# SWE2410-111 – Lab 1: Checkers Pieces – Jawadul Chowdhury – 9/8/2024

## Objectives

State your objectives for the lab here: what is the lab supposed to do in your own words. It should be a short paragraph

This lab is a checkers lab, where functionalities of the checkers board must be implemented. Features that have already been implemented are the movement of the pieces, repositioning of the pieces on the square, as well as the capturing of pieces by pieces of another color. The checker's board has been created using Java & JavaFX, hence when the program is run, a GUI application opens. For this lab, functionality needs to be added for kings, which means that when pieces reach the end of either side of the board, they need to turn into kings, and have the functionalities that a king piece would have on the checker's board.

## Requirements

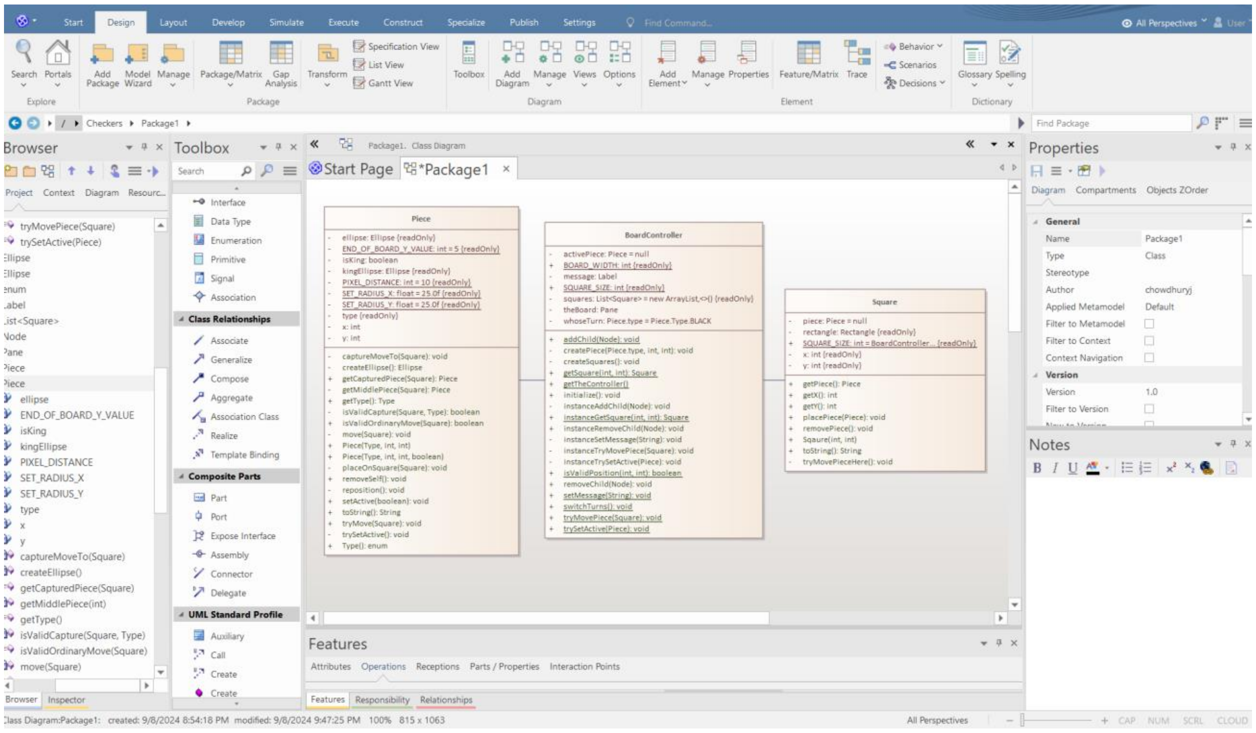
Capture the detailed requirements here. Look at the grading rubrics, the lab assignment instructions to make sure you capture what you must do. Also, put in any requirements that you have added.

The detailed requirement for the lab is listed as follows:

- Adding a boolean attribute to ensure that when a piece reaches the end of either side of the board, the piece now becomes a king, and modifying the code to identify pieces that have reached the end of the board, and their status is updated to become a king.
- Ensuring that within the JavaFX application, when a piece reaches the end of the board, it visually becomes a king by having another ellipse stacked on top of it. It's behavior when clicked should be of similar behavior when a piece is clicked upon.
- When the king ellipses are constructed on the board, the program must ensure that they have the functionalities of what a king piece should have. This functionality includes things such as moving forwards or backwards diagonally on the board and ensuring that they're positioned properly on each square.
- Other functionalities to be added for king ellipses is ensuring that they can overtake other pieces on the board that they happen to encounter, as well as ensure that king pieces can overtake other king pieces as well.
- A requirement that has been added which is outside of the lab is ensuring that pieces of the same color don't happen to overtake each other. It is during testing that it was found that red pieces can overtake other red pieces, which should not be possible.

## Design

Include appropriate design elements, such as UML diagrams which form the mental model of how it is intended to achieve the objectives. Sequence diagrams are also accepted.


Piece
- ellipse: Ellipse {readOnly}
- END_OF_BOARD_Y_VALUE: int = 5 {readOnly}
- isKing: boolean
- kingEllipse: Ellipse {readOnly}
- PIXEL_DISTANCE: int = 10 {readOnly}
- SET_RADIUS_X: float = 25.0f {readOnly}
- SET_RADIUS_Y: float = 25.0f {readOnly}
- type {readOnly}
- x: int
- y: int
- captureMoveTo(Square): void
- createEllipse(): Ellipse
+ getCapturedPiece(Square): Piece
- getMiddlePiece(Square): Piece
+ getType(): Type
- isValidCapture(Square, Type): boolean
+ isValidOrdinaryMove(Square): boolean
- move(Square): void
+ Piece(Type, int, int)
+ Piece(Type, int, int, boolean)
- placeOnSquare(Square): void
+ removeSelf(): void
- reposition(): void
+ setActive(boolean): void
+ toString(): String
+ tryMove(Square): void
- trySetActive(): void
+ Type(): enum
BoardController
- activePiece: Piece = null
+ BOARD_WIDTH: int {readOnly}
- message: Label
+ SQUARE_SIZE: int {readOnly}
- squares: List<Square> = new ArrayList<>() {readOnly}
- theBoard: Pane
- whoseTurn: Piece.type = Piece.Type.BLACK
+ addChild(Node): void
- createPiece(Piece.type, int, int): void
- createSquares(): void
+ getSquare(int, int): Square
+ getTheController()
+ initialize(): void
- instanceAddChild(Node): void
+ instanceGetSquare(int, int): Square
+ instanceRemoveChild(Node): void
- instanceSetMessage(String): void
- instanceTryMovePiece(Square): void
- instanceTrySetActive(Piece): void
+ isValidPosition(int, int): boolean
+ removeChild(Node): void
+ setMessage(String): void
+ switchTurns(): void
+ tryMovePiece(Square): void
+ trySetActive(Piece): void
Square
- piece: Piece = null
- rectangle: Rectangle {readOnly}
+ SQUARE_SIZE: int = BoardController... {readOnly}
- x: int {readOnly}
- y: int {readOnly}
+ getPiece(): Piece
+ getX(): int
+ getY(): int
+ placePiece(Piece): void
+ removePiece(): void
+ Sqaure(int, int)
+ toString(): String
- tryMovePieceHere(): void
Package1 : Class Diagram
Start Page
*Package1
Browser
Toolbox
Properties
Features
Notes

## Test Run & Results

What tests did you run to support the requirements of the lab? What was your first pass rate. Demonstrate the outcomes of the tests by providing appropriate screen shots and a short explanation of what the test proves.

With regards to the test runs, all testing was done on the application itself. The tests are conducted to ensure the following:

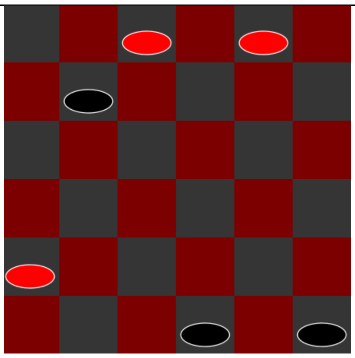
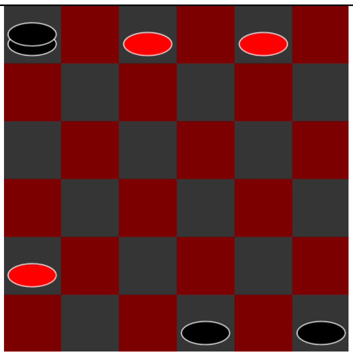
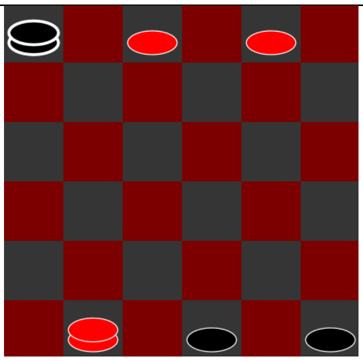
1. The pieces turn into kings when they reach the end of the either side of the board, and they should have a ellipse stacked on top of it to ensure that it visually represents a king

2. When the king pieces are selected, it should be ensured that both the pieces are highlighted so that the user knows that the king piece is now active and can be moved around

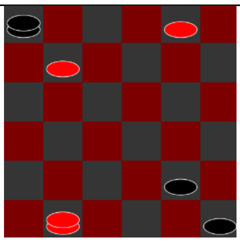
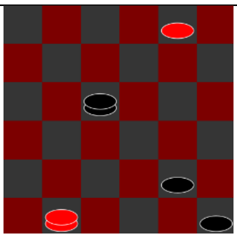
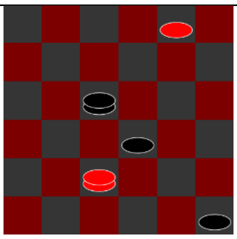
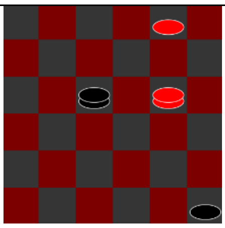
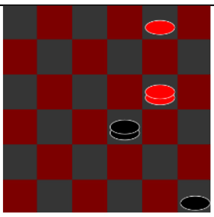
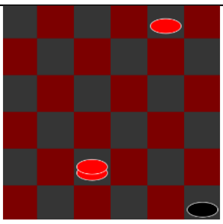
3. The king pieces should be able to move diagonally in any direction, be able to capture other pieces of the opposing color, as well as capture other king pieces of the other color

4. The pieces should have added functionality to ensure that they are able to capture any king pieces in the diagonal direction that they were intended to move, allowing for capturing

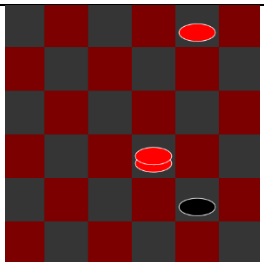
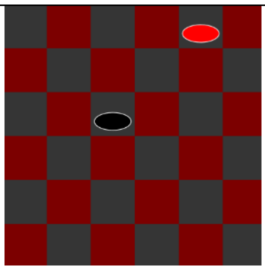
## Test Runs for Test Run Requirements 1 and 2

| Picture | Description |
|---|---|
|  | *The black piece is one diagonal move away from becoming a king* |
|  | *After the piece has moved in the diagonal direction, the piece has become a king with two ellipses* |
|  | *After the king piece is created, it can be selected, as shown, where the ellipses are highlighted* |

## Test Runs for Test Run Requirements 3

| Picture | Description |
|---|---|
|  | *The red piece is ready to be captured by the black king* |
|  | *The black king has captured the red piece* |
|  | *The black piece is ready to be captured by the red king* |
|  | *The black piece has been captured by the red king* |
|  | *The black king is ready to be captured by the red king* |
|  | *The black king is captured by the red king* |

## Test Runs for Test Run Requirements 4

| Picture | Description |
|---|---|
|  | *The red king is ready to be captured by the black piece* |
|  | *The red king has been captured by the black piece* |

## Discussion

Conclusions, including what you learned from the tests that you ran.

From running the application, it can be evident enough to see what the program is doing correctly, and what the program hasn't been doing correctly. The movements of the pieces, and their transformations as well as their captures can show which part of the code needs to be modified and which part doesn't need to be. These are listed as follows:

1. Test 1: when the red / black pieces are moving towards their end of the board, the pieces have to be transformed into a double ellipse, which represents a king. Since that didn't happen, the overloaded constructor had to be modified to called the createEllipse() method twice, and use .setVisible(false) to ensure the second ellipse isn't seen.

2. Test 2: once that has been solved, it was then noticed that the king pieces weren't positioned correctly on the board. To fix it, the reposition() method was modified to add two extra lines of code which used the .setLayout() method to ensure they were placed properly on the square itself. Next, to ensure that the king pieces were shown to be selected visibly, the setActive() method was updated by adding the .setStroke() for both the ellipses.

3. Test 3: next, it was then observed that king pieces were not able to overtake other king pieces. As a result, the getCapturedPiece() method was modified, where if the piece to be captured was the king, then the function would be modified to allow king pieces to capture other pieces.

4. Test 4: upon further testing, it was found that pieces of the same color were able to capture each other. To deal with that, the isValidCapture() method was modified to accept an extra parameter, named Type capturer, which prevented pieces of the same color from capturing each other.

From these tests, a lot of valuable fixes were made to the code. This allowed for the program to be more robust and prevent it from making any kind of mistakes. This has also meant that considerable amount of time was spent understanding how the code was written and how the code works, in order to add changes to the code.