Design Document for a Chess Program

(Note: This document is meant to give you an idea what we're looking for. Yours need not follow exactly the same structure. In fact, this one is incomplete. For example, many of the correctness constraints are not satisfied by what we have here. Also, pseudo-code is missing for some of the methods. These sorts of things will not be acceptable in your design doc.)

The Problem

Our goal is to implement a completely functional chess program. We will not implement an AI for the program. Rather, this game is meant for two human opponents to play against each other. A game of chess involves chess pieces, and a chess board.

The Board

The chess board is an 8 by 8 grid. The initial configuration of the pieces is as follows:

R	N	B	Q	K	B	N	R
P	P	P	P	P	P	P	P
P	P	P	P	P	P	P	P
R	N	B	Q	K	B	N	R

Where R=rook, N=knight, B=bishop, Q=queen, K=king, P=pawn

The Pieces

1. Pawns
    a. Normally move forward one space.
    b. Move diagonal one space to kill.
    c. May move two spaces forward on the first move.
        i. But may be killed En Passant after this move by attacking the space behind the pawn.
    d. May be exchanged for any piece except a King by reaching the opposite side of the board.
2. Bishops
    a. Move diagonally any distance.
    b. Kill by landing on a space occupied by an opponent's piece.

3. Knights
    a. Move in "L" shape (one space one direction, two in a perpendicular direction).
    b. May go "over" other pieces.
    c. Kill by landing on a space occupied by an opponent's piece.
4. Rooks
    a. Move horizontally or vertically any distance.
    b. Kill by landing on a space occupied by an opponent's piece.
    c. May "castle" with the king under certain circumstances (see below).
5. Queens
    a. Move any distance horizontally, vertically, or diagonally.
    b. Kill by landing on a space occupied by an opponent's piece.
6. Kings
    a. Move one space horizontally, vertically, or diagonally.
    b. Kill by landing on a space occupied by an opponent's piece.
    c. May "castle" with a rook under certain circumstances (see below).
    d. The game is over:
        i. when the opponent's king cannot escape being taken (i.e. Checkmate).
        ii. or one player can make no move without putting his/her king in danger (i.e. Stalemate)

The Object

The object of the game is to kill the opponent's king (i.e. get him/her in Checkmate).

Added Features

We will also implement a feature which records the progression of the game, and thus it will be possible to undo a move or look back at how the game progressed. This will also prove useful for implementing En Passant.

Implementation

API:

The API for this program is very simple. After creating a new Board object, one needs merely design a GUI which calls Board::TryToMove() to try to make a given move. The piece occupying a space on the board at a given time may be accessed by calling Board::PieceAt().

Classes:

Point: Since we will be dealing with a lot of two dimensional coordinates, we will implement a class called Point, which will allow us to more easily return two dimensional results from methods such as GetPosition().

Header:

```
class Point {
   private:
      int x, y;
   public:
      Point(int i, int j);  // x = i; y = j;
      int GetX();
      int GetY();
      void SetX(int i);
      void SetY(int j);
      void SetPoint(int i, int j); x = i; y = j;
}
```

Methods:

The methods in the class are pretty trivial.

Correctness/What to test for:

- if you create a Point with coordinates (a,b), you get those coordinates back when you call GetX() and GetY()

History: A list of the moves which have taken place. It will store the moves as a pair of points and a piece. That way, a move can be undone by taking the piece at "to" and moving it to "from". Then killed_piece can be placed at "to".

Header:

```
class History {
   private:
      Board *board;
      Point from;       // the point a piece was moved from
      Point to;         // the point a piece was moved to
      Piece killed_piece;  // remembers what piece was killed
                           // on this move.
      History *next;
      History *prev;
   public:
      History(Point f, Point t, Board* b, History *p);
      SetNext(History *n);
      void UndoLastMove();
      void RedoUndoneMove();
}
```

Methods:

```
History::History(Point f, Point t, Board* b, History *p) {
   from = f;
   to = t;
```

```
    board = b;
    prev = p;
    next = NULL;
    killed_piece = board->PieceAt(to);
}
```

History::SetNext(...) -- trivial

Correctness/What to test for:

- When you undo a normal move (not a kill, castle, etc), the piece which moved goes back to where it came from.
- When you undo a castle, both the rook and the king go back to the correct position.
- When you undo a normal kill (not En Passant), the killed piece comes back.
- When you undo an En Passant kill, the pawn goes back to the right location.
- When you redo any move, it behaves correctly (hand-wavey because it's easier than undoing -- simply call Board::TryToMove() as if somebody had just tried to make that move)

Board: The board class will represent the playing board itself. It will keep track of which pieces are in which positions at a given moment.

Header:

```
class Board {
   private:
      Piece* the_board[width][height];
      Color turn;
      History* first_history;
      History* last_history;
      Point en_passant:
   public:
      Board();          // creates board with pieces in initial configuration;  white's
turn
      Piece* PieceAt(Point p);          // return the piece at location p
      Piece* PieceAt(int x, int y);       // return the piece at location (x,y)
      PlacePieceAt(Piece* p, Point* pt);    // place piece p at location pt
      void Move(Point p1, Point p2);    // move piece at p1 to p2
      void Move(Point p1, Point p2, Point ep); // move piece at p1 to p2,
remembering
                                                // space that was jumped over
      void TryToMove(Point p1, Point p2);
```

```
    Point GetEnPassant();
}
```

Methods:

```
Board::Board() {
   Piece *temp_piece;
   turn = WHITE;
   first_history = last_history = NULL;
   en_passant = NULL;
   for (int i; i < width; i++)
      for (int j; j< width; j++) {
         temp_piece =  the correct piece (NULL for empty squares)
         the_board[i][j] = temp_piece;
      }
}

// This method moves the piece occupying p1 to p2,
// without any check as to whether the move is legal
void Board::Move(Point p1, Point p2, Point ep) {
   add the move from p1 to p2 to the history;
   en_passant = ep;
   change the turn;
   set location p2 on the board to point to the piece that is at p1 now;
   set location p1 to NULL;
}

void Board::Move(Point p1, Point p2) {
   Move(p1, p2, NULL);
}

// Moves the piece from p1 to p2 only if the move is legal
void Board::TryToMove(Point p1, Point p2) {
   Piece * temp_piece = piece at p1;
   if (temp_piece != NULL)
      temp_piece->TryToMove(p2);
}
```

The rest of the methods are trivial.

Correctness/What to test for:

- TryToMove() will perform the specified move ONLY if it is legal:
    - none of your own pieces occupy the space
    - the move conforms with the definition of the behavior of the piece at p1
    - the move does not put the player whose turn it is into check

- TryToMove() will take care of the necessary side effects:
  - when you move a king or a castle, it is noted that that particular piece is no longer elligible for castling
  - when you move a king to a castling position, the rook moves to his opposite side
  - when you move a pawn two spaces forward, the fact that it may be killed En Passant is recorded
  - when you move a piece to kill a pawn En Passant, the pawn is actually removed
- Move() will move the piece from p1 to p2, elliminating whatever piece happens to be at p2 at the moment
- All moves will be recorded in the history

Piece: We will need a class for all pieces to inherit from. Therefore, we will create a class called Piece, which will implement some shared methods between all pieces, as well as some virtual functions used by all pieces.

Header:

```
class Piece {
   private:
      Board* board;
      Color color;
      Point location;
   protected:
      virtual MoveType CanMove(Point p);
         //   MoveType can be ILLEGAL, NORMAL, CASTLE, DOUBLESTEP,
         //   ENPASSANT
   public:
      Piece(Point p, Color c, Board* b);
      Point GetLocation();
      Color GetColor();
      void SetLocation(Point p);
      void SetLocation(int x, int y);
      virtual void TryToMove(Point p);
}
```

Methods:

```
// returns ILLEGAL for illegal moves, ENPASSANT
// when the move kills a pawn by the en passant rule
MoveType Piece::CanMove(Point p) {
   if (p is on the board) {
      if (there is no piece at p)
         if (board->GetEnPassant() == p)
            return ENPASSANT;
```

```
            else
               return NORMAL;
         if (the color of the piece at point p is different than mine)
            return NORMAL;
      }
      return ILLEGAL;
}

// moves the piece to p only if the move is legal
// kills the en passant pawn if necessary
void Piece::TryToMove(Point p) {
      if (CanMove(p) != ILLEGAL)
         if (CanMove(p) == ENPASSANT)
            kill pawn;
         board->Move(location, p);
}
```

All other methods are trivial.

Correctness/What to test for:

- CanMove() does not allow a piece to take another of the same color.
- CanMove() does not allow moving a piece off the board.
- CanMove() does not allow a move which would put the current player in check.
- TryToMove() only makes the move when CanMove() says it is legal.
- pawns are correctly killed in En Passant.

Pawn: This class inherits from Piece and defines it's own CanMove() and TryToMove(). A pawn is the only piece with any concept of direction (they can never move backwards), so the direction attribute is unique to this class.

Header:

```
class Pawn : Piece {
   private:
      int direction;  // 1 for up, -1 for down
   protected:
      bool CanMove(Point p);
   public:
      Pawn(Point p, Color c, Board * b, int d);
      void TryToMove(Point p);
}
```

Methods:

```
Pawn::Pawn(Point p, Color c, Board* b) {
   Piece(p, c, b);
   if (c = WHITE)
```

```
      direction = 1;
   else
      direction = -1;
}

// redefine CanMove according the rules of pawn movement
MoveType Pawn::CanMove(Point p) {
   if(Piece.CanMove(p) == ILLEGAL)
      return ILLEGAL;
   int dy = p.GetY() - location.GetY();
   int dx = p.GetX() - location.GetX();
   if(dy == direction) { // one space forward
      if ((abs(dx) == 1) && (piece at p is opposite color))     // if we're killing
diagonally
         return NORMAL;
      if ((dx == 0) && (point p is unoccupied))        // one step forward
         return NORMAL;
   }
   if ((dy == 2*direction)
      && (dx == 0)
      && (point p is unnoccupied)
      && (point (location.getX(), location.getY() + direction) is unoccupied)
         return DOUBLESTEP;
   return ILLEGAL;
}

// redefine TryToMove such that, if this pawn made a double step,
// the fact that it can be killed en passant is recorded
void Pawn::TryToMove(Point p) {
   MoveType mt = CanMove(p);
   if (mt != ILLEGAL) {
      if (mt == DOUBLESTEP)
         board->Move(location, p, space between location and p);
      else {
         if (mt == ENPASSANT)
            kill pawn;
         board->Move(location, p);
}
```

Correctness/What to test for:

- pawns make all of the correct moves:
    - a pawn may move one space forward any time the space in front of it is unoccupied
    - a pawn may move one space diagonally forward to kill piece of the opposite color

    - a pawn may move two spaces forward if the two spaces in front of it are unoccupied and it is still in its starting location
- a pawn must be exchanged for another type of piece if it makes it to the other side of the board.

Bishop: This class inherits from Piece and defines it's own CanMove(). Piece::TryToMove() is sufficient for bishop, because they have no special moves like double step or en passant that we have to deal with.

Header:

```
class Bishop : Piece {
   protected:
      MoveType CanMove(Point p);
}
```

Methods:

```
MoveType Bishop::CanMove(Point p) {
   MoveType move = Piece.CanMove();
   if (move == ILLEGAL)
      return ILLEGAL;
   int dy = p.GetY() - location.GetY();
   int dx = p.GetX() - location.GetX();
   if (abs(dx) == abs(dy))    //same distance in x and y dirs
      if (no pieces between location and p)
         return move;
   return ILLEGAL;
}
```

Correctness/What to test for:

- Bishops make the correct moves:
    - a bishop may move any distance diagonally in any direction as long as it doesn't "go over" another piece.
    - a bishop may kill by landing on its space if the piece is on a space the bishop can legally move to.

Rook, Knight, etc... You get the idea. (btw, you CANNOT say this on your design doc...)