CHESS

DANIEL COUTURIER, RUTH GARCIA, TARIQ FIEVRE, BRADEN CORKUM

What is chess

- A board game between 2 players
- Players take turns moving pieces
- Different pieces have specific allowed moves
- The game ends when opponents capture a specific piece called the king

KING
QUEEN
BISHOP
KNIGHT
ROOK
PAWN
KING
QUEEN
BISHOP
KNIGHT
ROOK
PAWN

Why this topic

Reservation app - Too complex with a client - server architecture

Schedule app - Too simple with no room for expansion

Game - Simple logic with room for expansion, multiple components that can be developed separately. Chess is a game with reasonable complexity with well defined requirements.

PROCESS MODEL

Process Model Taken- Incremental Model

- For this project we followed the incremental model. Our first iteration of the project, we focused on implementing basic graphics and piece movements with placeholders for game logic.
- Our second iteration focused on the logic controller and chess board ensuring accurate interactions between components and transitions between components.
- For our last increment we are working on implementing edge case conditions and testing, such as program bugs, chess win conditions, and unique tactics in chess, such as “castling”.
- We chose to use the incremental model because of the small team size, low complexity of the project, and short deadlines.
- The incremental model works well for chess because of the well defined rules of the game.

Roles

We divided the project into four parts:

- Braden handled the chess pieces
- Daniel handled the graphics
- Ruth handled the chess board
- Tariq handled the logic controller

REQUIREMENTS

Different User Story examples for our chess game

- As a chess fanatic, I would like to have a portable games of chess so that I may play with people around me.
- I want it to be offline so I do not have to worry about losing connection with my opponent
- I want it to keep track of whos turn it is so we do not move pieces out of turn
- I want to be able to go back to the beginning of the game, so that we may review our moves.
- I want it to be beginner friendly, so that beginners do not have to worry about what piece moves where.

Requirements

a. Players are only allowed to make legal moves*
b. Two players can alternate moving their pieces
c. Game should end when player is in checkmate or stalemate
d. A player must make a move to get out of check when in check
e. Pawns should be promoted upon reaching the end of the board
f. A king and rook may be swapped if neither has moved yet
g. A pawn can take two steps on its first move
h. A pawn can take another pawn if it was skipped by taking 2 steps
i. Players should be able to see whose turn it is
j. Players should be able to see valid moves that are highlighted when clicking on a piece
k. Pieces should be set to the correct positions at the start of the game.
l. Players should be able to restart the game at the end of a match

Primary Use Case

Use Case Name:Two Player Offline Game

Goal/Intent: Two players play a game of chess

Stakeholders: Developers, Players

Primary users: Players 1 and 2

Preconditions:

- Chess app is installed
- There are two players present

Use Case continued

Main success:

- Players start a game
- Take turns moving and taking chess pieces
- Game ends with checkmate

Alternative scenario:

- Game ends in stalemate

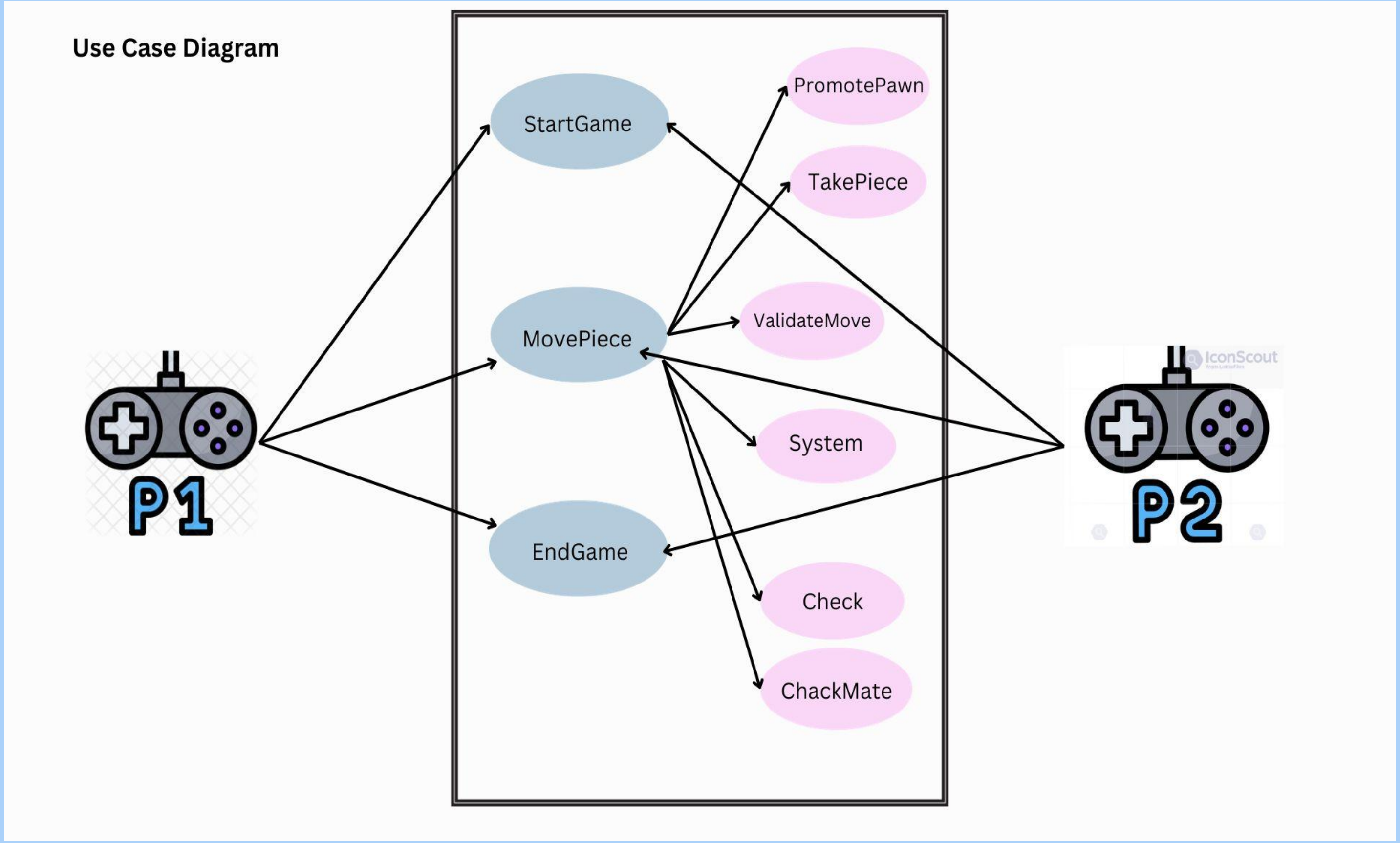
Exception :

- One player quits

Open Issues

- Performance
- Accessibility

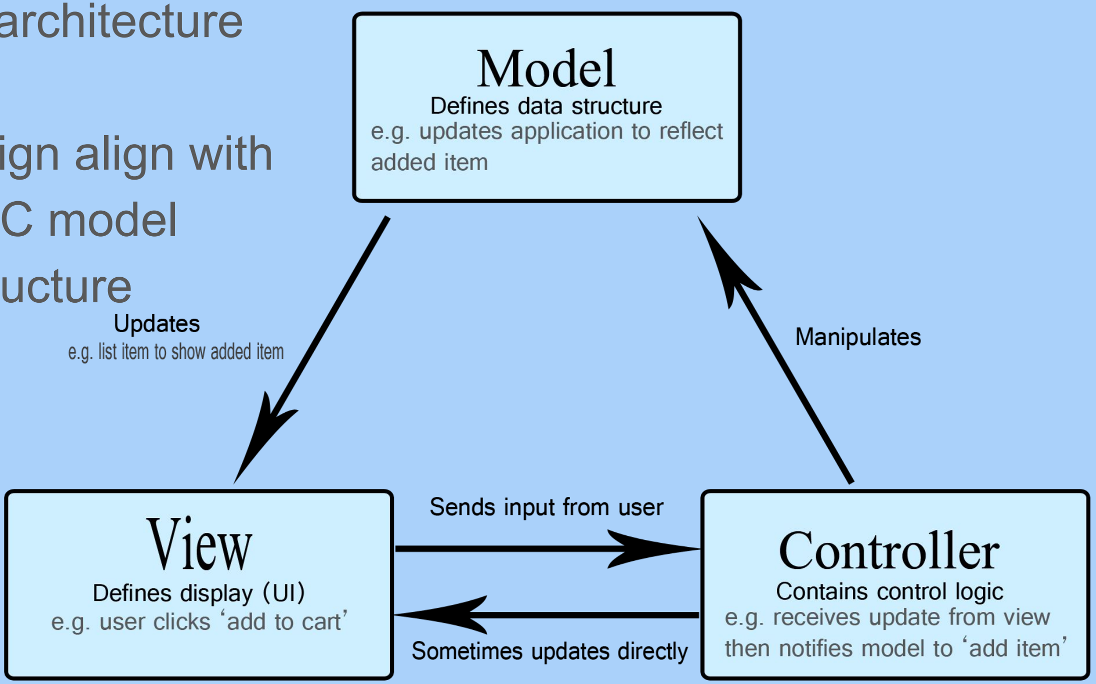
Primary use case - 2 player offline game



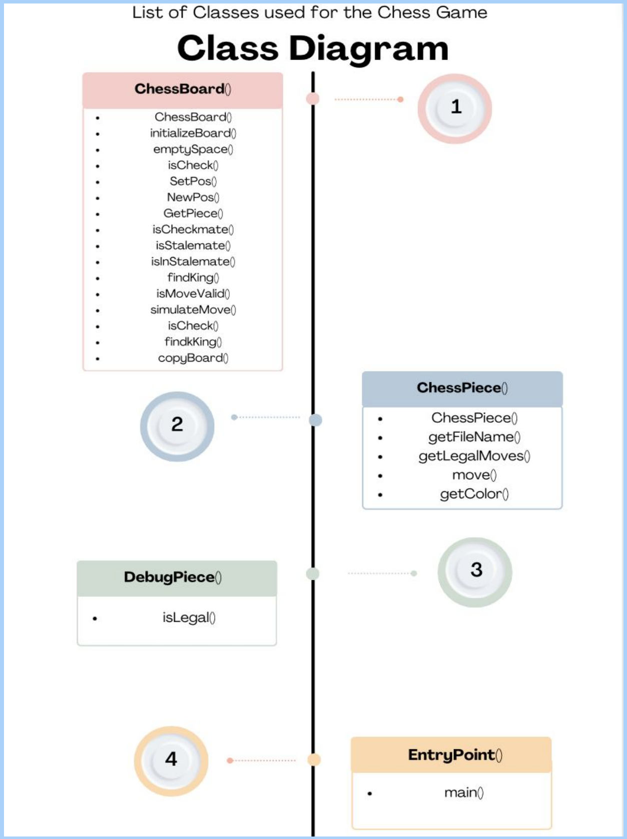
DESIGN

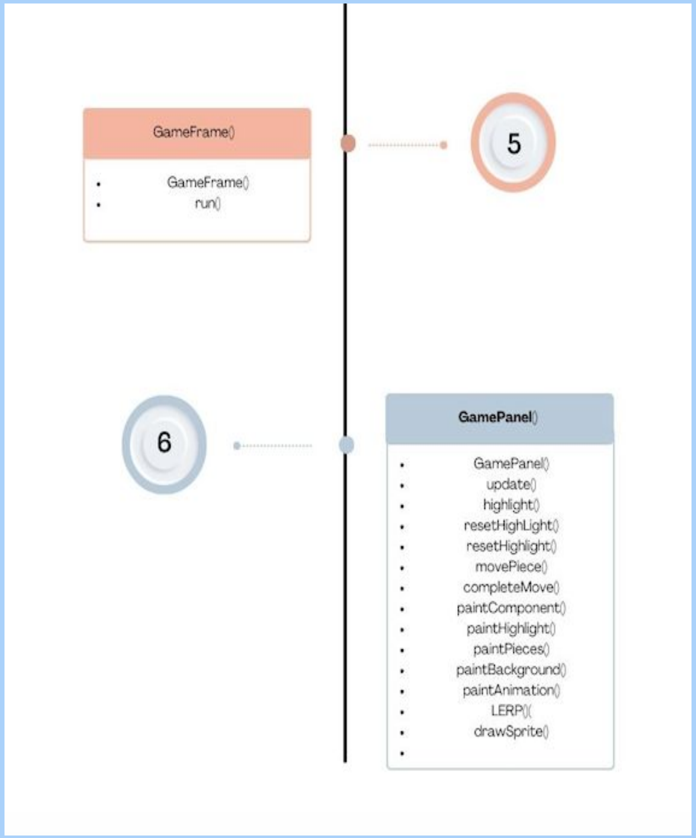
Component Diagram

- We used an object oriented architecture
- MVC architectural pattern
- The main classes of our design align with a specific purpose in the MVC model
- Other classes act only as structure or subroutines.

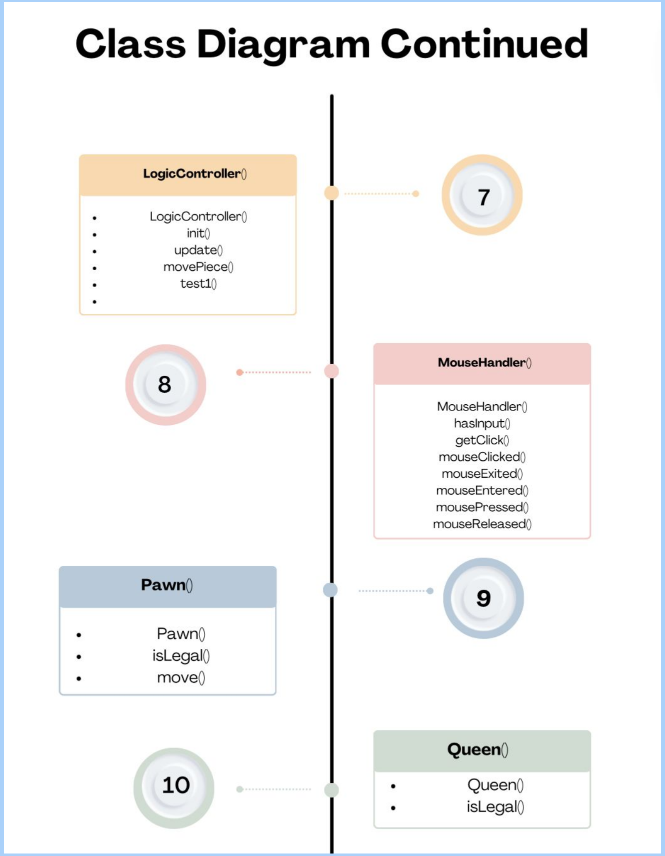

Model
Defines data structure
e.g. updates application to reflect added item
Updates
e.g. list item to show added item
Manipulates
View
Defines display (UI)
e.g. user clicks 'add to cart'
Sends input from user
Sometimes updates directly
Controller
Contains control logic
e.g. receives update from view then notifies model to 'add item'

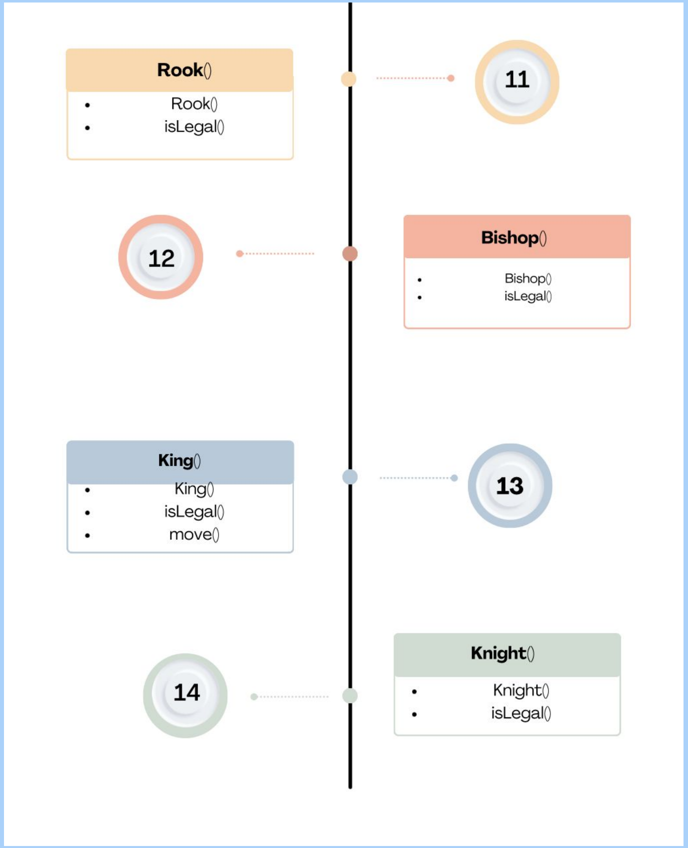
Class Diagram



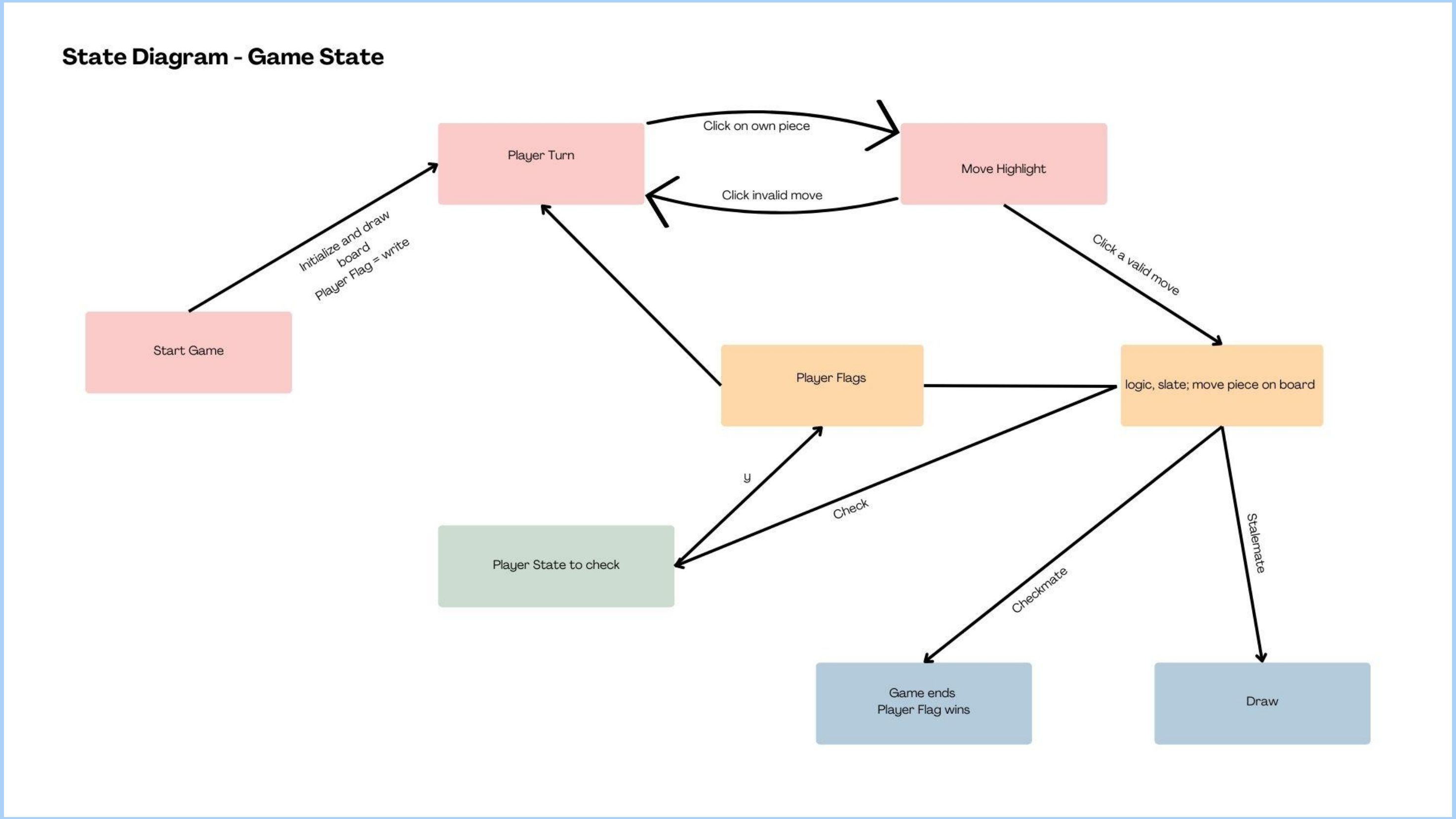


Class Diagrams Continued

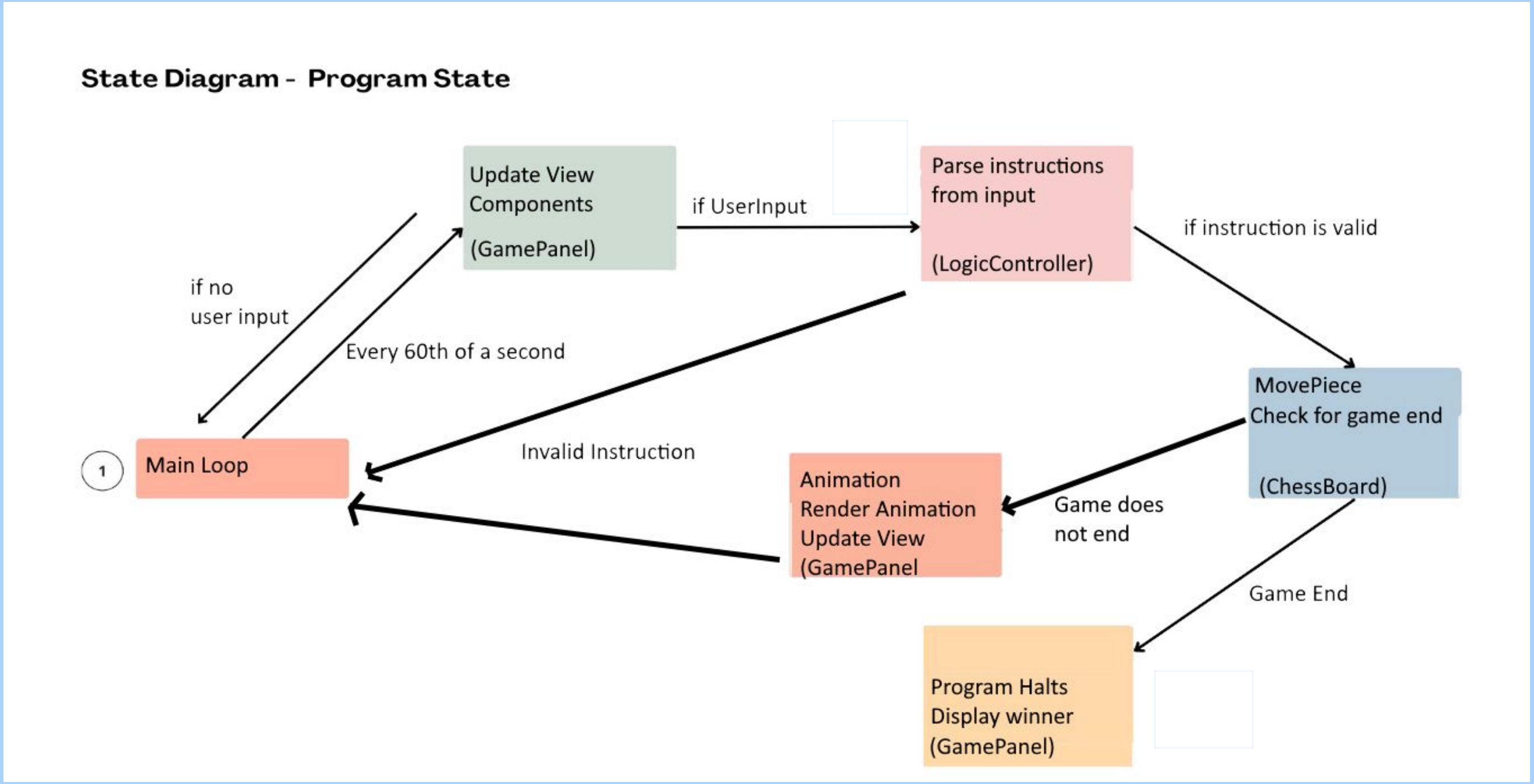




State Diagram - game state



State Diagram - program state


State Diagram - Program State
Update View Components (GamePanel)
if UserInput
Parse instructions from input (LogicController)
if instruction is valid
if no user input
Every 60th of a second
MovePiece Check for game end (ChessBoard)
1
Main Loop
Invalid Instruction
Animation Render Animation Update View (GamePanel
Game does not end
Game End
Program Halts Display winner (GamePanel)

DEVELOPMENT

EntryPoint.java and Game.java

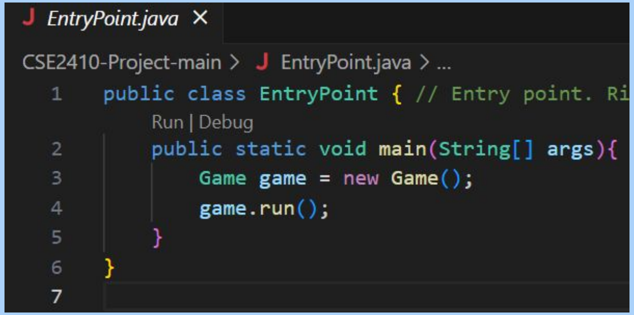
Main program

Initialize all objects and dependencies

Memory addresses for component communication

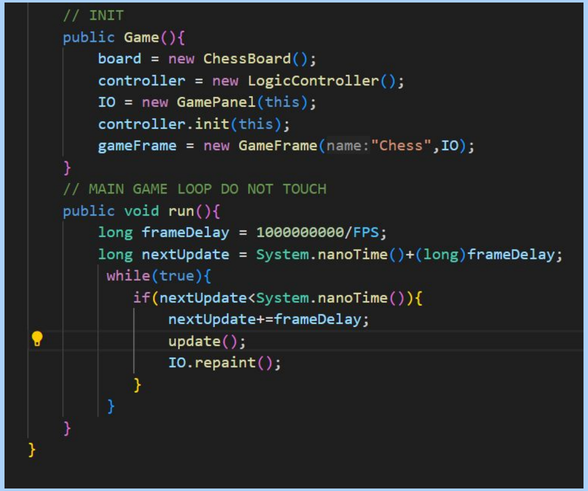
Main loop of program

Plan to incorporate into a menu for replayability

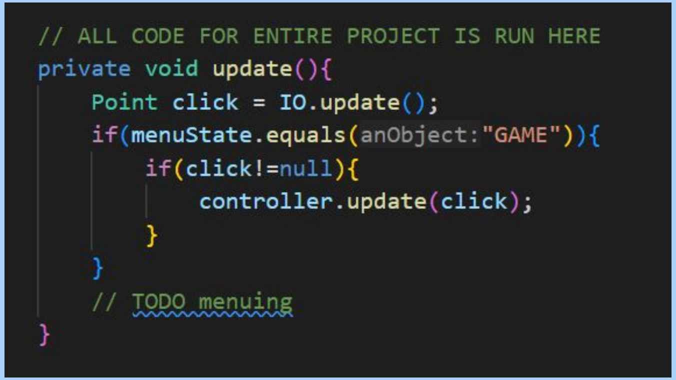

EntryPoint.java ×

CSE2410-Project-main > EntryPoint.java > ...

```java
public class EntryPoint { // Entry point. Ri
    Run | Debug
    public static void main(String[] args){
        Game game = new Game();
        game.run();
    }
}
```



```java
    // INIT
    public Game(){
        board = new ChessBoard();
        controller = new LogicController();
        IO = new GamePanel(this);
        controller.init(this);
        gameFrame = new GameFrame(name:"Chess",IO);
    }
    // MAIN GAME LOOP DO NOT TOUCH
    public void run(){
        long frameDelay = 1000000000/FPS;
        long nextUpdate = System.nanoTime()+(long)frameDelay;
        while(true){
            if(nextUpdate<System.nanoTime()){
                nextUpdate+=frameDelay;
                update();
                IO.repaint();
            }
        }
    }
}
```

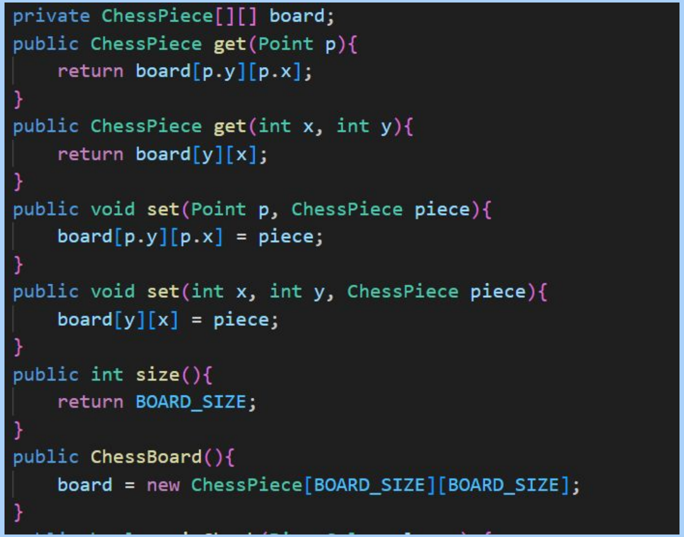


```java
// ALL CODE FOR ENTIRE PROJECT IS RUN HERE
private void update(){
    Point click = IO.update();
    if(menuState.equals(anObject:"GAME")){
        if(click!=null){
            controller.update(click);
        }
    }
    // TODO menuing
}
```

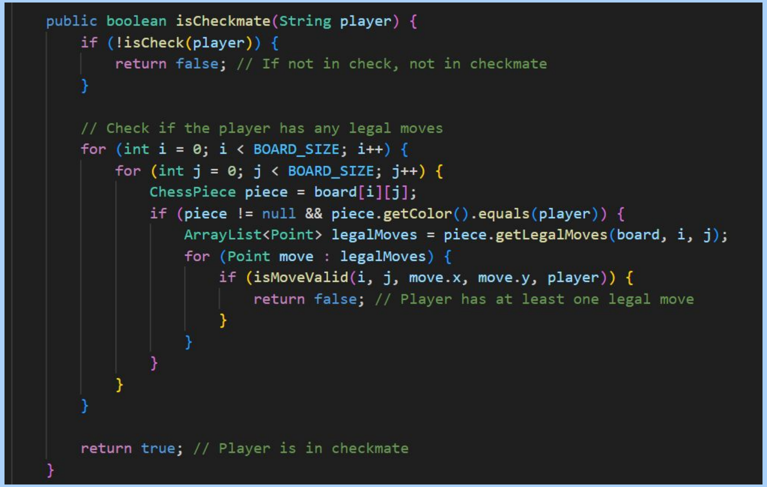
ChessBoard

1. Main Data Structure for game objects
2. Uses a 2D array to represent the chess board
3. Handles all board state logic, including determining check, checkmate, stalemate

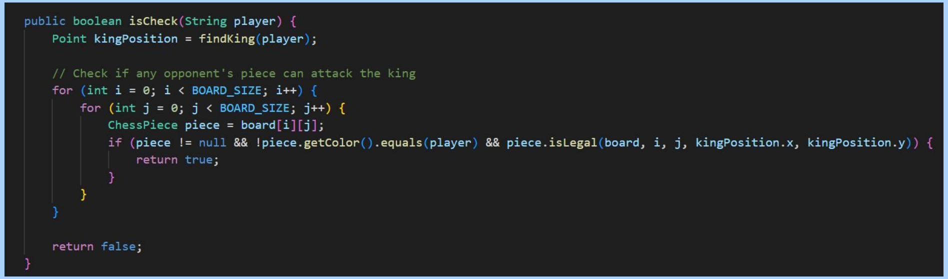
*pictures next slide



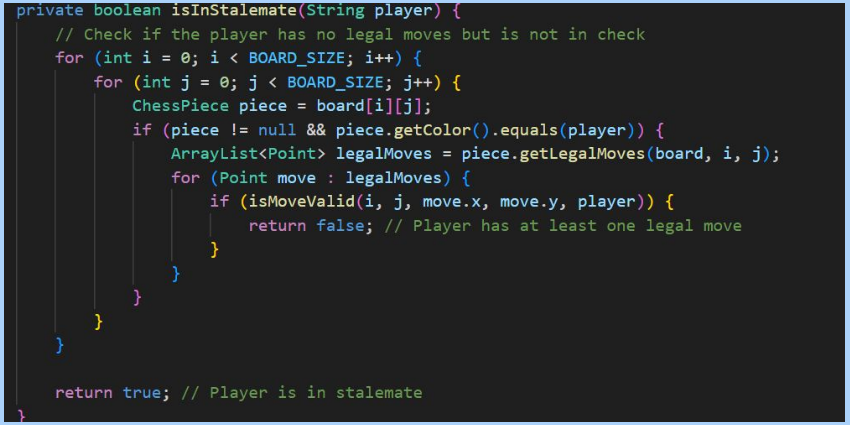
```java
private ChessPiece[][] board;
public ChessPiece get(Point p){
    return board[p.y][p.x];
}
public ChessPiece get(int x, int y){
    return board[y][x];
}
public void set(Point p, ChessPiece piece){
    board[p.y][p.x] = piece;
}
public void set(int x, int y, ChessPiece piece){
    board[y][x] = piece;
}
public int size(){
    return BOARD_SIZE;
}
public ChessBoard(){
    board = new ChessPiece[BOARD_SIZE][BOARD_SIZE];
}
```



```java
public boolean isCheckmate(String player) {
    if (!isCheck(player)) {
        return false; // If not in check, not in checkmate
    }

    // Check if the player has any legal moves
    for (int i = 0; i < BOARD_SIZE; i++) {
        for (int j = 0; j < BOARD_SIZE; j++) {
            ChessPiece piece = board[i][j];
            if (piece != null && piece.getColor().equals(player)) {
                ArrayList<Point> legalMoves = piece.getLegalMoves(board, i, j);
                for (Point move : legalMoves) {
                    if (isMoveValid(i, j, move.x, move.y, player)) {
                        return false; // Player has at least one legal move
                    }
                }
            }
        }
    }

    return true; // Player is in checkmate
}
```



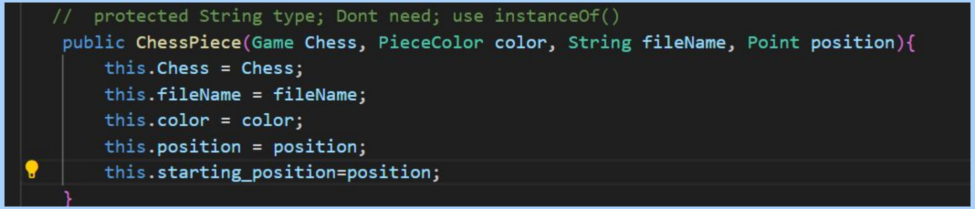
```java
public boolean isCheck(String player) {
    Point kingPosition = findKing(player);

    // Check if any opponent's piece can attack the king
    for (int i = 0; i < BOARD_SIZE; i++) {
        for (int j = 0; j < BOARD_SIZE; j++) {
            ChessPiece piece = board[i][j];
            if (piece != null && !piece.getColor().equals(player) && piece.isLegal(board, i, j, kingPosition.x, kingPosition.y)) {
                return true;
            }
        }
    }

    return false;
}
```



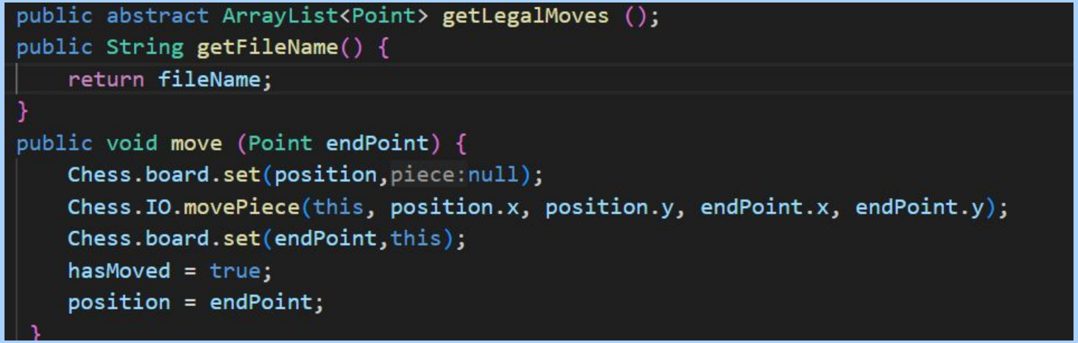
```java
private boolean isInStalemate(String player) {
    // Check if the player has no legal moves but is not in check
    for (int i = 0; i < BOARD_SIZE; i++) {
        for (int j = 0; j < BOARD_SIZE; j++) {
            ChessPiece piece = board[i][j];
            if (piece != null && piece.getColor().equals(player)) {
                ArrayList<Point> legalMoves = piece.getLegalMoves(board, i, j);
                for (Point move : legalMoves) {
                    if (isMoveValid(i, j, move.x, move.y, player)) {
                        return false; // Player has at least one legal move
                    }
                }
            }
        }
    }

    return true; // Player is in stalemate
}
```

ChessPiece

- Abstract class
- Implements basic piece control
- child objects implement legal moves depending on piece

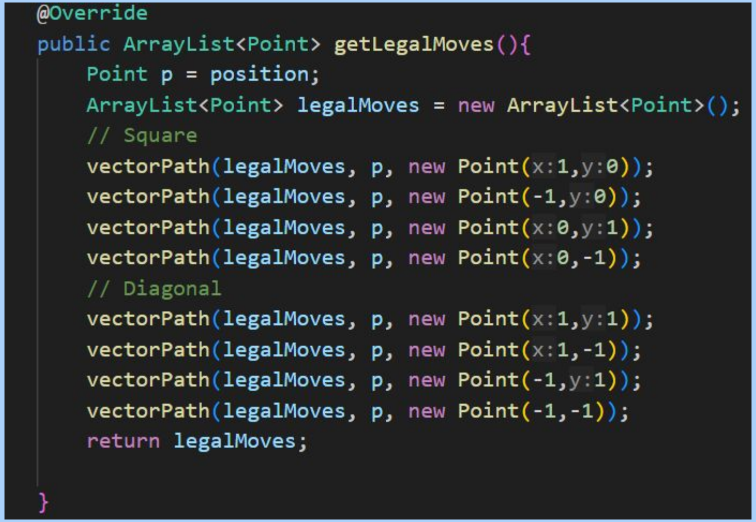


```
//  protected String type; Dont need; use instanceOf()
 public ChessPiece(Game Chess, PieceColor color, String fileName, Point position){
     this.Chess = Chess;
     this.fileName = fileName;
     this.color = color;
     this.position = position;
     this.starting_position=position;
 }
```

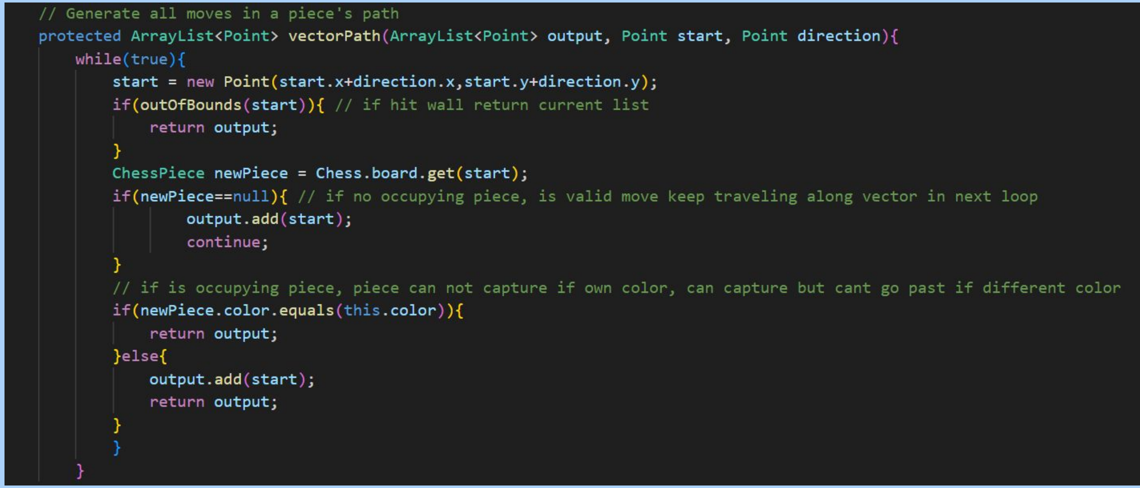


```
public abstract ArrayList<Point> getLegalMoves ();
public String getFileName() {
    return fileName;
}
public void move (Point endPoint) {
    Chess.board.set(position,piece:null);
    Chess.IO.movePiece(this, position.x, position.y, endPoint.x, endPoint.y);
    Chess.board.set(endPoint,this);
    hasMoved = true;
    position = endPoint;
}
```

Example ChessPiece - Queen



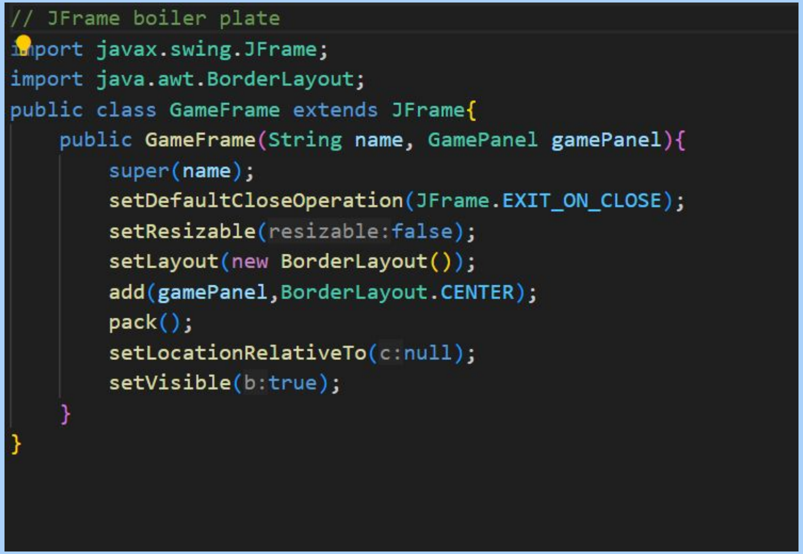
```java
@Override
public ArrayList<Point> getLegalMoves(){
    Point p = position;
    ArrayList<Point> legalMoves = new ArrayList<Point>();
    // Square
    vectorPath(legalMoves, p, new Point(x:1,y:0));
    vectorPath(legalMoves, p, new Point(-1,y:0));
    vectorPath(legalMoves, p, new Point(x:0,y:1));
    vectorPath(legalMoves, p, new Point(x:0,-1));
    // Diagonal
    vectorPath(legalMoves, p, new Point(x:1,y:1));
    vectorPath(legalMoves, p, new Point(x:1,-1));
    vectorPath(legalMoves, p, new Point(-1,y:1));
    vectorPath(legalMoves, p, new Point(-1,-1));
    return legalMoves;

}
```



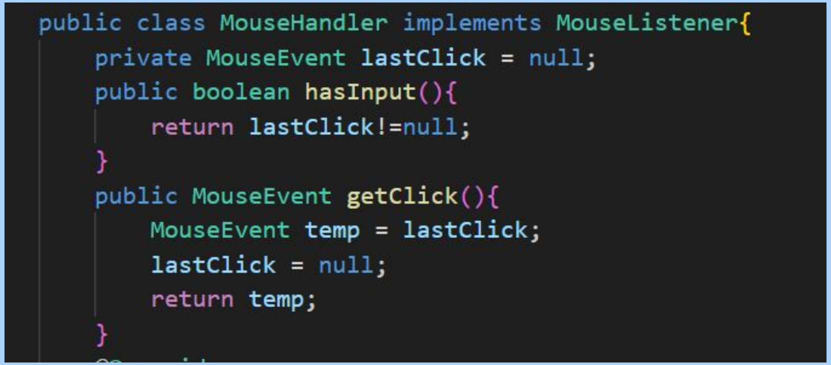
```java
// Generate all moves in a piece's path
protected ArrayList<Point> vectorPath(ArrayList<Point> output, Point start, Point direction){
    while(true){
        start = new Point(start.x+direction.x,start.y+direction.y);
        if(outOfBounds(start)){ // if hit wall return current list
            return output;
        }
        ChessPiece newPiece = Chess.board.get(start);
        if(newPiece==null){ // if no occupying piece, is valid move keep traveling along vector in next loop
                output.add(start);
                continue;
        }
        // if is occupying piece, piece can not capture if own color, can capture but cant go past if different color
        if(newPiece.color.equals(this.color)){
            return output;
        }else{
            output.add(start);
            return output;
        }
        }
    }
```

Graphics classes

GameFrame extends JFrame and contains all window configuration code
GamePanel extends JPanel and contains all
MouseHandler is a trivial implementation of MouseListener

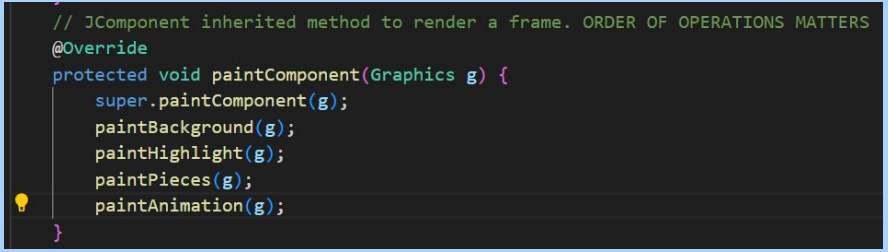


```java
// JFrame boiler plate
import javax.swing.JFrame;
import java.awt.BorderLayout;
public class GameFrame extends JFrame{
    public GameFrame(String name, GamePanel gamePanel){
        super(name);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setResizable(resizable:false);
        setLayout(new BorderLayout());
        add(gamePanel,BorderLayout.CENTER);
        pack();
        setLocationRelativeTo(c:null);
        setVisible(b:true);
    }
}
```

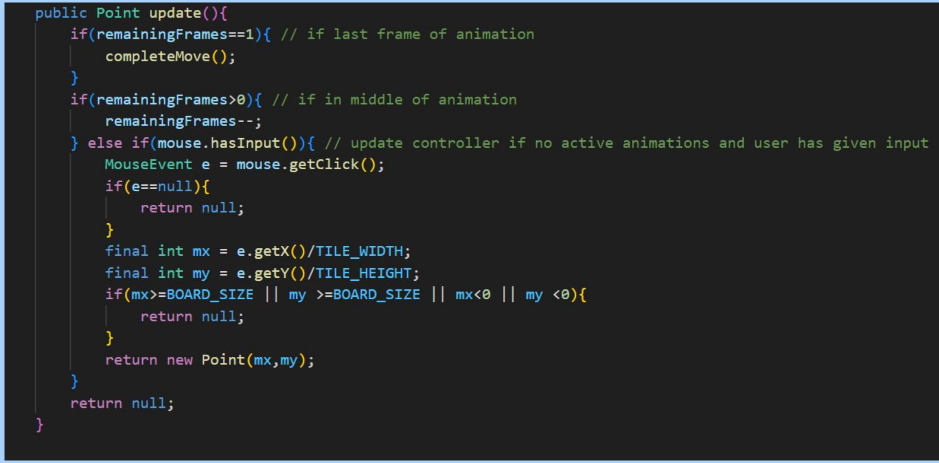


```java
public class MouseHandler implements MouseListener{
    private MouseEvent lastClick = null;
    public boolean hasInput(){
        return lastClick!=null;
    }
    public MouseEvent getClick(){
        MouseEvent temp = lastClick;
        lastClick = null;
        return temp;
    }
```

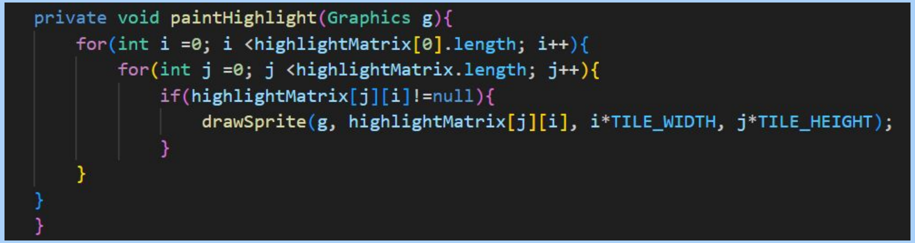
GamePanel



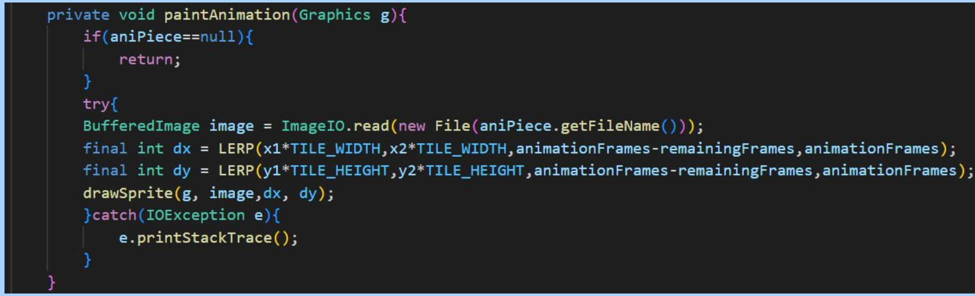
```java
// JComponent inherited method to render a frame. ORDER OF OPERATIONS MATTERS
@Override
protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    paintBackground(g);
    paintHighlight(g);
    paintPieces(g);
    paintAnimation(g);
}
```



```java
public Point update(){
    if(remainingFrames==1){ // if last frame of animation
        completeMove();
    }
    if(remainingFrames>0){ // if in middle of animation
        remainingFrames--;
    } else if(mouse.hasInput()){ // update controller if no active animations and user has given input
        MouseEvent e = mouse.getClick();
        if(e==null){
            return null;
        }
        final int mx = e.getX()/TILE_WIDTH;
        final int my = e.getY()/TILE_HEIGHT;
        if(mx>=BOARD_SIZE || my >=BOARD_SIZE || mx<0 || my <0){
            return null;
        }
        return new Point(mx,my);
    }
    return null;
}
```



```java
private void paintHighlight(Graphics g){
    for(int i =0; i <highlightMatrix[0].length; i++){
        for(int j =0; j <highlightMatrix.length; j++){
            if(highlightMatrix[j][i]!=null){
                drawSprite(g, highlightMatrix[j][i], i*TILE_WIDTH, j*TILE_HEIGHT);
            }
        }
    }
}
```

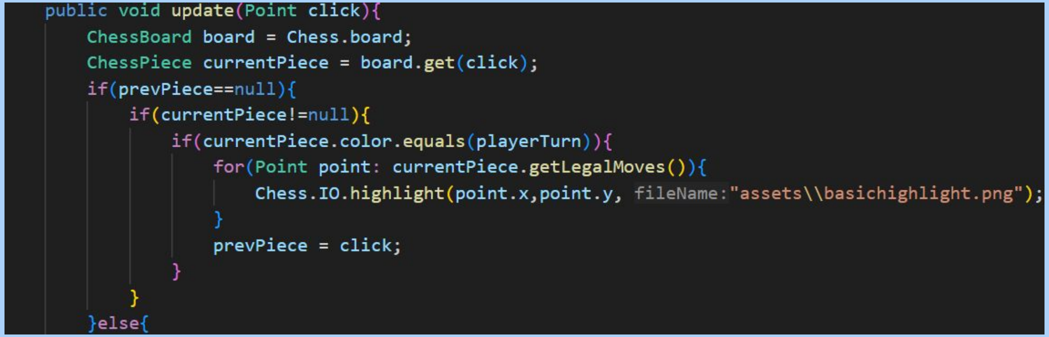


```java
private void paintAnimation(Graphics g){
    if(aniPiece==null){
        return;
    }
    try{
    BufferedImage image = ImageIO.read(new File(aniPiece.getFileName()));
    final int dx = LERP(x1*TILE_WIDTH,x2*TILE_WIDTH,animationFrames-remainingFrames,animationFrames);
    final int dy = LERP(y1*TILE_HEIGHT,y2*TILE_HEIGHT,animationFrames-remainingFrames,animationFrames);
    drawSprite(g, image,dx, dy);
    }catch(IOException e){
        e.printStackTrace();
    }
}
```

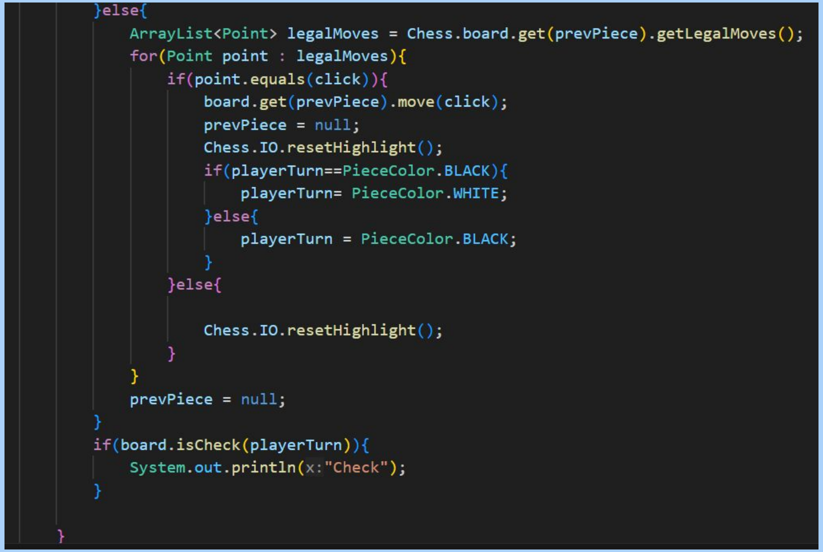
Logic Controller

Implements game state diagram

Mediates all actions in entire project at a high level



```
public void update(Point click){
    ChessBoard board = Chess.board;
    ChessPiece currentPiece = board.get(click);
    if(prevPiece==null){
        if(currentPiece!=null){
            if(currentPiece.color.equals(playerTurn)){
                for(Point point: currentPiece.getLegalMoves()){
                    Chess.IO.highlight(point.x,point.y, fileName:"assets\\basichighlight.png");
                }
                prevPiece = click;
            }
        }
    }else{
```



```
    }else{
        ArrayList<Point> legalMoves = Chess.board.get(prevPiece).getLegalMoves();
        for(Point point : legalMoves){
            if(point.equals(click)){
                board.get(prevPiece).move(click);
                prevPiece = null;
                Chess.IO.resetHighlight();
                if(playerTurn==PieceColor.BLACK){
                    playerTurn= PieceColor.WHITE;
                }else{
                    playerTurn = PieceColor.BLACK;
                }
            }else{

                Chess.IO.resetHighlight();
            }
        }
        prevPiece = null;
    }
    if(board.isCheck(playerTurn)){
        System.out.println(x:"Check");
    }

}
```

Some Refactoring Changes we made

- Clear entry point to allow global communication when necessary
- Privatizing variables to prevent inappropriate use (board[y][x] vs board.get(x,y)
- Using points instead of integers for better readability
- Dividing graphics objects into separate classes in accordance with JFrame library
- Separating updates to multiple different classes

TESTING

Testing methodology

- We have so far only performed white box testing
- Most requirements are simple to implement individually, the complexity comes from the number of different paths to test.
- We used bottom up testing to easily prove whether detailed requirements are met, and using manual testing to explore each functionality.

Testing - View

- Highlight and animations are the easiest requirements to test since there are very few number of paths to explore.
- Problems can be easily identified because of visual anomalies.
- Tested exhaustively
- Graphics are used as a driver for other tests

Testing - Data

- Purpose is to ensure chess objects are tracked properly through the board.
- Accomplished by rendering objects, and comparing their memory location in the data the rendered location.
- Position in piece memory can be programmatically compared to position of pieces in the board memory
- Tested Exhaustively

Testing Model

- Pieces can be tested exhaustively since there most valid moves are translations of other valid moves.
- In our testing, we found basic movement to be acceptable, but edge cases to be inaccurate.
- Methods like isCheck() and is CheckMate() rely on accurate piece movement, therefore requirements are still not met even though the components are functioning properly.

Testing Controller

- The controller acts as the driver for all other tests performed.
- Our initial white-box testing of the controller revealed several issues pertaining to handling component communication in special chess cases.
- Controller cannot be fully tested until special cases are accurate.
- The projects requirements are met when the controller is validated
- For final validation of the project, we will use blackbox testing against stockfish, a powerful and widely accepted chess model. If the project is correct, for any board state, the list of valid moves produced by the controller should equal the list of valid moves produced by stockfish.

Testing - Requirements Met or Not Met

Tested all the requirements and identified the requirements that were not satisfied to identify changes in the next iteration

Requirements	Requirements NOT Satisfied
a. Two players can alternate moving their pieces b. Pieces are animated showing where they are moving c. Players should be able to see valid moves that are highlighted when clicking on a piece d. Pieces should be set to the correct positions at the start of the game.	a. Players are only allowed to make legal moves b. Game should end when player is in checkmate or stalemate c. A player must make a move to get out of check when in check d. Pawns should be promoted upon reaching the end of the board e. A king and rook may be swapped if neither has moved yet f. A pawn can take two steps on its first move g. A pawn can take another pawn if it was skipped by taking 2 steps **h. Players should be able to see whose turn it is** **i. Players should be able to restart the game at the end of a match** bold need significant work.

Demo

- Usb Drive
- Github link will be posted here, after we finalize the code

Example of our end result after white moves up by 1 space