САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО

Институт прикладной математики и механики

Высшая школа теоретической механики

Курсовая работа

Дисциплина: Компьютерные технологии в механике

«Разработка игры Tic Tac Toe на языке C#»

Студент: Минина А.В.

Группа: 3640103/90301

Преподаватель: Ле-Захаров А.А.

Работа принята «____»_____2020 г.

Подпись_____

Санкт-Петербург

2020 г.

Оглавление

Введение

Tic Tac Toe – игра, известная в русском сегменте как «Крестики – Нолики». Игроки по очереди ставят на свободные клетки поля 3x3 знаки (один всегда **крестики**, другой всегда **нолики**). Первый, выстроивший в ряд 3 своих фигуры по вертикали, горизонтали или диагонали, выигрывает.

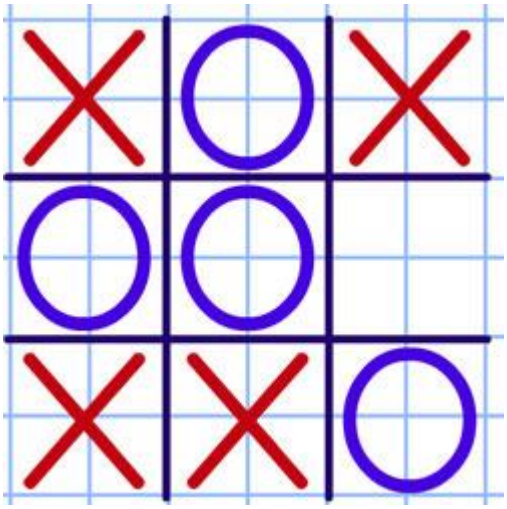

Рисунок 1 – Примерный внешний вид игры

Цель работы

Разработать игру «Крестики – Нолики». В итоге должна быть возможность играть с человеком или с компьютером. При игре человека с компьютером – компьютер должен обладать искусственным интеллектом и принимать решения на основе хода противника. В идеале нужно реализовать возможность игры компьютера против компьютера. Также следует разработать приятный интерфейс для пользователя.

Разработка консольной версии игры для двух игроков

В качестве первой цели было решено создать консольное приложение, которое осуществляет игру двух игроков – людей.

Чтобы игра могла существовать, необходимо создать:

- Функцию, отвечающую за старт игры StartGame
- Функцию, отвечающую за процесс игры PlayGame
- Функцию, отвечающую за победителя CheckForVictory
- Функцию, отображающую доску для игры PrintTicTacToeBoard
- Функцию для выбора игрока ChoosePlayer
- Функцию для инициализации доски Initialize Board
- В main`е будет создаваться новая игра и начинаться игра

Блок – схемы и описание функций

Рисунок 2 -схема StartGame

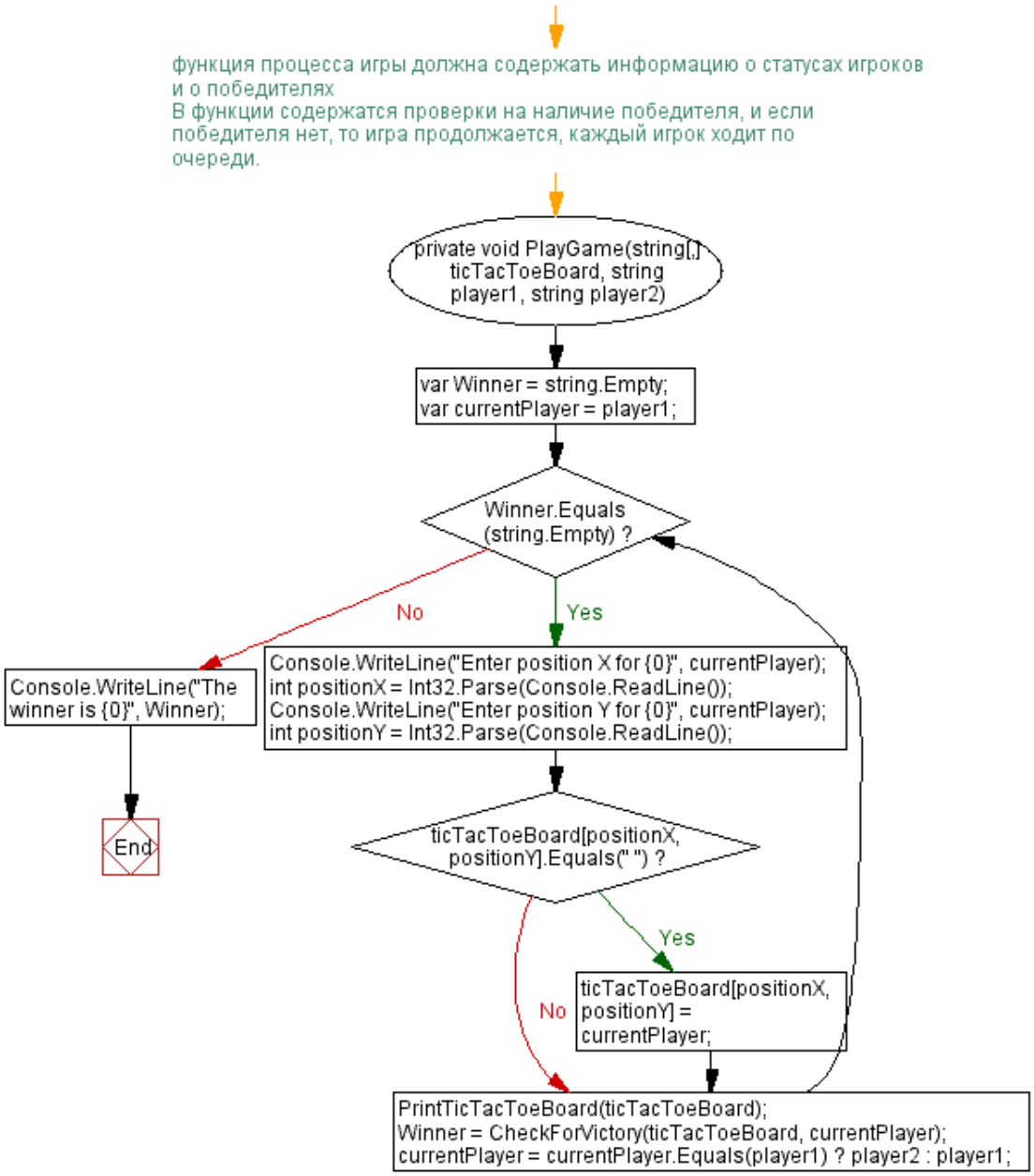

Рисунок 3 – схема PlayGame

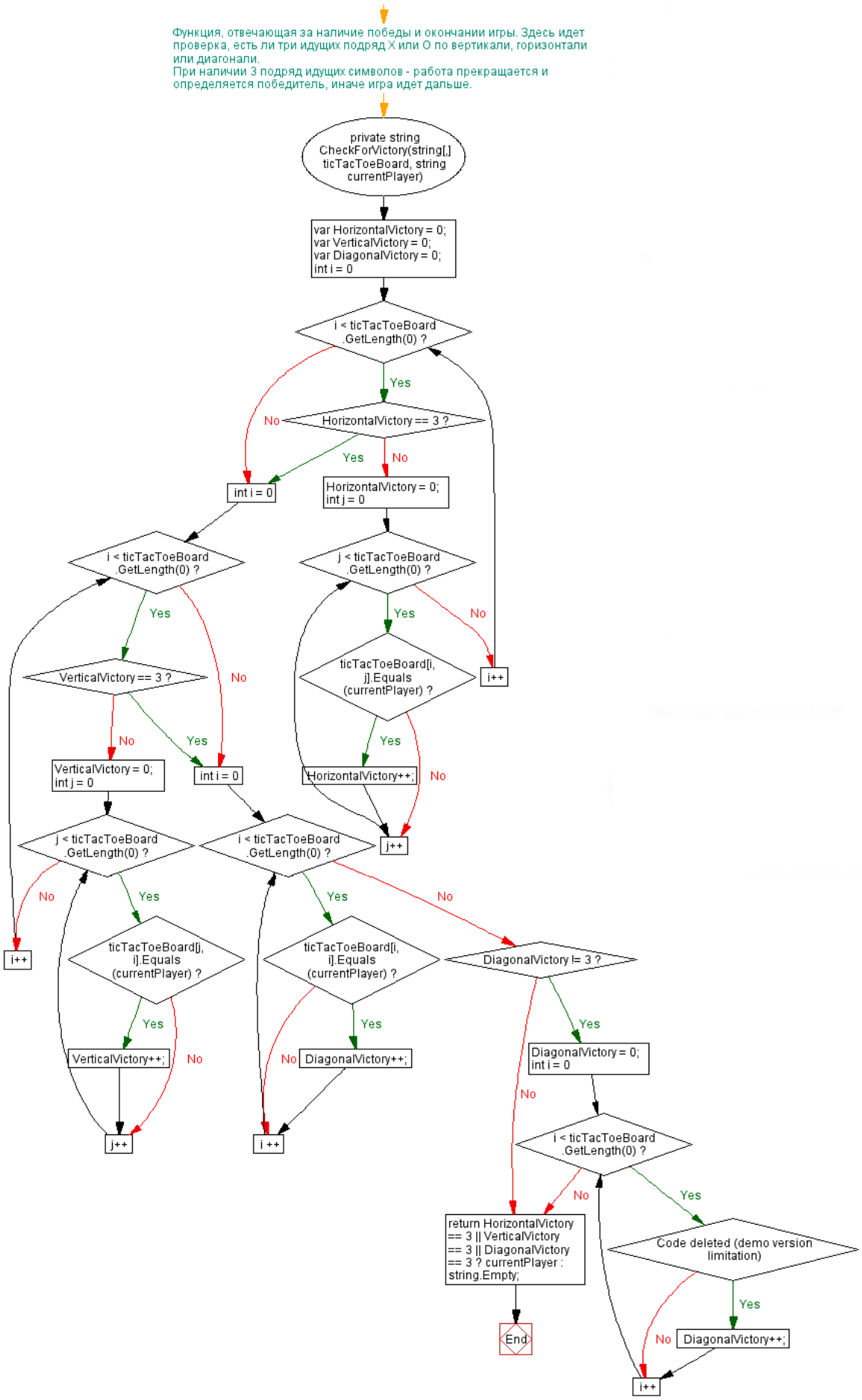

Рисунок 4 – Схема CheckForVictory

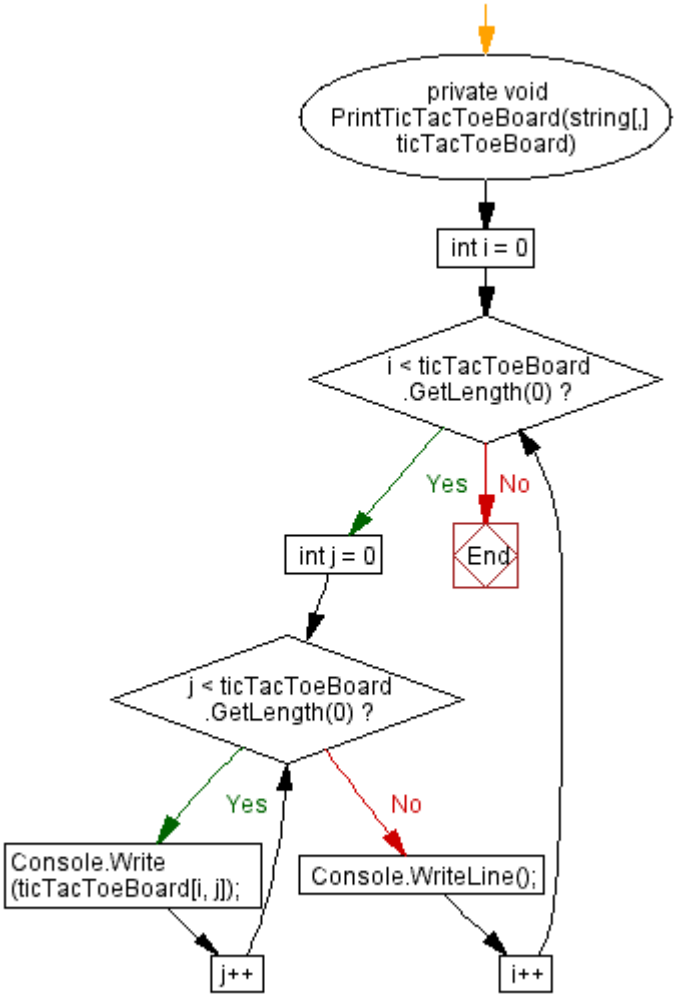

Рисунок 5 – схема PrintTicTacToe

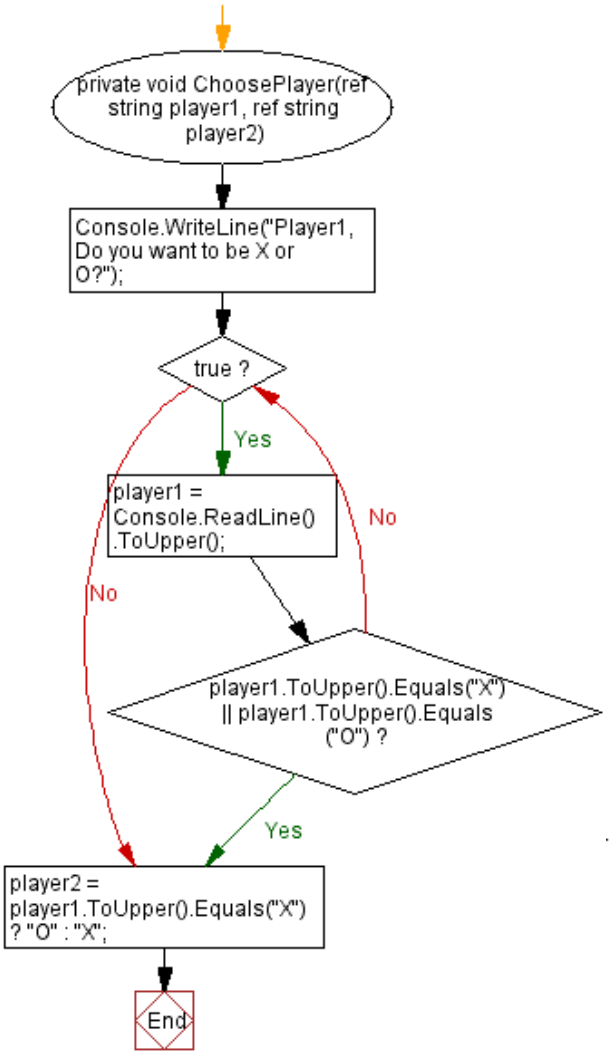

Рисунок 6 – схема ChoosePlayer

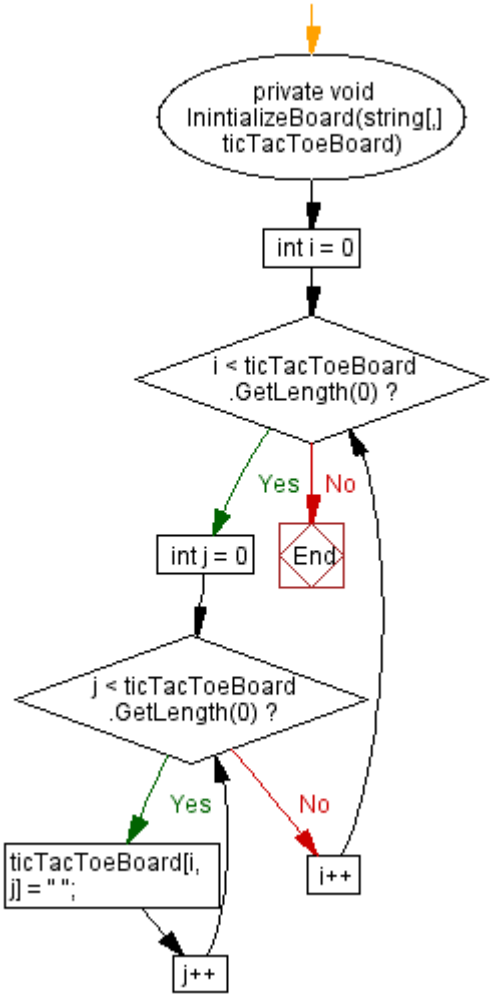

Рисунок 7 – схема IninializeBoard

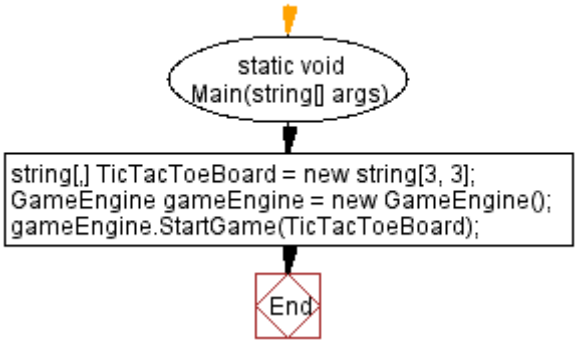

Рисунок 8 – схема Main`a

Пример работы консольного приложения «Игрок против Игрока»

Игрок, который играет первым имеет право выбрать, будет он играть за «крестик» или «нолик». После выбора, второму игроку автоматически присваивается второе свободное значение.

Затем игроки поочередно вводят крестики и нолики, после каждого ввода выводится обновленная доска со значением, заданным игроком.

```
C:\Users\shevc\source\repos\TicTacToeWPF\TicTacToeWPF\bin\Debug\netcoreapp3.0\TicTacToeWPF.exe
Player1, Do you want to be X or O?
X
Enter position X for X
0
Enter position Y for X
1
 X


Enter position X for O
0
Enter position Y for O
0
OX


Enter position X for X
1
Enter position Y for X
2
OX
  X

Enter position X for O
```

Рисунок 9 – пример заполнения поля игроками

```
 OX
Enter position X for X
0
Enter position Y for X
2
OXX
 X
 OX
Enter position X for O
1
Enter position Y for O
0
OXX
OX
 OX
Enter position X for X
1
Enter position Y for X
2
OXX
OXX
 OX
The winner is X
```

Рисунок 10 – пример победы игрока, игравшего за «крестики»

Разработка игры против компьютера с реализацией интерфейса

Для того, чтобы была возможность игры человека с искусственным интеллектом, необходимо переработать игру. Я посчитала нужным писать игру заново, частично изменив её логику и затраты памяти. В предыдущей, консольной версии человек против человек – очень много вложенных циклов и условий.

- **Доска**

Доска для игры также представляет собой двумерный массив из чисел. Пусть 0 значение будет представлять пустое место, 1 – X, а 2 – O.

Позицию на доске можно выбрать либо указав координаты (строка и столбец), либо указав номер позиции.

0	1	2
3	4	5
6	7	8

Рисунок 11 – позиции на доске в игре

- **Сама игра**

Помимо доски нужно создать класс, который будет моделировать сам процесс игры.

В него должны входить:

1. Отслеживание хода каждого игрока и процесс ходов на игровом поле
2. Конец игры, когда игрок выигрывает или ничья.

Когда будет предприниматься попытка хода, должна выполняться проверка, что движение валидно, и сделан ли ход, в ином случае выбрасывается исключение.

- **Создание игроков**

В данной реализации игры будет два вида игроков: компьютер и человек.

Игрок – человек отвечает на ход от живого человека.

Игрок – компьютер будет искать ходы из дерева, которое он построит, исходя из хода оппонента.

Абстрактный класс игрока HumanPlayer будет включать в себя общие свойства и функционал для игроков – людей и компьютера.

В класс будет включено уведомление, когда был сделан ход. Это нужно для того, чтобы не блокировался пользовательский интерфейс в ожидании, пока игрок выберет свой ход.

- **Искусственный игрок (противник – компьютер)**

Как было сказано ранее, компьютер должен использовать «умные» ходы. Для этого нужно найти способ определить, насколько хорош будущий ход. Для этого будет определена функция оценки e(m), которую компьютер будет использовать, чтобы увидеть, насколько благоприятен конкретный ход m. Если следовать обычной практике, то выбирается значение e(m) большим, если ход может привести к выигрышу, и меньшим, если он может привести к проигрышу. Для реализации функции оценки лучше всего использовать алгоритм MinMax, который поможет выбрать ход, максимизирующий e(m).

- **Определение функции оценки**

Существует много способов построить функцию оценки игры, которая будет соответствовать критериям игры.

Один из способов, который я намерена использовать состоит в том, что для каждой строки, столбца и диагонали на доске подсчитывается количество

крестиков или ноликов игроков, которые присутствуют, если это возможно для победы в этом ряду, столбце или диагонали. Затем нужно вычесть, что из этого мы получаем из такой же оценки для части противника.

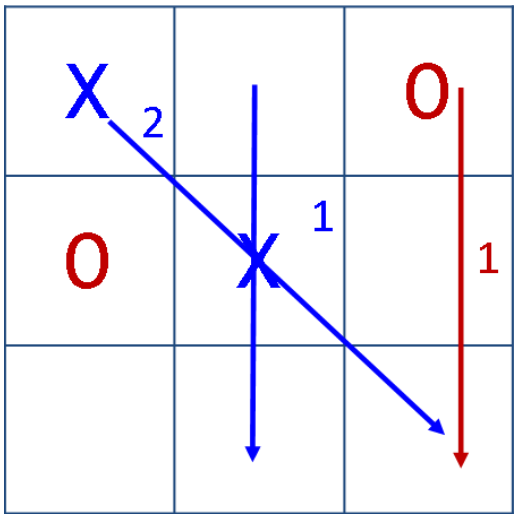

Рисунок 12 – Пример расчета e(m) с следующим ходом «X». Для «X» два значка по диагонали = 2; 1 «X» во втором столбце = 1; Для «O», «O» в третьем столбце = 1; e(m) = 3-1=2. Это расположение наиболее благоприятно для крестиков.

Удобнее всего определить очень большое значение для e(m), когда расположение представляет выигрыш для «крестиков», и очень маленькое значение, когда будет проигрыш для «крестиков». Можно выбрать любое достаточно большое число, если есть гарантия, что для проигрышных расположений невозможно получить одинаковое e(m). Можно использовать $\pm\infty$.

➢ Описание MinMax алгоритма для игры

Чтобы помочь компьютеру принять решение об «умном» ходе, сделаем дерево возможных решений вниз на заданную глубину. Глубина показывает, на сколько уровней ходов компьютер будет смотреть дальше. Дерево с глубиной 2 содержит все возможные ходы, которые может сделать компьютер, и все последующие ходы, которыми может противостоять противник компьютера. Корень дерева будет представлять начальное расположение.

Всего будет два вида вершин: Min и Max. Вершины Max используются для представления расположений, по которым должен перемещаться

компьютер, а вершины Min представляют расположение на доске, по которым должен перемещаться противник компьютера. Таким образом, корень дерева всегда является Max. Вершины Max выбирают движение, которое приводит к значению Максимума значений e(m) от его дочерних элементов.

«Дети» корневой вершины будут представлять расположение всех возможных ходов, которые может сделать компьютер. Эти вершины являются Min, потому что они представляют собой расположения, для которых будет двигаться «Нолик» (противник), и по этой причине они будут выбирать ходы, которые минимизируют значения e(m). Построение дерева продолжается, генерируя дочерние элементы для этих вершин Min, которые, конечно, являются и вершинами Max, поскольку в представленных расположениях на доске снова будет очередь компьютера.

Построение дерева продолжается подобным образом, пока не достигается указанная глубина.

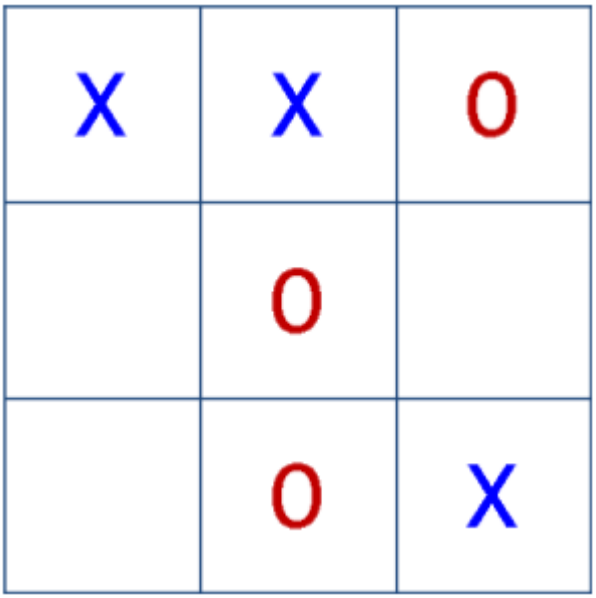

Рисунок 13 – Ход компьютера – крестик, дерево глубины 2

Пусть на рисунке 13 – корень дерева. Он представляет собой расположение, исходя из которого компьютер должен выбрать ход, он является Max. Компьютер может переместиться в ячейки 3, 5 и 6. Каждое из перемещений приводит к новому расположению и будет представляться, как дочерние вершины корневого элемента. Так как дочерние вершины представляют расположения, по которым противник должен двигаться, они являются Min.

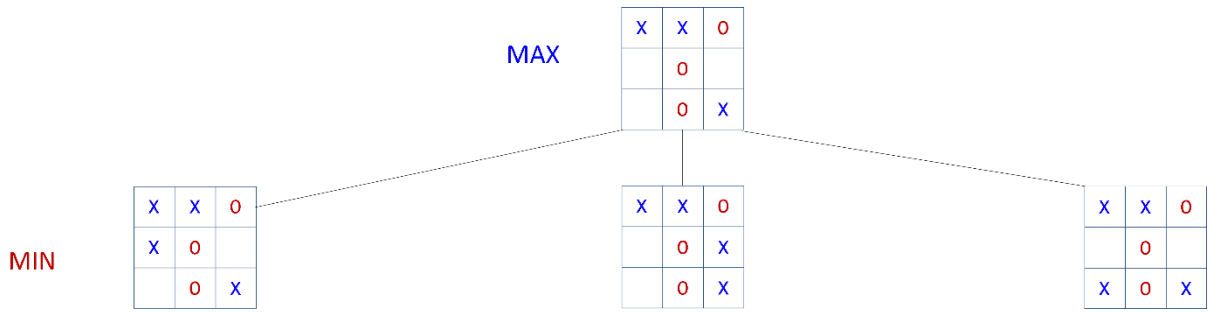

Рисунок 14 – Дерево игры, показывающее трёх «деток», порожденных из корня дерева, представляющих перемещение крестика в ячейки 3, 5 и 6

Теперь нужно создать другой набор из дочерних вершин для каждого из Min-вершин, чтобы достичь заданной глубины 2. Так как эти дочерние вершины являются дочерними вершинами Min-вершин, то они будут Max-вершинами. Так будет всегда. У вершин Max есть дочерние вершины Min и наоборот.

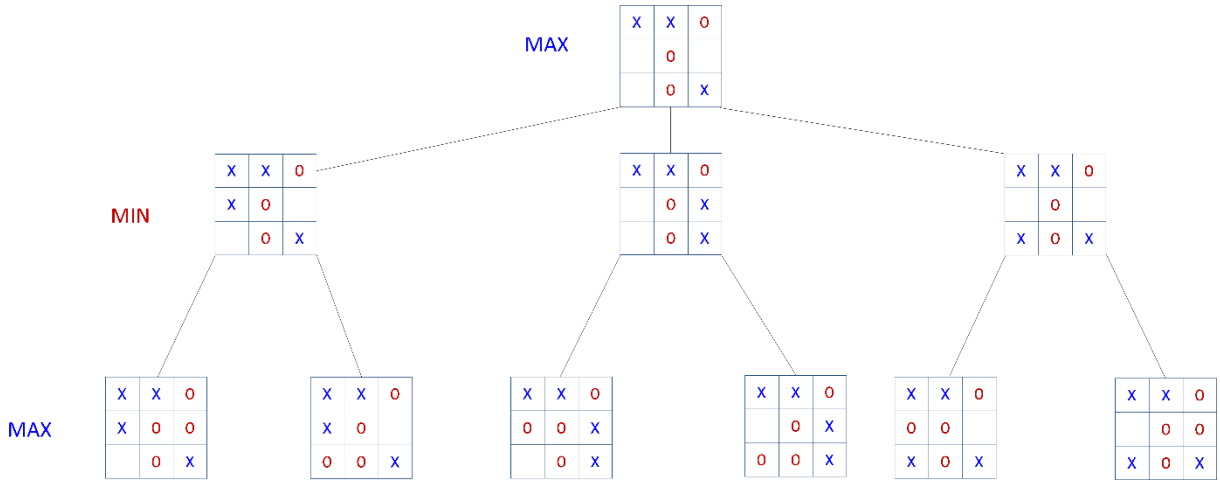

Рисунок 15 – Игра с деревом глубины 2

Алгоритм MinMax применяется снизу вверх. На рисунке 16 показано дерево со значениями e(m), рассчитанными функцией оценки для нижних дочерних вершин.

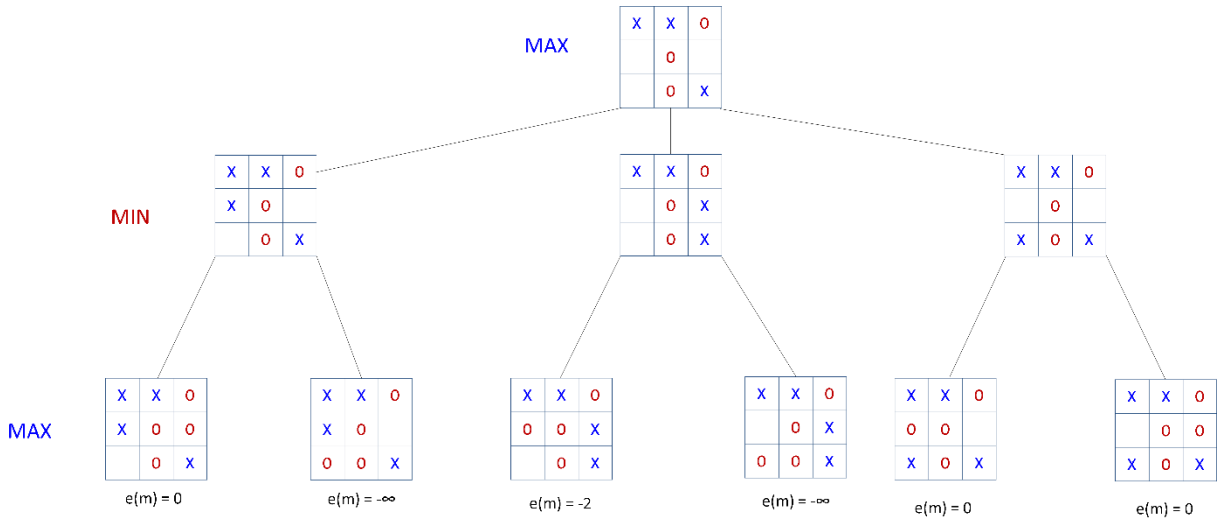

Рисунок 16 – Дерево с расчетом e(m) для самых нижних «деток»

С помощью значений e(m), рассчитанных для нижних дочерних вершин, каждая из вершин Min может выбрать свой лучший ход. Лучший ход для вершины Min – это дочерняя вершина, которая имеет минимальное значение, поскольку она представляет лучшее расположение для «ноликов». На рисунке 17 показано дерево после того, как вершинам Min были присвоены значения их лучших дочерних вершин. Значения, выбранные вершинами Min, показаны в кружке справа от вершин Min. Они отображаются таким образом, чтобы отличать выбранные значения от значений, которые рассчитываются непосредственно из функции оценки. Только вершины на нижнем уровне используют функцию оценки для вычисления своих значений.

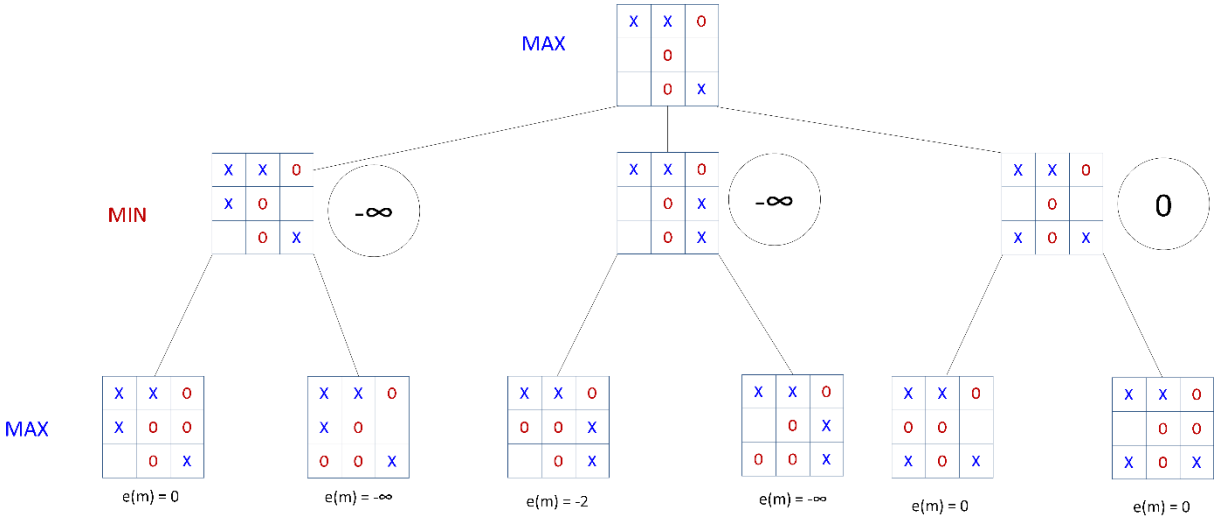

Рисунок 17 – Вершинам Min присваиваются значения, представляющие их лучший ход, который является дочерней вершиной с минимальным значением e(m)

Теперь корневая вершина выбирает максимальное значение из числа своих «детей». Вершина с e(m) = 0 является максимумом, поэтому корень дерева выбирает его как лучший ход для компьютера (перемещение в ячейку 6). Алгоритм прогнозирует потери, если выполняется перемещение в ячейки 3 или 5, где e(m) = ± ∞ для этих перемещений.

➢ Внедрение MinMax – алгоритма

Также, как и в случае с игроком-человеком, класс, отвечающий за игрока-компьютера, будет расширять Player и должен реализовывать метод движения Move. Можно создать дерево поиска, используя объекты Node, которые будут представлять определенный сделанный ход.

Как уже ранее говорилось, существуют два типа вершин: Max и Min. В коде определяются классы MaxNode и MinNode, которые будут представлять вершины Max и Min соответственно. Будет использоваться абстрактный класс Node для захвата общих свойств и функциональности вершин Min и Max.

Рисунки 18, 19, 20, 21, 22, 23, 24, 25, 26 и 27 представляют блок-схемы абстрактного класса Node

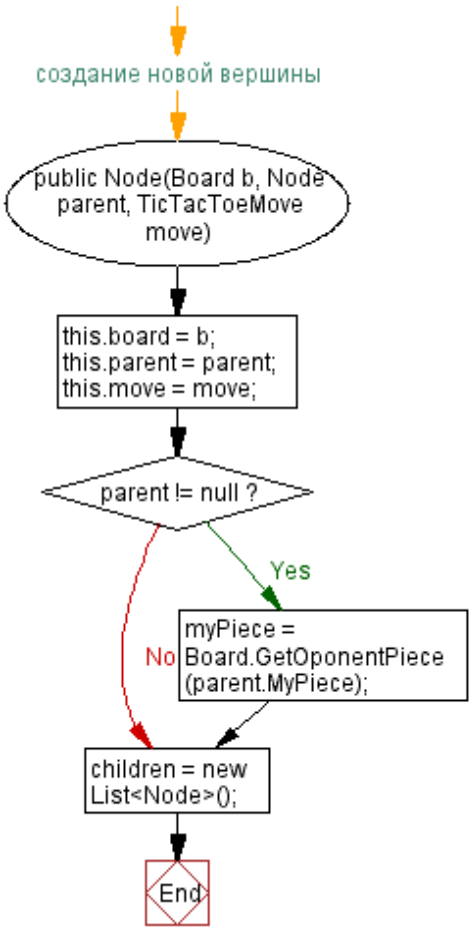

Рисунок 18

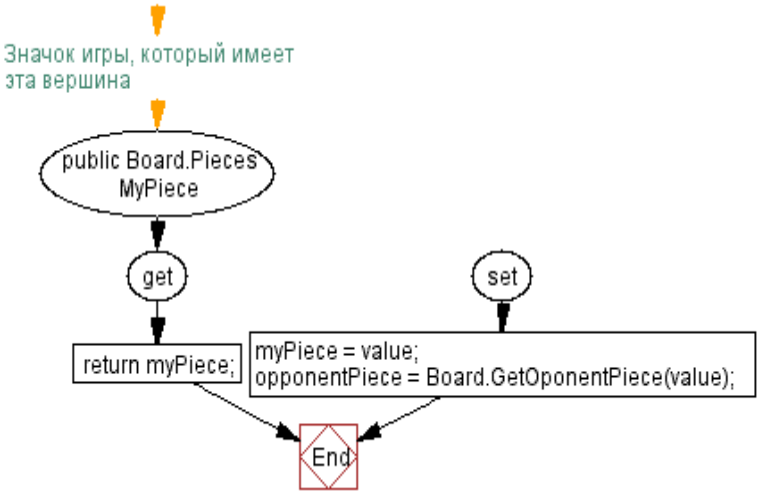

Рисунок 19

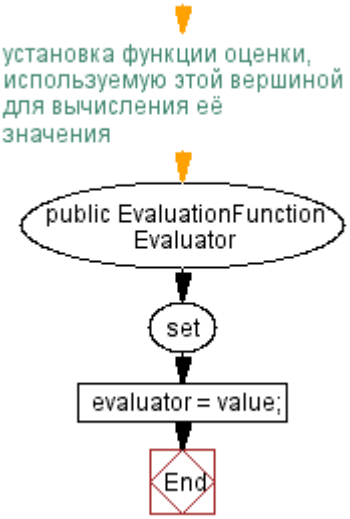

Рисунок 20

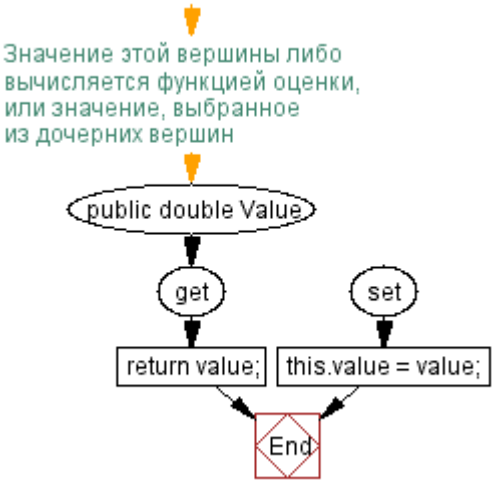

Рисунок 21

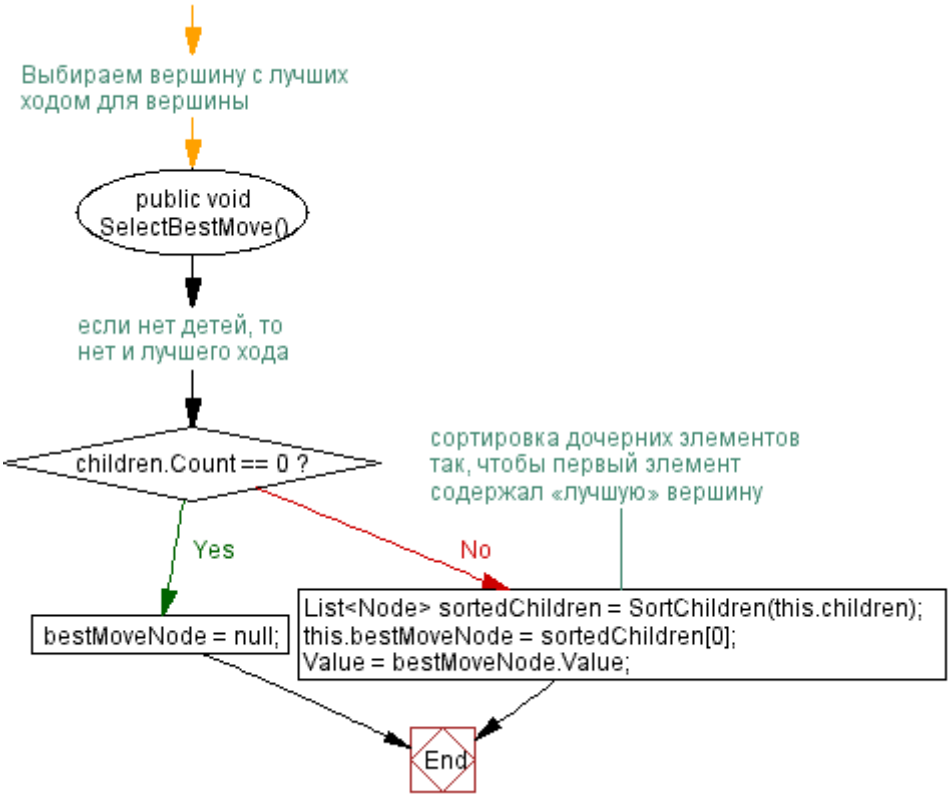

Рисунок 22

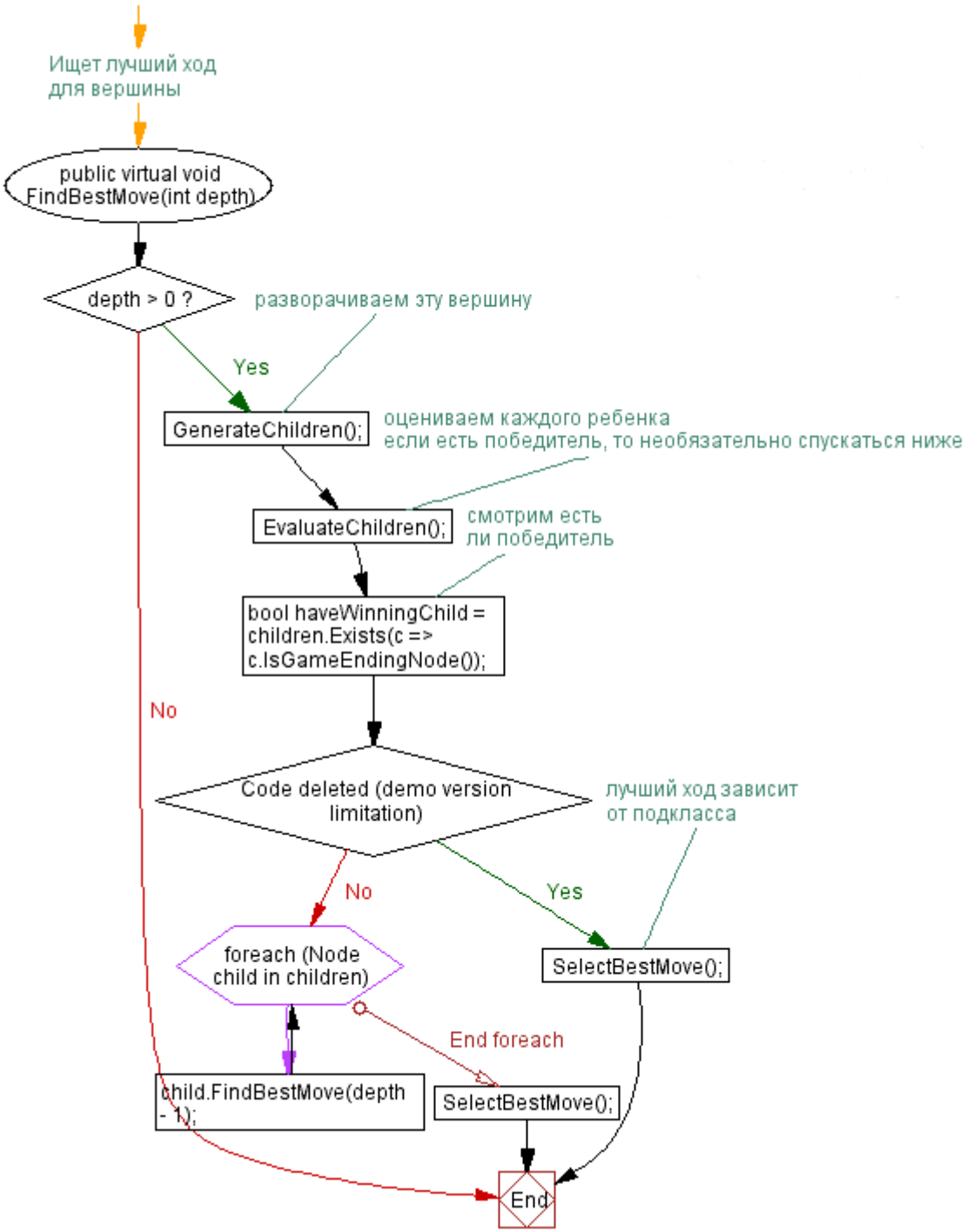
Ищет лучший ход
для вершины
public virtual void
FindBestMove(int depth)
depth > 0 ?
разворачиваем эту вершину
Yes
GenerateChildren();
оцениваем каждого ребенка
если есть победитель, то необязательно спускаться ниже
EvaluateChildren();
смотрим есть
ли победитель
bool haveWinningChild =
children.Exists(c =>
c.IsGameEndingNode());
No
Code deleted (demo version
limitation)
лучший ход зависит
от подкласса
No
Yes
foreach (Node
child in children)
SelectBestMove();
End foreach
child.FindBestMove(depth
- 1);
SelectBestMove();
End

Рисунок 23

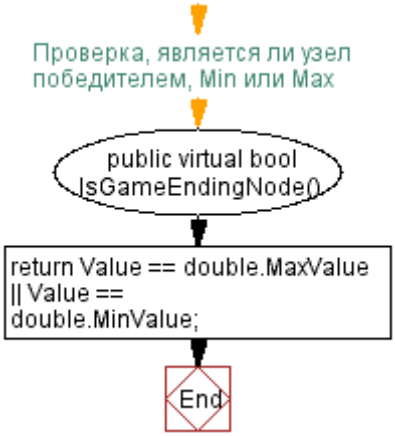

Рисунок 24

Рисунок 25

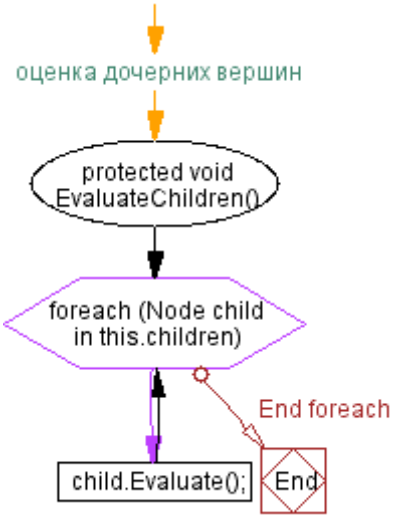

Рисунок 26

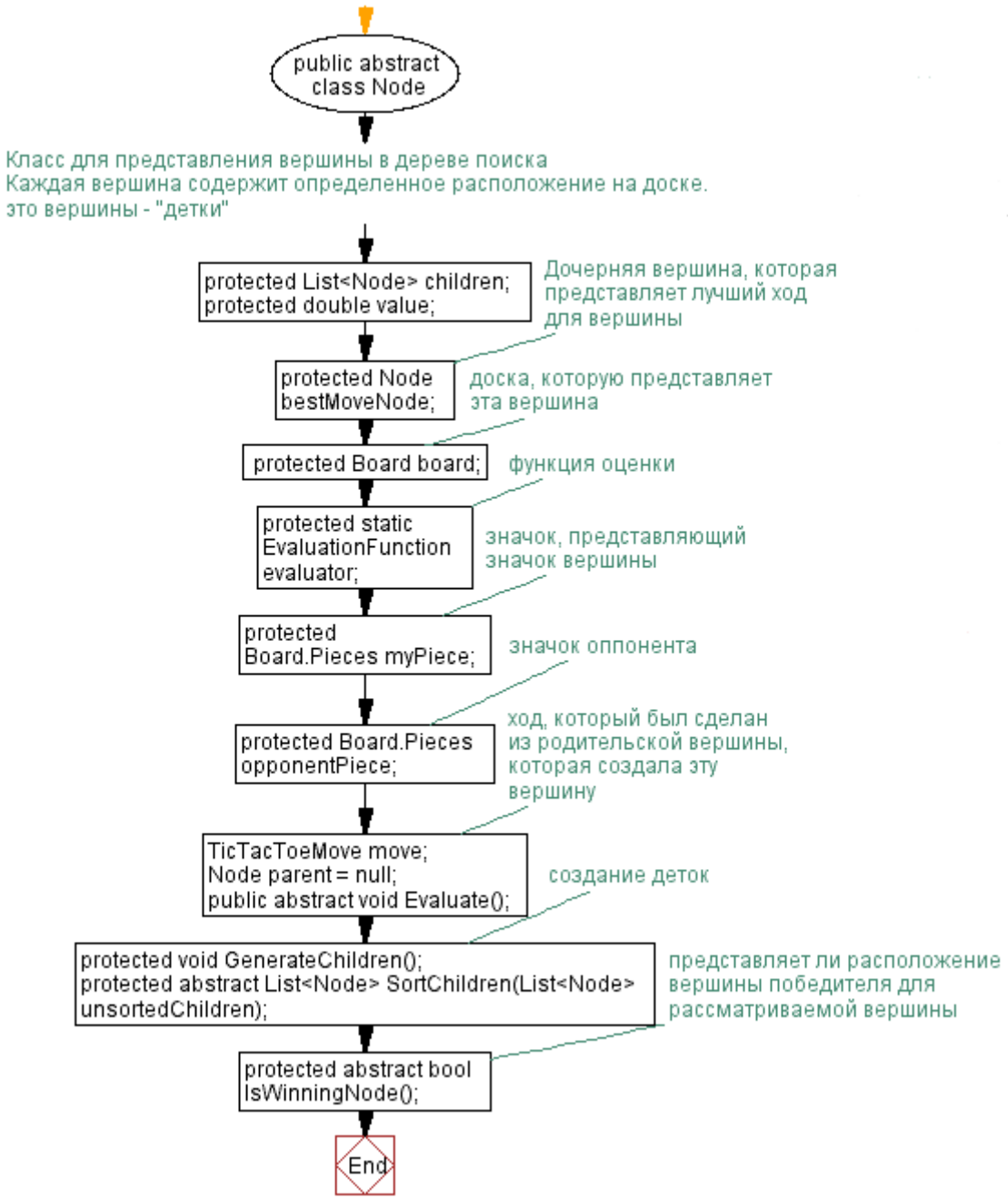

Рисунок 27

Все объекты Node имеют родителя (за исключением корневой вершины) и ноль или более дочерних объектов. Каждая вершина имеет значение, связанное с ней, либо вычисленное непосредственно из функции оценки, либо путем выбора максимального (для вершин Max) или минимального (для вершин Min) значений дочерних элементов. Класс Node также отслеживает ход, сделанный на родительской доске для создания собственной доски. Существует также свойство BestMove, которое позволяет вершине

отслеживать её лучший ход. Метод FindBestMove, определенный в классе Node, создает дерево игры и управляет поиском лучшего хода.

Рисунки 28, 29, 30, 31 и 32 представляют блок-схемы класса MaxNode, который наследуется от Node

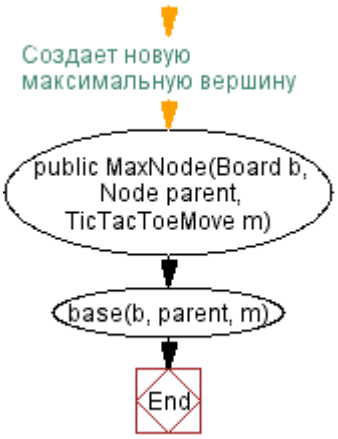

Рисунок 28

Рисунок 29

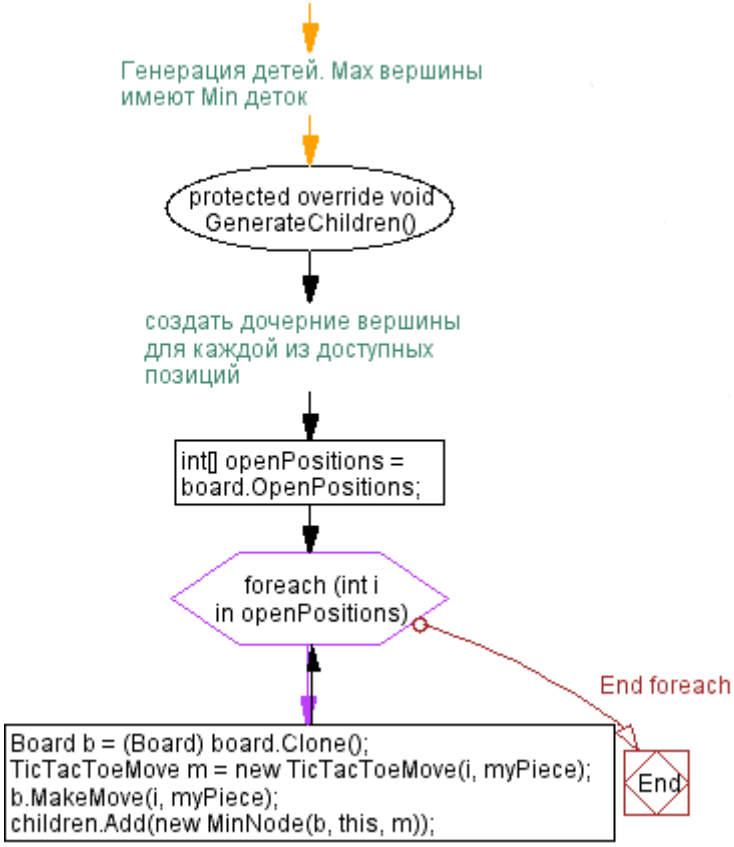

Рисунок 30

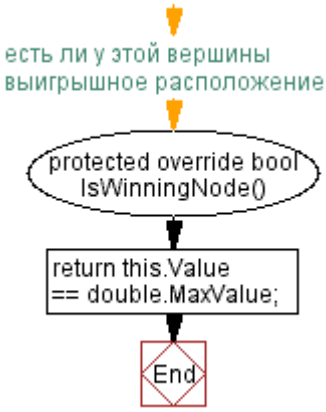

Рисунок 31

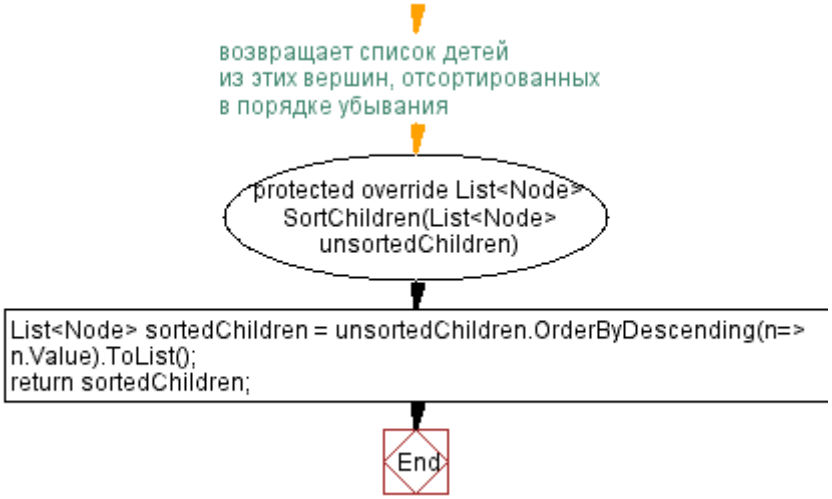

Рисунок 32

Рисунки 33, 34, 35, 36 и 37 представляют блок-схемы класса MinNode, который наследуется от Node

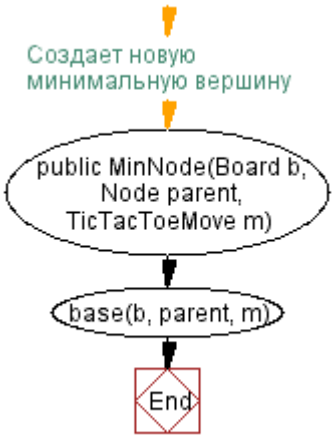

Рисунок 33

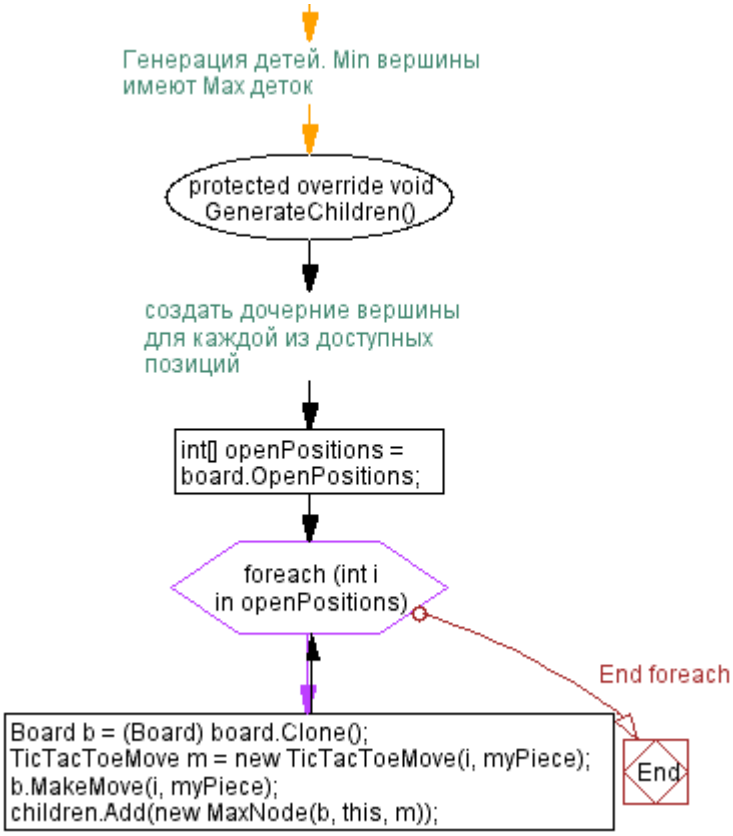

Рисунок 34

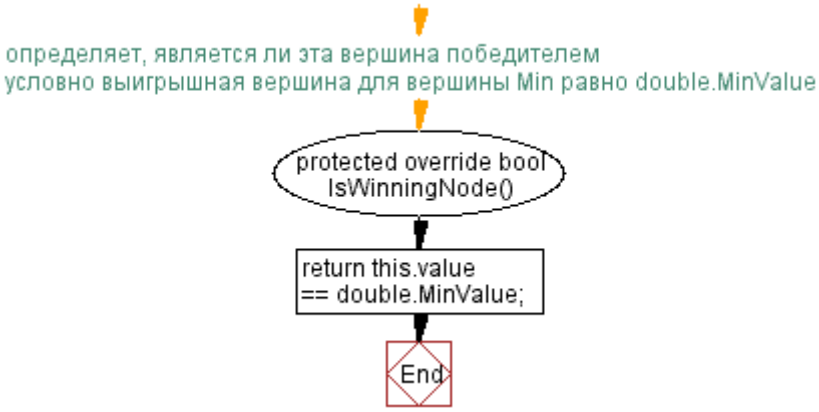

Рисунок 35

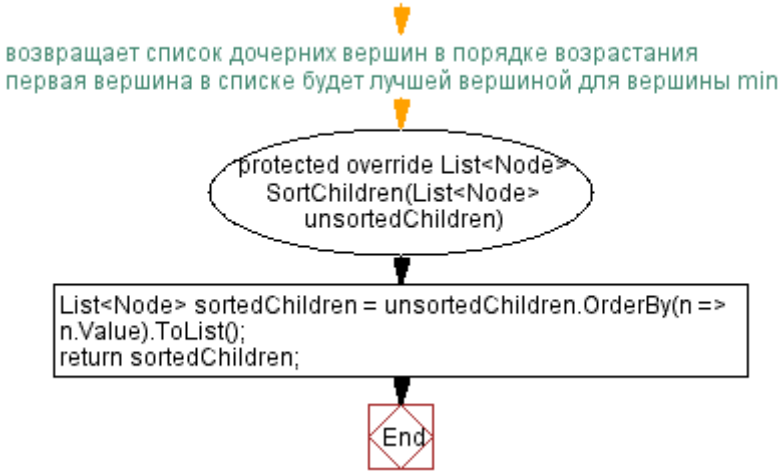

Рисунок 36

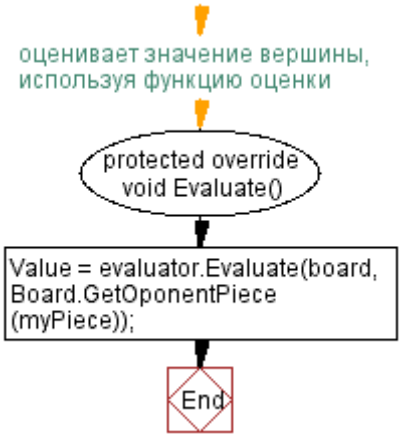

Рисунок 37

Видно, что абстрактный класс Node реализует поиск в методе FindBestMove, в то время как MaxNode и MinNode добавляют некоторые специфичные для вершин операции, такие как сортировка и создание дочерних элементов.

Создавая деревья игр и выбирая лучшие ходы, можно реализовать класс ComputerPlayer.

Рисунки 38, 39, 40, 41 и 42 представляют блок-схемы класса ComputerPlayer, глубина DEFAULT_SEARCH_DEPTH = 2.

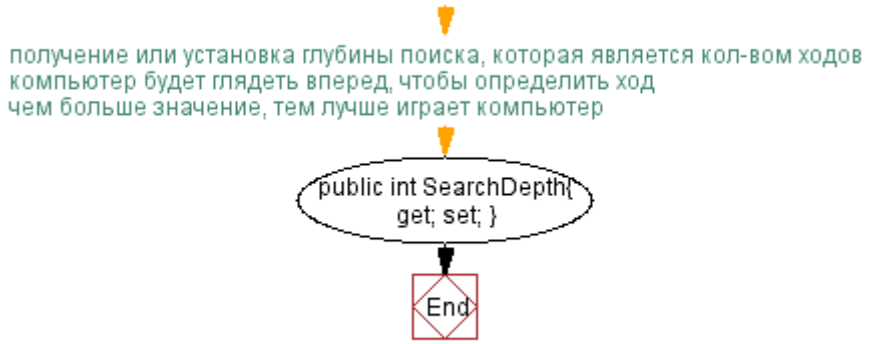

Рисунок 38

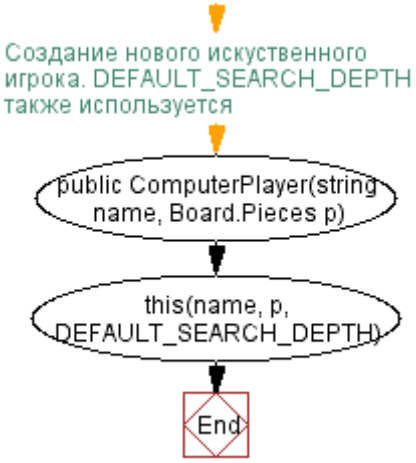

Рисунок 39

Рисунок 40

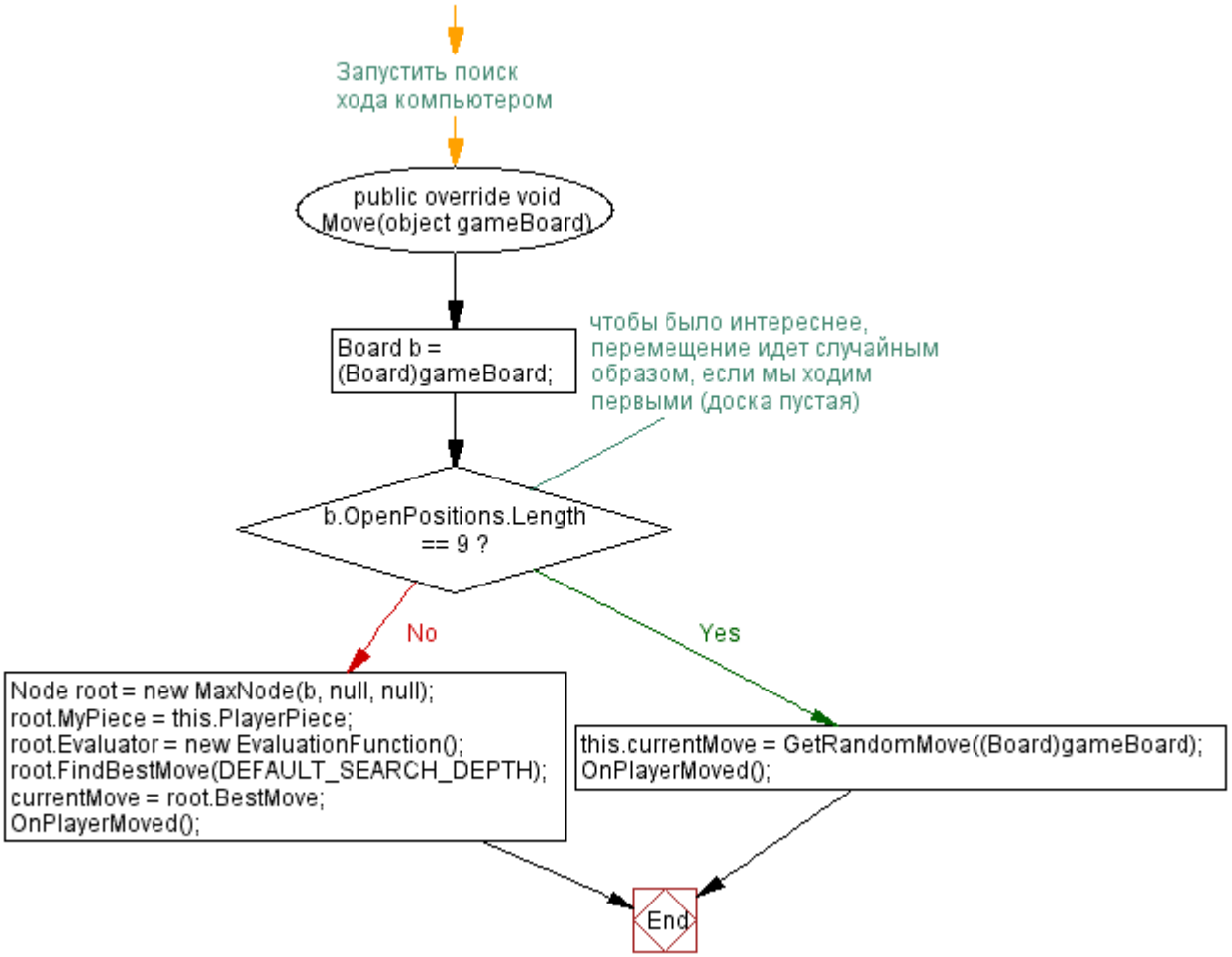

Рисунок 41

Рисунок 42

Как и в HumanPlayer, поиск перемещения начинается, когда экземпляру ComputerPlayer отправляется сообщение Move. Чтобы заставить «шарик катиться», ComputerPlayer создает корень дерева (MaxNode), которому дается ссылка на текущую игровую доску (ну, на самом деле это клон текущей доски – не стоит портить актуальную игровую доску). Функция оценки установлена, и метод FindBestMove корневой вершины (это реализовано в абстрактном классе Node) вызывается с глубиной поиска по умолчанию, равной 3.

Первое, что нужно сделать в FindBestMove, это проверить, что глубина > 0. Так как это первый проход и глубина = 3, он входит в блок if, вызывается метод GenerateChildren, и, для каждой пустой позиции в корневой вершине создается дочерняя вершина. Получается доска, представляющая все возможные начальные шаги, которые может сделать компьютер. Каждый из этих дочерних элементов будет MinNodes, поскольку их родительский элемент является MaxNode.

После вызова метода GenerateChildren вызывается метод EvaluateChildren, который заставляет дочерние вершины компьютеризировать свое значение через функцию оценки. Согласно алгоритму, нужно только оценивать вершины на глубине 3 или любые вершины, для которых есть полностью полная доска или доска, в которой выиграл один игрок.

Теперь используем рекурсии, и для каждого дочернего элемента, у которого у корневой вершины есть метод FindBestMove этой вершины, он вызывается, проходя на глубину 2. Снова создаются дочерние вершины, вершины уровня 2, которые являются вершинами MaxNode, поскольку они являются непосредственными потомками MinNode от уровня 1. Снова рекурсивный вызов FindBestMove вызывается для каждого из дочерних элементов на этот раз с глубиной 1. В последний раз создаются дочерние элементы, которые являются MinNodes, и рекурсивный вызов FindBestMove выполняется с глубины 0. На этот раз глубина > 0 ложна, поэтому FindBestMove просто возвращает ничего не делая, и управление возвращается в блок foreach, который возвращает нас к вершине первого уровня 2 (вершина

2_0). Затем метод SelectBestMove вызывается на этой вершине. Поскольку вершинами уровня 2 являются MaxNodes, метод SelectBestMove, определенный в MaxNode, используется для выбора максимального значения e(m) среди его дочерних элементов. Затем управление возвращается к родительской вершине первого уровня 2 вершины (вершина 1_0), которую мы создали. Ряд рисунков ниже показывает, как дерево глубины 3 генерируется в методе FindBestMove.

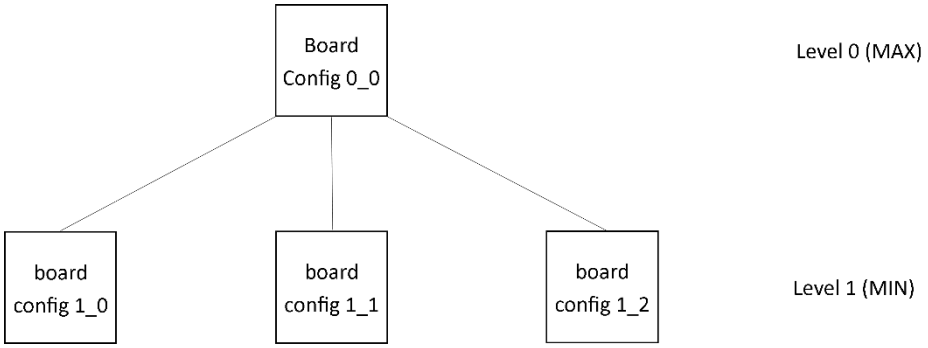

Рисунок 43 - Состояние дерева после первого вызова метода FindBestMove и после вызова GenerateChildren, но до первого рекурсивного вызова FindBestMove

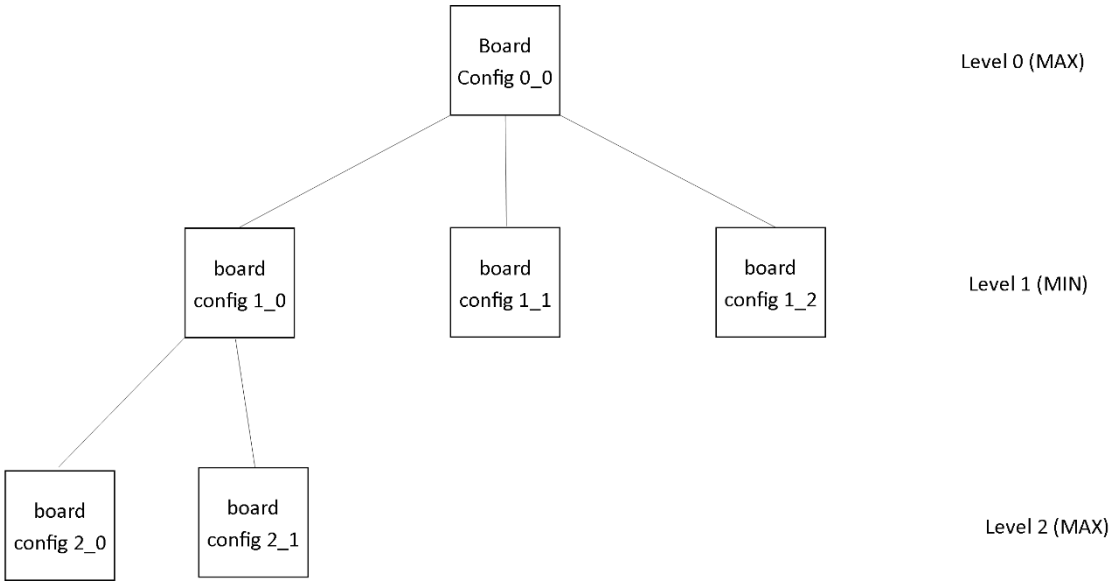

Рисунок 44 - Состояние дерева после первого рекурсивного вызова FindBestMove с глубиной = 2 из вершины 1_0, после вызова метода GenerateChildren, но до повторного рекурсивного вызова FindBestMove

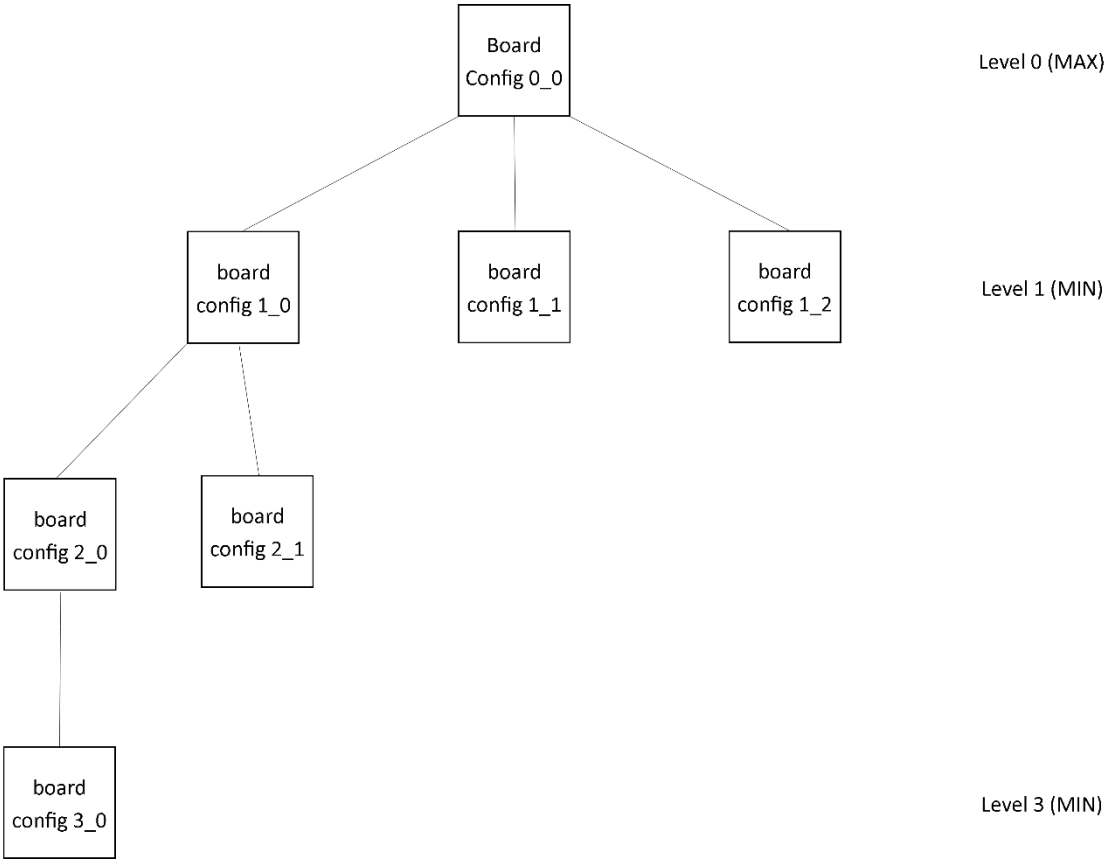

Рисунок 45 - Состояние дерева после второго рекурсивного вызова FindBestMove с глубиной = 1 из вершины 2_0 после вызова метода GenerateChildren, но перед последующим рекурсивным вызовом FindBestMove. Затем выполняется вызов FindBestMove с глубиной = 0 из вершины 3_0, но он ничего не делает. Вызывается SelectBestMove, и вершина 2_0 устанавливает его значение равным максимальному значению его дочерних элементов (значение его единственного дочернего элемента, вершины 3_0).

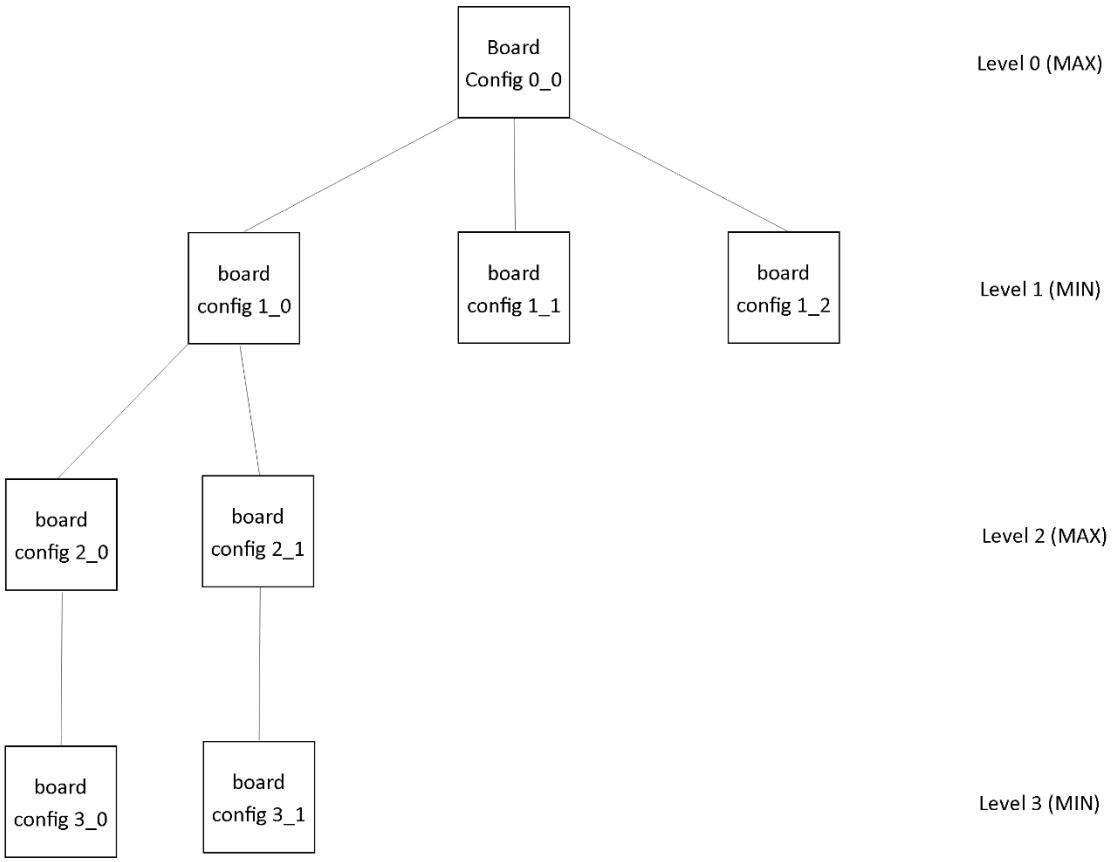

Рисунок 46 - Состояние дерева после рекурсивного вызова FindBestMove с глубиной = 1 из вершины 2_1 после вызова метода GenerateChildren, но перед последующим рекурсивным вызовом FindBestMove. Затем выполняется вызов FindBestMove с глубиной = 0 из вершины 3_1, но он ничего не делает. Вызывается SelectBestMove, и вершина 2_1 устанавливает его значение равным максимальному значению его дочерних элементов (значение его единственного дочернего элемента, вершины 3_1).

После состояния дерева на рисунке 46 FindBestMove вызывается на вершине 1_1 и продолжается таким же образом, как показано в шагах 44 - 46 выше, но на вершине 1_1. Аналогично, FindBestMove вызывается на вершине 1_2, дерево завершается, и корневая вершина выбирает максимальное значение из числа дочерних элементов уровня 1. На рисунке ниже показано окончательное дерево.

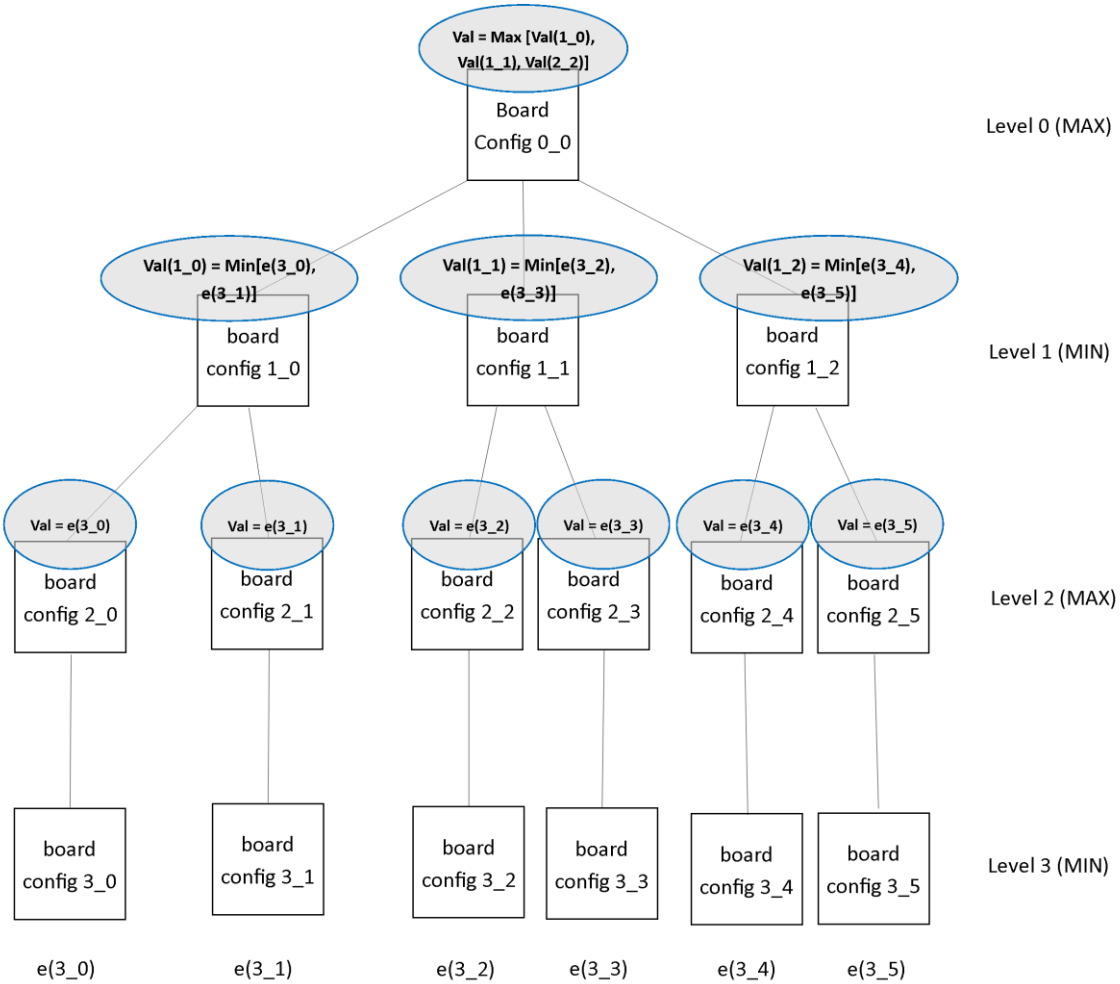

Рисунок 47 - Окончательное дерево глубины 3 со значениями для каждой определенной вершины.

➢ Создание игроков и запуск игры

Создав «умного» игрока – компьютера, который способен делать собственные шаги, нужно настроить и создать игру.

В главном классе Program.cs создаются игроки и запускается игра.

Можно сыграть с человеком, с компьютером или заставить двух компьютерных игроков играть против друг друга.

Блок-схема класса Program.cs представлена на рисунке 48.

Можно создавать разных компьютерных ботов, меняя глубину дерева игр, тем самым меняя их уровень сложности.

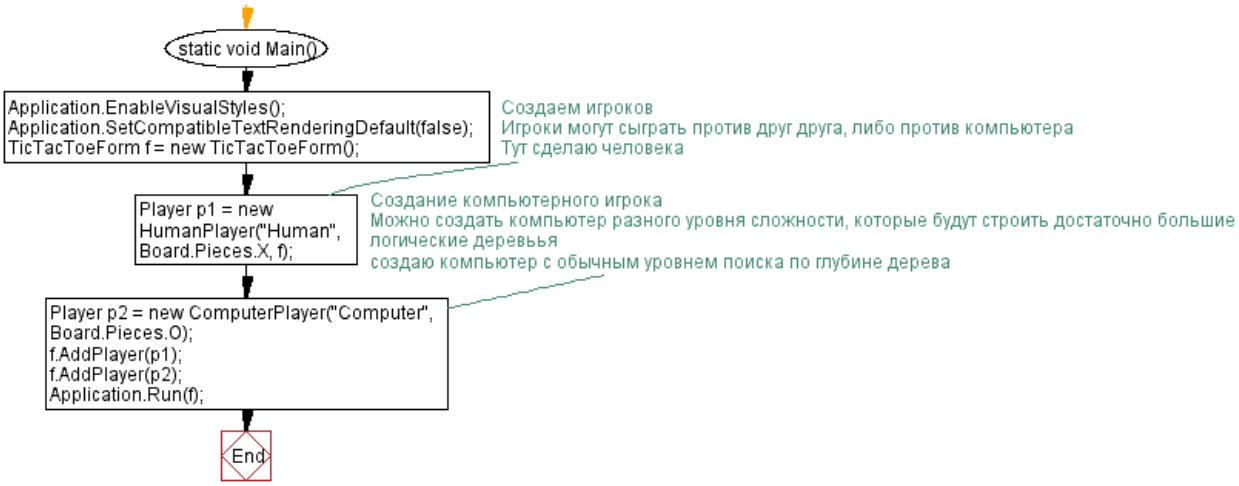

Рисунок 48

Визуализация в игре создавалась с помощью одной Windows From`ы, отрисовки границ при входе игру и отрисовки символов при ходе игрока или компьютера.

```
protected TicTacToeForm ticTacToeForm;

protected bool alreadyMoved = false;

ссылка: 1
public HumanPlayer(string name, Board.Pieces p, TicTacToeForm tttf)
    : base(name, p)
{

    this.ticTacToeForm = tttf;

}
```

Рисунок 49 – Небольшой пример из кода по отрисовке

➢ Пример работы игры

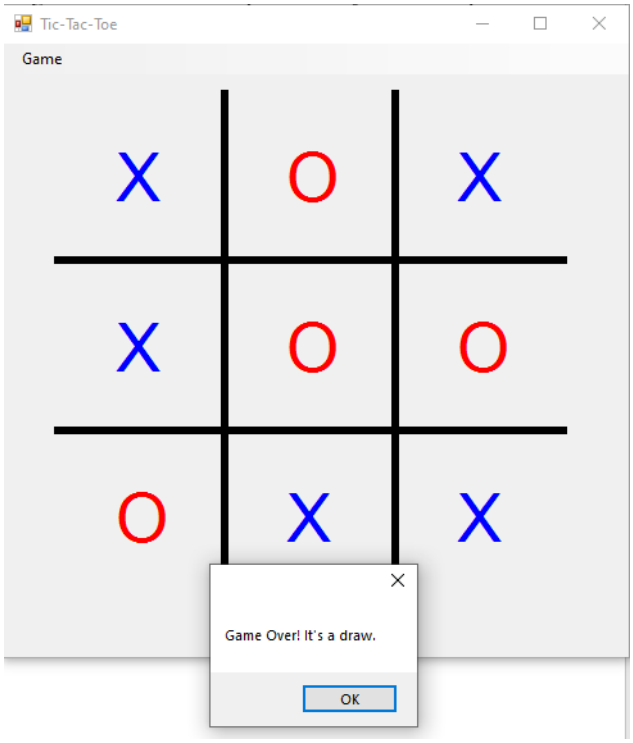

Рисунок 50 – Пример работы игры, если вышла ничья.

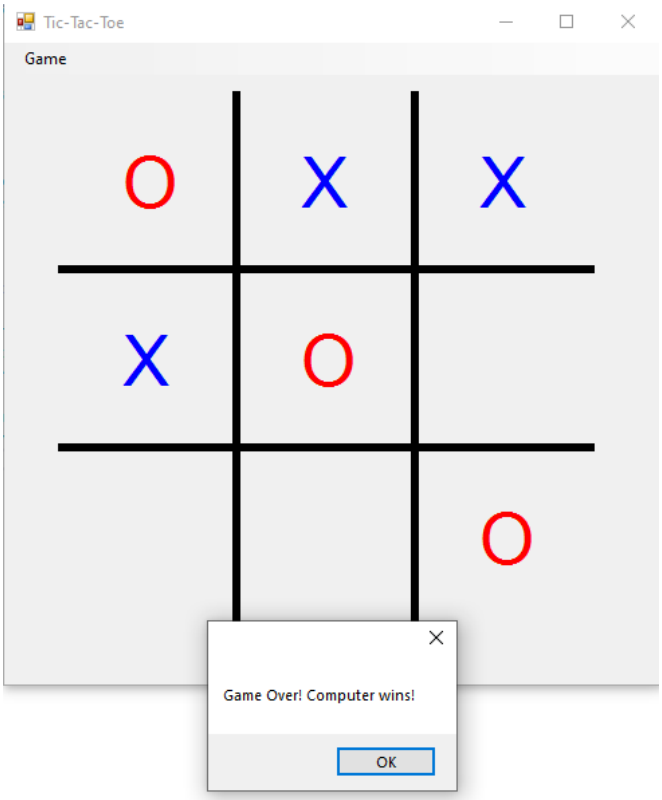

Рисунок 51 – Пример работы игры, победа компьютера

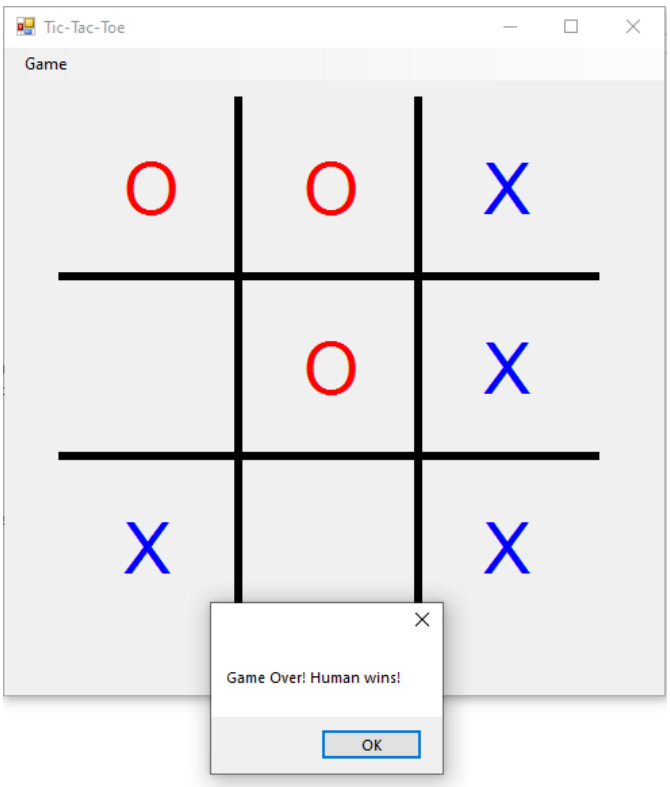

Рисунок 52 – Пример работы игры, победа человека

Выводы

- При попытках создания игры были рассмотрены разные алгоритмы и варианты написания кода.
- Был изучен алгоритм MinMax из теории игр для создания игрока – компьютера.
- Ознакомилась с большим количеством ньюансов в языке C#, изучила такие вещи, как интерфейс ICloneable, ознакомилась с разметкой комментариев (summary, param name и т.д.) , благодаря которой комментарий появляется при наведении мыши на метод или функцию.
- Немного ознакомилась с обработкой исключений
- Попробовала создать несложный интерфейс для отображения программы
- Лучше разобралась в различиях между public, private, protected
- Изучила, как работает наследование в языке C#

Приложение

```
namespace TicTacToe
{

    // Класс, в котором я объединила всё необходимое, для игрового поля в Крестиках-
ноликах
    // Сюда же включена логика для ходов игрока и заполнения доски ходами игрока

    public class TicTacToeGame
    {

        public enum Players { Player1, Player2 };

        protected bool isDraw = false;
        protected bool haveWinner;


        protected Board board;

        protected Stack<TicTacToeMove> moves;
        protected Players currentTurn = Players.Player1; // Player1 ходит первым
        protected Board.Pieces player1Piece = Board.Pieces.X; // player1 всегда X  и
player2 всегда O
        protected Board.Pieces player2Piece = Board.Pieces.O;

        protected Board.Pieces winningPiece = Board.Pieces.Empty;
        protected Players winningPlayer;

        protected bool gameOver = false;



        // Создание новой игры, используя доску для обоих игроков.
        public TicTacToeGame() : this(Board.Pieces.X, Board.Pieces.O)
        {

        }



        // Создание новой игры с использованием значка каждого указанного игрока
        /// <param name="player1Piece">Player one's piece</param>
        /// <param name="player2Piece">Player two's piece</param>
        public TicTacToeGame(Board.Pieces player1Piece, Board.Pieces player2Piece)
        {
            this.player1Piece = player1Piece;
            this.player2Piece = player2Piece;
            board = new Board();
            moves = new Stack<TicTacToeMove>();
        }


        // Получает доску по этой игре
        public Board GameBoard
        {
            get { return board; }
        }


        // количество столбцов на доске
        public int Columns
```

```
        {
            get { return board.Columns; }
        }


        /// <summary>
        /// количество строчек на доске
        /// </summary>
        public int Rows
        {
            get { return board.Rows; }
        }


        // Если появился победитель - он возвращается. Если нет победителя, то
возвращается пустая строка.
        public Board.Pieces WinningPiece
        {
            get { return winningPiece; }
        }


        // Вернет true если игра окончится (в том случае, если есть победитель или ничья)
        public bool IsGameOver()
        {
            return board.IsGameOver();
        }

        // Отмена последнего хода
        public void UndoLastMove()
        {
            TicTacToeMove lastMove = moves.Pop();

            board.UndoMove(lastMove);

            SwapTurns();

        }


        // получение или установка значка Игрока 1
        public Board.Pieces Player1Piece
        {
            get { return player1Piece; }
            set { player1Piece = value; }
        }


        // получение или установка значка Игрока 2
        public Board.Pieces Player2Piece
        {
            get { return player2Piece; }
            set { player2Piece = value; }
        }



        // Возвращает того игрока, чей сейчас ход
        public Players CurrentPlayerTurn
        {
            get { return this.currentTurn; }
        }


        // Делать указанный ход
```

```
public void MakeMove(TicTacToeMove m)
{
    MakeMove(m, GetPlayerWhoHasPiece(m.Piece));
}


// Делать ход для указанного игрока
//"m" - нужный ход
//"p"- игрок делает ход
public void MakeMove(TicTacToeMove m, Players p)
{

    if (currentTurn != p)
    {
        throw new InvalidMoveException("You went out of turn!");
    }

    if (!board.IsValidSquare(m.Position))
        throw new InvalidMoveException("Pick a square on the board!");

    board.MakeMove(m.Position, m.Piece);

    moves.Push(m);

    SwapTurns();

}


// На всякий случай для тестирования
public void TakeSquare(int position, Players p)
{
    if (currentTurn != p)
        throw new InvalidMoveException("You tried to move out of turn!");

    if (!board.IsValidSquare(position))
        throw new InvalidMoveException();

    board.MakeMove(position, GetPlayersPiece(p));

    if (board.HasWinner())
        winningPlayer = currentTurn;


    SwapTurns();



}



// Возвращает игровой знак для указанного игрока
protected Board.Pieces GetPlayersPiece(Players p)
{

    if (p == Players.Player1)
        return player1Piece;
    else
        return player2Piece;
}


// возвращает игрока, у которого есть указанный знак
protected TicTacToeGame.Players GetPlayerWhoHasPiece(Board.Pieces piece)
```

```
        {
            if (piece == player1Piece)
                return Players.Player1;
            else
                return Players.Player2;
        }









        // Меняются ходы
        // Если только что ходил X, то будет ходить O и наоборот
        private void SwapTurns()
        {
            if (currentTurn == Players.Player1)
                currentTurn = Players.Player2;

            else
                currentTurn = Players.Player1;
        }


    }

    // Ход в Крестиках-Ноликах
    public class TicTacToeMove
    {


        public TicTacToeMove(int position, Board.Pieces piece)
        {
            this.Position = position;
            this.Piece = piece;
        }


        public int Position { get; set; }

        public Board.Pieces Piece {get; set;}

    }

    // Исключение при неверном ходе
    public class InvalidMoveException : Exception
    {
        public InvalidMoveException()
            : base()
        {
        }

        public InvalidMoveException(string msg)
            : base(msg)
        {
        }
    }
}
```

```
// делегат для ответа на ходы игрока
    public delegate void PlayerMovedHandler(object sender, PlayerMovedArgs args);


    /// <summary>
    /// Класс для создания Игрока, в который включила некоторые общие функции
    /// Также здесь есть уведомление о том, что сделан шаг в игре
    /// </summary>
    public abstract class Player
    {

        // Слежка за ходом, который сделал игрок
        public event PlayerMovedHandler PlayerMoved;



        protected TicTacToeMove currentMove;
        public Player(string name, Board.Pieces p)
        {
            this.Name = name;
            this.PlayerPiece = p;
        }


        public abstract void Move(object gameBoard);


        public TicTacToeMove CurrentMove
        {
            get { return currentMove; }
        }


        /// <summary>
        /// Выхываем подклассы, чтобы указать, что игрок сделал ход
        /// </summary>
        public virtual void OnPlayerMoved()
        {
            if (PlayerMoved != null)
                PlayerMoved(this, new PlayerMovedArgs(currentMove, this));
        }


        public Board.Pieces PlayerPiece { get; set; }

        public string Name { get; set; }

    }


    /// <summary>
    /// Класс для искуственного игрока
    /// Он определяет ходы по minmax-алгоритму
    /// </summary>
    public class ComputerPlayer : Player
    {
        public const int DEFAULT_SEARCH_DEPTH = 2;


        /// <summary>
        /// Создание нового искуственного игрока. DEFAULT_SEARCH_DEPTH также используется
        /// </summary>
        /// <param name="name">Имя игрока</param>
        public ComputerPlayer(string name, Board.Pieces p) : this(name,
            p, DEFAULT_SEARCH_DEPTH)
```

```
        {
        }


        /// <summary>
        /// Создание нового искуственного игрока
        /// </summary>
        /// <param name="searchDepth">Глубина продумывания ходов по дереву
решений</param>
        public ComputerPlayer(string name, Board.Pieces p, int searchDepth) :base(name,
p)
        {
            this.SearchDepth = searchDepth;
        }


        /// <summary>
        /// получение или установка глубины поиска, которая является кол-вом ходов
        /// компьютер будет глядеть вперед, чтобы определить ход
        /// чем больше значение, тем лучше играет компьютер
        /// </summary>
        public int SearchDepth { get; set; }





        /// <summary>
        /// Запустить поиск хода компьютером
        /// </summary>
        /// <param name="gameBoard">доска на данный момент</param>
        public override void Move(object gameBoard)
        {
            Board b = (Board)gameBoard;

             //чтобы было интереснее, перемещение идет случайным образом, если мы ходим
первыми (доска пустая)
            if (b.OpenPositions.Length == 9)
            {
                this.currentMove = GetRandomMove((Board)gameBoard);
                OnPlayerMoved();
                return;
            }

            Node root = new MaxNode(b, null, null);
            root.MyPiece = this.PlayerPiece;
            root.Evaluator = new EvaluationFunction();
            root.FindBestMove(DEFAULT_SEARCH_DEPTH);


            currentMove = root.BestMove;

            OnPlayerMoved();
        }


        // Рандомный ход, чтобы сделать игру интересной в начале игры, чтобы попробовать
ослабить компьютер
        protected TicTacToeMove GetRandomMove(Board b)
        {
            int openPositions = b.OpenPositions.Length;
            Random rGen = new Random();

            int squareToMoveTo = rGen.Next(openPositions);
```

```
            TicTacToeMove move = new TicTacToeMove(squareToMoveTo, this.PlayerPiece);
            return move;
        }


    }

    /// <summary>
    /// Класс для человека
    /// </summary>
    public class HumanPlayer : Player
    {


        protected TicTacToeForm ticTacToeForm;

        protected bool alreadyMoved = false;

        public HumanPlayer(string name, Board.Pieces p, TicTacToeForm tttf)
            : base(name, p)
        {

            this.ticTacToeForm = tttf;

        }


        /// <summary>
        /// Делаем ход. Ожидание, пока человек щелкнет дважды по квадратику.
        /// затем запускаем событие PlayerMoved
        /// </summary>
        /// <param name="gameBoard"></param>
        public override void Move(object gameBoard)
        {

            ticTacToeForm.SquareDoubleClicked += new
SquareDoubleClickHandler(SquareDoubleClicked);

            // ждем пока пользователь кликнет
            while (!alreadyMoved)
                ;


            // сброс флага
            alreadyMoved = false;
            OnPlayerMoved();


        }


        // когда игрок дважды щелкает по квадрату в форме, метод получит сообщение о
событии
        // устанавливается текущее перемещение, флаг alreadyMoved становится true, значит
цикл while в методе Move разорвется.
        void SquareDoubleClicked(object sender, TicTacToeBoardClickedEventArgs args)
        {
            // отмена регистрации события двойного клика
            ticTacToeForm.SquareDoubleClicked -= SquareDoubleClicked;

            currentMove = new TicTacToeMove(args.BoardPosition, this.PlayerPiece);
            alreadyMoved = true;

        }
```

```
}


/// <summary>
/// Инкапсуляция движущегося игрока
/// Передается вместе с событием PlayerMoved
/// </summary>
public class PlayerMovedArgs : System.EventArgs
{
    protected TicTacToeMove move;
    protected Player player;


    /// <summary>
    /// Создание нового объекта PlayerMovedArgs с указанными Move и Player
    /// </summary>
    public PlayerMovedArgs(TicTacToeMove m, Player player)
        : base()
    {
        this.player = player;
        move = m;

    }

    public TicTacToeMove Move
    {
        get { return move; }
    }

    public Player Player
    {
        get { return player; }
    }
}
```

```
// Класс для создания доски для игры
//  Cloneable поставлена для того, чтобы можно было копировать доску для поиска нужного хода
public class Board : ICloneable
{
    public enum Pieces { X, O, Empty };



    public const int COLUMNS = 3;
    public const int ROWS = 3;
    protected const int WINNING_LENGTH = 3;

    public bool haveWinner;  // есть победитель?



    protected Pieces winningPiece;




    protected int[,] board; // двумерный массив для доски



    // Создание новой доски исходя из предыдущего состояния по игре
    // Состояния игры:
    // Первый индекс - это строка доски, второй индекс - это столбец.
    // 0 - это пустой квадратик на доске
    // 1 - это на доске стоит значение X
    // 2 - это на доске стоит O
    public Board(int[,] gameState) : this()
    {
        for (int i = 0; i <= gameState.GetUpperBound(0); i++)
```

```
        for (int j = 0; j <= gameState.GetUpperBound(1); j++)
        {
            this.board[i, j] = gameState[i, j];
        }
}



// Создание пустой доски
public Board()
{
    board = new int[ROWS, COLUMNS];
}



// Возвращает значок победителя (крестик или нолик)
public Pieces WinningPiece
{
    get { return winningPiece; }
    set { winningPiece = value; }
}



// возвращает количество строчек на доске
public int Rows
{
    get { return ROWS; }

}
```

```
// возвращает количество столбцов на доске
public int Columns
{
    get { return COLUMNS; }
}





// Возвращает "true" если такая позиция есть на доске и она не занята крестиком или ноликом.
//0 - это левая верхняя часть квадрата и увеличиваетсчя построчно так, чтобы вторая строка
// начиналась с 3, а 8 была нижней правой частью квадрата
// 0 1 2
// 3 4 5
// 6 7 8

public bool IsValidSquare(int position)
{
    Point p = GetPoint(position);

    if (p.X >= 0 && p.X < ROWS && p.Y >= 0 && p.Y < COLUMNS && IsPositionOpen(p.X,
p.Y))
        return true;

    return false;
}
```

```
public bool IsOnBoard(int position)
{
    return IsOccupied(position) || IsValidSquare(position);
}

public void UndoMove(TicTacToeMove m)
{



    if (!IsOnBoard(m.Position))
        throw new InvalidMoveException("Can't undo a move on an invalid square!");

    // сброс позиции
    Point p = GetPoint(m.Position);
    board[p.X, p.Y] = 0;
}









// Сделать ход на доске
public void MakeMove(int position, Pieces piece)
{



    if (!IsValidSquare(position))
        throw new InvalidMoveException();
```

```
    int pieceNumber = GetPieceNumber(piece);

    Point point = GetPoint(position);

    board[point.X, point.Y] = pieceNumber;

}


public int[] OpenPositions
{
    get
    {
        List<int> positions = new List<int>();
        for (int i = 0; i < board.Length; i++)
        {
            if (!IsOccupied(i))
            {
                positions.Add(i);
            }
        }

        return positions.ToArray();
    }
}


// Повторяет значок в соответствующем ряду и столбце на доске
```

```
public Pieces GetPieceAtPoint(int row, int column)
{
    return GetPieceAtPosition(GetPositionFromPoint(new Point(row, column)));

}

// Возвращает значок на заданной позиции на доске

public Pieces GetPieceAtPosition(int position)
{
    if (!IsOccupied(position))
        return Pieces.Empty;

    if (GetBoardPiece(position) == 1)
        return Pieces.X;
    else
        return Pieces.O;
}



// Проверяет доску на наличие победителя
public bool HasWinner()
{
    for (int i = 0; i < board.Length; i++)
        if (IsWinnerAt(i))
        {
```

```
            haveWinner = true;
            SetWinnerAtPosition(i);
            return true;
        }

    return false;
}



public static bool IsValidPosition (int position)
{
    return position >= 0 && position < Board.COLUMNS * Board.ROWS;
}

private void InitBoard()
{
    for (int i = 0; i < board.GetLength(0); i++)
        for (int j = 0; j < board.GetLength(1); j++)
            board[i, j] = 0;

}

// Отображение номера позиции на доске в точку, содержащую строку в X и столбец в Y
protected Point GetPoint(int position)
{
    Point p = new Point();

    p.X = position / COLUMNS;
    p.Y = position % ROWS;
```

```
    return p;
}


// внутреннее предствление значка
protected int GetPieceNumber(Pieces p)
{
    if (p == Pieces.O)
        return 2;
    else
        return 1;
}


// возвращает номер по строке и столбцу
// p.X строка
// p.Y столбец
protected int GetPositionFromPoint(Point p)
{
    return p.X * Columns + p.Y;
}


private void SetWinnerAtPosition(int position)
{

    WinningPiece = GetPieceAtPosition(position);
}
```

```
private int GetBoardPiece(int position)
{
    Point p = GetPoint(position);

    return board[p.X, p.Y];
}


// доступна ли позиция
private bool IsPositionOpen(int row, int col)
{

    return board[row, col] == 0;

}


private bool IsOccupied(int position)
{
    Point p = GetPoint(position);
    return IsOccupied(p.X, p.Y);
}


private bool IsOccupied(int row, int col)
{
    return !IsPositionOpen(row, col);
}
```

```
// вспомогательный метод для определения победителя
private bool IsWinnerAt(int position)
{

    // проверить справа каждую позицию для победителя
    if (IsWinnerToTheRight(position) || IsWinnerFromTopToBottom(position)
        || IsWinnerDiagonallyToRightUp(position) || IsWinnerDiagonallyToRightDown(position))
        return true;
    else
        return false;
}



// проверяем победителя по диагонали, начиная с нижнего левого угла доски до правого верхнего угла
private bool IsWinnerDiagonallyToRightUp(int position)
{

    if (!IsOccupied(position))
        return false;

    Pieces piece = GetPieceAtPosition(position);

    Point last = GetPoint(position);
    for (int i = 1; i < WINNING_LENGTH; i++)
    {
        last.X -= 1;
```

```
        last.Y += 1;

        if (!IsPointInBounds(last))
            return false;

        if (piece != GetPieceAtPosition(GetPositionFromPoint(last)))
            return false;

    }
    return true;

}

//  вернет true если есть победитель или ничья
public bool IsGameOver()
{
    return HaveWinner() || IsDraw();
}


// вернет true если есть победитель
public bool HaveWinner()
{
    return HasWinner();
}


// Вернет значок, представляющий противника
public static Board.Pieces GetOponentPiece(Board.Pieces yourPiece)
{
```

```
    if (yourPiece == Board.Pieces.X)
        return Board.Pieces.O;
    else if (yourPiece == Board.Pieces.O)
        return Board.Pieces.X;
    else
        throw new Exception("Invalid Piece!");
}

// Проверяем доску, нет ли ничьей.
// Ничья - это если вся доска заполнена, а победителя нет.
public bool IsDraw()
{

    if (HasWinner())
        return false;

    for (int i = 0; i < board.Length; i++)
    {
        if (!IsOccupied(i))
            return false;
    }

    return true;
}


// проверка, есть ли строка и столбец на доске
// p.X строка, p.Y столбец
private bool IsPointInBounds(Point p)
{
```

```
    if (p.X < 0 || p.X >= Rows || p.Y < 0 || p.Y >= Columns)
        return false;

    return true;
}



// проверяем победителя по диагонали от указанной позиции справа
private bool IsWinnerDiagonallyToRightDown(int position)
{
    Point p = GetPoint(position);

    if (!IsOccupied(position))
        return false;



    Pieces piece = GetPieceAtPosition(position);



    // продолжаем двигаться по диагонали, пока не достигнем выигрышна или не увидим тот
же значок
    Point last = GetPoint(position);
    for (int i = 1; i < WINNING_LENGTH; i++)
    {
        last.X += 1;
        last.Y += 1;
        if (!IsPointInBounds(last))
            return false;

        if (piece != GetPieceAtPosition(GetPositionFromPoint(last)))
```

```
            return false;

        }

        return true;



    }

    // проверяем победителя сверху вниз, начиная с указанной позиции
    private bool IsWinnerFromTopToBottom(int position)
    {
        Point p = GetPoint(position);

        // проверяем, есть ли квадратик
        if (!IsOccupied(position))
            return false;

        // есть ли место внизу
        if (p.X + WINNING_LENGTH - 1 >= ROWS)
            return false;



        Pieces piece = GetPieceAtPosition(position);

        // если попадем сюда, то знаем, что по крайней мере есть потенциал для победиты сверху
вниз
        for (int i = 1; i < WINNING_LENGTH; i++)
        {
            if (piece != GetPieceAtPosition(position + 3 * i))
```

```
            return false;

        }

        return true;
    }



    // проверяем победителя с указанной позиции справа
    private bool IsWinnerToTheRight(int position)
    {
        Point p = GetPoint(position);

        // проверка, занят ли квадратик, если не занят, то победителя здесь нет
        if (!IsOccupied(position))
            return false;

        // првоерить, есть ли место справа
        if (p.Y + WINNING_LENGTH - 1 >= COLUMNS)
            return false;

        Pieces piece = GetPieceAtPosition(position);

        for (int i = 1; i < WINNING_LENGTH; i++)
        {
            if (GetPieceAtPosition(position + i) != piece)
                return false;
        }
        return true;
    }
```

```
    #region ICloneable Members

    public object Clone()
    {
        Board b = new Board(this.board);
        return b;

    }

    #endregion
}
```

```
public abstract class Node
{

    protected List<Node> children;

    protected double value;



    protected Node bestMoveNode;

    protected Board board;

    protected static EvaluationFunction evaluator;


    protected Board.Pieces myPiece;
    protected Board.Pieces opponentPiece;


    TicTacToeMove move;
    Node parent = null;



    public Node(Board b, Node parent, TicTacToeMove move)
    {
        this.board = b;
        this.parent = parent;
        this.move = move;
        if (parent != null)
            myPiece = Board.GetOponentPiece(parent.MyPiece);
        children = new List<Node>();
    }



    public Board.Pieces MyPiece
    {
        get { return myPiece; }
        set
        {
            myPiece = value;
            opponentPiece = Board.GetOponentPiece(value);
        }
    }



    public EvaluationFunction Evaluator
    {
        set { evaluator = value; }
    }




    public double Value
    {
        get { return value; }
        set { this.value = value;}
    }
```

```
protected abstract void Evaluate();


public void SelectBestMove()
{

    if (children.Count == 0)
    {
        bestMoveNode = null;
        return;
    }



    List<Node> sortedChildren = SortChildren(this.children);

    this.bestMoveNode = sortedChildren[0];
    Value = bestMoveNode.Value;
}



public virtual void FindBestMove(int depth)
{
    if (depth > 0)
    {

        GenerateChildren();


        EvaluateChildren();


        bool haveWinningChild = children.Exists(c => c.IsGameEndingNode());

        if (haveWinningChild)
        {

            SelectBestMove();
            return;
        }
        else
        {
            foreach (Node child in children)
            {
                child.FindBestMove(depth - 1);
            }
            SelectBestMove();
        }
    }

}



protected abstract void GenerateChildren();


public virtual bool IsGameEndingNode()
{
    return Value == double.MaxValue || Value == double.MinValue;
}
```

```
        public TicTacToeMove BestMove
        {
            get { return this.bestMoveNode.move; }
        }


        protected void EvaluateChildren()
        {
            foreach (Node child in this.children)
            {
                child.Evaluate();
            }
        }

        protected abstract List<Node> SortChildren(List<Node> unsortedChildren);

        protected abstract bool IsWinningNode();



    }

    public class MaxNode : Node
    {


        public MaxNode(Board b, Node parent, TicTacToeMove m)
            : base(b, parent, m)
        {
        }





        protected override void GenerateChildren()
        {

            int[] openPositions = board.OpenPositions;

            foreach (int i in openPositions)
            {
                Board b = (Board) board.Clone();
                TicTacToeMove m = new TicTacToeMove(i, myPiece);

                b.MakeMove(i, myPiece);
                children.Add(new MinNode(b, this, m));

            }
        }



        protected override void Evaluate()
        {
            this.Value = evaluator.Evaluate(this.board, myPiece);
```

```
        }


        protected override bool IsWinningNode()
        {

            return this.Value == double.MaxValue;
        }


        protected override List<Node> SortChildren(List<Node> unsortedChildren)
        {
            List<Node> sortedChildren = unsortedChildren.OrderByDescending(n=>
n.Value).ToList();

            return sortedChildren;
        }

    }




    public class MinNode : Node
    {


        public MinNode(Board b, Node parent, TicTacToeMove m)
            :base(b, parent, m)
        {

        }



        protected override void GenerateChildren()
        {
            int[] openPositions = board.OpenPositions;



            foreach (int i in openPositions)
            {
                Board b = (Board)board.Clone();
                TicTacToeMove m = new TicTacToeMove(i, myPiece);

                b.MakeMove(i, myPiece);
                children.Add(new MaxNode(b, this, m));

            }

        }



        protected override bool IsWinningNode()
        {
            return this.value == double.MinValue;

        }


        protected override List<Node> SortChildren(List<Node> unsortedChildren)
```

```
        {
            List<Node> sortedChildren = unsortedChildren.OrderBy(n => n.Value).ToList();
            return sortedChildren;
        }


        protected override void Evaluate()
        {
            Value = evaluator.Evaluate(board, Board.GetOponentPiece(myPiece));
        }
    }




    public class EvaluationFunction
    {

        int functionCalls = 0;

        public EvaluationFunction()
        {
        }


        public int FunctionCalls
        {
            get { return this.functionCalls; }
        }

        public double Evaluate(Board b, Board.Pieces maxPiece)
        {
            functionCalls++;

            if (b.HasWinner())
            {
                if (b.WinningPiece == maxPiece)
                    return double.MaxValue;
                else
                    return double.MinValue;
            }

            double maxValue = EvaluatePiece(b, maxPiece);
            double minValue = EvaluatePiece(b, Board.GetOponentPiece(maxPiece));

            return maxValue - minValue;



        }


        private double EvaluatePiece(Board b, Board.Pieces p)
        {

            return EvaluateRows(b, p) + EvaluateColumns(b, p) + EvaluateDiagonals(b, p);
        }


        private double EvaluateRows(Board b, Board.Pieces p)
        {

            int cols = b.Columns;
            int rows = b.Rows;
```

```
    double score = 0.0;
    int count;

    for (int i = 0; i < b.Rows; i++)
    {
        count = 0;
        bool rowClean = true;
        for (int j = 0; j < b.Columns; j++)
        {
            Board.Pieces boardPiece = b.GetPieceAtPoint(i, j);

            if (boardPiece == p)
                count++;
            else if (boardPiece == Board.GetOponentPiece(p))
            {
                rowClean = false;
                break;
            }
        }


        if (rowClean && count != 0)
            score += count;
    }

    return score;
}



private double EvaluateColumns(Board b, Board.Pieces p)
{
    int cols = b.Columns;
    int rows = b.Rows;

    double score = 0.0;
    int count;

    for (int j = 0; j < b.Columns; j++)
    {
        count = 0;
        bool rowClean = true;
        for (int i = 0; i < b.Columns; i++)
        {
            Board.Pieces boardPiece = b.GetPieceAtPoint(i, j);

            if (boardPiece == p)
                count++;
            else if (boardPiece == Board.GetOponentPiece(p))
            {
                rowClean = false;
                break;
            }
        }


        if (rowClean && count != 0)
            score += count;
    }

    return score;
}
```

```
    private double EvaluateDiagonals(Board b, Board.Pieces p)
    {

        int count = 0;
        bool diagonalClean = true;

        double score = 0.0;

        for (int i = 0; i < b.Columns; i++)
        {
            Board.Pieces boardPiece = b.GetPieceAtPoint(i, i);

            if (boardPiece == p)
                count++;

            if (boardPiece == Board.GetOponentPiece(p))
            {
                diagonalClean = false;
                break;
            }
        }

        if (diagonalClean && count > 0)
            score += count;




        int row = 0;
        int col = 2;
        count = 0;
        diagonalClean = true;

        while (row < b.Rows && col >= 0)
        {
            Board.Pieces boardPiece = b.GetPieceAtPoint(row, col);

            if (boardPiece == p)
                count++;

            if (boardPiece == Board.GetOponentPiece(p))
            {
                diagonalClean = false;
                break;
            }

            row++;
            col--;
        }

        if (count > 0 && diagonalClean)
            score += count;

        return score;
    }
}
```

```
static class Program
    {

        // Главный вход в программу
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);

            TicTacToeForm f = new TicTacToeForm();

            // Создаем игроков
            // Игроки могут сыграть против друг друга, либо против компьютера


            // Тут сделаю человека
            Player p1 = new HumanPlayer("Human", Board.Pieces.X, f);


            // Создание компьютерного игрока
            // Можно создать компьютер разного уровня сложности, которые будут строить
достаточно большие логические деревья
            // на всякий случай оставлю тут другие варианты компьютеров, нужно
закомментить ненужные и раскомментить нужные


            // создаю компьютер с обычным уровнем поиска по глубине дерева
            //Player p2 = new ComputerPlayer("Computer", Board.Pieces.O);

            // Самый простенький бот, который смотрит только на один ход вперед, это
указано после запятой
            // он не учитывает последующий ход после сделанного человеком
            Player p2 = new ComputerPlayer("Computer", Board.Pieces.O, 1);

            // Создаёт достаточно сильного противника, который думает на пять ходов
вперед
            // Player p2 = new ComputerPlayer("Advanced HAL", Board.Pieces.X, 5);

            f.AddPlayer(p1);
            f.AddPlayer(p2);

            Application.Run(f);



        }
    }
```