Definitely Not A Chess Clone: Code Structure

Directory Structure:

All the files related to the game's code are in ChessGame/src/Game/src which acts as the source folder for our Java project. The Java files for our classes are in the default package in this source folder, and the images required for our game are in the "Icons" folder.

The major classes in our game are:

- `Board`
- `View`
- `PieceHandler`
- `Piece`
- `PieceButton`

Board:

The `Board`, an observable object, stores the current state of the chess board using a 2D array of `PieceButton` objects called `boardModel`. The `Board` also tracks the current player and the piece selected by the player.

The primary method of the `Board` class is the initializer, which sets the `boardModel` to have the starting placement of `PieceButtons` which in turn track the various pieces in the game.

The `Board` class also has a helper method called `defaultPiece` which given an x and y coordinate, returns the piece that is in that location at the start of a game.

Additionally, the `Board` class has the `switchTurn()` method, which changes the current active player to whichever player it is not currently, and modifies all the `PieceButtons` in the `boardModel` so that only those belonging to the new active player can be selected.

Moreover, whenever the Board is changed it notifies its observers through the `notifyObservers()` method. Since the `View` observes the `Board`, it changes itself appropriately upon receiving a notification with its `update()` method.

Lastly, the `Board` class has the `reset()` method, which returns the `boardModel` to the state at the start of the game.

PieceButton:

Each `PieceButton` stores the `Piece` currently on it and the button's position on the `Board`. Its main methods are `select` and `setPiece`.

The `select` method allows selecting and deselecting a button based on whether it was previously selected.

The method `setPiece` allows changing the `Piece` placed on the button. Both `select` and `setPiece` change the image of the `PieceButton` to reflect new changes using the `changeImage` method. The ID of each button is set to be the `type` of Piece placed on it (returned by the `getType()` method). This ID helps when the `PieceHandlerCaller` class to call the appropriate `PieceHandler` when a player clicks a `PieceButton`.

PieceHandler:

Each `Piece` has a corresponding `PieceHandler` class. So, we have 6 handler classes in addition to the superclass, `PieceHandler`. The primary function of each of these classes is the `handle` method. This method responds to the event in which a `PieceButton` is clicked. There are three cases that we consider here:

1) A `PieceButton` was not selected before - so the player is selecting a `Piece` to move. The method selects all the `PieceButton` objects that the `Piece` can move too
2) The player clicked the same `PieceButton` twice. In this case, the handler deselects the `PieceButton`.
3) A different `PieceButton` was selected before. In this case, this `PieceButton` was one of the potential spots to move. So, the handler sets the `Piece` on this button to be the `Piece` that was previously selected. If this `PieceButton` was initially empty, then the `Piece` simply moved. If there was a `Piece` on this button, then the `Piece` was attacked.

Case 3 is special. In the event of a `King` getting attacked, the game must end. So our `PieceHandler` classes keep track of which `Piece` a player attacks. If a `King` is attacked we call `isCheckmated()` to set `King`'s checkmated attribute to be true. Setting this attribute to be true helps the `View` to identify when the game has ended. We also call `Board` method `notifyObservers()` from the `PieceHandler` class whenever Case 3 occurs, because in this case the `Board` is changed, and the `View` needs to update the current player or end the game accordingly.

View:

The `View` class is responsible for setting up the screen by adding the `Board` as a panel of buttons that we called `chessPanel`. The `View` also adds labels to update the players on whose turn it is.

The primary method in `View` is the `update()` method. The `View` acts as an *observer* to the `Board`. So whenever `Board` changes, it calls its `notifyObservers()` method which invokes `View` method `update()`. The `update()` method would switch players' turns if a `King` was not checkmated. If a `King` was checkmated, it opens a popup window that gives the players

the option to either play again or quit. If the players click play again, the `View` calls the `chessPanel` method `reset()` which resets the button's panel to the `Board` default state.

The `DefinitelyNotAChessClone` class launches the View.

Extending the game:

You can extend the game by adding another `Piece` to it. To do so, you need to do the following:

1) Add a class that extends the `Piece` class
2) Implement the abstract `toString()` and `getType()` methods required by the `Piece` class. The string-representation (returned by `toString()`) for white and black pieces need to be different!
3) Add 8 images with names of the following type:
    a) your white piece's string-representation + "UW.jpg" (your white piece, unselected on a white background)
    b) your white piece's string-representation + "SW.jpg" (your white piece, selected on a white background)
    c) Your white piece's string-representation + "UB.jpg" (your white piece, unselected on a black background)
    d) Your white piece's string-representation + "SB.jpg" (your white piece, selected on a black background)
    e) your black piece's string-representation + "UW.jpg" (your black piece, unselected on a white background)
    f) your black piece's string-representation + "SW.jpg" (your black piece, selected on a white background)
    g) Your black piece's string-representation + "UB.jpg" (your black piece, unselected on a black background)
    h) Your black piece's string-representation + "SB.jpg" (your black piece, selected on a black background)

4) Add a `PieceHandler` that selects potential buttons for your `Piece` to move to. If another Piece was clicked before the `selectedPiece` attribute of the `Board` will not be `null`

5) Map the `PieceHandler` to type of the `Piece` in hashmap `pieceHandlers` attribute of the `PieceHandlerCaller`

6) Return an instance of the `Piece` according to its default location in the `getDefaultPiece()` method of the `Board`