**Programming Project 5**

**Due Sunday, December 6 at 11:59pm**

**As Monday, December 7 is Reading Day and that day should to be spent preparing for final exams, *no late assignments will be accepted*.**

**I. Overview**

The goal of this assignment is to build a JavaFX application and to work on creating type hierarchies. The end result should be really neat: a chess game that works with both the Swing and JavaFX graphical libraries and can choose between different variations of chess.

**II. Code Readability (20% of your project grade)**

To receive the full readability marks, your code must follow the following guideline:

* All variables (fields, parameters, local variables) must be given appropriate and descriptive names.
* All variable and method names must start with a lowercase letter. All class names must start with an uppercase letter.
* The class body should be organized so that all the fields are at the top of the file, the constructors are next, the non-static methods next, and the static methods at the bottom with the main method last.
* There should not be two statements on the same line.
* All code must be properly indented (see Appendix F of the Lewis book for an example of good style). The amount of indentation is up to you, but it should be at least 2 spaces, and it must be used consistently throughout the code.
* You must be consistent in your use of {, }. The closing } must be on its own line and indented the same amount as the line containing the opening {.
* There must be an empty line between each method.
* There must be a space separating each operator from its operands as well as a space after each comma.
* There must be a comment at the top of the file that **is in proper JavaDoc format** and includes both your name and a description of what the class represents. The comment should include tags for the author. (See Appendix J of the Lewis book of pages 226-234 if the Evans and Flanagan book.)
* There must be a comment directly above each method (including constructors) that **is in proper JavaDoc format** and states *what* task the method is doing, not how it is doing it. The comment should include tags for any parameters, return values and exceptions, and the tags should include appropriate comments that indicate the purpose of the inputs, the value returned, and the meaning of the exceptions.
* There must be a comment directly above each field that, in one line, states what the field is storing.
* There must be a comment either above or to the right of each non-field variable indicating what the variable is storing. Any comments placed to the right should be aligned so they start on the same column.
* There must be **a comment above each loop that indicates the purpose of the loop**. Ideally, the comment would consist of any preconditions (if they exist) and the subgoal for the loop iteration.
* Any code that is complicated should have a short comment either above it or aligned to the right that explains the logic of the code.

**III. Program Testing (20% of your project grade)**

You are to write a test report that indicates the *types* of tests needed to thoroughly test your project. The tests should demonstrate that all parts of your code behave correctly. Any unit of your program involving conditional statements will need tests that go through each branch of the execution. Any unit of your program involving loops will need tests that cover the "test 0, test 1, test many" and "test first, test middle, test last" guidelines. Your testing report should *not* list the actual tests and results.

You are to have a JUnit test class or classes that implement as many of the tests as you can. You should have comments, names, or other indicators in your JUnit tests that easily link the JUnit tests back to the testing report.

*The testing report must be separate from the JUnit class.* In most companies, the testing document will be written in a style that allows both programmers and non-programmers to read it and recognize whether all the needed test cases were included.

**Note** that you will not be able to (easily) test methods involving user input or screen output with JUnit. For these parts of your program, your testing report should indicate the specific tests you did to test these routines.

*Hint:* Because doing JUnit tests on JavaFX is challenging, you should keep the game mechanics completely separate from the graphical display code. This is what you were to do in Project 3, and you should continue doing that in Project 5. This way, you can use JUnit to test all the pieces and game mechanics.

**IV. Java Programming (60% of your grade)**

For this project you will take the Indo-European version of chess that you created for project 3 and modify it in the following ways:

1. You will make a JavaFX chessboard for the game and create the classes so that the chess pieces can easily work with either the Java Swing version of the chessboard or the JavaFX version of the chessboard.
2. Add in additional pieces and a new implementation of the board display so that users can choose to play either Indo-European chess or Chinese Xiangqi chess.

**What you must do:**

1. Create the types listed below to create a JavaFX chessboard
2. Modify the chessboard that was provided to in in Project 3 so that it can fit in a hierarchy with the JavaFX chessboard
3. Create the Xiangqi chesspieces and add them into the chess pieces hierarchy you made in Project 3. You are welcome to redesign that heirarchy.
4. Create display and game types below so users can create and start either an Indo-European or a Xiangqi chess game.

**What you do not have to do:**

1. You do not have to get the Indo-European chess pieces to work.
2. You do not have to get the Indo-European chess game to play correctly.

**You are to create the following types**

You are welcome to add additional types and/or methods as you feel is appropriate.

1. **ChessBoard:** The ChessBoard will now represent an arbitrary chessboard that does not depend on either Swing or JavaFX. The ChessBoard type will have the following methods:
   1. ChessGame getGameRules: returns the ChessGame being played on this board.
   2. void addPiece(ChessPiece piece, int row, int column): adds a chess piece to a given row and column position on the board.
   3. ChessPiece removePiece(int row, int column): removes the piece at the given row and column from the board and returns the removed piece.
   4. boolean hasPiece(int row, int column): returns true if there is a piece on the board at the given row and column.
   5. ChessPiece getPiece(int row, int column): returns the chess piece located on the board at the given row and column.
   6. boolean squareThreatened(int row, int column, ChessPiece piece): returns true if a player from a different Side as the chess piece can make a legal capture move to the given row and column on the board.
2. **SwingChessBoard:** The SwingChessBoard is an implementation of the ChessBoard type for the Java Swing package. (You should take the ChessBoard.java file from Project 3 and *slightly* modify it to make the SwingChessBoard.) The SwingChessBoard should have the following *constructor*:
   1. SwingChessBoard(SwingChessBoardDisplay boardDisplay, ChessGame chessGame): The constructor should create a grid of JButtons for the board. You can use the GridLayout on a JPanel to organize the buttons, and place the JPane onto a JFrame . The constructor should use the ChessGame input to determine the number of rows and columns in the grid, and it should use the SwingChessBoardDisplay to determine how to display each button.

Each button should have an ActionListener that:

* + 1. If there is a piece on that button, it calls the ChessGame legalPieceToPlay method, and if it returns true, it uses the SwingChessBoardDisplay to highlight that button.
    2. On the second click, it calls the ChessGame makeMove method to move the piece from the highlighted square to the new location on the board. If the method returns false, and if the ChessGame canChangeSelection method returns true, the square is unhighlighted (using the SwingChessBoardDisplay), and the user can select a new piece. Otherwise, the square remains highlighted and the user must select another square to move the piece.

At the end of the constructor, the JFrame used to display the chessboard should be made visible.

1. The SwingChessBoard should override the various methods of ChessBoard as needed and it should use the SwingChessBoardDisplay to update the JButton for the square any time a piece is added or removed from the button.
2. **SwingChessBoardDisplay:** The SwingChessBoardDisplay indicates how the SwingChessBoard should display its squares. The SwingChessBoardDisplay should have the following methods:
   1. void displayEmptySquare(JButton button, int row, int column) does the display for an empty square at the given row and column
   2. void displayFilledSquare(JButton button, int row, int column, ChessPiece piece) does the display for a square at the given row and column with the given piece on it.
   3. void highlightSquare(boolean highlight, JButton button, int row, int column, ChessPiece piece) adds or removes a highlight from a square on the chess board. The square is at the given row and column, and the piece is the piece currently on the square.

Note that this class is essentially the ChessBoardDisplay.java file from Project 3.

1. **SwingEuropeanChessDisplay:** An implementation of the SwingChessBoardDisplay for Indo-European chess. The default is to make the squares alternate background colors. You can do anything you want for the display of a piece, but my suggestion is to set the JButton background to be the color of that piece's side, and make the text of the JButton the piece's label.

Note that this class is the EuropeanChessDisplay.java file from Project 3.

1. **ChessGame:** The ChessGame contains methods indicating how the game is supposed to be played. The ChessGame type will have the following nested type:
   1. enum Side: determines the "player" for the game. The possible sides should be NORTH, SOUTH, EAST, and WEST.

The ChessGame will have the following methods:

* 1. boolean legalPieceToPlay(ChessPiece piece, int row, int column): returns true if it is legal to move the given piece from its current position to the given row and column.
  2. boolean makeMove(ChessPiece piece, int row, int column): Will move the given piece on the board from its current position to the given row and column, if it is legal and possible to make that move. The method will return true if the piece was successfully moved and false if the piece could not be moved.
  3. boolean canChangeSelection(ChessPiece piece) returns true if the player is allowed to change their choose on what piece to move. The input piece is the piece the player selected to move.
  4. int getNumRows(): Returns the number of rows in this chess game.
  5. int getNumColumns(): Returns the number of columns in this chess game.
  6. void startGame(ChessBoard board): Calling this method will take the given board, place all the initial pieces for the game onto the board in their correct starting positions, and sets any internal fields (if needed) so that the game can begin.

Note that the ChessGame type is slightly changed from what you did in Project 3 with three new methods added.

1. **EuropeanChess:** This type extends ChessGame. It is the class you created in Project 3 with three new methods added.
2. **JavaFXChessBoard:** The JavaFXChessBoard is an implementation of the ChessBoard type for the Java FX package. The JavaFXChessBoard should extend Application, and the JavaFXChessBoard should have its start method do the following:
   1. Get the command line arguments using the getParameters().getRaw() method of Application.
   2. Use the first command line argument to determine what version of chess to use. If the argument is "chess" this will be Indo-European chess, and if it is "xiangqi" it will be Chinese Xiangqi chess. You are welcome to add more versions if you want.
   3. Create an instance of JavaFXChessBoardDisplay and the ChessGame for the desired game. If the desired game is "chess", these should be JavaFXEuropeanChessDisplay and EuropeanChess, and if the desired game is "xiangqi" these should be JavaFXXiangqiDisplay and Xiangqi.
   4. Create a 2-dimensional array of Buttons for the board. Place the buttons in GridPane that you then place in the Scene that is used for the primary stage. Use the JavaFXChessBoardDisplay to determine how to display each button.
   5. Each button should have an EventHandler that:
      1. If there is a piece on that button, it calls the ChessGame legalPieceToPlay method, and if it returns true, it uses the JavaFXChessBoardDisplay to highlight that button.
      2. On the second click, it calls the ChessGame makeMove method to move the piece from the highlighted square to the new location on the board. If the method returns false, and if the ChessGame canChangeSelection method returns true, the square is unhighlighted (using the JavaFXChessBoardDisplay, and the user can select a new piece. Otherwise, the square remains highlighted and the user must select another square to move the piece.

At the end of the start method, the startGame method of the ChessGame is called and the primary stage is displayed.

1. The JavaFXChessBoard should override the various methods of ChessBoard as needed and it should use the JavaFXChessBoardDisplay to update the Button for the square any time a piece is added or removed from the button.
2. You are welcome to use the Swing version of the chess board as a template for the JavaFX version. One thing to note is that there is a lot of thread code in the Swing version to make sure that the routines to draw the board are executed in the correct thread. JavaFX does a better job of ensuring that the correct thread is used for each method (this is why it uses the start method instead of the constructor to initialize it), and so you do not have to implement any thread routines in your JavaFX code.
3. **JavaFXChessBoardDisplay:** The JavaFXChessBoardDisplay indicates how the JavaFXChessBoard should display its squares. The JavaFXChessBoardDisplay should have the following methods:
   1. void displayEmptySquare(Button button, int row, int column) does the display for an empty square at the given row and column
   2. void displayFilledSquare(Button button, int row, int column, ChessPiece piece) does the display for a square at the given row and column with the given piece on it.
   3. void highlightSquare(boolean highlight, Button button, int row, int column, ChessPiece piece) adds or removes a highlight from a square on the chess board. The square is at the given row and column, and the piece is the piece currently on the square.
4. **JavaFXEuropeanChessDisplay:**

An implementation of the JavaFXChessBoardDisplay for Indo-European chess. The default is to make the squares alternate background colors. To set the background of a Button you use the setBackground method that takes an array of BackgroundFill instances.

For each blank square, I recommend making a BackgroundFill with the desired Color and use CorderRadii.EMPTY and Insets.EMPTY.

For the pieces, you would add a second BackgroundFill to the background but this time use the color for the piece's side, and set the CorderRadii and Insets so that the piece is rounded and allows the background color behind it to be visible.

Then make the button's text be the label for the piece. See the API for this to make some sense.

1. **JavaFXXiangqiDisplay:**

An implementation of the JavaFXChessBoardDisplay for Xiangqi. The default is to make the squares a light gray color and use a darker gray for the middle three squares on the top three rows and bottom three rows of the board (assuming your game is north to south). To set the background of a Button you use the setBackground method that takes an array of BackgroundFill instances.

For each blank square, I recommend making a BackgroundFill with the desired Color and use CorderRadii.EMPTY and use a small Insets value so the grid on the board is visible.

For the pieces, you would add a second BackgroundFill to the background but this time use the color for the piece's side, and set the CorderRadii and Insets so that the piece is rounded and allows the background color behind it to be visible.

Then make the button's text be the label for the piece. See the API for this to make some sense.

You can have the constructor for the JavaFXXiangqiDisplay pass in the number of rows and columns of the board, so you don't have to use magic numbers.

Note that in traditional Xiangqi, the game is played on the lines of the grid instead of the centers of the square. You are welcome to change the game visuals it can appear as if each button is actually a corner of the grid, but that is not required.

1. **Xiangqi:** This type extends ChessGame. The type is almost the same as the EuropeanChess type with the following changes:
   1. The board has 10 rows and 9 columns (assuming you are playing north to south).
   2. You should prevent a move if it creates a *facing kings* situation: the two kings can "see" each other with no pieces of either team between them.

Just as with Indo-European chess, you are not allowed to make a move that places your king into check, but you do not have to implement this.

You are welcome to create a SwingXiangqiDisplay so that implements the Xiangqi visuals for the Swing application, but you are not required to do this.

**The New Pieces**

The rest of the classes are new chess pieces to add to your hierarchy. You should place these pieces into the same hierarchy as your chess pieces from Project 3. You are welcome to rearrange the hierarchy if needed, and you do not have to get the pieces from Project 3 working.

Just as with Indo-European chess, a piece cannot move onto a square with a piece of the same side. If it moves onto a square containing a piece of the opposing side, that opposing piece is captured and removed from the board.

In the descriptions below, I am assuming the north/south game. You should rotate the descriptions for an east/west game.

* XiangqiKingPiece: a king can move one square vertically or horizontally. A Xiangqi king cannot move out of the three center columns of the board nor out of the bottom (or top) three rows of the board. I recommend using "X" as the label for the Xiangqi king.
* GuardPiece: a guard can move one square diagonally. A guard cannot move out of the three center columns of the board nor out of the bottom (or top) three rows of the board. I recommend using "G" as the label for the guard.
* ElephantPiece: an elephant can move exactly two squares diagonally on each move. The elephant cannot jump over a piece so the first diagonal square of the move must be empty. Elephants cannot cross the center of the board (i.e. they are restricted to the top or bottom half of the rows assuming a north-south game). I recommend using "E" as the label for the elephant,
* HorsePiece: a horse/knight can move exactly two squares on each move. The first square must be a horizontal or vertical move, and that is followed by a diagonal move with the result being an "L" shaped move. The horse/knight cannot jump over pieces so the first square of the move must be vacant. I recommend using "H" as the label for the horse/knight.
* RookPiece: a rook can move any number of spaces vertically or horizontally. The rook cannot jump over pieces so all squares on the move (except possibly the last one) must be empty. I recommend using "R" as the label for the rook piece. Note that this is exactly the same piece as the Indo-European rook. I include it here for completeness.
* CannonPiece: a cannon can move any number of spaces vertically or horizontally. If the cannon is not capturing a piece, all the squares on its move must be empty. If the cannon is capturing, it must jump over exactly one piece (of either Side) on its move. I recommend using "C" as the label for the cannon piece.
* SoldierPiece: a soldier can move exactly one space vertically toward the opposite side of the board from its starting side. It cannot move backwards toward its own side. Once the pawn crosses the middle of the board, the pawn can also move one space horizontally (in either direction). I recommend using "S" as the label for the soldier piece.

**Extra Credit:**

If you decide to do the extra credit, *you must state in the Canvas comments what you did*. Don't make us hunt through your code to figure out what extra you did.

Make further improvements to the aesthetics and play of the game. Extra credit will be awarded to improvements that require coding challenge, a lot of work with the API, or creativity. **Note:** Do not look up online Xiangqi games and copy their code or style. This extra credit is for you to have some fun and be creative. It is an academic integrity offense if you copy code or ideas from online versions of the game.