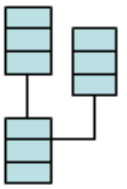
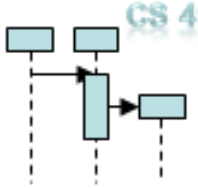



CS 414 Object Oriented Design

Spring 2018
Computer Science Department

Assignment A1

Colorado State University

Test Driven Development using Java and JUnit

DUE: 11:59PM, Tuesday 30 January 2018

30 points

1. Objectives

- Use Test Driven Development.
- Use JUnit to write unit tests while implementing a new program.

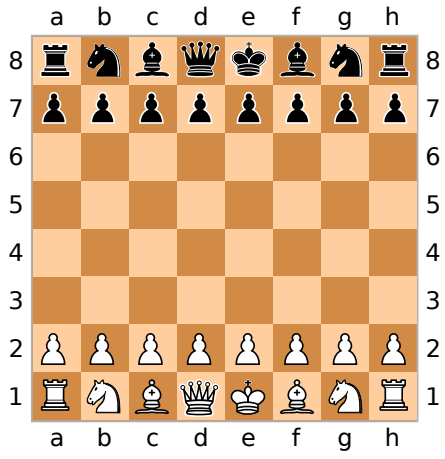
2. Tasks

Note that all the classes that you will implement below must be inside a package called `a1`. For the ease of understanding the program, we describe the tasks related to the program implementation before listing the tasks related to the unit tests. However, you will use test driven development, where you will write your test cases to specify the program.

2.1. Implementing the Modified Chess program

We are just creating a toy chess program that lets you set up pieces on a chessboard, move them, and find possible moves for a given piece in a certain position. We are not implementing Deep Blue. Because we are simplifying so many rules, we'll call this game, **M**odified ch**ESS** or **MESS**!

Chess is played on an 8 X 8 board where the initial placement of pieces is as shown in the following figure, taken from Wikipedia. Note the indexing scheme. The white king is on e1, and the black king is on e8. Familiarize yourself with the names of the pieces.



Each chess piece can move in a specific way. Details of the original rules are provided here. However, we will follow a simplified version in this assignment. Most importantly, we will ignore an important rule in chess: Moving any piece in a way that puts your own king in *check* is illegal. Since we don't know what *check* means, for us a move is legal if the piece we are moving has an empty square to move to or can capture (replace) an opponent's piece (including their king, although that's also illegal in real chess). The simplified rules are as follows:

- Assume that the queen and knight can't move.
- The king can only move one square horizontally, vertically, or diagonally. Assume that it cannot castle.
- A pawn in the initial position may move one or two squares vertically forward to an empty square but cannot leap over any piece. Subsequently it can move only one square vertically forward to an empty square. A pawn may also capture (replace)

an opponent's piece diagonally one square in front of it. Pawns can never move backwards. These are the only moves; we will not implement the En passant rule and will also not allow promotion to another piece if the pawn reaches the end of the column. If you don't know what these rules are, don't worry. We won't use them.

- A rook can move any number of squares horizontally or vertically, forward or backward, as long as it does not have to leap over other pieces. At the end of the move, it can occupy a previously empty square or capture (replace) an opponent's piece but it cannot replace another piece of the same player.
- A bishop can move any number of squares diagonally in any direction as long as it does not have to leap over other pieces. At the end of the move, it can occupy a previously empty square or capture (replace) an opponent's piece but it cannot replace another piece of the same player.

You will implement eight classes `ChessBoard`, `ChessPiece`, `Rook`, `Knight`, `Bishop`, `Queen`, `King`, and `Pawn`, and two exception classes called `IllegalPositionException` and `IllegalMoveException` that are described below.

2.1.1. Class `ChessBoard`

This class only stores the state of the board and its pieces for this assignment. In a real program, it would need to store more information (e.g., whoseTurn, etc). The board is represented by a 2-dimensional array of size 8X8.

Since the positions on a chess board are represented using a letter followed by a number, our array needs to represent the directions accordingly. We will make the following association: a=0, b=1, c=2, d=3, e=4, f=5, g=6, and h=7. In the initial position, the white king at e1 is at index [0][4]. The black queen at d8 is at index [7][3].

Implement the following attribute:

- `private ChessPiece[][] board;`

Implement the following constructor:

- The no-arg constructor `ChessBoard()` initializes the board to an 8X8 array with all empty squares. An empty square is null.

Implement the following methods:

- `public void initialize()`

  This method initializes the board to the standard chess opening state with indexing as shown in the figure. This method should use the constructors of the appropriate pieces, and call `placePiece` below to place the newly constructed pieces in the right position.

- `public ChessPiece getPiece(String position) throws IllegalPositionException`

  This method returns the chess piece at a given position. The position is represented as a two-character string (e.g., e8) as described above. The first letter is in lowercase (a..h) and the second letter is a digit (1..8). If the position is illegal because the string contains illegal characters or represents a position outside the board, the exception is thrown.

- `public boolean placePiece(ChessPiece piece, String position)`

  This method tries to place the given piece at a given position, and returns true if successful, and false if there is already a piece of the same player in the given position or the position was illegal for any of the two reasons mentioned in the description of `getPiece`. If an opponent's piece exists, that piece is captured. If successful, this method should call an appropriate method in the `ChessPiece` class (i.e., `setPosition`) to set the piece's position.

- `public void move(String fromPosition, String toPosition) throws IllegalMoveException`

  This method checks if moving the piece from the `fromPosition` to `toPosition` is a legal move. Legality is defined based on the rules described above in Section 2.1. If the move is legal, it executes the move, changing the value of the board as needed. Otherwise, the stated exception is thrown.

- `public String toString()`

  You must include the following `toString` method to help debug your program. We assume that `ChessPiece` has an appropriately implemented `toString` method, whose implementation is described below. The code for the method is here. **There is no need to write JUnit tests for this method.**

A barebones main method is provided for the `ChessBoard` class in case you want to try out the entire application at the end. Note that this method is not to be tested using JUnit.

```
public static void main(String[] args) {
    ChessBoard board = new ChessBoard();
    board.initialize();
    System.out.println(board);
    board.move("c2", "c4");
    System.out.println(board);
}
```

2.1.2. Abstract class `ChessPiece`

The **abstract** class `ChessPiece` is the parent class for all the actual chess pieces classes. This class keeps a reference to the board the piece is on (if any), stores the position where the piece is located, and the color.

Include the following enum type as shown below:

```
public enum Color {WHITE, BLACK}
```

Implement the following attribute:

- `protected ChessBoard board`; // the board it belongs to, default null
- `protected int row`; // the index of the horizontal rows 0..7
- `protected int column`; // the index of the vertical column 0..7
- `protected Color color`; // the color of the piece

Implement the following constructor:

- `public ChessPiece(ChessBoard board, Color color)`

  This constructor sets the board and color attributes.

Implement the following methods:

- `public Color getColor()`

  This method returns the color of the piece. There is no need for a `setColor` method because a piece cannot change color.

- `public String getPosition()`

  This method returns the position of the piece in the format single letter (a..h) followed by a single digit (1..8).

- `public void setPosition(String position) throws IllegalPositionException`

  This method sets the position of the piece to the appropriate row and column based on the argument, which in the format single letter (a..h) followed by a single digit (1..8). If the position is illegal for any of the two reasons mentioned earlier, throw the stated exception.

- `abstract public String toString();`

  This method will be implemented in the concrete subclasses corresponding to each chess piece. This method returns a `String` composed of a single character that corresponds to which piece it is. In the unicode character encoding scheme there are characters that represent each chess piece. Use one of the following characters:

```
 character     piece
----------   -----------
  "\u2654"   white king
  "\u2655"   white queen
  "\u2656"   white rook
  "\u2657"   white bishop
  "\u2658"   white knight
  "\u2659"   white pawn
  "\u265A"   black king
  "\u265B"   black queen
  "\u265C"   black rook
  "\u265D"   black bishop
  "\u265E"   black knight
  "\u265F"   black pawn
```

  It is important that your toString method return the right character string, as we will use it to display the board position when testing your code.

- `abstract public ArrayList<String> legalMoves();`

  This method will be implemented in the concrete subclasses corresponding to each chess piece. This method returns all the legal moves that piece can make based on the rules described above in the assignment. Each string in the `ArrayList` should be the position of a possible destination for the piece (in the same format described above). If there are multiple legal moves, the order of moves in the `ArrayList` does not matter. If there are no legal moves, return an empty `ArrayList`, i.e., the size should be zero.

2.1.3. Concrete classes `Rook`, `Knight`, `Bishop`, `Queen`, `King`, and `Pawn`

Implement each class to extend `ChessPiece` and override the methods `toString` and `legalMoves`. Note that the `legalMoves` methods of `Knight` and `Queen` return an empty `ArrayList<String>`.

2.1.4. Exception classes `IllegalPositionException` and `IllegalMoveException`

Implement each exception to extend `Exception`.

2.2. Implementation of JUnit test cases

Implement JUnit test classes, called `ChessBoardTest`, `RookTest`, `KnightTest`, `BishopTest`, `QueenTest`, `KingTest`, and `PawnTest` containing JUnit test cases for each of the classes listed above. Also implement a JUnit test suite class called `ChessSuite` that defines a main method and a test suite that includes all the above test classes.

3. Submission

Create a Jar file called `a1.jar` containing the 18 Java files. Doublecheck that you have included source files (i.e., `.java` files), not compiled class files (i.e., `.class` files). The folder hierarchy should not include `src`; it should start from `a1`. That is, the jar file should contain `a1/*.java`.

Submit the `a1.jar` file using **Assignment Submission** in Canvas.

4. Grading criteria

- Functionality of the implementation classes (implement methods correctly): 15 points.
- Functionality of test classes: 15 points.
- Packaging, creating the jar file, and submitting the the jar file correctly, using appropriate comments, following the coding style described in our slides: 5 points.

Up to 5 points may be deducted for not following the packaging structure or names used for the classes and the methods of each class.