

# Project 2: Design Patterns in Java

COMP 121, Spring 2021

Test case due: Tue, Mar 2 @ 11:59pm

Main project due: Mon, Mar 8 @ 11:59pm

## Introduction

In this project, you will implement a chess game simulator. Your simulator will take as input a file that describes the initial locations of pieces on the board and a series of moves of those pieces. Your simulator will then set up the board and execute those moves, checking that all the moves are legal and, at the end, printing the state of the game board.

The real goal of implementing this project, however, is to gain experience with design patterns. More specifically, as part of this project you will implement the following patterns: Singleton, Factory, External Dependency Injection, Observer, and (External) Iterator. Here is a [code skeleton](#) to get you started.

This project will also give you a little practice with the essential software engineering skill of solving programming problems by finding library code to do what you want. We'll give you some hints, but fewer than in the last project. So you'll need to spend some time searching on Google and poking around the [JDK 15 API](#). Most of the methods you want are probably in the following:

- [java.base](#)
  - [java.io](#)
  - [java.lang](#)
  - [java.util](#)

But, if you want to get fancy, you can use other parts of the API, e.g., [java.nio](#) or [java.util.regex](#). In this and all other projects, you are allowed to use any part of the JDK 15 API you like.

































This project doesn't break down into separate steps quite the way the previous one does, but it does break down into modules. So this writeup is organized as a description of the modules you'll need to write. The description is written in an order that's sensible for presentation, but you can implement the different modules in whatever order you want.

*Hint:* The hardest part of the project, in a technical sense, is writing the code to model the moves of all the chess pieces. The rest of the project has much less code, but you will need some time to understand all the design patterns we've crammed into the project. And yes, there are too many design patterns here, but hopefully not as many as this [factorial implementation that's been design patterned to death](#).

*Note:* You are free to add any helper methods, additional classes, etc, that you'd like to for this project. Just be sure to upload a complete, working program to Gradescope.

## Overview

Recall that the game of chess is played on an eight-by-eight board with the following initial layout:

	a	b	c	d	e	f	g	h
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

Using standard notation, we've labeled the chess board's rows 1-8 and the columns a-h, and using slightly non-standard notation we're describing each piece with two characters: the color (black (b) or white (w)) and the kind (king (k), queen (q), knight (n), bishop (b), rook (r), and pawn (p)).

Your simulator will be run via the command

```
java Chess layout moves
```

where `layout` is the name of a file describing the initial locations of pieces on the chessboard, and `moves` is the name of a file describing a sequence of moves. We've given you one example each of the layout and move files, `layout1` and `moves1`, respectively, where `layout1` contains the standard setup of chess pieces on the board. For ideas of other ways you could lay out chess pieces initially, search the web for *chess puzzles* or [chess problems](#).

## Parsing Layout and Move Files

You will need to write code in `Chess.java` that (a) reads data from the layout file and sets up the board and then (b) plays the sequence of moves in the moves files. Of course, you can create whatever additional methods and classes you need.

If you look in `Chess.java`, you'll see a `main` method of the usual type. You'll want to add your code toward the bottom of this method. Notice that the names of the files you need to read are given on the command line, which means they will be stored in the `args` parameter of `main`. The layout file name will be in `args[0]`, and the moves file name will be in `args[1]`.

To write this code, you'll need to figure out how to open and read files in Java. We won't tell you how to do this, but if you search the web, look in the JDK documentation, and/or look through the Java textbooks linked from the class web page (see the [resources](#) at the bottom of the page), you can figure it out.

There are actually a few different ways to access files in Java. You just have to find one that works. Please don't ask your fellow students for help with this. Pretend that you're working at a company, you need to work with files in Java, and none of your colleagues knows how to do it. This kind of situation happens often, so now is a good time to practice finding the information on your own!

As you process the layout and moves files, you **must enforce the following rules about the file**. If any of the rules is violated, throw an exception (any exception will do):

- **Layouts**

- A layout file is a possibly empty sequence of lines
- If a line begins (0th character) with a `#`, that line is a comment and should be ignored
- Otherwise, a line should have the form `xn=cp`, indicating that position `xn` starts out with piece `cp`, where
  - `x` is a column (`a-h`)
  - `n` is a row (`1-8`)
  - `c` is a color (`b` or `w`)
  - `p` is a piece kind (`k`, `q`, `n`, `b`, `r`, or `p`)
- All files we supply will not have any extra whitespace at the beginning or end of a line, so you can allow or disallow extra whitespace as you like.
- A layout file may not place two pieces in the same location.

- **Moves**

- A moves file is a possibly empty sequence of lines
- If a line begins (0th character) with a `#`, that line is a comment and should be ignored
- Otherwise, a line should have the form `xn-ym`, indicating that the piece at position `xn` moves to position `ym`, where
  - `x` and `y` are columns (`a-h`)
  - `n` and `m` are rows (`1-8`)
- All files we supply will not have any extra whitespace at the beginning or end of a line.
- All moves must be valid according to a subset of the chess rules we discuss below.

## Chess Pieces

There are six classes representing different kinds of chess pieces: `King`, `Queen`, `Knight`, `Bishop`, `Rook`, and `Pawn`. Each of these classes is a subclass of `Piece`, which is a class rather than an interface because it has some code in it. You need to add code to all seven of these classes (the six chess piece classes and their superclass) to match the following design:

- Each chess piece has a color, which is either `Color.BLACK` or `Color.WHITE`. This typesafe enum is defined in `Color.java`. The color is passed to the piece's constructor. You should implement the `color()` method in `Piece`, rather than in the subclasses. To do so, you'll probably want to add a field to `Piece`.
- Each piece's `toString()` method should return the piece name in the form it appears in the layout file, e.g., a black king should return `bk`, a white pawn should return `wp`.
- Each piece has a method `List<String> moves(Board b, String loc)` that returns a list containing the locations the piece could legally move to starting from the position `loc`, taking the positions of other pieces on the board into account. (However, there should **not** be any check by this method that the piece is actually at position `loc`.) The `loc` is given in the same notation as the layout file, e.g., `"a3"` is a location. Each piece has a different set of moves. We will use the following movement rules only, which are slightly simplified from the [rules of chess](#):

- The king can move to one adjacent space in any direction (left/right, up/down, or diagonal) and may or may not capture a piece at the end of its move. We will not worry about [check](#) or [checkmate](#), and we will not implement [castling](#).
- The queen can move any number of vacant spaces in any direction, left/right, up/down, or diagonal. The queen can stop moving before capturing a piece, or can capture a piece at the end of the move.
- The knight moves in an L-shape: two squares horizontally and then one square vertically, or two squares vertically and one square horizontally. See the [rules of chess wikipedia page](#) for a picture. The knight jumps over other pieces. It can either land on an unoccupied space, or it can capture a piece by landing on an occupied space.
- The bishop is like a queen that can only move diagonally. It moves any number of vacant spaces along any diagonal, and may or may not capture a piece at the end of its move.
- The rook is like a queen that can only move horizontally or vertically. It moves any number of vacant spaces either horizontally or vertically, and may or may not capture a piece at the end of its move. We will not implement [castling](#).
- The pawn can move one space vertically forward, only toward the opponent's side of the board, i.e., a black pawn can move toward row 1, and a white pawn can move toward row 8. However, the pawn can move two spaces forward if it is in its home row (row 2 for white, row 7 for black) and the intervening space is vacant. The preceding two moves can only be made to vacant spaces; they cannot capture pieces. A pawn can capture an opponent's piece by moving one square diagonally toward the opponent. It cannot move diagonally if it is not capturing a piece. We will not implement [en passant](#) or [pawn promotion](#).
- Pieces can only capture the opponent's pieces (i.e., of the opposite color).

Aside: Notice that `Piece` is an `abstract` class, with two `abstract` methods. This means that `Piece` is somewhere between an interface and a class: It has some methods (and possibly fields) that are inherited by subclasses, and it has some methods that must be implemented by subclasses.

## Chess Piece Factories

You will need to implement the factory pattern to create chess pieces. Here's how the design should work.

- Method `Piece createPiece(String name)` should create an instance of the piece described by its argument in the same format as the layout file, e.g., `Piece.createPiece("br")` should return a black rook.
- For each subclass `C` of `Piece`, there is a corresponding class `CFactory` that has two methods (which we've written for you): `char symbol()`, which returns the piece's one-letter symbol (e.g., 'p' for Pawn) and `Piece create(Color c)`, which returns a new instance of the corresponding piece with the given color, e.g., `PieceFactory p = new PawnFactory(); p.create(Color.WHITE)` returns a new white pawn.
- (You do need to implement the constructors of the underlying chess piece classes, e.g., you need to implement the `Pawn(Color c)` constructor.)
- In `Chess.java`, there is a sequence of calls of the form `Piece.registerPiece(new PawnFactory())`, which makes a new instance of the factory object and passes it to `Piece.registerPiece`. You must implement the `registerPiece` method to store information about the piece (probably in a map) so that the `Piece.createPiece` method can look in that map to find out how to create such a piece.

Whew, that's pretty complicated! And probably unnecessary for chess. But, it does have the nice (?) feature that it would be easy to create new kinds of chess pieces and add them to the board without having to change too much code.

## The Chess Board

Next up, you need to write code for class `Board`, which stores the locations of the pieces, among other things. To give you practice with another pattern, we've decided that the `Board` should be a singleton class. Hence you need to implement a method `theBoard` that returns the singleton `Board` instance.

The board itself stores the piece locations in field `pieces`, which is a two-dimensional array of `Piece`. Notice that, very often, you will need to convert coordinates like "a3" into an access into this array. It's up to you how you do this. You could implement a full-scale adapter pattern. You could write a utility method or two to convert. Or you could duplicate code all over the place. Only the last choice is not recommended!

You should implement four *mutators* for `Board`:

- `Piece getPiece(String loc)` returns the piece at the given location (in the usual notation, e.g., "a3"), or `null` if the location is empty. It should raise an exception (any exception) if `loc` is invalid.
- `void addPiece(Piece p, String loc)` adds piece `p` to the board at the given location. It should raise an exception (any exception) if the location is already occupied or is invalid.
- `void movePiece(String from, String to)` moves the piece at location `from` to location `to`. This method should check that there is a piece at `from` (and throw an exception if not), and it should check that the move is valid for that piece. If the move is valid, then location `from` becomes vacant and the piece is placed at position `to`. It is possible there was another piece at `to`, in which case it has been captured; see below. If the move is invalid or `from` or `to` are invalid locations, `movePiece` should raise an exception (any exception).



- `void clear()` should remove all pieces from the board.

## The Chess Board Observer

The `Board` class also supports the observer pattern, so that listeners can be called back when key events happen. More precisely, a `BoardListener` is an observer that implements two methods: `void onMove(String from, String to, Piece p)`, which is called whenever a move is made on a board; and `void onCapture(Piece attacker, Piece captured)`, which is called whenever a piece is captured. If a piece is captured, there should be a call to `onMove` first and then a call to `onCapture`.

You must implement the observer pattern by modifying `Board.java` as follows:

- `void registerListener(BoardListener bl)` should add a listener that should subsequently be called at the appropriate times. (So you will need to modify `movePiece` also.) There can be any number of listeners at any time (zero or more). We will not test the case of the same listener being registered more than once.
- `void removeListener(BoardListener bl)` removes a listener so it will subsequently not be notified of events. Your code can behave however you like at an attempt to remove a listener that wasn't registered, or an attempt to remove a listener that was registered twice.
- `void removeAllListeners()` removes all listeners.

In `Chess.java`, the `main` method registers a simple observer that listens for updates to the board and prints out what happened. This could be helpful for debugging.

## The Chess Board Iterator

But wait, there's more! (We promise this is the last part...) The `Board` class also supports external iteration. The `BoardExternalIterator` interface (sorry for the terribly long name) defines a method `void visit(String loc, Piece p)`. You must implement the `iterate` method of `Board` so that it takes a `BoardExternalIterator` and, for every *square* on the board, calls the external iterator's `visit` method, passing that square's location and the piece at the location, or `null` if the location is vacant. The iteration order is up to you, but it must visit every square of the board. (So, `visit` will be called 64 times.)

## Testing Your Code

To help you test your code, we've given you a file `Test.java` with another main method that, like the previous project, runs a test case. You can invoke the tests with

```
java -ea Test
```

We've included one sample test case. (Note this test case is a little advanced; you'll have to do a fair amount of work before it passes.)

As part of this project, **you must write a test case for one `Piece`'s `moves` method and share it with the class on campuswire**. More specifically, your test case should set up a board by calling `addPiece` some number of times and then call one appropriate `moves` method, testing the result. **Be sure your test case ignores the order of moves in the returned list** because different implementations might return moves in different orders.

Of course, you will want to write many other test cases for different parts of the project!

## What to turn in

Put all your code in the [code skeleton](#) we have supplied, and upload your solution using Gradescope.