EECS285, Fall 2013

Programming Project 2 Specification

Overview:

For this project, you will begin to implement a chess game. Chess is a game that consists of six different kinds of pieces, each of which has different rules about how it may move, which are described below. This project is about inheritance and polymorphism, and you must not try to "get around" using those aspects of OOP. For this reason, take special note of the "Restrictions" section at the end of this document to make sure you avoid huge penalties.

This project is due ***Monday, October 21, 2013 no later than 2:59pm***. Use the same submission process as was provided for project 1.

Project Organization

This project will require you to develop several Java source files. Because all of these files are related in the sense that they all support this particular project, we will put them all in a package. The package name will be "eecs285.proj2.<uniqname>". For example, my UM uniqname is morgana, therefore, I would use a package name of "eecs285.proj2.morgana".

When you submit your project, attach each individual .java source file to the submission page on ctools. Please do NOT try to include any folders with your submission – we will organize your submissions into appropriate paths during grading. It is very important that you name your package exactly as described so that your project can be graded successfully, so please double check that each .java file specifies the package it belongs to at the top of the file and that the package is named eecs285.proj2.<uniqname>.

Piece Descriptions

King: The King can move exactly one space in any direction, unless the space is occupied by another piece of the same color. If the space is occupied by a piece of the opposite color, that piece can be captured, allowing the King to move to the previously occupied space.

Queen: The Queen can move in any straight line (forward, backward, left, right, or any of the four diagonals). She may move any number of spaces in a straight line, and stop at either a vacant space, or the first space occupied by a piece of the opposite color which would be captured by the move. The Queen may not “jump”, and must stop before reaching a space containing a piece of the same color.

Bishop: The Bishop moves exactly like the Queen, but only on one of the four diagonals. The Bishop may not move straight forward, backward, left, or right.

Rook: The Rook moves exactly like the Queen, but only forward, backward, left, or right. The Rook may not move diagonally.

Knight: The Knight is the only piece which is not obstructed by another piece along its path. A Knight always moves one space in one direction, and two spaces in a perpendicular direction. The Knight’s movement is shaped like an ‘L’.

Pawn: The Pawn has the most complicated movement rules. If a pawn is in its starting position, it may move either one or two spaces forward, unless obstructed by a piece of either color. If a pawn is not in its starting position, it may only move one space forward, unless obstructed by a piece of either color. Pawn's are not allowed to capture opposing pieces during either of these forward movements. If an opposite color piece occupies a space one row forward and one row either left or right (i.e. one row forward on the diagonal), then the pawn may move to that space and capture the opposing piece. A pawn can only move diagonally when capturing a piece as described; when not capturing a piece, the pawn may only move straight forward.

Detailed Description:

You will develop a set of classes that will allow a user to call functions to set up a situation in a chess game, print an ASCII version of the chess board to the screen, determine how many valid moves there are, and to optionally print the valid moves to the screen.

Your program must make use of inheritance and polymorphism. Create a base class called ChessPiece. Next, create a subclass for each specific type of piece that extends the base class. These classes must be called: Pawn, Rook, Knight, Bishop, Queen, and King.

Finally, you must have a ChessBoard class which stores an 8x8 two-dimensional array of ChessPiece references, some of which will be null, and some of which will refer to chess pieces of specific types (i.e. Rook, Pawn, etc). Note that, with polymorphism, the references in the ChessBoard array will be of type ChessPiece, but will point to the specific piece types, such as King, Bishop, etc.

We will test your program using only a relatively small number of publicly accessible functions, which must be implemented exactly as specified. Following the specifications for these functions *exactly* will allow our test programs to be compiled and executed using your classes. If the test cases don't compile there will be a deduction, even if the needed change is minimal. You will certainly want to implement additional functions to modularize your code to suit your design.

A Quick Word on Design

Your design will be a consideration in your final project grade. When designing your class hierarchy, carefully consider the "is-a" relationship. For example, even though a Rook moves in a similar way to a Queen, it is not true that a Rook "is-a" Queen, or that a Queen "is-a" Rook. Therefore, your class hierarchy should not be set up such that a Rook inherits from Queen or vice-versa. Further, make sure if you develop methods at a base class-level that the method is applicable to all subclasses. In other words, don't have a "movePawn(…)" method defined in the ChessPiece class, because it applies (presumably) only to the Pawn type and not the other types that inherit from ChessPiece. Spend a little time thinking about your design and ensure you don't make these types of mistakes which will result in deductions.

Also, the array in the ChessBoard class must be zero-based. When printing the board (see below), the first row number is 1, but that will be contained in row index 0 of the array. Your array should be exactly 8x8, with the indices ranging from 0-7, inclusive.

The required functions are specified here:

ChessBoard class required methods:

ChessBoard(): The default constructor – simply allocates the array of ChessPiece objects. Does not need to create any ChessPiece objects.

void initialize(): This method will allocate and place all the chess pieces in their home positions, which are illustrated in the description of the print method, below.

ChessPiece getPieceAt(final int row, final int col): This method returns a reference to the piece at the specified row and column, or returns null if the specified space does not contain a piece. If the row and column provided do not represent a valid space, print an appropriate error message to the screen and return null.

void print(): Prints the current state of the chess board, using ASCII characters. Spaces containing a piece indicate the piece using two characters, a lower case 'w' or 'b' (for "white" or "black"), followed by an upper case 'R', 'N', 'B', 'Q', 'K', or 'P' (for "Rook", "Knight", "Bishop", "Queen", "King", or "Pawn"). Note that 'N' is used to represent Knight to avoid a conflict with King. Row and column labels are included, along with separator characters. Below is an example of the output from the print function, immediately after initialize() had been called.

```
   |--|--|--|--|--|--|--|--|
8  |bR|bN|bB|bQ|bK|bB|bN|bR|
   |--|--|--|--|--|--|--|--|
7  |bP|bP|bP|bP|bP|bP|bP|bP|
   |--|--|--|--|--|--|--|--|
6  |  |  |  |  |  |  |  |  |
   |--|--|--|--|--|--|--|--|
5  |  |  |  |  |  |  |  |  |
   |--|--|--|--|--|--|--|--|
4  |  |  |  |  |  |  |  |  |
   |--|--|--|--|--|--|--|--|
3  |  |  |  |  |  |  |  |  |
   |--|--|--|--|--|--|--|--|
2  |wP|wP|wP|wP|wP|wP|wP|wP|
   |--|--|--|--|--|--|--|--|
1  |wR|wN|wB|wQ|wK|wB|wN|wR|
   |--|--|--|--|--|--|--|--|
    a  b  c  d  e  f  g  h
```

public void removePiece(final int rowNum, final int colNum): This method removes the piece from the specified row and column. If the space does not contain a piece, the method has no effect. If the specified indices do not represent a valid space, an appropriate error message is printed, and the method has no effect.

public void placePiece(final ChessPiece pieceToPlace, final int rowNum, final int colNum): This method places the piece specified as the first parameter in the space indicated via the rowNum and colNum parameters. If the space already contains a piece, it is replaced with the piece passed in, and is not

considered an error. If the row and column provided do not represent a valid space, an appropriate error message is printed, and the method has no effect.

ChessPiece class required methods:

public abstract int getNumberOfMoves(final ChessBoard board, final boolean printMoves): This method is abstract, and therefore has no implementation in the ChessPiece class. It must be implemented in each of the subclasses, as appropriate for the subclass type. The current state of the chess board is provided, as well as a boolean value indicating whether to print out the valid moves that were found or not. The method returns an integer, which represents the number of valid moves that were found.

The rest of the design is up to you. You should utilize inheritance and polymorphism as fully as you can to develop concise and easy-to-understand code. Avoid duplicating code when possible – write functions to perform tasks that are needed multiple times, and place parent class-specific code at the parent class, rather than duplicating it in each subclass.

A sample application class, called ChessGame.java will be provided to you, with a straight forward initial test case. The expected output for this test case is also provided. You should develop further test cases to ensure that other cases are tested and produce expected results. Remember to change the package name at the top of the provided ChessGame.java to replace my uniqname with yours.

Restrictions

IMPORTANT: In order to force you to utilize polymorphism to its full extent, you may ***not*** declare any references to specific chess piece types. In other words, you may not declare a reference of type Pawn, Rook, Queen, etc. We will absolutely be checking for this, and if we find any references of these specific chess piece types (as opposed to the more generic parent type ChessPiece, which is ok) there will be a ***minimum*** 30 point deduction. As an example:

```
Pawn myPiece = new Pawn(…); //NOT allowed!

ChessPiece myPiece = new Pawn(…); //Allowed
```

Very similarly, you may not use any significant type casting to cast a ChessPiece reference to a specific chess piece type. Similar deductions will be taken in this case. For example:

```
((Pawn)board[0][0]).moveThePawn(); //NOT allowed!

board[0][0].someGenericMethod(); //Allowed
```

Finally, and again similarly, you should not need any long "if..else if..else if.." statements to decide what to do. Specifically, do not use code that says "if the type is a Pawn then do X() else if the type is Knight do Y()" etc.