając z funkcji ***MIN*** dla głębokości d + 1. Do zmiennej value przyporządkuj ruch o większej wartości funkcji użyteczności.

Złożoność: $O(2^n)$.

Implementacja:

```
private static Move maxValue(Board board, int depth)
{
    if (board.isEnd() || depth >= MAX_DEPTH)
    {
        Move move = board.getLastMove();
        if (move == null)
        {
            move = new Move((Board)board.DeepClone(), BoardSide.MAX);
        }

        move.setRating(UtilityRating.getRating(board));
        return move;
    }

    Move value = new Move(board, BoardSide.MAX);
    foreach (Board possibleBoard in board.getPossibleBoards())
    {
        Move currentMove = minValue(possibleBoard, depth + 1);
        value = currentMove.getRating() >= value.getRating() ? currentMove : value;
    }

    return value;
}
```

2. Funkcja użyteczności.

Aby algorytm dobrze był wykorzystany w grze potrzebna jest funkcja użyteczności. W naszej aplikacji uwzględniliśmy cztery punkty, dzięki którym system zawsze wybierze najlepsze rozwiązanie. Po sprawdzeniu czerech punktów użyteczności komputer wykona ruch który w jak największym stopniu ma zaszkodzić przeciwnikowi.

1. Wstawienie kamienia w pole które daje trójkę i w największym stopniu blokuje ruchy przeciwnika.
2. Przewidzenie trzech przyszłych ruchów przeciwnika i w postawienie kamienia w pole, które utrudnia ułożenie trójki przez przeciwnika.
3. Kamienie powinny być ustawiane w szeregu nie tworzące wolnych miejsc między sobą, nie dając się zablokować (chyba ze już ułoży się trójkę).
4. Wykrywanie możliwości zablokowania kamienia przez przeciwnika.

3. Języki programowania, narzędzia informatyczne i środowiska używane do implementacji systemu.

Interfejs użytkownika oraz wszystkie funkcje programu zostały zaimplementowane w języku C#, XAML po ówczesnym skonsultowaniu z prowadzącym zajęcia. Środowisko programistyczne wykorzystane do stworzenia aplikacji to Visual Studio 2012. Grafika gry stworzona w programie graficznym GIMP.

III. Podział prac

Autor	**Podzadanie**
Artur Buchwald	1. Implementacja funkcji aplikacji 2. Implementacja funkcji użyteczności 3. Testy aplikacji
Wojciech Przybylski	1. Stworzenie grafiki gry 2. Opracowanie funkcji użyteczności 3. Testy aplikacji
Marcin Wcisło	1. Opracowanie algorytmu MIN-MAX 2. Implementacja algorytmu MIN-MAX 3. Testy aplikacji

IV. Opis implementacji

Wszystkie klasy oznaczone są adnotacją [Serializable] ponieważ jest to jeden ze sposobów na umożliwienie wykonania tzw. głębokiej kopii obiektu w języku C#.

Klasa *ExtensionMethods* – klasa ta zawiera jedną metodę statyczną, która pozwala na wykonywanie tzw. "głębokich" kopii obiektów (tzn. wykonywane jest faktyczne kopiowanie danych i struktur, a nie pobieranie ich referencji).

Klasa *BoardSide* – typ wyliczeniowy w którym dopuszczalne są następujące wartości: MAX, MIN, NONE.

Klasa *BoardObjectType* – typ wyliczeniowy w którym dopuszczalne są następujące wartości: PAWN, NONE.

Klasa *BoardObject* – obiekty tej klasy reprezentują obiekty mogące znajdować się na planszy do gry. Wśród tych obiektów znajdują się pionki (gracza MIN i gracza MAX) oraz puste pola które nie należą do nikogo. Ponadto obiekty tej klasy dodatkowo zawierają pole boolowskie locked za pomocą którego można stwierdzić czy dany pionek został zablokowany (patrz zasady gry). Jest to klasa abstrakcyjna, którą dziedziczona jest przez klasę ***None*** oraz ***Pawn*.**

Klasa *None* – klasa dziedziczy po BoardObject. Obiekty tej klasy posiadają tylko jedną metodą, konstruktor bez argumentów który ustawia właściwości boardObjectSide oraz boardObjectType na odpowiednie wartości.

Klasa *Pawn* – klasa dziedziczy po BoardObject. Obiekty tej klasy posiadają tylko jedną metodą, konstruktor z argumentem typu ***BoardSide*** który ustawia właściwości boardObjectSide oraz boardObjectType na odpowiednie wartości.

Klasa BoardSquare – obiekty tej klasy reprezentują dane pole na planszy. Każdy obiekt tej klasy ma swoje koordynaty oraz obiekt który się na nim znajduje (***None*** lub ***Pawn*).**

Klasa *Board*– obiekty tej klasy reprezentują plansze na której znajduję się pionki (czyli stan gry). W obiektach tej klasy przechowywane są informacje takie jak: wielkość planszy do gry, dwuwymiarowa tablica obiektów klasy ***BoardSquare*,** kogo jest aktualna tura (MIN czy MAX) oraz jaki został wykonany ostatni ruch. Dla tych właściwości zostały utworzone odpowiednie gettery i settery. Ponadto zostały zaimplementowane metody o następujących nazwach: getPossibleBoards (zwraca tablicę obiektów klasy ***Board*** które można uzyskać wykonując ruch podczas tury aktualnego gracza), findAndLockThrees (znajduje i blokuje "trójki", patrz zasady gry), getMaxSidePawnCount oraz getMinSidePawnCount (zwraca liczbę pionków graczy odpowiedniej "strony").

Klasa *Minimax* – zawiera statyczną metodę getNextMove przyjmującą jako argument obiekt klasy ***Board*** która zwraca najbardziej odpowiedni ruch wg. algorytmu MINMAX dla gracza który ma aktualną turę. Zmienna statyczna MAX_DEPTH określa do jakiej głębokości drzewa decyzyjnego maksymalnie możemy się odwołać. Prywatne metody minValue oraz maxValue są główną implementacją algorytmu MINMAX. Dzięki logice w nich zawartej wybierane są najbardziej odpowiednie ruchy który należy wykonać dla gracza MIN lub MAX. Decyzja podejmowana jest na podstawie wartości generowanej przez metodę obecną w implementacji klasy ***UtilityRating*.**

Klasa *UtilityRating* – na podstawie logiki zaimplementowanej w tej klasy zwracana jest wartość liczbowa dla obiektu klasy ***Board*.** Dzięki wyrafinowanym algorytmom liczącym określane są liczby "trójek", "dwójek" na planszy dla danego gracza oraz odległości innych pionków do uzyskanie takowych. Do tych faktów przyporządkowane są odpowiednie wagi dzięki określające stopień ważności.

Klasa *InvalidMoveExecutionExpection* – wyjątek wyrzucany w momencie gdy wykonany zostanie ruch przesunięcia pionka poza planszę.

Klasa *NoneTurnException* – wyjątek wyrzucany w momencie gdy w jakiś sposób tura użytkownika zostanie ustawiona na NONE**.**

Klasa *PawnOverrideException* – wyjątek wyrzucany w momencie gdy w jakiś sposób będziemy chcieli postawić pionek na polu na którym jakiś pionek już stoi.

Opis ważniejszych funkcji zaimplementowanych w programie:

public bool isEnd() – funkcja sprawdza, czy aktualny stan gry jest stanem końcowym tj. sprawdzane jest czy każda ze stron ma do dyspozycji trzy pionki którymi może się poruszać. Metoda zwraca wartość boolwską.

public bool isEnd(int iswin) – funkcja sprawdza, czy aktualny stan gry jest stanem końcowym tj. sprawdzane jest czy każda ze stron ma do dyspozycji trzy pionki którymi może się poruszać, oraz wykazanie wygranego w tej partii gry. Metoda zwraca wartość int wskazując na wygranego.

public List<Board> getPossibleBoards() – funkcja zwraca listę obiektów klasy ***Board*** które można uzyskać wykonując ruch podczas tury aktualnego gracza. Lista obiektów klasy ***Board*** konstruowana jest w następujący sposób: pobierane są wszystkie niezablokowane pionki gracza posiadającego teraz turę, pobierane są wszystkie wolne pola na planszy, dobierane są wszystkie możliwe kombinacje pionków gracza i wolnych pól na planszy. Metoda zwraca listę obiektów klasy ***Board.***

private void lockThrees(BoardSquare square) – funkcja znajduje i blokuje wszystkie powstałe na planszy "trójki" pionków. Metoda nie zwraca żadnego wyniku.

public void findFours() – funkcja znajduje i uniemożliwia powstanie na planszy "czwórki" pionków. Metoda nie zwraca żadnego wyniku.

public static double getRating(Board board) – na podstawie sformułowanej funkcji użyteczności obliczana jest i zwracana wartość dla podanego stanu gry (obiekt klasy ***Board***). Metoda zwraca wartość typu double.

public static T DeepClone<T>(this T a) – dzięki tej metodzie wszystkie klasy oznaczone jako Serializable mają metodę DeepClone która umożliwia dokładne skopiowanie obiektu klasy (a nie zawierających się w niej referencji). Metoda zwraca kopię obiektu typu na którym została wykonana.

public void setCurrentObject(BoardObject currentObject) – metoda ta umożliwia ustawianie pionka na dane pole. W momencie gdy będziemy chcieli ustawić pionek na istniejący już pionek zostanie wyrzucony wyjątek. Metoda nie zwraca żadnego wyniku.

public int executeMove() – dzięki tej metodzie dokonywany jest ruch na planszy operując na wszystkich potrzebnych do wykonania ruchu klasach. W przypadku ruchu komputera również wykorzystywana jest tutaj j funkcja użyteczności. Metoda ta zwraca int, określając błąd bądź poprawny ruch.

public int findAndRateTwos() – metoda ta jest używana w funkcji użyteczności. Została zaimplementowana by nadać odpowiednią punktację ruchowi odpowiadającemu stworzenie „dwójki". Metoda zwraca wyliczony dla danego ruchu ocenę punktową.

public int findAndRateCloseToTwos() – metoda ta jest używana w funkcji użyteczności. Została zaimplementowana by nadać odpowiednią punktację ruchowi odpowiadającemu znalezienia się pionka w pozycji umożliwiającej w kolejnym ruchu utworzenie „dwójki". Metoda zwraca wyliczony dla danego ruchu ocenę punktową.

public int findAndRateCloseToThrees()–metoda ta jest używana w funkcji użyteczności. Została zaimplementowana by nadać odpowiednią punktację ruchowi odpowiadając stworzenie w kolejnym ruchu „trójki". Metoda zwraca wyliczony dla danego ruchu ocenę punktową.

public int findAndRateThrees() –metoda ta jest używana w funkcji użyteczności. Została zaimplementowana by nadać odpowiednią punktację ruchowi odpowiadającemu stworzenie „trójki". Metoda zwraca wyliczony dla danego ruchu ocenę punktową.

public int findAndRateCloseMoves() – metoda ta jest używana w funkcji użyteczności. Została zaimplementowana by nadać odpowiednią punktację ruchowi odpowiednią dla odległości od pionka. Metoda zwraca wyliczony dla danego ruchu ocenę punktową.

V. Użytkowanie i testowanie

1. Testy aplikacji:

Przeprowadzone testy:

1. Uruchomienie aplikacji w Windows XP/7/8
2. Test wydajnościowy
3. Rozpoczęcie pierwszej fazy gry (rozstawienie na przemian z komputerem 12 kamieni)
4. Celowa gra nieprzepisowa
5. Celowe przegranie z komputerem
6. Wygranie gry
7. System pomocy
8. Zamykanie aplikacji

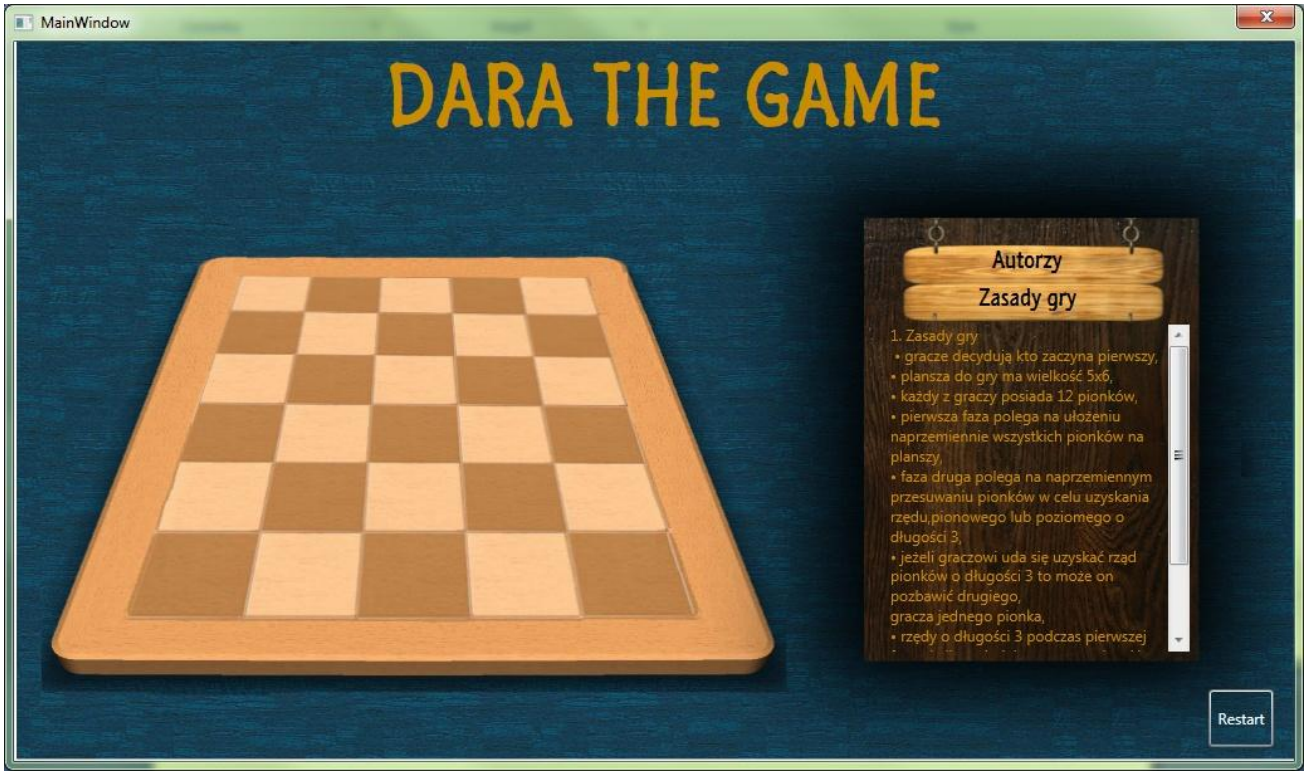
Numer testu	Co podlega ocenie	Ocena testu
1.	Czy aplikacja została uruchomiona w wymienionych systemach Odpowiedź: Tak lub Nie (jeżeli 'Nie' napisać jaki system)	
2.	Czy obciążenie systemu jest duże, np. 25% zasobów komputera? Odpowiedź: Tak lub Nie (jeżeli 'Nie' podać parametry komputera)	
3.	Czy wystąpił jakiś problem? Odpowiedź: Tak lub Nie (jeżeli 'Tak' to jaki problem się pojawił)	
4.	Czy wyświetlane są powiadomienia? Odpowiedź: Tak lub Nie (jeżeli 'Nie' to w jakim przypadku)	
5.	Czy wyświetlona informacja o przegranej? Odpowiedź: Tak lub Nie	
6.	Czy została wyświetlona informacja o wygranej Odpowiedź: Tak lub Nie	

7.	Czy system pomocy wyświetla się Odpowiedź: Tak lub Nie (jeżeli 'Nie' to kiedy się nie wyświetla)	
8.	Czy aplikacja została zamknięta poprawnie? Odpowiedź: Tak lub Nie (jeżeli 'Nie' to podać system operacyjny oraz opisać sytuację zamykania programu.	

Testy zostały przeprowadzone kilka krotnie przez wykonawców projektu jak i użytkowników spoza projektu. Wszystkie testy przebiegły pomyślnie, brak jakichkolwiek błędów, jak i nieoczekiwanego stanu gry.

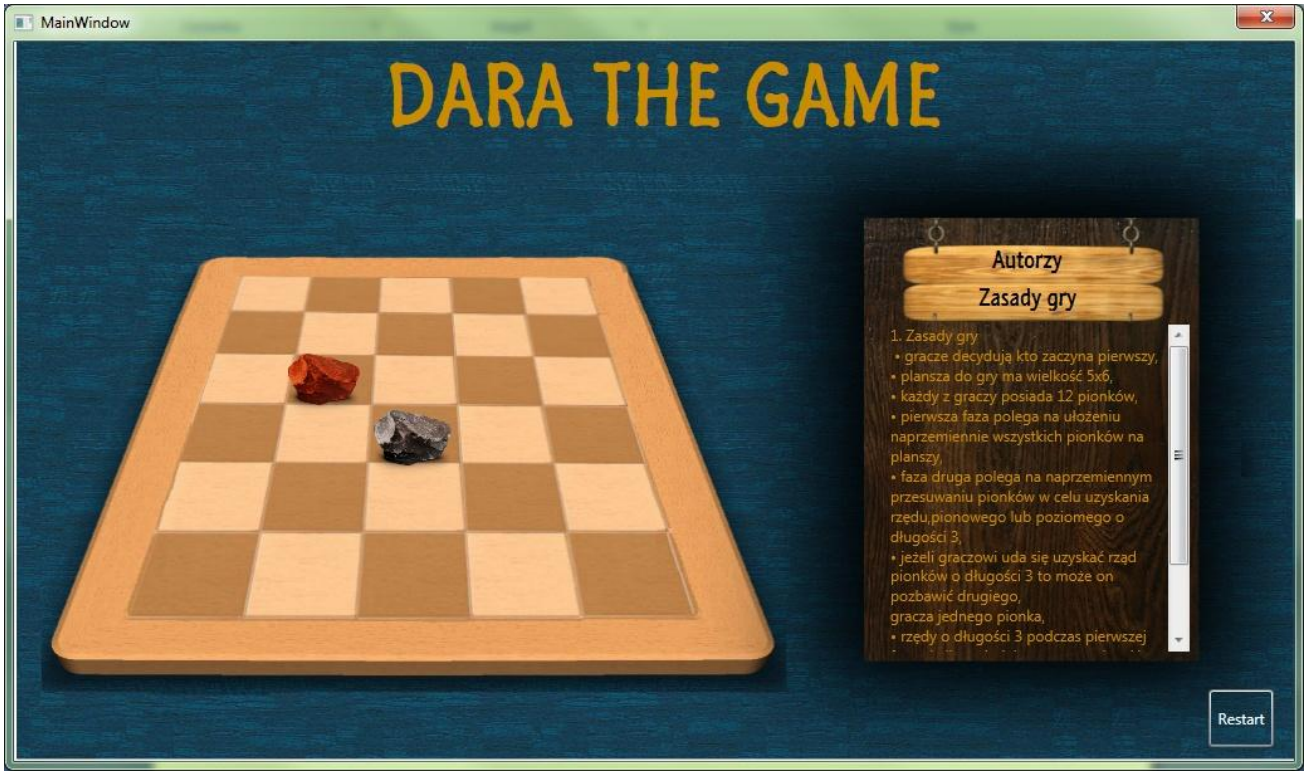
2. Instrukcja obsługi gry

Użytkownikowi dzięki przyjaznemu interfejsowi aplikacji pozostaje wyklikiwać pożądane akcje. Na samym początku użytkownik powinien zapoznać się z regułami gry klikając odpowiednio na dany przycisk



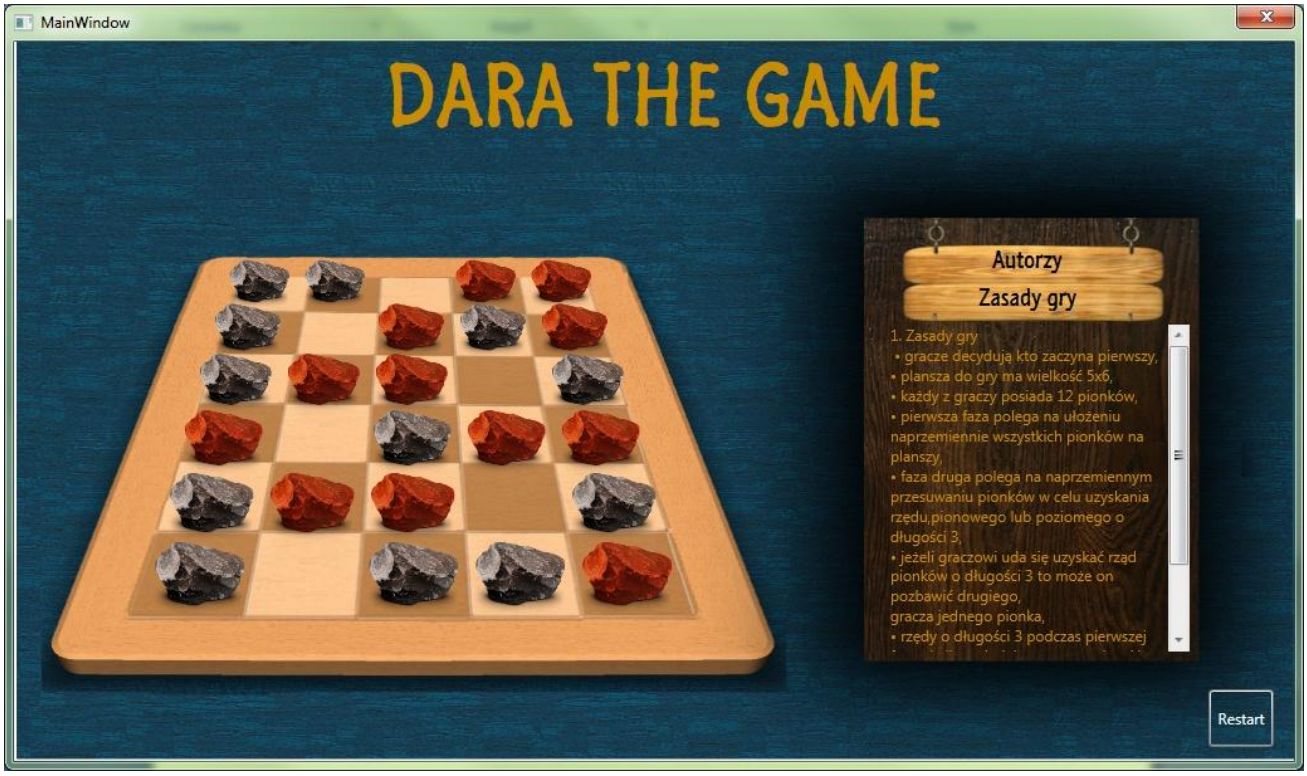
Rys. 1. Interfejs gry

Za symulowanie rozpoczęcia gry odbywa się przez wybranie pionka na danym polu. Gdzie od razu zostanie umieszczony, komputer automatycznie odpowiada na ruch gracza:



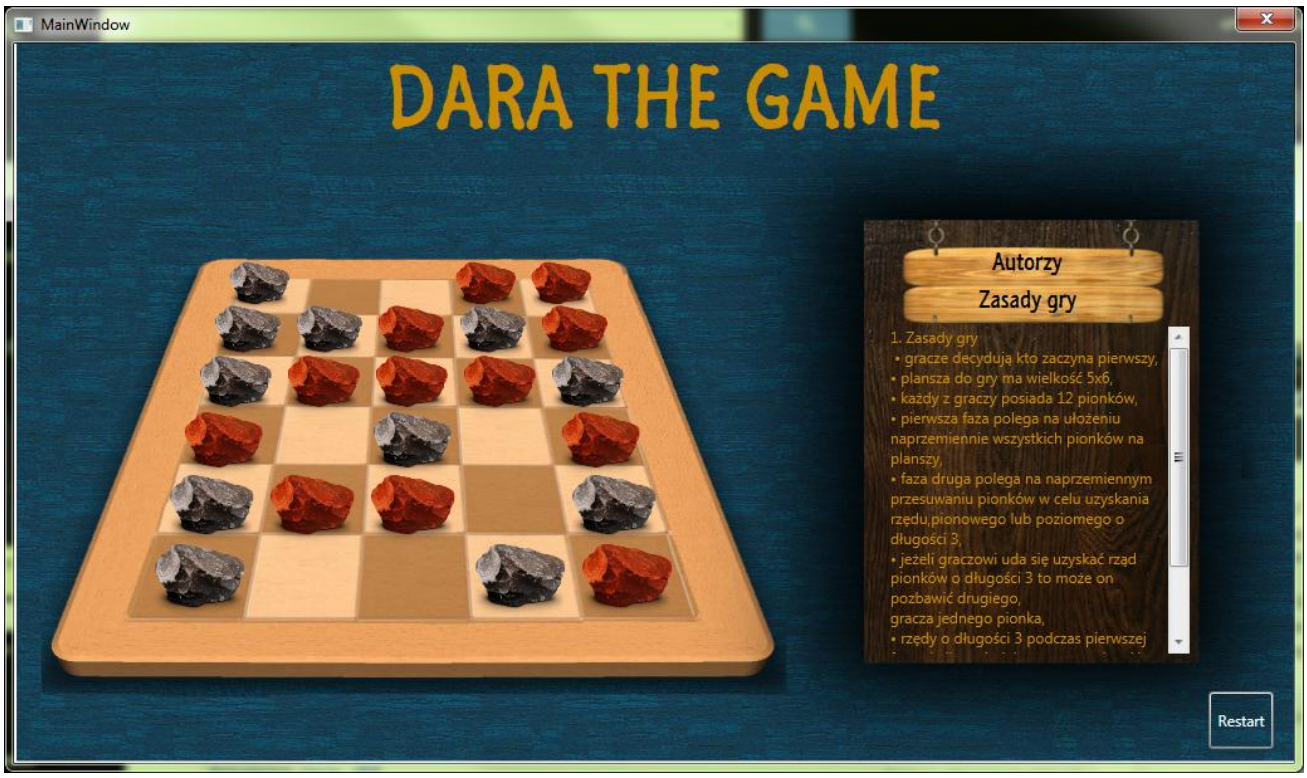
Rys. 2. Rozpoczęcie gry

Ten krok wykonujemy dwunastokrotnie w celu zapełnienia planszy 12 pionkami:



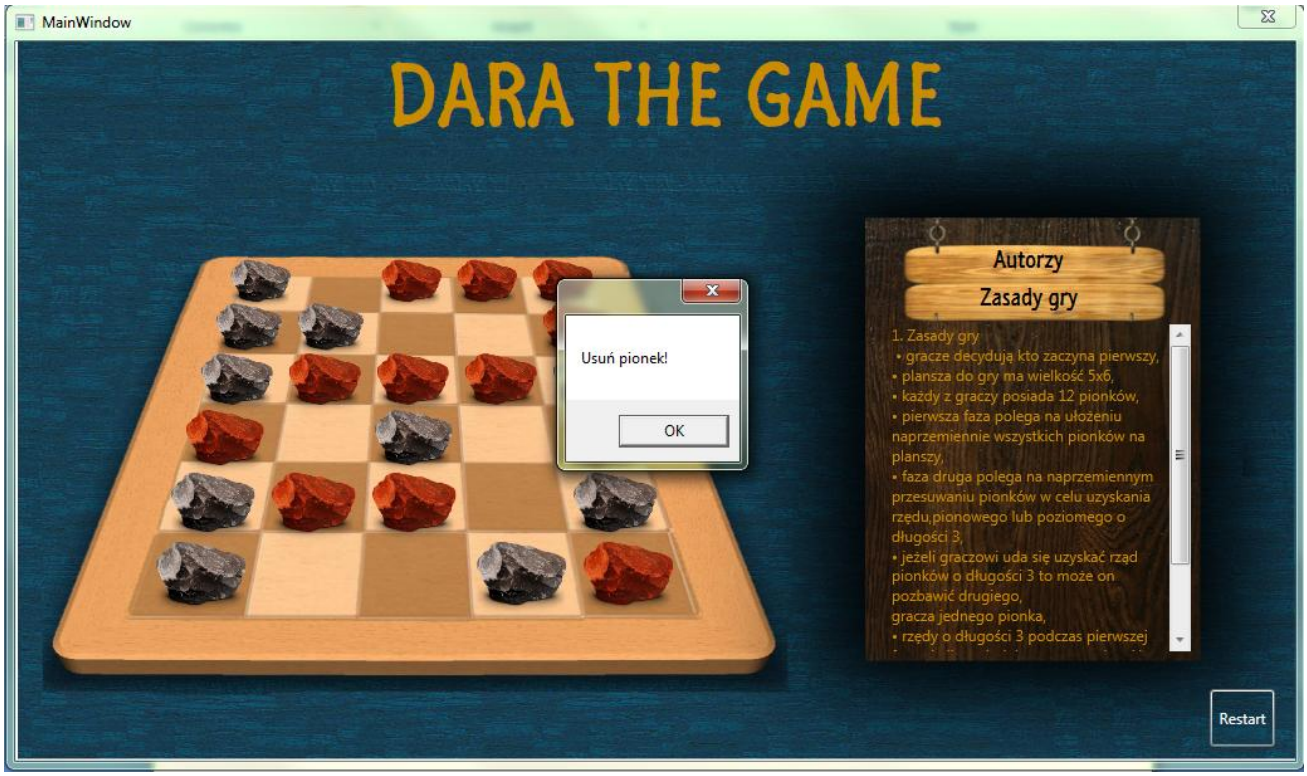
Rys. 3. Wypełnienie planszy 12 pionkami

Gracza pionki to czarne kamienie jak można zauważyć podczas rozmieszczania. W celu wykonania ruchu w fazie drugiej gry, należy kliknąć na jeden z naszych pionków i wybrać wolne pole znajdujące się przy pionku. W przypadku gdyby był to ruch niemożliwy do wykonania pionek nie przemieści się, należy wybrać wówczas poprawną. Komputer automatycznie odpowiada na ruch:



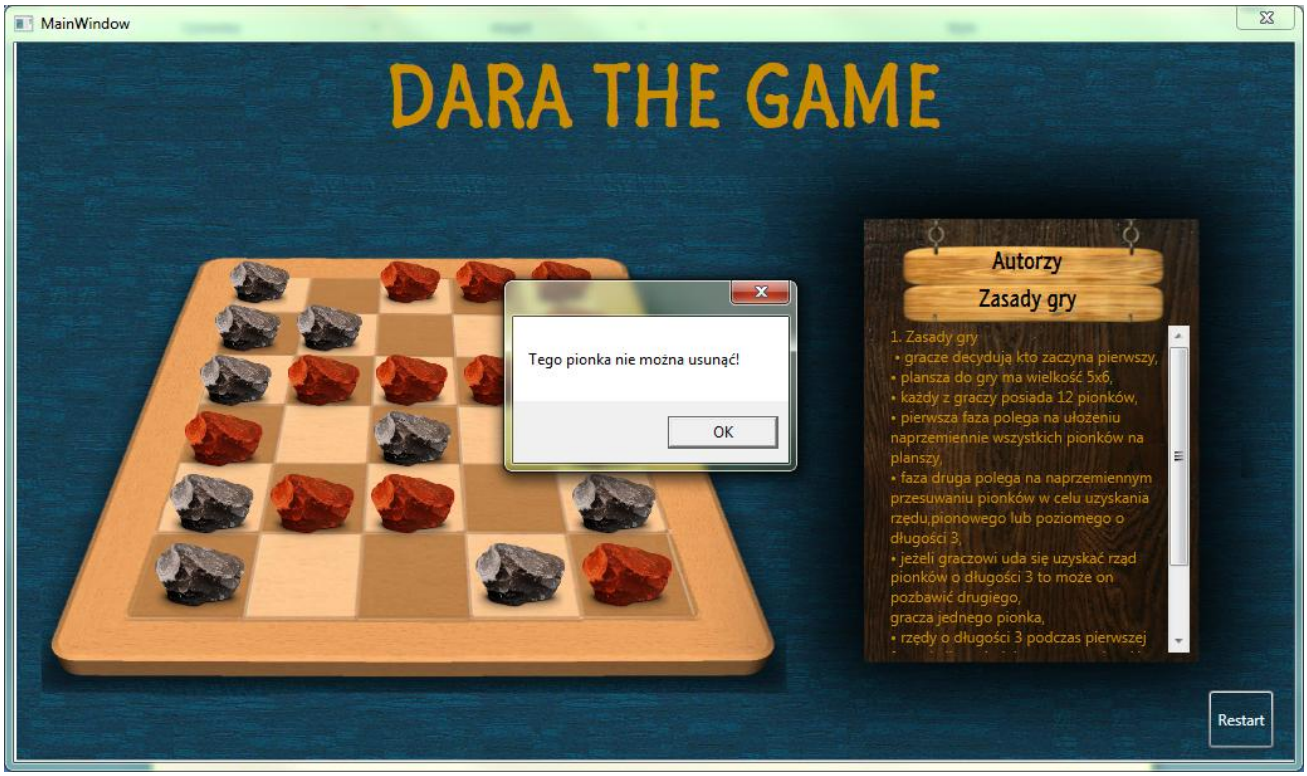
Rys. 4. Odpowiedź komputera na ruch

Jak można zauważyć nasz pionek się przemieścił, a komputer utworzył trójkę i usunął jeden z naszych pionków. W przypadku gdy gracz utworzy trójkę dostanie komunikat:



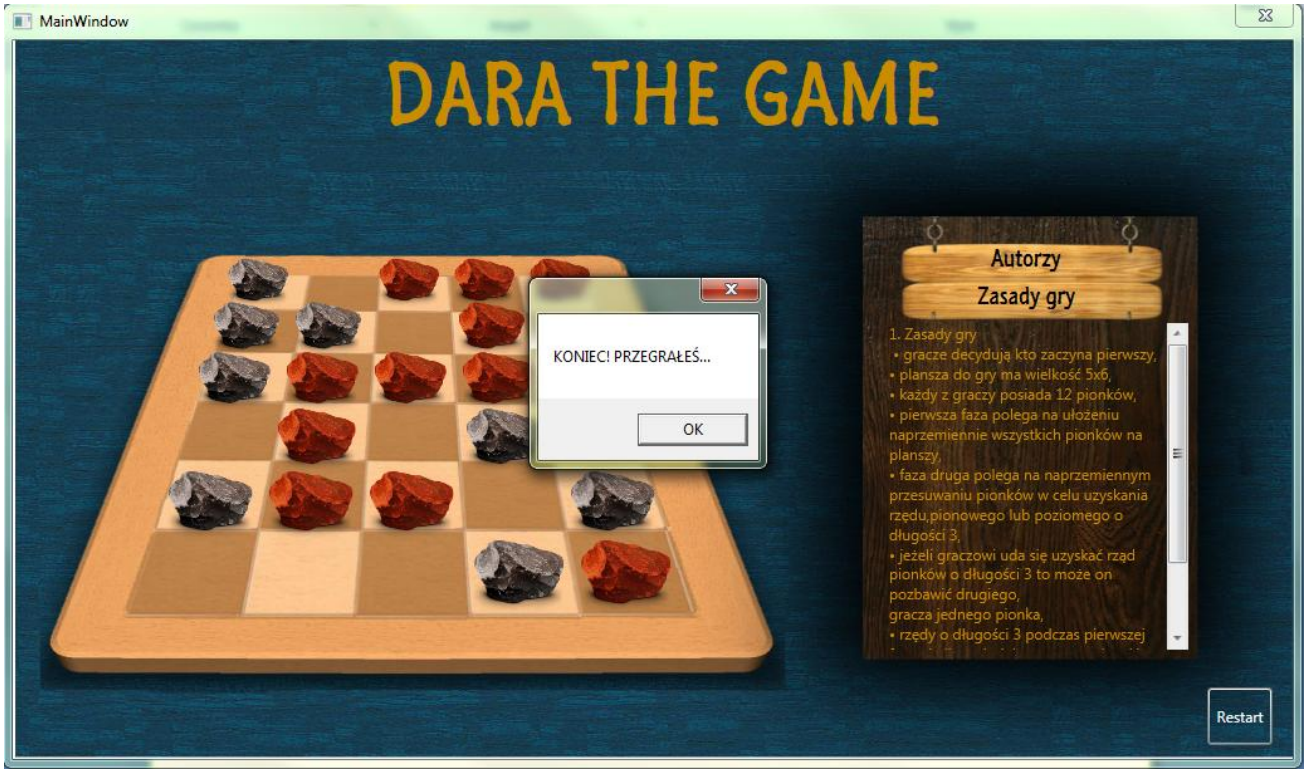
Rys. 5. Podpowiedź „Usuń pionek”

Gracz po takim komunikacie powinien wybrać pionek przeciwnika w celu usunięcia. Jak wynika z zasad gry w przypadku wybrania pionka z trójki dostanie komunikat o niemożliwości usunięcia tego pionka, gracz wówczas będzie musiał wybrać inny pionek:



Rys. 6. Podpowiedź „Tego pionka nie można usunąć”

I tak po kilku ruchach może się okazać że:



Rys. 7. Podpowiedź „KONIEC ...”

VI. Tekst programu

```
[Serializable]
    public class Board
    {
        public static int BOARD_COLUMN_COUNT = 5;
        public static int BOARD_ROW_COUNT = 6;
        public static bool delete =false;

        private static Random rand = new Random();

        private BoardSquare[][] boardSquares;
        private BoardSide turnOwner;
        private Move lastMove;

        public Board()
        {
            boardSquares = new BoardSquare[BOARD_ROW_COUNT][];
            turnOwner = BoardSide.MAX;

            for (int i = 0; i < boardSquares.Length; i++)
            {
                boardSquares[i] = new BoardSquare[BOARD_COLUMN_COUNT];
                for (int j = 0; j < boardSquares[i].Length; j++)
                {
                    boardSquares[i][j] = new BoardSquare(i, j);
                }
            }
        }

        public int putPawn(BoardSquare where, Pawn pawn)
        {
            BoardSquare square = boardSquares[where.getColumnCoord()][where.getRowCoord()];
            int returned =square.setCurrentObject(pawn);
            int reutrnedfours = findFours();
            if (reutrnedfours == 1)
            {
                square = boardSquares[where.getColumnCoord()][where.getRowCoord()];
                square.takeCurrentObject();
                return 2 ;
            }
            if (returned == 2)
            {
                return 1;

            }
            else
            {
                return 0;
            }

            //where.setCurrentObject(pawn);

        }

        // for testing purposes
        public void putPawn(Pawn pawn)
        {
            int row = rand.Next(BOARD_ROW_COUNT);
            int column = rand.Next(BOARD_COLUMN_COUNT);
            BoardSquare square = boardSquares[row][column];
            int i = square.setCurrentObject(pawn);
            while (i == 2)
            {
```

```
                    row = rand.Next(BOARD_ROW_COUNT);
                    column = rand.Next(BOARD_COLUMN_COUNT);
                    square = boardSquares[row][column];
                    i = square.setCurrentObject(pawn);
                    int reutrned = findFours();
                    if (reutrned == 1)
                    {
                        square = boardSquares[row][column];
                        square.takeCurrentObject();
                        i = 2;
                    }

                //findAndLockThrees();

            }
        }

        public BoardSide getTurnOwner()
        {
            return turnOwner;
        }

        public void toggleTurnOwner()
        {
            if (turnOwner == BoardSide.MAX)
            {
                turnOwner = BoardSide.MIN;
            }
            else if (turnOwner == BoardSide.MIN)
            {
                turnOwner = BoardSide.MAX;
            }
            else
            {
                throw new NoneTurnException("Something or someone has set turn owner to
BoardObjectSide.NONE - why? :(");
            }
        }

        public void setLastMove(Move lastMove)
        {
            this.lastMove = lastMove;
        }

        public Move getLastMove()
        {
            return lastMove;
        }

        public bool isEnd()
        {
            List<BoardSquare> maxNotLockedSquares = new List<BoardSquare>();
            List<BoardSquare> minNotLockedSquares = new List<BoardSquare>();

            for (int i = 0; i < boardSquares.Length; i++)
            {
                for (int j = 0; j < boardSquares[i].Length; j++)
                {
                    if (boardSquares[i][j].getCurrentObject() is Pawn &&
                        boardSquares[i][j].getCurrentObject().getBoardObjectSide() ==
BoardSide.MAX &&
                        !boardSquares[i][j].getCurrentObject().getLocked())
```

```
                    {
                        maxNotLockedSquares.Add(boardSquares[i][j]);
                    }

                    if (boardSquares[i][j].getCurrentObject() is Pawn &&
                        boardSquares[i][j].getCurrentObject().getBoardObjectSide() ==
BoardSide.MIN &&
                        !boardSquares[i][j].getCurrentObject().getLocked())
                    {
                        minNotLockedSquares.Add(boardSquares[i][j]);
                    }
                }
            }

            if (maxNotLockedSquares.Count < 3 || minNotLockedSquares.Count < 3)
            {
                return true;
            }

            return false;
        }

        public int isEnd(int cos)
        {
            List<BoardSquare> maxNotLockedSquares = new List<BoardSquare>();
            List<BoardSquare> minNotLockedSquares = new List<BoardSquare>();
            int win = 0;
            int lose = 0;
            for (int i = 0; i < boardSquares.Length; i++)
            {
                for (int j = 0; j < boardSquares[i].Length; j++)
                {
                    if (boardSquares[i][j].getCurrentObject() is Pawn &&
                        boardSquares[i][j].getCurrentObject().getBoardObjectSide() ==
BoardSide.MAX &&
                        !boardSquares[i][j].getCurrentObject().getLocked())
                    {
                        maxNotLockedSquares.Add(boardSquares[i][j]);
                    }
                    if (boardSquares[i][j].getCurrentObject() is Pawn &&
                        boardSquares[i][j].getCurrentObject().getBoardObjectSide() ==
BoardSide.MAX &&
                        boardSquares[i][j].getCurrentObject().getLocked())
                    {
                        win += 1;
                    }
                    if (boardSquares[i][j].getCurrentObject() is Pawn &&
                        boardSquares[i][j].getCurrentObject().getBoardObjectSide() ==
BoardSide.MIN &&
                        !boardSquares[i][j].getCurrentObject().getLocked())
                    {
                        minNotLockedSquares.Add(boardSquares[i][j]);
                    }
                    if (boardSquares[i][j].getCurrentObject() is Pawn &&
                        boardSquares[i][j].getCurrentObject().getBoardObjectSide() ==
BoardSide.MIN &&
                        boardSquares[i][j].getCurrentObject().getLocked())
                    {
                        lose += 1;
                    }
                }
            }

            if (maxNotLockedSquares.Count < 3 || minNotLockedSquares.Count < 3)
            {
```

```
                if (win > lose)
                {
                    return 1;
                }
                else
                {
                    return 2;
                }

            }
            return 0;
        }
        public List<Board> getPossibleBoards()
        {
            List<BoardSquare> freeSquares = new List<BoardSquare>();
            List<BoardSquare> turnOwnerSquares = new List<BoardSquare>();
            List<Board> possibleBoards = new List<Board>();
            List<Move> validMoves = new List<Move>();

            for (int i = 0; i < boardSquares.Length; i++)
            {
                for (int j = 0; j < boardSquares[i].Length; j++)
                {
                    if (boardSquares[i][j].getCurrentObject() is None)
                    {
                        freeSquares.Add(boardSquares[i][j]);
                    }

                    if (boardSquares[i][j].getCurrentObject() is Pawn &&
                        boardSquares[i][j].getCurrentObject().getBoardObjectSide() ==
turnOwner)
                    {
                        turnOwnerSquares.Add(boardSquares[i][j]);
                    }
                }
            }

            foreach (BoardSquare from in turnOwnerSquares)
            {
                foreach (BoardSquare to in freeSquares)
                {
                    Board boardClone = (Board)this.DeepClone();
                    BoardSquare fromClone = boardClone.getBoardSquare(from.getRowCoord(),
from.getColumnCoord());
                    BoardSquare toClone = boardClone.getBoardSquare(to.getRowCoord(),
to.getColumnCoord());

                    Move move = new Move(boardClone, fromClone, toClone);
                    if (move.isValid())
                    {
                        validMoves.Add(move);
                    }
                }
            }

            foreach (Move move in validMoves)
            {
                try
                {
                    move.executeMove();
                    possibleBoards.Add(move.getBoard());
                }
                catch (InvalidMoveExecutionException ex)
                {
                    System.Console.WriteLine(ex.Message);
```

```
                }
            }

            return possibleBoards;
        }

        public int findFours()
        {
            List<BoardSquare> maxSquares = new List<BoardSquare>();
            List<BoardSquare> minSquares = new List<BoardSquare>();

            for (int i = 0; i < boardSquares.Length; i++)
            {
                for (int j = 0; j < boardSquares[i].Length; j++)
                {
                    if (boardSquares[i][j].getCurrentObject() is Pawn &&
                        boardSquares[i][j].getCurrentObject().getBoardObjectSide() ==
BoardSide.MAX)
                    {
                        maxSquares.Add(boardSquares[i][j]);
                    }

                    if (boardSquares[i][j].getCurrentObject() is Pawn &&
                        boardSquares[i][j].getCurrentObject().getBoardObjectSide() ==
BoardSide.MIN)
                    {
                        minSquares.Add(boardSquares[i][j]);
                    }
                }
            }

            foreach (BoardSquare maxSquare in maxSquares)
            {
                int returned=returnFours(maxSquare);
                if(returned!=0)
                    return returned;
            }

            foreach (BoardSquare minSquare in minSquares)
            {
                int returned=returnFours(minSquare);
                if (returned != 0)
                    return returned;
            }
            return 0;
        }

        private int returnFours(BoardSquare square)
        {
            if (square.getCurrentObject().getLocked())
            {
                return 0;
            }

            int rowCoords = square.getRowCoord();
            int columnCoords = square.getColumnCoord();
            try
            {
                BoardSquare upperNeighbour1 = boardSquares[rowCoords + 1][columnCoords];
                BoardSquare upperNeighbour2 = boardSquares[rowCoords + 2][columnCoords];
                BoardSquare upperNeighbour3 = boardSquares[rowCoords + 3][columnCoords];

                if (upperNeighbour1.getCurrentObject() is Pawn &&
                    upperNeighbour1.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide() &&
```

```
                    upperNeighbour2.getCurrentObject() is Pawn &&
                    upperNeighbour2.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide() &&
                    upperNeighbour3.getCurrentObject() is Pawn &&
                    upperNeighbour3.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide())
                {
                    return 1;

                }
            }
            catch (IndexOutOfRangeException ex)
            {
                //System.Console.WriteLine(ex.Message);
            }
            try
            {
                BoardSquare lowerNeighbour1 = boardSquares[rowCoords - 1][columnCoords];
                BoardSquare lowerNeighbour2 = boardSquares[rowCoords - 2][columnCoords];
                BoardSquare lowerNeighbour3 = boardSquares[rowCoords - 3][columnCoords];

                if (lowerNeighbour1.getCurrentObject() is Pawn &&
                    lowerNeighbour1.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide() &&
                    lowerNeighbour2.getCurrentObject() is Pawn &&
                    lowerNeighbour2.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide() &&
                    lowerNeighbour3.getCurrentObject() is Pawn &&
                    lowerNeighbour3.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide())
                {
                    return 1;

                }
            }
            catch (IndexOutOfRangeException ex)
            {
                //System.Console.WriteLine(ex.Message);
            }
            try
            {
                BoardSquare upperNeighbour1 = boardSquares[rowCoords + 1][columnCoords];
                BoardSquare upperNeighbour2 = boardSquares[rowCoords + 2][columnCoords];
                BoardSquare lowerNeighbour1 = boardSquares[rowCoords - 1][columnCoords];
                BoardSquare lowerNeighbour2 = boardSquares[rowCoords - 2][columnCoords];

                if (upperNeighbour1.getCurrentObject() is Pawn &&
                    upperNeighbour1.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide() &&
                    lowerNeighbour1.getCurrentObject() is Pawn &&
                    lowerNeighbour1.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide() &&
                    lowerNeighbour2.getCurrentObject() is Pawn &&
                    lowerNeighbour2.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide() )
                {
                    return 1;

                }
            }
            catch (IndexOutOfRangeException ex)
            {
                //System.Console.WriteLine(ex.Message);
            }
```

```
            try
            {
                BoardSquare upperNeighbour1 = boardSquares[rowCoords + 1][columnCoords];
                BoardSquare upperNeighbour2 = boardSquares[rowCoords + 2][columnCoords];
                BoardSquare lowerNeighbour1 = boardSquares[rowCoords - 1][columnCoords];
                BoardSquare lowerNeighbour2 = boardSquares[rowCoords - 2][columnCoords];

                if (upperNeighbour1.getCurrentObject() is Pawn &&
                    upperNeighbour1.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide() &&
                    lowerNeighbour1.getCurrentObject() is Pawn &&
                    lowerNeighbour1.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide() &&
                    upperNeighbour2.getCurrentObject() is Pawn &&
                    upperNeighbour2.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide())
                {
                    return 1;

                }
            }
            catch (IndexOutOfRangeException ex)
            {
                //System.Console.WriteLine(ex.Message);
            }

            try
            {
                BoardSquare leftNeighbour1 = boardSquares[rowCoords][columnCoords - 1];
                BoardSquare leftNeighbour2 = boardSquares[rowCoords][columnCoords - 2];
                BoardSquare rightNeighbour1 = boardSquares[rowCoords][columnCoords + 1];
                BoardSquare rightNeighbour2 = boardSquares[rowCoords][columnCoords + 2];

                if (leftNeighbour1.getCurrentObject() is Pawn &&
                    leftNeighbour1.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide() &&
                    rightNeighbour1.getCurrentObject() is Pawn &&
                    rightNeighbour1.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide() &&
                    leftNeighbour2.getCurrentObject() is Pawn &&
                    leftNeighbour2.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide() )
                {

                    return 1;
                }
            }
            catch (IndexOutOfRangeException ex)
            {
                //System.Console.WriteLine(ex.Message);
            }
            try
            {
                BoardSquare leftNeighbour1 = boardSquares[rowCoords][columnCoords - 1];
                BoardSquare leftNeighbour2 = boardSquares[rowCoords][columnCoords - 2];
                BoardSquare rightNeighbour1 = boardSquares[rowCoords][columnCoords + 1];
                BoardSquare rightNeighbour2 = boardSquares[rowCoords][columnCoords + 2];

                if (leftNeighbour1.getCurrentObject() is Pawn &&
                    leftNeighbour1.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide() &&
                    rightNeighbour1.getCurrentObject() is Pawn &&
                    rightNeighbour1.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide() &&
                    rightNeighbour2.getCurrentObject() is Pawn &&
```

```
                    rightNeighbour2.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide())
                {

                    return 1;
                }
            }
            catch (IndexOutOfRangeException ex)
            {
                //System.Console.WriteLine(ex.Message);
            }

            try
            {
                BoardSquare leftNeighbour1 = boardSquares[rowCoords][columnCoords - 1];
                BoardSquare leftNeighbour2 = boardSquares[rowCoords][columnCoords - 2];
                BoardSquare leftNeighbour3 = boardSquares[rowCoords][columnCoords - 3];

                if (leftNeighbour1.getCurrentObject() is Pawn &&
                    leftNeighbour1.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide() &&
                    leftNeighbour2.getCurrentObject() is Pawn &&
                    leftNeighbour2.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide() &&
                    leftNeighbour3.getCurrentObject() is Pawn &&
                    leftNeighbour3.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide())
                {
                    return 1;

                }
            }
            catch (IndexOutOfRangeException ex)
            {
                //System.Console.WriteLine(ex.Message);
            }
            try
            {
                BoardSquare rightNeighbour1 = boardSquares[rowCoords][columnCoords + 1];
                BoardSquare rightNeighbour2 = boardSquares[rowCoords][columnCoords + 2];
                BoardSquare rightNeighbour3 = boardSquares[rowCoords][columnCoords + 3];

                if (rightNeighbour1.getCurrentObject() is Pawn &&
                    rightNeighbour1.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide() &&
                    rightNeighbour2.getCurrentObject() is Pawn &&
                    rightNeighbour2.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide() &&
                    rightNeighbour3.getCurrentObject() is Pawn &&
                    rightNeighbour3.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide())
                {
                    return 1;

                }
            }
            catch (IndexOutOfRangeException ex)
            {
                //System.Console.WriteLine(ex.Message);
            }
            return 0;
        }

        public void findAndLockThrees()
        {
```

```
            List<BoardSquare> maxSquares = new List<BoardSquare>();
            List<BoardSquare> minSquares = new List<BoardSquare>();

            for (int i = 0; i < boardSquares.Length; i++)
            {
                for (int j = 0; j < boardSquares[i].Length; j++)
                {
                    if (boardSquares[i][j].getCurrentObject() is Pawn &&
                        boardSquares[i][j].getCurrentObject().getBoardObjectSide() ==
BoardSide.MAX)
                    {
                        maxSquares.Add(boardSquares[i][j]);
                    }

                    if (boardSquares[i][j].getCurrentObject() is Pawn &&
                        boardSquares[i][j].getCurrentObject().getBoardObjectSide() ==
BoardSide.MIN)
                    {
                        minSquares.Add(boardSquares[i][j]);
                    }
                }
            }

            foreach (BoardSquare maxSquare in maxSquares)
            {
                lockThrees(maxSquare);
            }

            foreach (BoardSquare minSquare in minSquares)
            {
                lockThrees(minSquare);
            }
        }
        public void isThrees(BoardSquare position)
        {
            lockThrees(position);

        }

        private void lockThrees(BoardSquare square)
        {
            if (square.getCurrentObject().getLocked())
            {
                return ;
            }

            int rowCoords = square.getRowCoord();
            int columnCoords = square.getColumnCoord();
            try
            {
                BoardSquare upperNeighbour = boardSquares[rowCoords + 1][columnCoords];
                BoardSquare upperNeighbour1 = boardSquares[rowCoords + 2][columnCoords];

                if (upperNeighbour.getCurrentObject() is Pawn &&
                    upperNeighbour.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide() &&
                    upperNeighbour1.getCurrentObject() is Pawn &&
                    upperNeighbour1.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide())
                {
                    square.getCurrentObject().setLocked(true);
                    upperNeighbour.getCurrentObject().setLocked(true);
                    upperNeighbour1.getCurrentObject().setLocked(true);
                    delete = true;
                }
```

```
            }
            catch (IndexOutOfRangeException ex)
            {
                //System.Console.WriteLine(ex.Message);
            }
            try
            {
                BoardSquare lowerNeighbour = boardSquares[rowCoords - 1][columnCoords];
                BoardSquare lowerNeighbour1 = boardSquares[rowCoords - 2][columnCoords];

                if (lowerNeighbour.getCurrentObject() is Pawn &&
                    lowerNeighbour.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide() &&
                    lowerNeighbour1.getCurrentObject() is Pawn &&
                    lowerNeighbour1.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide())
                {
                    square.getCurrentObject().setLocked(true);
                    lowerNeighbour.getCurrentObject().setLocked(true);
                    lowerNeighbour1.getCurrentObject().setLocked(true);
                    delete = true;
                }
            }
            catch (IndexOutOfRangeException ex)
            {
                //System.Console.WriteLine(ex.Message);
            }
            try
            {
                BoardSquare upperNeighbour = boardSquares[rowCoords + 1][columnCoords];
                BoardSquare lowerNeighbour = boardSquares[rowCoords - 1][columnCoords];

                if (upperNeighbour.getCurrentObject() is Pawn &&
                    upperNeighbour.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide() &&
                    lowerNeighbour.getCurrentObject() is Pawn &&
                    lowerNeighbour.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide())
                {
                    square.getCurrentObject().setLocked(true);
                    upperNeighbour.getCurrentObject().setLocked(true);
                    lowerNeighbour.getCurrentObject().setLocked(true);
                    delete = true;
                }
            }
            catch (IndexOutOfRangeException ex)
            {
                //System.Console.WriteLine(ex.Message);
            }
            try
            {
                BoardSquare leftNeighbour = boardSquares[rowCoords][columnCoords - 1];
                BoardSquare leftNeighbour1 = boardSquares[rowCoords][columnCoords - 2];

                if (leftNeighbour.getCurrentObject() is Pawn &&
                    leftNeighbour.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide() &&
                    leftNeighbour1.getCurrentObject() is Pawn &&
                    leftNeighbour1.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide())
                {
                    square.getCurrentObject().setLocked(true);
                    leftNeighbour.getCurrentObject().setLocked(true);
                    leftNeighbour1.getCurrentObject().setLocked(true);
                    delete = true;
```

```
                }
            }
            catch (IndexOutOfRangeException ex)
            {
                //System.Console.WriteLine(ex.Message);
            } try
            {
                BoardSquare rightNeighbour = boardSquares[rowCoords][columnCoords +1];
                BoardSquare rightNeighbour1 = boardSquares[rowCoords][columnCoords +2];

                if (rightNeighbour.getCurrentObject() is Pawn &&
                    rightNeighbour.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide() &&
                    rightNeighbour1.getCurrentObject() is Pawn &&
                    rightNeighbour1.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide())
                {
                    square.getCurrentObject().setLocked(true);
                    rightNeighbour.getCurrentObject().setLocked(true);
                    rightNeighbour1.getCurrentObject().setLocked(true);
                    delete = true;
                }
            }
            catch (IndexOutOfRangeException ex)
            {
                //System.Console.WriteLine(ex.Message);
            }
            try
            {
                BoardSquare leftNeighbour = boardSquares[rowCoords][columnCoords - 1];
                BoardSquare rightNeighbour = boardSquares[rowCoords][columnCoords + 1];

                if (leftNeighbour.getCurrentObject() is Pawn &&
                    leftNeighbour.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide() &&
                    rightNeighbour.getCurrentObject() is Pawn &&
                    rightNeighbour.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide())
                {
                    square.getCurrentObject().setLocked(true);
                    leftNeighbour.getCurrentObject().setLocked(true);
                    rightNeighbour.getCurrentObject().setLocked(true);
                    delete = true;
                }
            }
            catch (IndexOutOfRangeException ex)
            {
                //System.Console.WriteLine(ex.Message);
            }

        }

        public int getMaxSidePawnCount()
        {
            int counter = 0;
            for (int i = 0; i < boardSquares.Length; i++)
            {
                for (int j = 0; j < boardSquares[i].Length; j++)
                {
                    if (boardSquares[i][j].getCurrentObject() is Pawn &&
                        boardSquares[i][j].getCurrentObject().getBoardObjectSide() ==
BoardSide.MAX)
                    {
                        counter++;
                    }
```

```
                }
            }

            return counter;
        }




        public int getMinSidePawnCount()
        {
            int counter = 0;
            for (int i = 0; i < boardSquares.Length; i++)
            {
                for (int j = 0; j < boardSquares[i].Length; j++)
                {
                    if (boardSquares[i][j].getCurrentObject() is Pawn &&
                        boardSquares[i][j].getCurrentObject().getBoardObjectSide() ==
BoardSide.MIN)
                    {
                        counter++;
                    }
                }
            }

            return counter;
        }

        public BoardSquare getBoardSquare(int row, int column)
        {
            return boardSquares[row][column];
        }

        // printing


        public Char[,] printBoard()
        {
            StringBuilder sb = new StringBuilder();
            Char[,] array = new Char[6,5];
            for (int i = 0; i < boardSquares.Length; i++)
            {
                for (int j = 0; j < boardSquares[i].Length; j++)
                {
                    if (boardSquares[i][j].getCurrentObject() is None)
                    {
                        array[i,j] = '_';
                        sb.Append("_");
                    }

                    if (boardSquares[i][j].getCurrentObject() is Pawn &&
                        boardSquares[i][j].getCurrentObject().getBoardObjectSide() ==
BoardSide.MAX)
                    {
                        if (boardSquares[i][j].getCurrentObject().getLocked())
                        {
                            array[i, j] = 'M';
                            sb.Append("M");
                        }
                        else
                        {
                            array[i, j] = 'm';
                            sb.Append("m");
                        }
```

```
                    }

                    if (boardSquares[i][j].getCurrentObject() is Pawn &&
                        boardSquares[i][j].getCurrentObject().getBoardObjectSide() ==
BoardSide.MIN)
                    {
                        if (boardSquares[i][j].getCurrentObject().getLocked())
                        {
                            array[i, j] = 'I';
                            sb.Append("I");
                        }
                        else
                        {
                            array[i, j] = 'i';
                            sb.Append("i");
                        }
                    }
                }
                sb.AppendLine();
            }
            return array;
           // System.Console.WriteLine(sb.ToString());
        }

        public void printPossibleBoards()
        {
            List<Board> possibleBoards = getPossibleBoards();
            int i = 0;
            foreach (Board board in possibleBoards)
            {
                System.Console.WriteLine(i);
                board.printBoard();
                i++;
            }
        }

        public void findAndRateTwos()
        {
            List<BoardSquare> maxSquares = new List<BoardSquare>();
            List<BoardSquare> minSquares = new List<BoardSquare>();

            for (int i = 0; i < boardSquares.Length; i++)
            {
                for (int j = 0; j < boardSquares[i].Length; j++)
                {
                    if (boardSquares[i][j].getCurrentObject() is Pawn &&
                        boardSquares[i][j].getCurrentObject().getBoardObjectSide() ==
BoardSide.MAX)
                    {
                        maxSquares.Add(boardSquares[i][j]);
                    }

                    if (boardSquares[i][j].getCurrentObject() is Pawn &&
                        boardSquares[i][j].getCurrentObject().getBoardObjectSide() ==
BoardSide.MIN)
                    {
                        minSquares.Add(boardSquares[i][j]);
                    }
                }
            }

            foreach (BoardSquare maxSquare in maxSquares)
            {
                FindTwos(maxSquare);
            }
```

```
            foreach (BoardSquare minSquare in minSquares)
            {
                FindTwos(minSquare);
            }
        }



        private void FindTwos(BoardSquare square)
        {

            if (square.getCurrentObject().getLocked())
            {
                return;
            }

            int rowCoords = square.getRowCoord();
            int columnCoords = square.getColumnCoord();
            try
            {
                BoardSquare upperNeighbour = boardSquares[rowCoords + 1][columnCoords];

                if (upperNeighbour.getCurrentObject() is Pawn &&
                    upperNeighbour.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide())
                {
                    UtilityRating.rating += 2;
                    //set rating
                }
            }
            catch (IndexOutOfRangeException ex)
            {
                //System.Console.WriteLine(ex.Message);
            }
            try
            {
                BoardSquare lowerNeighbour = boardSquares[rowCoords - 1][columnCoords];

                if (lowerNeighbour.getCurrentObject() is Pawn &&
                    lowerNeighbour.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide() )
                {
                    UtilityRating.rating += 2;
                }
            }
            catch (IndexOutOfRangeException ex)
            {
                //System.Console.WriteLine(ex.Message);
            }
            try
            {
                BoardSquare leftNeighbour = boardSquares[rowCoords][columnCoords - 1];

                if (leftNeighbour.getCurrentObject() is Pawn &&
                    leftNeighbour.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide() )
                {
                    UtilityRating.rating += 2;
                }
            }
            catch (IndexOutOfRangeException ex)
            {
                //System.Console.WriteLine(ex.Message);
            } try
```

```
            {
                BoardSquare rightNeighbour = boardSquares[rowCoords][columnCoords + 1];

                if (rightNeighbour.getCurrentObject() is Pawn &&
                    rightNeighbour.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide() )
                {
                    UtilityRating.rating += 2;
                }
            }
            catch (IndexOutOfRangeException ex)
            {
                //System.Console.WriteLine(ex.Message);
            }

        }



        public void findAndRateCloseToTwos()
        {
            List<BoardSquare> maxSquares = new List<BoardSquare>();
            List<BoardSquare> minSquares = new List<BoardSquare>();

            for (int i = 0; i < boardSquares.Length; i++)
            {
                for (int j = 0; j < boardSquares[i].Length; j++)
                {
                    if (boardSquares[i][j].getCurrentObject() is Pawn &&
                        boardSquares[i][j].getCurrentObject().getBoardObjectSide() ==
BoardSide.MAX)
                    {
                        maxSquares.Add(boardSquares[i][j]);
                    }

                    if (boardSquares[i][j].getCurrentObject() is Pawn &&
                        boardSquares[i][j].getCurrentObject().getBoardObjectSide() ==
BoardSide.MIN)
                    {
                        minSquares.Add(boardSquares[i][j]);
                    }
                }
            }

            foreach (BoardSquare maxSquare in maxSquares)
            {
                FindCloseToTwos(maxSquare);
            }

            foreach (BoardSquare minSquare in minSquares)
            {
                FindCloseToTwos(minSquare);
            }
        }



        private void FindCloseToTwos(BoardSquare square)
        {
            if (square.getCurrentObject().getLocked())
            {
                return;
            }

            int rowCoords = square.getRowCoord();
```

```
            int columnCoords = square.getColumnCoord();
            try
            {
                BoardSquare upperNeighbour = boardSquares[rowCoords + 1][columnCoords+1];

                if (upperNeighbour.getCurrentObject() is Pawn &&
                    upperNeighbour.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide())
                {

                    UtilityRating.rating += 1;
                }
            }
            catch (IndexOutOfRangeException ex)
            {
                //System.Console.WriteLine(ex.Message);
            }
            try
            {
                BoardSquare lowerNeighbour = boardSquares[rowCoords - 1][columnCoords+1];

                if (lowerNeighbour.getCurrentObject() is Pawn &&
                    lowerNeighbour.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide())
                {
                    UtilityRating.rating += 1;
                }
            }
            catch (IndexOutOfRangeException ex)
            {
                //System.Console.WriteLine(ex.Message);
            }
            try
            {
                BoardSquare leftNeighbour = boardSquares[rowCoords-1][columnCoords - 1];

                if (leftNeighbour.getCurrentObject() is Pawn &&
                    leftNeighbour.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide())
                {
                    UtilityRating.rating += 1;
                }
            }
            catch (IndexOutOfRangeException ex)
            {
                //System.Console.WriteLine(ex.Message);
            } try
            {
                BoardSquare rightNeighbour = boardSquares[rowCoords+1][columnCoords -1];

                if (rightNeighbour.getCurrentObject() is Pawn &&
                    rightNeighbour.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide())
                {
                    UtilityRating.rating += 1;
                }
            }
            catch (IndexOutOfRangeException ex)
            {
                //System.Console.WriteLine(ex.Message);
            }


        }
```

```
        public void findAndRateCloseToThrees()
        {
            List<BoardSquare> maxSquares = new List<BoardSquare>();
            List<BoardSquare> minSquares = new List<BoardSquare>();

            for (int i = 0; i < boardSquares.Length; i++)
            {
                for (int j = 0; j < boardSquares[i].Length; j++)
                {
                    if (boardSquares[i][j].getCurrentObject() is Pawn &&
                        boardSquares[i][j].getCurrentObject().getBoardObjectSide() ==
BoardSide.MAX)
                    {
                        maxSquares.Add(boardSquares[i][j]);
                    }

                    if (boardSquares[i][j].getCurrentObject() is Pawn &&
                        boardSquares[i][j].getCurrentObject().getBoardObjectSide() ==
BoardSide.MIN)
                    {
                        minSquares.Add(boardSquares[i][j]);
                    }
                }
            }

            foreach (BoardSquare maxSquare in maxSquares)
            {
                FindCloseToThrees(maxSquare);
            }

            foreach (BoardSquare minSquare in minSquares)
            {
                FindCloseToThrees(minSquare);
            }
        }




        private void FindCloseToThrees(BoardSquare square)
        {
            if (square.getCurrentObject().getLocked())
            {
                return;
            }

            int rowCoords = square.getRowCoord();
            int columnCoords = square.getColumnCoord();
            try
            {
                BoardSquare upperNeighbour = boardSquares[rowCoords + 1][columnCoords];
                BoardSquare upperNeighbour1 = boardSquares[rowCoords +2][columnCoords+1];

                if (upperNeighbour.getCurrentObject() is Pawn &&
                    upperNeighbour.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide() &&
                    upperNeighbour1.getCurrentObject() is Pawn &&
                    upperNeighbour1.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide())
                {
                    UtilityRating.rating += 3;
                }
            }
            catch (IndexOutOfRangeException ex)
```

```
            {
                //System.Console.WriteLine(ex.Message);
            }
            try
            {
                BoardSquare upperNeighbour = boardSquares[rowCoords + 1][columnCoords];
                BoardSquare upperNeighbour1 = boardSquares[rowCoords + 2][columnCoords- 1];

                if (upperNeighbour.getCurrentObject() is Pawn &&
                    upperNeighbour.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide() &&
                    upperNeighbour1.getCurrentObject() is Pawn &&
                    upperNeighbour1.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide())
                {
                    UtilityRating.rating += 3;
                }
            }
            catch (IndexOutOfRangeException ex)
            {
                //System.Console.WriteLine(ex.Message);
            }
            try
            {
                BoardSquare upperNeighbour = boardSquares[rowCoords + 1][columnCoords];
                BoardSquare upperNeighbour1 = boardSquares[rowCoords +3][columnCoords];

                if (upperNeighbour.getCurrentObject() is Pawn &&
                    upperNeighbour.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide() &&
                    upperNeighbour1.getCurrentObject() is Pawn &&
                    upperNeighbour1.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide())
                {
                    UtilityRating.rating += 3;
                }
            }
            catch (IndexOutOfRangeException ex)
            {
                //System.Console.WriteLine(ex.Message);
            }
            try
            {
                BoardSquare lowerNeighbour = boardSquares[rowCoords - 1][columnCoords];
                BoardSquare lowerNeighbour1 = boardSquares[rowCoords -2][columnCoords+1];

                if (lowerNeighbour.getCurrentObject() is Pawn &&
                    lowerNeighbour.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide() &&
                    lowerNeighbour1.getCurrentObject() is Pawn &&
                    lowerNeighbour1.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide())
                {
                    UtilityRating.rating += 3;
                }
            }
            catch (IndexOutOfRangeException ex)
            {
                //System.Console.WriteLine(ex.Message);
            }
            try
            {
                BoardSquare lowerNeighbour = boardSquares[rowCoords - 1][columnCoords];
                BoardSquare lowerNeighbour1 = boardSquares[rowCoords - 2][columnCoords -1];
```

```
                if (lowerNeighbour.getCurrentObject() is Pawn &&
                    lowerNeighbour.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide() &&
                    lowerNeighbour1.getCurrentObject() is Pawn &&
                    lowerNeighbour1.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide())
                {
                    UtilityRating.rating += 3;
                }
            }
            catch (IndexOutOfRangeException ex)
            {
                //System.Console.WriteLine(ex.Message);
            }
            try
            {
                BoardSquare lowerNeighbour = boardSquares[rowCoords - 1][columnCoords];
                BoardSquare lowerNeighbour1 = boardSquares[rowCoords - 3][columnCoords];

                if (lowerNeighbour.getCurrentObject() is Pawn &&
                    lowerNeighbour.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide() &&
                    lowerNeighbour1.getCurrentObject() is Pawn &&
                    lowerNeighbour1.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide())
                {
                    UtilityRating.rating += 3;
                }
            }
            catch (IndexOutOfRangeException ex)
            {
                //System.Console.WriteLine(ex.Message);
            }

            try
            {
                BoardSquare leftNeighbour = boardSquares[rowCoords][columnCoords - 1];
                BoardSquare leftNeighbour1 = boardSquares[rowCoords-1][columnCoords - 2];

                if (leftNeighbour.getCurrentObject() is Pawn &&
                    leftNeighbour.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide() &&
                    leftNeighbour1.getCurrentObject() is Pawn &&
                    leftNeighbour1.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide())
                {
                    UtilityRating.rating += 3;
                }
            }
            catch (IndexOutOfRangeException ex)
            {
                //System.Console.WriteLine(ex.Message);
            }

            try
            {
                BoardSquare leftNeighbour = boardSquares[rowCoords][columnCoords - 1];
                BoardSquare leftNeighbour1 = boardSquares[rowCoords + 1][columnCoords - 2];

                if (leftNeighbour.getCurrentObject() is Pawn &&
                    leftNeighbour.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide() &&
                    leftNeighbour1.getCurrentObject() is Pawn &&
                    leftNeighbour1.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide())
```

```
                {
                    UtilityRating.rating += 3;
                }
            }
            catch (IndexOutOfRangeException ex)
            {
                //System.Console.WriteLine(ex.Message);
            }
            try
            {
                BoardSquare leftNeighbour = boardSquares[rowCoords][columnCoords - 1];
                BoardSquare leftNeighbour1 = boardSquares[rowCoords][columnCoords - 3];

                if (leftNeighbour.getCurrentObject() is Pawn &&
                    leftNeighbour.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide() &&
                    leftNeighbour1.getCurrentObject() is Pawn &&
                    leftNeighbour1.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide())
                {
                    UtilityRating.rating += 3;
                }
            }
            catch (IndexOutOfRangeException ex)
            {
                //System.Console.WriteLine(ex.Message);
            }
            try
            {
                BoardSquare rightNeighbour = boardSquares[rowCoords][columnCoords + 1];
                BoardSquare rightNeighbour1 = boardSquares[rowCoords+1][columnCoords + 2];

                if (rightNeighbour.getCurrentObject() is Pawn &&
                    rightNeighbour.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide() &&
                    rightNeighbour1.getCurrentObject() is Pawn &&
                    rightNeighbour1.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide())
                {
                    UtilityRating.rating += 3;
                }
            }
            catch (IndexOutOfRangeException ex)
            {
                //System.Console.WriteLine(ex.Message);
            }
            try
            {
                BoardSquare rightNeighbour = boardSquares[rowCoords][columnCoords + 1];
                BoardSquare rightNeighbour1 = boardSquares[rowCoords - 1][columnCoords + 2];

                if (rightNeighbour.getCurrentObject() is Pawn &&
                    rightNeighbour.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide() &&
                    rightNeighbour1.getCurrentObject() is Pawn &&
                    rightNeighbour1.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide())
                {
                    UtilityRating.rating += 3;
                }
            }
            catch (IndexOutOfRangeException ex)
            {
                //System.Console.WriteLine(ex.Message);
            }
```

```
            try
            {
                BoardSquare rightNeighbour = boardSquares[rowCoords][columnCoords + 1];
                BoardSquare rightNeighbour1 = boardSquares[rowCoords][columnCoords + 3];

                if (rightNeighbour.getCurrentObject() is Pawn &&
                    rightNeighbour.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide() &&
                    rightNeighbour1.getCurrentObject() is Pawn &&
                    rightNeighbour1.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide())
                {
                    UtilityRating.rating += 3;
                }
            }
            catch (IndexOutOfRangeException ex)
            {
                //System.Console.WriteLine(ex.Message);
            }

        }




        public void findAndRateCloseMoves()
        {
            List<BoardSquare> maxSquares = new List<BoardSquare>();
            List<BoardSquare> minSquares = new List<BoardSquare>();

            for (int i = 0; i < boardSquares.Length; i++)
            {
                for (int j = 0; j < boardSquares[i].Length; j++)
                {
                    if (boardSquares[i][j].getCurrentObject() is Pawn &&
                        boardSquares[i][j].getCurrentObject().getBoardObjectSide() ==
BoardSide.MAX)
                    {
                        maxSquares.Add(boardSquares[i][j]);
                    }

                    if (boardSquares[i][j].getCurrentObject() is Pawn &&
                        boardSquares[i][j].getCurrentObject().getBoardObjectSide() ==
BoardSide.MIN)
                    {
                        minSquares.Add(boardSquares[i][j]);
                    }
                }
            }

            foreach (BoardSquare maxSquare in maxSquares)
            {
                FindCloseToMoves(maxSquare);
            }

            foreach (BoardSquare minSquare in minSquares)
            {
                FindCloseToMoves(minSquare);
            }
        }
```

```
        private void FindCloseToMoves(BoardSquare square)
        {
            if (square.getCurrentObject().getLocked())
            {
                return;
            }

            int rowCoords = square.getRowCoord();
            int columnCoords = square.getColumnCoord();
            try
            {
                for (int i = 0; i < 6; i++)
                {
                    for (int j = 0; j < 7; j++)
                    {
                        BoardSquare upperNeighbour = boardSquares[rowCoords +
i][columnCoords+j];


                        if (upperNeighbour.getCurrentObject() is Pawn &&
                            upperNeighbour.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide())
                        {
                            double sum =  1/(i +  j+0.001) ;
                            UtilityRating.rating += sum;
                        }
                    }
                }

            }
            catch (IndexOutOfRangeException ex)
            {
                //System.Console.WriteLine(ex.Message);
            }


        }









        public void findAndRateThrees()
        {
            List<BoardSquare> maxSquares = new List<BoardSquare>();
            List<BoardSquare> minSquares = new List<BoardSquare>();

            for (int i = 0; i < boardSquares.Length; i++)
            {
                for (int j = 0; j < boardSquares[i].Length; j++)
                {
                    if (boardSquares[i][j].getCurrentObject() is Pawn &&
                        boardSquares[i][j].getCurrentObject().getBoardObjectSide() ==
BoardSide.MAX)
                    {
                        maxSquares.Add(boardSquares[i][j]);
                    }

                    if (boardSquares[i][j].getCurrentObject() is Pawn &&
```

```
                        boardSquares[i][j].getCurrentObject().getBoardObjectSide() ==
BoardSide.MIN)
                    {
                        minSquares.Add(boardSquares[i][j]);
                    }
                }
            }

            foreach (BoardSquare maxSquare in maxSquares)
            {
                RateThrees(maxSquare);
            }

            foreach (BoardSquare minSquare in minSquares)
            {
                RateThrees(minSquare);
            }
        }


        private void RateThrees(BoardSquare square)
        {
            if (square.getCurrentObject().getLocked())
            {
                return;
            }

            int rowCoords = square.getRowCoord();
            int columnCoords = square.getColumnCoord();
            try
            {
                BoardSquare upperNeighbour = boardSquares[rowCoords + 1][columnCoords];
                BoardSquare upperNeighbour1 = boardSquares[rowCoords + 2][columnCoords];

                if (upperNeighbour.getCurrentObject() is Pawn &&
                    upperNeighbour.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide() &&
                    upperNeighbour1.getCurrentObject() is Pawn &&
                    upperNeighbour1.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide())
                {
                    UtilityRating.rating += 4;
                }
            }
            catch (IndexOutOfRangeException ex)
            {
                //System.Console.WriteLine(ex.Message);
            }
            try
            {
                BoardSquare lowerNeighbour = boardSquares[rowCoords - 1][columnCoords];
                BoardSquare lowerNeighbour1 = boardSquares[rowCoords - 2][columnCoords];

                if (lowerNeighbour.getCurrentObject() is Pawn &&
                    lowerNeighbour.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide() &&
                    lowerNeighbour1.getCurrentObject() is Pawn &&
                    lowerNeighbour1.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide())
                {
                    UtilityRating.rating += 4;
                }
            }
            catch (IndexOutOfRangeException ex)
            {
```

```
                //System.Console.WriteLine(ex.Message);
            }
            try
            {
                BoardSquare upperNeighbour = boardSquares[rowCoords + 1][columnCoords];
                BoardSquare lowerNeighbour = boardSquares[rowCoords - 1][columnCoords];

                if (upperNeighbour.getCurrentObject() is Pawn &&
                    upperNeighbour.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide() &&
                    lowerNeighbour.getCurrentObject() is Pawn &&
                    lowerNeighbour.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide())
                {
                    UtilityRating.rating += 4;
                }
            }
            catch (IndexOutOfRangeException ex)
            {
                //System.Console.WriteLine(ex.Message);
            }
            try
            {
                BoardSquare leftNeighbour = boardSquares[rowCoords][columnCoords - 1];
                BoardSquare leftNeighbour1 = boardSquares[rowCoords][columnCoords - 2];

                if (leftNeighbour.getCurrentObject() is Pawn &&
                    leftNeighbour.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide() &&
                    leftNeighbour1.getCurrentObject() is Pawn &&
                    leftNeighbour1.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide())
                {
                    UtilityRating.rating += 4;
                }
            }
            catch (IndexOutOfRangeException ex)
            {
                //System.Console.WriteLine(ex.Message);
            } try
            {
                BoardSquare rightNeighbour = boardSquares[rowCoords][columnCoords + 1];
                BoardSquare rightNeighbour1 = boardSquares[rowCoords][columnCoords + 2];

                if (rightNeighbour.getCurrentObject() is Pawn &&
                    rightNeighbour.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide() &&
                    rightNeighbour1.getCurrentObject() is Pawn &&
                    rightNeighbour1.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide())
                {
                    UtilityRating.rating += 4;
                }
            }
            catch (IndexOutOfRangeException ex)
            {
                //System.Console.WriteLine(ex.Message);
            }
            try
            {
                BoardSquare leftNeighbour = boardSquares[rowCoords][columnCoords - 1];
                BoardSquare rightNeighbour = boardSquares[rowCoords][columnCoords + 1];

                if (leftNeighbour.getCurrentObject() is Pawn &&
```

```
                    leftNeighbour.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide() &&
                    rightNeighbour.getCurrentObject() is Pawn &&
                    rightNeighbour.getCurrentObject().getBoardObjectSide() ==
square.getCurrentObject().getBoardObjectSide())
                {
                    UtilityRating.rating += 4;
                }
            }
            catch (IndexOutOfRangeException ex)
            {
                //System.Console.WriteLine(ex.Message);
            }
        }




        public List<BoardSquare> findAllMax()
        {
            List<BoardSquare> maxSquares = new List<BoardSquare>();
            List<BoardSquare> minSquares = new List<BoardSquare>();

            for (int i = 0; i < boardSquares.Length; i++)
            {
                for (int j = 0; j < boardSquares[i].Length; j++)
                {
                    if (boardSquares[i][j].getCurrentObject() is Pawn &&
                        boardSquares[i][j].getCurrentObject().getBoardObjectSide() ==
BoardSide.MAX && !boardSquares[i][j].getCurrentObject().getLocked())
                    {
                        maxSquares.Add(boardSquares[i][j]);
                    }

                    if (boardSquares[i][j].getCurrentObject() is Pawn &&
                        boardSquares[i][j].getCurrentObject().getBoardObjectSide() ==
BoardSide.MIN)
                    {
                        minSquares.Add(boardSquares[i][j]);
                    }
                }
            }

            return maxSquares;
        }
    }
}
```

```
[Serializable]
    abstract public class BoardObject
    {
        protected BoardSide boardObjectSide;
        protected BoardObjectType boardObjectType;
        protected bool locked = false;

        public BoardSide getBoardObjectSide()
        {
            return boardObjectSide;
        }

        public BoardObjectType getBoardObjectType()
        {
            return boardObjectType;
        }

        public bool getLocked()
        {
            return locked;
        }

        public void setLocked(bool locked)
        {
            this.locked = locked;
        }
    }


public enum BoardObjectType
    {
        PAWN,
        NONE
    }


public enum BoardSide
    {
        MAX,
        MIN,
        NONE
    }


[Serializable]
    public class BoardSquare
    {
        private int rowCoord;
        private int columnCoord;
        private BoardObject currentObject;

        public BoardSquare(int rowCoord, int columnCoord)
        {
            this.rowCoord = rowCoord;
            this.columnCoord = columnCoord;
            this.currentObject = new None();
        }

        public int getRowCoord()
        {
            return rowCoord;
        }
```

```
        public int getColumnCoord()
        {
            return columnCoord;
        }

        public BoardObject getCurrentObject()
        {
            return currentObject;
        }

        public BoardObject takeCurrentObject()
        {
            BoardObject co = currentObject;
            currentObject = new None();
            return co;
        }

        public int setCurrentObject(BoardObject currentObject)
        {


            try
            {
                if (this.currentObject is None)
                {
                    this.currentObject = currentObject;
                    return 1;
                }
                else
                {
                    return 2;
                   // throw new PawnOverrideException("Existing object override requested...
row: " + this.getRowCoord() + " column: " + this.getColumnCoord());

                }
            }
            catch (PawnOverrideException ex)
            {

               // System.Console.WriteLine(ex.Message);

            }
            return 0;
        }


    }


public static T DeepClone<T>(this T a)
        {
            using (MemoryStream stream = new MemoryStream())
            {
                BinaryFormatter formatter = new BinaryFormatter();
                formatter.Serialize(stream, a);
                stream.Position = 0;
                return (T)formatter.Deserialize(stream);
            }
        }
```

```
public class InvalidMoveExecutionException : Exception
    {
        public InvalidMoveExecutionException()
        { }

        public InvalidMoveExecutionException(string message)
            : base(message)
        { }
    }


public class Minmax
    {
        private static int MAX_DEPTH = 1;

        public static Move getNextMove(Board board)
        {
            Move nextMove;

            if (board.getTurnOwner() == BoardSide.MAX)
            {
                nextMove = maxValue(board, 0);
            }
            else if (board.getTurnOwner() == BoardSide.MIN)
            {
                nextMove = minValue(board, 0);
            }
            else
            {
                throw new NoneTurnException("Check Minmax public class, there is a problem...
(none of utility ratings are better for respective executed moves... lol");
            }

            return nextMove;
        }

        private static Move maxValue(Board board, int depth)
        {
            if (board.isEnd() || depth >= MAX_DEPTH)
            {
                Move move = board.getLastMove();
                if (move == null)
                {
                    move = new Move((Board)board.DeepClone(), BoardSide.MAX);
                }

                move.setRating(UtilityRating.getRating(board));
                return move;
            }

            Move value = new Move(board, BoardSide.MAX);
            foreach (Board possibleBoard in board.getPossibleBoards())
            {
                Move currentMove = minValue(possibleBoard, depth + 1);
                value = currentMove.getRating() >= value.getRating() ? currentMove : value;
            }

            return value;
        }

        private static Move minValue(Board board, int depth)
        {
            if (board.isEnd() || depth >= MAX_DEPTH)
            {
                Move move = board.getLastMove();
```

```
                if (move == null)
                {
                    move = new Move((Board)board.DeepClone(), BoardSide.MAX);
                }

                move.setRating(UtilityRating.getRating(board));
                return move;
            }

            Move value = new Move(board, BoardSide.MIN);
            foreach (Board possibleBoard in board.getPossibleBoards())
            {
                Move currentMove = maxValue(possibleBoard, depth + 1);
                value = currentMove.getRating() <= value.getRating() ? currentMove : value;
            }

            return value;
        }
    }


[Serializable]
    public class Move
    {
        private Board board;
        private BoardSquare from;
        private BoardSquare to;
        private bool valid;
        private double rating;

        public Move(Board board, BoardSquare from, BoardSquare to, double rating = 0)
        {
            this.board = board;
            this.from = from;
            this.to = to;
            this.valid = validityCheck(from, to);
            this.rating = rating;
        }

        public Move(Board board, BoardSide side)
        {
            this.board = board;

            if (side == BoardSide.MAX)
            {
                rating = UtilityRating.MIN_RATING;
            }

            if (side == BoardSide.MIN)
            {
                rating = UtilityRating.MAX_RATING;
            }
        }

        public Board getBoard()
        {
            return board;
        }

        public bool isValid()
        {
            return valid;
        }
```

```
        public double getRating()
        {
            return rating;
        }

        public void setRating(double rating)
        {
            this.rating = rating;
        }

        private static bool validityCheck(BoardSquare from, BoardSquare to)
        {
            if (to.getCurrentObject() is None)
            {
                if (isNeighbour(from, to) && isOnBoard(from, to) && !isLocked(from))
                {
                    return true;
                }
            }

            return false;
        }

        private static bool isNeighbour(BoardSquare from, BoardSquare to)
        {
            if (((from.getRowCoord() + 1 == to.getRowCoord() && from.getColumnCoord() ==
to.getColumnCoord()) ||
                 (from.getRowCoord() - 1 == to.getRowCoord() && from.getColumnCoord() ==
to.getColumnCoord())) ^
                ((from.getColumnCoord() + 1 == to.getColumnCoord() && from.getRowCoord() ==
to.getRowCoord()) ||
                 (from.getColumnCoord() - 1 == to.getColumnCoord() && from.getRowCoord() ==
to.getRowCoord())))
            {
                return true;
            }

            return false;
        }

        private static bool isOnBoard(BoardSquare from, BoardSquare to)
        {
            if ((from.getRowCoord() >= 0 && from.getRowCoord() < Board.BOARD_ROW_COUNT) &&
                (from.getColumnCoord() >= 0 && from.getColumnCoord() <
Board.BOARD_COLUMN_COUNT) &&
                (to.getRowCoord() >= 0 && to.getRowCoord() < Board.BOARD_ROW_COUNT) &&
                (to.getColumnCoord() >= 0 && to.getColumnCoord() < Board.BOARD_COLUMN_COUNT))
            {
                return true;
            }

            return false;
        }

        private static bool isLocked(BoardSquare from)
        {
            return from.getCurrentObject().getLocked();
        }

        /*
        private static bool isFieldFree(BoardSquare to)
        {
            if (to.getCurrentObject() is None)
            {
                return true;
```

```
            }

            return false;
        }
        */


        public int executeMove()
        {
            if (valid)
            {
                BoardObject co = from.takeCurrentObject();
                to.setCurrentObject(co);
                int reutrnedfours = board.findFours();
                if (reutrnedfours == 1)
                {
                    from.setCurrentObject(co);
                    to.takeCurrentObject();
                    return 1;
                }
                //board.toggleTurnOwner();
                board.setLastMove(this);
                board.isThrees(to);

                return 0;

            }
            else
            {
                //throw new InvalidMoveExecutionException("Invalid move execution requested
:(.");
            }
            return -1;
        }
    }


[Serializable]
    public class None : BoardObject
    {
        public None()
        {
            this.boardObjectSide = BoardSide.NONE;
            this.boardObjectType = BoardObjectType.NONE;
        }
    }

public class NoneTurnException : Exception
    {
        public NoneTurnException()
        { }

        public NoneTurnException(string message)
            : base(message)
        { }
    }

[Serializable]
    public class Pawn : BoardObject
    {
        public Pawn(BoardSide boardObjectSide)
        {
            this.boardObjectSide = boardObjectSide;
            this.boardObjectType = BoardObjectType.PAWN;     }    }
```

```
public class PawnOverrideException : Exception
    {
        public PawnOverrideException()
        { }

        public PawnOverrideException(string message)
            : base(message)
        { }
    }


public class UtilityRating
    {
        public static int MAX_RATING = Int32.MaxValue - 1;
        public static int MIN_RATING = (-Int32.MaxValue) + 1;
        internal static double rating = 0;


        public static double getRating(Board board)
        {
           // double rating = board.getMaxSidePawnCount() - board.getMinSidePawnCount();

            rating = 0;
            board.findAndRateTwos();
            board.findAndRateCloseToTwos();
            board.findAndRateCloseToThrees();
            board.findAndRateThrees();
            board.findAndRateCloseMoves();



            return rating;
        }
    }
```

```
public partial class MainWindow : Window
    {
        Image[,] images = new Image[6, 5];
        bool putpawns = true;
        Board board = new Board();
        bool click = false;
        bool canDelete = false;
        int x;
        int y;

        public void checkpawn()
        {
            Char[,] a = board.printBoard();
            for (int i = 0; i < 6; i++)
            {

                for (int j = 0; j < 5; j++)
                {
                    if (a[i, j] == 'M')
                    {
                        FileStream stream = new FileStream("blackb.png", FileMode.Open,
FileAccess.Read);
                        BitmapImage src = new BitmapImage();
                        src.BeginInit();
                        src.StreamSource = stream;
                        src.EndInit();
                        images[i, j].Source = src;
                    }
                    if (a[i, j] == 'm')
                    {
                        FileStream stream = new FileStream("blackb.png", FileMode.Open,
FileAccess.Read);
                        BitmapImage src = new BitmapImage();
                        src.BeginInit();
                        src.StreamSource = stream;
                        src.EndInit();
                        images[i, j].Source = src;
                    }
                    if (a[i, j] == '_')
                    {
                        FileStream stream = new FileStream("p.png", FileMode.Open,
FileAccess.Read);
                        BitmapImage src = new BitmapImage();
                        src.BeginInit();
                        src.StreamSource = stream;
                        src.EndInit();
                        images[i, j].Source = src;
                    }
                    if (a[i, j] == 'I')
                    {
                        FileStream stream = new FileStream("redb.png", FileMode.Open,
FileAccess.Read);
                        BitmapImage src = new BitmapImage();
                        src.BeginInit();
                        src.StreamSource = stream;
                        src.EndInit();
                        images[i, j].Source = src;
                    }
                    if (a[i, j] == 'i')
                    {
                        FileStream stream = new FileStream("redb.png", FileMode.Open,
FileAccess.Read);
                        BitmapImage src = new BitmapImage();
                        src.BeginInit();
                        src.StreamSource = stream;
```

```
                    src.EndInit();
                    images[i, j].Source = src;
                }
            }
        }
        public void makeboard()
        {

            int x = 0;
            int y = 110;
            int wh = 50;
            int h=12;
            for (int i = 0; i < 6; i++)
            {

                for (int j = 0; j < 5; j++)
                {
                    images[i, j] = new Image();
                    images[i, j].Margin = new Thickness(y,x,0,0) ;
                    images[i,j].HorizontalAlignment= System.Windows.HorizontalAlignment.Left;
                    images[i, j].VerticalAlignment = System.Windows.VerticalAlignment.Top;
                    images[i, j].MouseLeftButtonDown += new
MouseButtonEventHandler(button_Click);
                    images[i, j].Tag = i+" "+j +" a";
                    images[i, j].Width = wh;
                    images[i, j].Height = wh;
                    images[i, j].Stretch = Stretch.UniformToFill;
                    //Uri uri = new Uri("p.jpg", UriKind.Relative);
                    //images[i, j].Source = new BitmapImage(uri);
                    FileStream stream = new FileStream("p.png", FileMode.Open,
FileAccess.Read);
                    BitmapImage src = new BitmapImage();
                    src.BeginInit();
                    src.StreamSource = stream;
                    src.EndInit();
                    images[i, j].Source = src;

                    b.Children.Add(images[i, j]);
                    y += wh+10;

                }
                x += wh-16;
                y = 110-h;
                wh += 5;
                h += 12;
            }
        }


        void button_Click(object sender, MouseButtonEventArgs e)
        {
            if (putpawns == true)
            {
                if (board.getMinSidePawnCount() < 12)
                {
                    BoardSide who = board.getTurnOwner();
                    String a = Convert.ToString(who);
                    if (a == "MAX")
                    {
                        String b = (String)(sender as Image).Tag;
                        String[] bb = b.Split(' ');

                        int returned = board.putPawn(new BoardSquare(Convert.ToInt32(bb[1]),
Convert.ToInt32(bb[0])), new Pawn(BoardSide.MAX));
```

```
                        checkpawn();
                        // MessageBox.Show(string.Format("Kliknęłeś pionek na pozycji: {0}",
(sender as Image).Tag));
                        if (returned == 2)
                        {
                            MessageBox.Show("Niedozwolony ruch!");
                        }
                        else
                            if (returned == 1)
                            {
                                MessageBox.Show("Pionek na tym polu już istnieje!");

                            }
                            else
                            {


                                board.putPawn(new Pawn(BoardSide.MIN));
                                checkpawn();
                            }

                    }
                }
                if (board.getMinSidePawnCount() == 12)
                    putpawns = false;
            }
            else
            {
                int testtocheck = 0;
                int checkStateGame = board.isEnd(testtocheck);
                if (checkStateGame == 0)
                {
                    int returned = -1;
                    int returneed = -1;
                    if (click == true)
                    {
                        String b = (String)(sender as Image).Tag;
                        String[] bb = b.Split(' ');

                        Move mv = new Move(board, board.getBoardSquare(x, y),
board.getBoardSquare(Convert.ToInt32(bb[0]), Convert.ToInt32(bb[1])));
                        returned = mv.executeMove();
                        if (returned == 1)
                            MessageBox.Show("Niemożliwy ruch!");

                        checkpawn();



                        click = false;
                        this.images[x, y].Tag = x + " " + y + " " + "a";


                    }
                    if (click == false)
                    {

                        String b = (String)(sender as Image).Tag;
                        String[] bb = b.Split(' ');
                        //MessageBox.Show(images[Convert.ToInt32(bb[0]),
Convert.ToInt32(bb[1])].Tag.ToString());
                        this.images[Convert.ToInt32(bb[0]), Convert.ToInt32(bb[1])].Tag =
bb[0] + " " + bb[1] + " " + "b";
                        // MessageBox.Show(images[Convert.ToInt32(bb[0]),
Convert.ToInt32(bb[1])].Tag.ToString());
```

```
                    x = Convert.ToInt32(bb[0]);
                    y = Convert.ToInt32(bb[1]);
                    click = true;
                }


                if (Board.delete == true)
                {

                    if (canDelete == true)
                    {

                        String b = (String)(sender as Image).Tag;
                        String[] bb = b.Split(' ');
                        returneed = deletePawn(Convert.ToInt32(bb[0]),
Convert.ToInt32(bb[1]));
                        if (returneed == 1)
                            MessageBox.Show("Tego pionka nie można usunąć!");
                    }
                    else
                    {
                        if (board.getTurnOwner().ToString() == "MAX")
                        {
                            MessageBox.Show("Usuń pionek!");
                            canDelete = true;
                        }
                        else
                        {
                            List<BoardSquare> AllMaxPawns = board.findAllMax();
                            Random rand = new Random();
                            int a = rand.Next(AllMaxPawns.Count - 1);
                            int i = 1;
                            while (i == 1)
                            {
                                if (i == 1)
                                {
                                    i = deletePawn(AllMaxPawns[a].getRowCoord(),
AllMaxPawns[a].getColumnCoord());
                                    a = rand.Next(AllMaxPawns.Count - 1);
                                }
                            }

                        }


                    }




                }
                bool enemydel = false;
                if (Board.delete != true)
                {

                    if (returned == 0 || returneed == 0)
                    {
                        board.toggleTurnOwner();
                        Move nextMove = Minmax.getNextMove(board);
                        //Board.delete = false;
                        int returnedd = nextMove.executeMove();
                        while (returnedd == 1)
                        {
                            nextMove = Minmax.getNextMove(board);
                            Board.delete = false;
```

```
                            returnedd = nextMove.executeMove();
                        }
                        board = nextMove.getBoard();
                        board.toggleTurnOwner();

                        checkpawn();
                        if (Board.delete == true)
                            enemydel = true;
                    }
                }
                if (Board.delete == true)
                {

                    if (enemydel == true)
                    {

                        if (board.getTurnOwner().ToString() == "MAX")
                        {
                            List<BoardSquare> AllMaxPawns = board.findAllMax();
                            Random rand = new Random();
                            int a = rand.Next(AllMaxPawns.Count - 1);
                            int i = 1;
                            while (i == 1)
                            {
                                if (i == 1)
                                {
                                    i = deletePawn(AllMaxPawns[a].getRowCoord(),
AllMaxPawns[a].getColumnCoord());
                                    a = rand.Next(AllMaxPawns.Count - 1);
                                }
                            }
                            enemydel = false;
                        }


                    }







                }
            }
            if (checkStateGame == 1)
                MessageBox.Show("KONIEC! WYGRAŁEŚ!!!!");
            if (checkStateGame == 2)
                MessageBox.Show("KONIEC! PRZEGRAŁEŚ...");

        }
    }



    public MainWindow()
    {
        InitializeComponent();

        //BoardSquare bs = new BoardSquare(0, 0);
        //board.putPawn(bs, new Pawn(BoardSide.MIN));
```

```
    //bs = new BoardSquare(1, 1);
    //board.putPawn(bs, new Pawn(BoardSide.MAX));
    //board.putPawn(new Pawn(BoardSide.MAX));
    //board.putPawn(new Pawn(BoardSide.MAX));
    //board.putPawn(new Pawn(BoardSide.MAX));
    //board.putPawn(new Pawn(BoardSide.MAX));
    //board.putPawn(new Pawn(BoardSide.MAX));
    //board.putPawn(new Pawn(BoardSide.MIN));
    //board.putPawn(new Pawn(BoardSide.MIN));
    //board.putPawn(new Pawn(BoardSide.MIN));
    //board.putPawn(new Pawn(BoardSide.MIN));
    //board.putPawn(new Pawn(BoardSide.MIN));

    makeboard();
   // checkpawn();


}
public int deletePawn(int x, int y)
{
    BoardSquare bs = board.getBoardSquare(x, y);
    if (bs.getCurrentObject().getLocked() == false)
    {
        bs.takeCurrentObject();
        checkpawn();
        Board.delete = false;
        canDelete = false;
        return 0;
    }
    else
    {
        return 1;



    }
}
private void button1_Click(object sender, RoutedEventArgs e)
{


    Move nextMove = Minmax.getNextMove(board);
    Board.delete = false;
    int returned=nextMove.executeMove();
    while (returned == 1)
    {
        nextMove = Minmax.getNextMove(board);
        Board.delete = false;
        returned = nextMove.executeMove();
    }
    board = nextMove.getBoard();
    checkpawn();
}

private void image1_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)
{
    if (textBlock4.Visibility == Visibility.Visible)
    {
        textBlock2.Visibility = Visibility.Hidden;
        textBlock3.Visibility = Visibility.Hidden;
        textBlock4.Visibility = Visibility.Hidden;

    }
    else
    {
```

```
                textBlock5.Text = "";
                textBlock2.Visibility = Visibility.Visible;
                textBlock3.Visibility = Visibility.Visible;
                textBlock4.Visibility = Visibility.Visible;
            }
        }

        private void documentViewer1_PageViewsChanged(object sender, EventArgs e)
        {

        }

        private void image2_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)
        {
            if (textBlock5.Text == "")
            {
                textBlock2.Visibility = Visibility.Hidden;
                textBlock3.Visibility = Visibility.Hidden;
                textBlock4.Visibility = Visibility.Hidden;
                string txt = "1. Zasady gry \n • gracze decydują kto zaczyna pierwszy,\n• 
plansza do gry ma wielkość 5x6,\n• każdy z graczy posiada 12 pionków,\n• pierwsza faza polega 
na ułożeniu naprzemiennie wszystkich pionków na planszy,\n• faza druga polega na 
naprzemiennym przesuwaniu pionków w celu uzyskania rzędu,pionowego lub poziomego o długości 
3,\n• jeżeli graczowi uda się uzyskać rząd pionków o długości 3 to może on pozbawić 
drugiego,\ngracza jednego pionka,\n• rzędy o długości 3 podczas pierwszej fazy nie liczą się 
(nie można pozbawić drugiego gracza jego pionka),\n• gracz który nie może uzyskać już rzędu o 
długości 3 za pomocą swoich pionków przegrywa.";
                textBlock5.Text = txt;
            }
            else
            {
                textBlock5.Text = "";
            }
        }

        private void image2_MouseEnter(object sender, MouseEventArgs e)
        {

        }

        private void button1_Click_1(object sender, RoutedEventArgs e)
        {
            Process.Start(Application.ResourceAssembly.Location);

            Application.Current.Shutdown();
        }
```