Ain Shams University,
Faculty of Engineering,
Computer and Systems Engineering Department.

Data Structures Project: Board Games

SUBMITTED BY

Hossam Mohamed El-Nagy Mohamed	33737
Asim Mahmoud Abd El-Azem	33767
Abd-Elrahman Abd-Elrazek Hussein	33768
Abdullah Abd-Elrazek Hussein	33773
Essam El-Din Eid Abd-Elsalam	33775

SUBMITTED TO

Dr. Eslam Maddah

DATE SUBMITTED

9-May-2016

Contents

1.0 Project Description

Implementation of "connect four" and "Tic tac toe" board games where there's a board which has empty places/squares and players take turns in filling these empty places according to some rules. Program decides which player won the game according to the game rules.

- **Connect Four:**
  The Connect 4 Board Game rules are very easy to understand. To win Connect Four, all you have to do is connect four of your pieces in a row. This can be done horizontally, vertically or diagonally. Each player will drop in one piece at a time. The game is over either when one of the players reaches four in a row, or when all slots are filled, ending in a stalemate.
- **Tic Tac Toe:**
  Also known as Xs and Os is a game for two players, who take turns marking the spaces in a grid. The player who succeeds in placing three of their marks in a horizontal, vertical, or diagonal row wins the game.

Tic tac toe is similar to connect 4 but the player can insert the X or O in any place in the grid while in connect 4 the player can only choose a column to put his piece in.

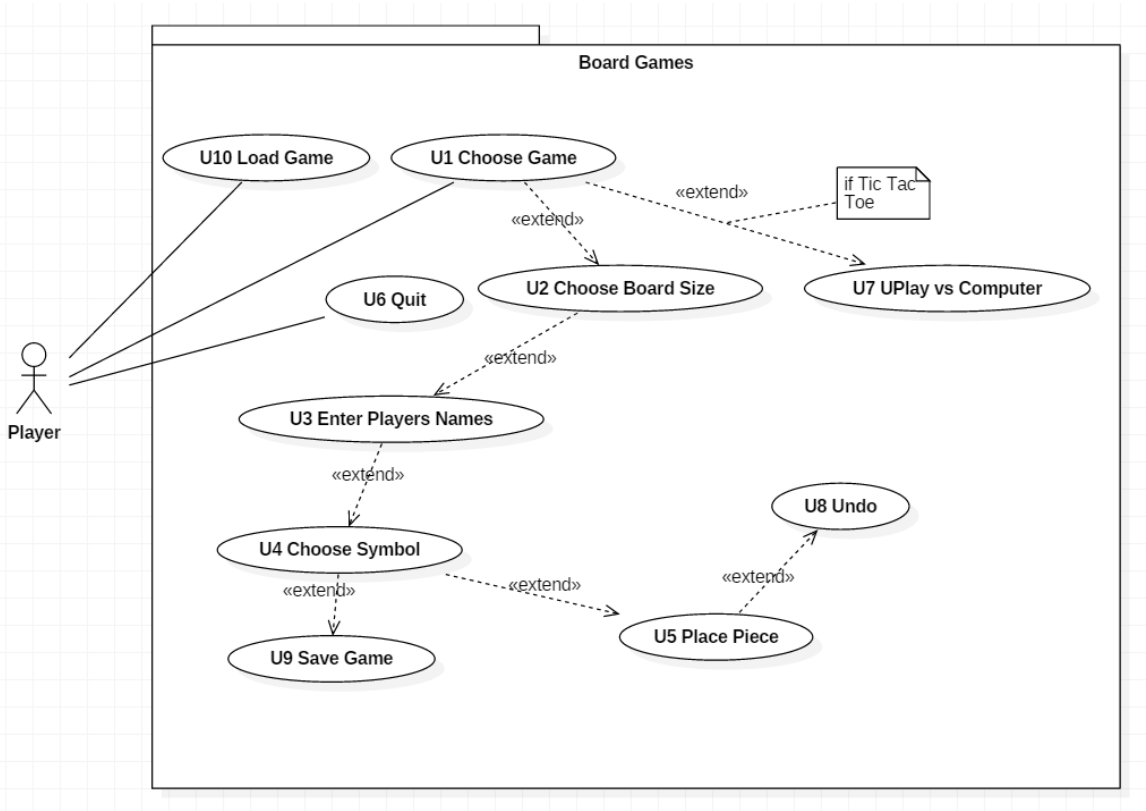
2.0 Features and limitations

1. The board width and height can range from 3 to 15 in case of Tic Tac Toe and 4 to 15 in case of Connect 4.
2. User can play Tic Tac Toe against the computer in a 3x3 board.
3. Players can specify the number of markers needed to win in case of Tic Tac Toe game.
4. Game supports more than 2 players. Maximum number of players depends on the board size and number of consecutive markers needed to win.
   $$Max\ Players = \frac{BoardWidth * BoardHeight}{noMarkersNeededToWin}$$
5. Player can undo previous steps.

6. User can save and load games to/from disk.
7. The game keeps track of high scores.

3.0 Use Case

3.1 Use Case Diagram:



3.2 Use Case Details

U1: Choose Game:

- User chooses either to play “Tic Tac Toe” or “Connect Four”.

U2: Choose Board Size:

- Extends: U1 Choose Game
- User enters the desired board size of the game.

U3: Enter Players Names:

- Each player enters his\her name.

U4: Choose Symbol:

- Each player enters his\her unique symbol which will be displayed on the board while playing as his\her piece.

U5: Place a Piece:

- User chooses a position on the board to place his\her piece.
- In case of "Tic Tac Toe" game: user enters column and raw number which he\she wishes to place his\her piece.
- In case of "Connect Four" game: user enters column number which he\she wishes to drop his\her piece in.

U6: Quit:

- User can choose to play a new game.

U7: Play vs Computer:

- Extends: U1 Choose Game if player chooses "Tic Tac Toe"
- User play a Tic Tac Toe game against the computer in 3x3 board.

U8: Undo:

- Extends: U5 Place a Piece
- Player undo the last move he made.

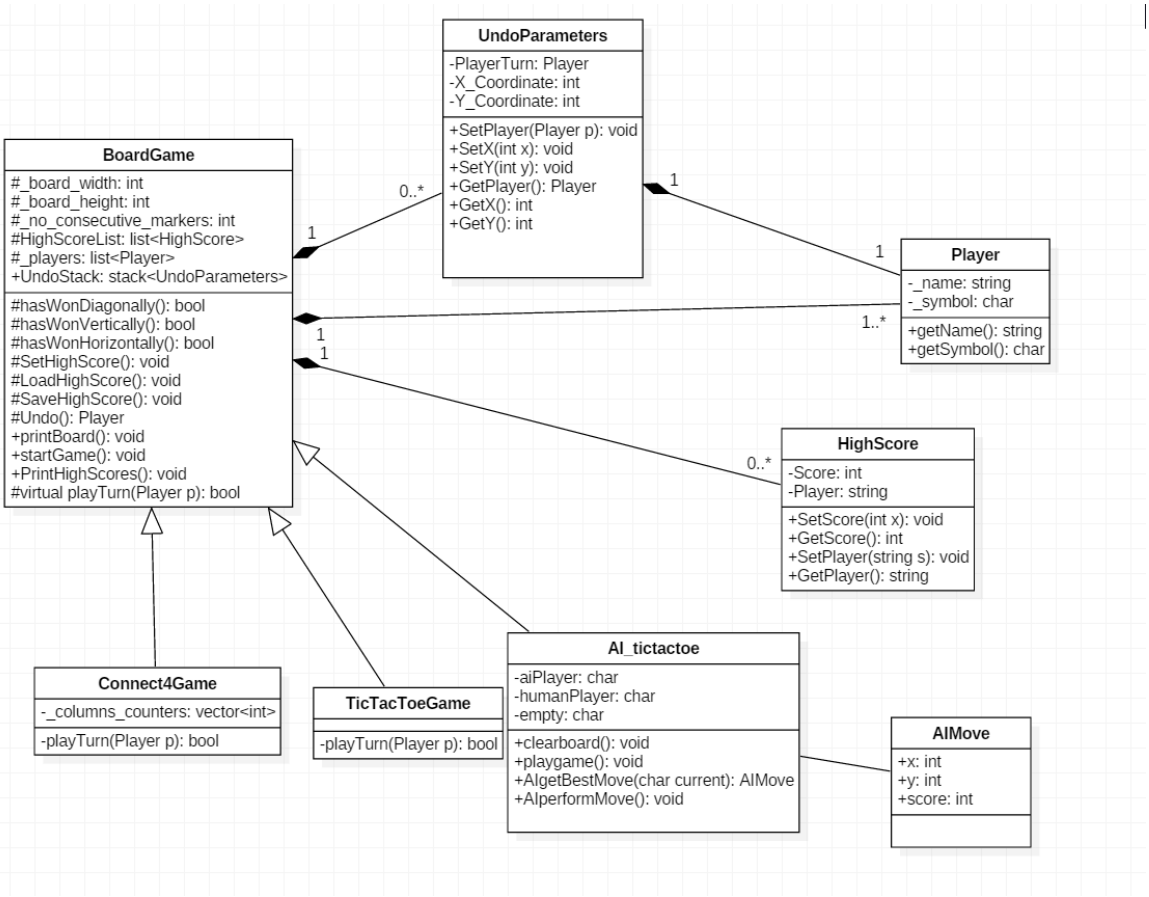
U9: Save Game:

- Extends: U5 Place a Piece
- Player saves the status of the game in a file so he can load it later.

U10: Load Game:

- Player loads a game he saved earlier.

4.0 Class Diagram



5.0 Data Structures used

5.1 2D-Vectors

We used 2D – Vectors to represent the board (vector of columns).

One advantage of Vectors is that it's dynamic allowing it to be allocated at run time when the user enters the board size. Also the advantage of using Vector is that it provides random access in constant time that is helpful when accessing a piece in any place in the board.

We didn't use 2D – dynamic array because

- Vectors are thin wrappers around dynamic arrays, so there's no significant difference in performance when using vectors.
- When using dynamic array, there is the problem that you have to keep track of the size, and you need to delete them manually, and do all sort of housekeeping.

5.2 Stack

We used stack to implement the undo feature because it's Last-in–First-out container. That is done by pushing the player move after each turn and popping the stack when the player wishes to undo his previous move.

5.3 List

We used list to store the players. We didn't use vectors since we didn't need the random access, we only needed to iterate over the players one by one giving each player a turn.

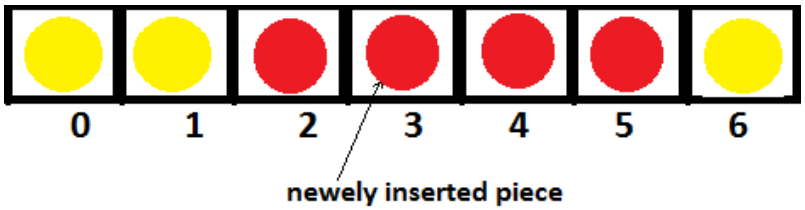
6.0 Algorithms

6.1 detect if a player has won

When the user input a piece in a specific place we check this piece and its surrounding pieces to see if the player has won. We check horizontally, vertically and diagonally.

To check horizontally:

For example



Consider we have this vertical raw. Let's call the index of the newly inserted piece (K)

1. First we initialize $count = 0$ which counts how many cells after the newly inserted piece have the same type of piece.
2. Then we start iterating to the right from the newly inserted piece
   From $i = K + 1$ to $K + 3$.
   From the example above this loop will iterate from $i = 4$ to $i = 6$
   If the piece at the current location doesn't match the newly inserted piece or the location is empty or i exceeds the width of the board:
   Break from the loop and jump to step 4.
   Else
   $count = count + 1$
   Continue iterations.
3. If the loop continued without breaking, it's a wining condition and the player who inserted the piece wins.
   From the example above this loop will break at $i = 6$ and $count = 2$
4. Next we iterate to the left from the newly inserted piece from $j = k - 1$ to $k - (3 - count)$
   From the example above this loop will iterate from $j = 2$ to $j = 2$
   If the piece at the current location doesn't match the newly inserted piece or the location is empty or i is less than 0:
   Wining condition failed.

Else
Continue iterations.

5. If the loop continued without breaking, it's a wining condition and the player who inserted the piece wins.
   This will happen in our example and the player who inserted the piece wins.

We use similar algorithms for checking vertically and diagonally.

We use the same algorithms for Tic Tac Toe but the wining conditions will differ in the number of required consecutive pieces.

- Implementation snapshots:

```
bool BoardGame::hasWonDiagonalyLeft(int x, int y)
{
    assert(x >= 0 && x < _board_width);
    assert(y >= 0 && y < _board_height);
    bool has_won = true;
    int count = 0;
    int j = y + 1;
    for (int i = x - 1; i > x - _no_consecutive_markers_to_win; i--, j++) {
        if (i < 0 || j >= _board_height || board[x][y] != board[i][j] || board[i][j] == ' ') {
            has_won = false;
            break;
        }else
            count++;
    }

    if (has_won)
        return true;

    j = y - 1;
    for (int i = x + 1; i < x + _no_consecutive_markers_to_win - count; i++, j--) {
        if (i >= _board_width || j < 0 || board[x][y] != board[i][j] || board[i][j] == ' ')
            return false;

    }

    return true;
}
```

```
bool BoardGame::hasWonDiagonalyRight(int x, int y)
{
    assert(x >= 0 && x < _board_width);
    assert(y >= 0 && y < _board_height);
    bool has_won = true;
    int count = 0;
    int j = y + 1;
    for (int i = x + 1; i < x + _no_consecutive_markers_to_win; i++, j++) {
        if (i >= _board_width || j >= _board_height || board[x][y] != board[i][j] || board[i][j] == ' ') {
            has_won = false;
            break;
        }
        else {
            count++;
        }
    }
    if (has_won)
        return true;
    j = y - 1;
    for (int i = x - 1; i > x - (_no_consecutive_markers_to_win - count); i--, j--) {
        if (i < 0 || j <0 || board[x][y] != board[i][j] || board[i][j] == ' ' ) {
            return false;
        }
    }

    return true;
}
```

```
bool BoardGame::hasWonVertically(int x, int y)
{
    //function input checks
    assert(x >= 0 && x < _board_width);
    assert(y >= 0 && y < _board_height);
    int CurrentHeight = y;
    int counter = 0;
    for (counter = 0; counter < _no_consecutive_markers_to_win-1; counter++)
    {
        if (CurrentHeight == 0)
        {
            return false;
        }
        if (board[x][CurrentHeight] != board[x][CurrentHeight - 1] || CurrentHeight >= _board_height)
        {
            return false;
        }
        CurrentHeight--;
    }


    return true;
}
```

```
bool BoardGame::hasWonHorizontally(int x, int y)
{

    //function input checks
    assert(x >= 0 && x < _board_width);
    assert(y >= 0 && y < _board_height);
    int CurrentWidth = x;
    int counter = 0;
    //move right to check first
    for (counter = 0; counter < _no_consecutive_markers_to_win-1; counter++)
    {
        if(CurrentWidth == _board_width - 1)
            goto MoveLeftAndCheck;
        if (board[CurrentWidth][y] != board[CurrentWidth + 1][y] || board[CurrentWidth][y]==' ')
        {
            goto MoveLeftAndCheck;
        }
        CurrentWidth++;
    }
    return true;
MoveLeftAndCheck:
    for (counter = 0; counter < _no_consecutive_markers_to_win-1; counter++)
    {
        if (CurrentWidth == 0)
        {
            return false;
        }
        if (board[CurrentWidth][y] != board[CurrentWidth - 1][y] || CurrentWidth <= 0 || board[CurrentWidth][y] == ' ')
        {
            return false;
        }
        CurrentWidth--;
    }
    return true;

}
```

6.2 undo

We have a stack containing all players' moves from the start of the game, before each turn the user is asked if he want to undo the previous move, if he chooses to do so, the top element of the stack is popped, and this move is then deleted from the board and the turn goes to the player who made that move.

```
Player TicTacToeGame::Undo()
{
    board[UndoStack.top().GetX()][UndoStack.top().GetY()] = ' ';
    Player temp = UndoStack.top().GetPlayer();
    UndoStack.pop();
    return temp;
}
```

```
Player Connect4Game::Undo()
{
    board[UndoStack.top().GetX()][UndoStack.top().GetY()] = ' ';
    _columns_counters[UndoStack.top().GetX()]--;
    Player temp = UndoStack.top().GetPlayer();
    UndoStack.pop();
    return temp;
}
```

```
void BoardGame::startGame()
{
    list<Player>::iterator it = _players.begin(); //players iterator
    int turns_counter = 0;
    printBoard();
    while (true) {
        //giving option to undo
        if (!UndoStack.empty())
        {
            cout << "Press U to Undo anything else to continue :" << endl;
            char choice;
            choice = getChar();
            if (choice == 'u' || choice == 'U')
            {
                Player GivingTurn = Undo();
                printBoard();
                it--; //turn goes to the previous player
                turns_counter--;
            }
        }
        //checks if number of plays is equal to number of places in the board
        if (turns_counter >= _board_width *_board_height) {
            cout << "Game ended in tie" << endl;
            break;
        }
        //if the iterator reaches to the end of the players list .. the iterator start over from the begining
        if (it == _players.end())
            it = _players.begin();
        //PlayTurn will write the player turn in the board and return true if the player who played the turn has won
        if (playTurn(*it)) {
            printBoard();
            cout << (*it).getName() << " Won!" << endl;
            SetHighScore(*it, turns_counter);
            break;
        }
        printBoard();
        turns_counter++;
        it++;
    }
}
```

7.0 Future Work

- Adding GUI layer.
- Adding playing vs the computer in a board bigger than 3x3 and in connect 4 as well.