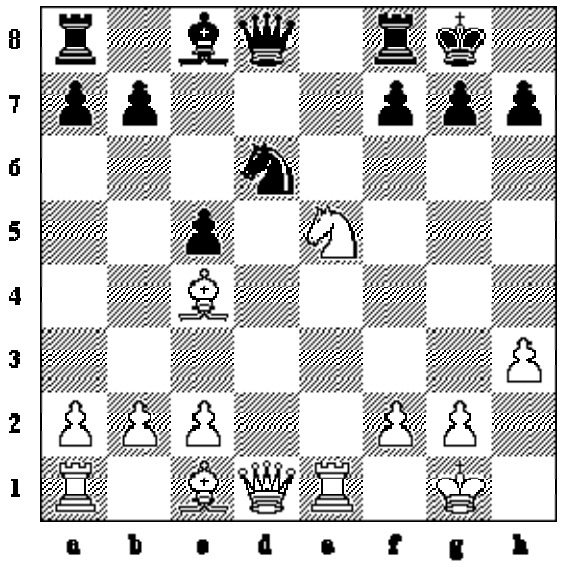
Assignment

Implement a chess system, which reads the chess-piece position file recorded in Forsyth notation, calculates an appropriate move for the white player and outputs the action represented in algebraic notation.

For example, for the following board,



in Forsyth notation this position would be recorded as:

```
r1bq1rk1; pp3ppp; 3n4; 2p1N3; 2B5; 7P; PPP2PP1; R1BQR1K1.
```

The pieces are represented by their letter symbols (For example, Q = Queen, R = Rook etc.). The letter symbols for Black are written in lower case and for White they are written in capitals. Empty squares are represented by a number. For example, 1 means there is one empty square, 2 means there are two empty squares and so on.

The position is recorded rank by rank starting with the eighth rank (a8) and the ranks are separated by semi-colons.

Thus `r1bq1rk1`; means on a8 there is a Black Rook, then an empty square (b8), a Black Bishop (c8), a Black Queen (d8), an empty square (e8), a Black Rook, (f8), a Black King (g8) and an empty square (h8). See diagram below.

3n4; means on row 6 there are three empty squares, then a Black Knight, and 4 empty squares as shown in the diagram below.

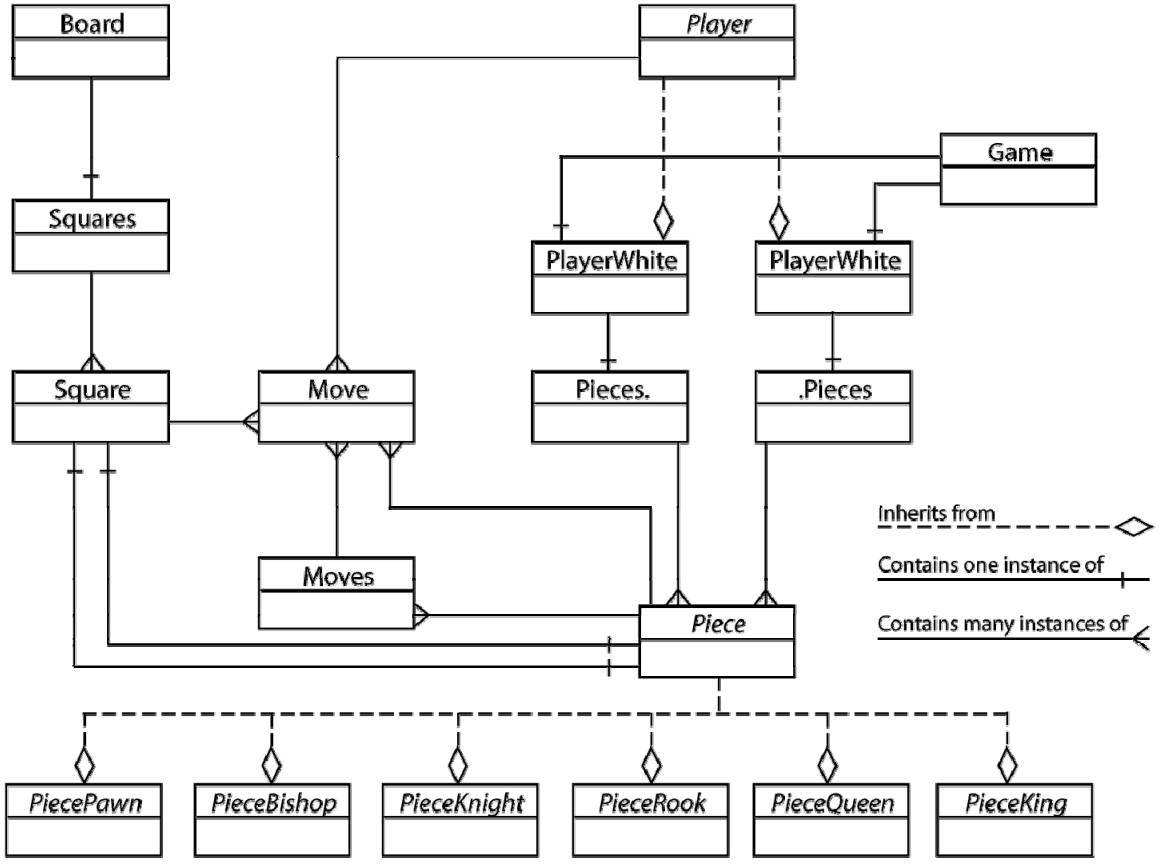
If a whole row contains empty squares like this:

The possible action of white player in above position could be represented in algebraic notation as follows:

`b2-b3`

The program has to parse an input file name named `position.txt` and output a file named `white_action.txt` with an appropriate response of the white player. The execution file for the program is `Chess.exe`.

The class hierarchy could be designed as follows (members and methods omitted) but other implementations are allowed:



Searching Algorithm

Chess is a strategy game where the goal is to capture the king piece of the opponent. Although playing any legal move is a possible tactic, the method is going to fail if the opponent adopts a more rigorous approach. A good player shall be able to anticipate the reaction of his adversary and thus, to select the best move according to some criterion. This way of playing describes a decision problem that is called *MinMax*, and which consists of minimizing the potential maximal lost of a player. Note that this is a conservative approach that aims at fending against the best possible move of the opponent. Due to the complexity of Chess, it is possible to look in advance up to a certain depth only, and thus, the player that is able to explore the tree of the game deeper than his adversary has an important advantage.

The MinMax problem considers a performance function that is also called an *evaluation function*, and which is used to rank the states of the game. In case of Chess where it is impossible to look ahead at all final positions, this metric estimates the goodness of a position.

The pseudo code of the MinMax algorithm is presented below. Given a *node* of the game tree and a specific exploration *depth*, the function *minmax* computes the maximum gain of the opponent according to the performance function *evaluate*. If either the node is a final state or the depth falls to zero then the goodness of the node is given by the evaluation function. Otherwise, a depth-first exploration of the tree is performed. The algorithm distinguishes between the two players. If it is the turn of the first player to move then the function returns the minimal gain that can be achieved. Symmetrically, if the opponent has to select a move then the algorithm returns the maximum value he can achieve.

```
int minmax(node, depth)
{
        if (node.IsTerminal || depth == 0)
                return evaluate(node);
        if (node.WePlay)
        {
                int e = MAXINT;
                foreach child of node
                        e = min(e, minmax(child, depth-1)
                return e
        }
        else // It's the turn of the opponent
        {
                int e = MININT;
                foreach child of node
                        e = max(e, minmax(child, depth-1))
                return e
        }
}
```

Although the principle of the algorithm is simple, the size of the game tree grows exponentially with the depth of the exploration. Pruning techniques are used to restrict the branching. One possibility consists of considering promising moves only, scarifying thus the accuracy of the algorithm. Better solutions are based on proving that exploring subtrees are worthless. For instance, the *alpha-beta pruning technique* also uses the algorithm of the adversary. It stops

evaluating a subtree if the opponent can play in a better way that ends up in a position more favorable to him than the current one. Consequently, the accuracy of the MinMax computation is preserved. The pseudo-code of the algorithm is shown here:

```
int minimax(node, depth)
{
    return alphabeta(node, depth, MININT, MAXINT)
}

int alphabeta(node, depth, α, β)
{
    if (node.IsTerminal || depth == 0)
            return evaluate(node);

    foreach child of node
    {
        α := max(α, -alphabeta(child, depth-1, -β, -α))
        if α ≥ β
            return α
    }
    return α
}
```

Furthermore note that the pruning depends on the exploration order of the nodes. Therefore, one way of improving the alpha-beta pruning algorithm consists of considering promising branches first.

Being able to explore the tree of the game deeper than his opponent is one important facet of the problem. Another crucial aspect consists of designing a good evaluation function that actually defines the pertinence of the computer's moves. The most basic function consists of counting the value of the pieces. The standard valuation is the following:

- King: Infinity
- Queen: 9 points
- Rock: 5 points
- Bishop: 3 points
- Knight: 3 points
- Pawn: 1 points

Naturally, the evaluation function can also take into account additional criteria such as the position of the pieces, their combination, the safety of the King, the control of the center. As the literature is abundant and interesting, you are encouraged to read it and to establish your own evaluation function.

The first part of the project consists of building engine for playing chess. The solution shall be based on the MinMax principle. Performances are not taken into account for the grading of this first assignment. However, the application shall give the next move in a reasonable amount of time (a few seconds). Furthermore, the answer shall respect the rules of Chess [1]. At this stage, the pertinence of the answer is not graded.

Requirements

The following requirements have to be fulfilled in order to get a positive grading:

1. The delivered code has to comply with the design guidelines [2].
2. The program has to compile with Visual Studio 2005 and the .Net Framework 2.0 under Windows XP.
3. The program has to run under Windows XP with the .Net Framework 2.0.
4. You must provide a one to two page document that describes how your program works. You have to include a complete class diagram in UML notation [3].

Code Delivery

Deadline: May 10th

Implementation tasks and points assignment:

- Parsing the Forsyth notation position file (10 points)
- Define the class hierarchy (10 points)
- Implement a search algorithm (10 points)

References

[1] Chess-Rules:
http://www.chesscorner.com/tutorial/learn.htm

[2] Coding Guidelines:
http://se.inf.ethz.ch/teaching/ss2007/251-0290-00/project/CSharpCodingStandards.pdf

[3] UML:
http://en.wikipedia.org/wiki/Unified_Modeling_Language