



CCP6224 Object Oriented Analysis and Design

Kwazam Chess Project

Trimester 2310

By Group C

Mr. Zainudin Bin Johari

Lab session: T19L

Name	ID	Phone
Sia Jing Liang	1211106208	016-2716656
Lee Xiang Ze	1211106818	011-63388308
Ngo Shun Hong	1211106586	019-7586016
Yeoh Han Yi	1211106319	012-5568050

1.	Work Distribution	3
2.	Compile And Run Instructions	7
3.	UML Class diagram	11
3.1	Class Table	11
3.2	Class Diagram	28
3.3	Classes Utilizing Design Patterns	29
3.3.1	MVC Design Pattern	29
3.3.2	Singleton Design Pattern	29
3.3.3	Template Method Design Pattern	30
3.3.4	Factory Method Design Pattern	30
3.3.5	Composite Design Pattern	31
3.3.6	Adapter Design Pattern	31
3.3.7	Façade Design Pattern	31
3.3.8	Observer Design Pattern	32
3.3.9	Mediator Design Pattern	33
3.3.10	State Design Pattern	34
3.3.11	Abstract Factory Design Pattern	34
4.	Use Case diagram	35
4.1	Use Case	35
4.2	Use case Diagram	36
5.	Sequence diagrams	37
5.1	Create new game	37
5.2	Open saved game (Load)	38
5.3	Save game	39
5.4	Exit game	40
5.5	Move piece	41
6.	User Documentation	43
6.1	User Interface Overview	43

6.2	User Roles and Permission	47
6.3	Game Instructions	47
6.4	File Management Information	49
6.4.1	Save file	49
6.4.1	Load file	50

1. Work Distribution

Category	Class	Method	Contributor
Controller	CheckSau	isCheck	Sia, Yeoh
		findSau	Sia, Yeoh
	Controller	addListener	Ngo
		gameStart	Lee
		isVaildMove	Lee, Sia
		sameTeam	Lee
		changePiecePosition	Sia
		capture	Sia
		winOrNextTurn	Sia
		setResetSelectedPiece	Sia
		getModel	Ngo
		getView	Ngo
		getPieceList	Ngo
		getPiece	Ngo
		transformPieces	Lee
		transform	Lee
		toggleFlip	Lee
		saveGame	Yeoh
		loadGame	Yeoh
	Input	mousePressed	Lee, Sia
		mouseDragged	Lee, Sia
		mouseReleased	Lee, Sia
		actionPerformed	Yeoh, Lee
		componentResized	Sia
	Move		Ngo

Category	Class	Method	Contributor
Model	Model	initialize	Ngo
		addPieces	Ngo
		getArrayList	Sia
		getPiece	Yeoh
	Piece	loadSprite	Ngo
		setCol	Ngo
		setRow	Ngo
		setXPos	Ngo
		setYPos	Ngo
		setIsBlue	Ngo
		setName	Ngo
		getSprite	Yeoh
		getXPos	Yeoh
		getYPos	Yeoh
		getIsBlue	Yeoh
		getName	Yeoh
		canMoveTo	Lee
	Tor	isPathClear	Lee, Sia, Yeoh
		canMoveTo	Lee, Sia, Yeoh
	Xor	isPathClear	Lee, Sia, Yeoh
		canMoveTo	Lee, Sia, Yeoh
	Biz	canMoveTo	Lee, Ngo
	Ram	canMoveTo	Lee, Sia, Yeoh
		setReachEnd	Lee, Sia, Yeoh
		getReachEnd	Lee, Sia, Yeoh
	Sau	canMoveTo	Ngo, Sia, Yeoh
	PieceFactory	getPiece	Yeoh

Category	Class	Method	Contributor
View	MenuPanel	getNewMenuItem	Yeoh
		getOpenMenuItem	Yeoh
		getSaveMenuItem	Yeoh
		getExitMenuItem	Yeoh
	Panel	calculateValidMoves	Ngo
		paintValidMove	Ngo
		paintComponent	Lee, Sia
		getCols	Sia
		getRows	Sia
	View	getPanel	Lee
		getMp	Lee
		getFrame	Lee
		paint	Ngo
		getCols	Lee
		getRows	Lee
		setSelectedPiece	Sia
		paintValidMove	Ngo
		printWinner	Ngo
		printSaulsInCheck	Ngo
		handleExit	Ngo

2. Compile And Run Instructions

The prerequisite to running the application includes:

1. The device must have a Java Runtime Environment (JRE) to be able to run Java programs.
2. The zip file containing all Java files is downloaded.

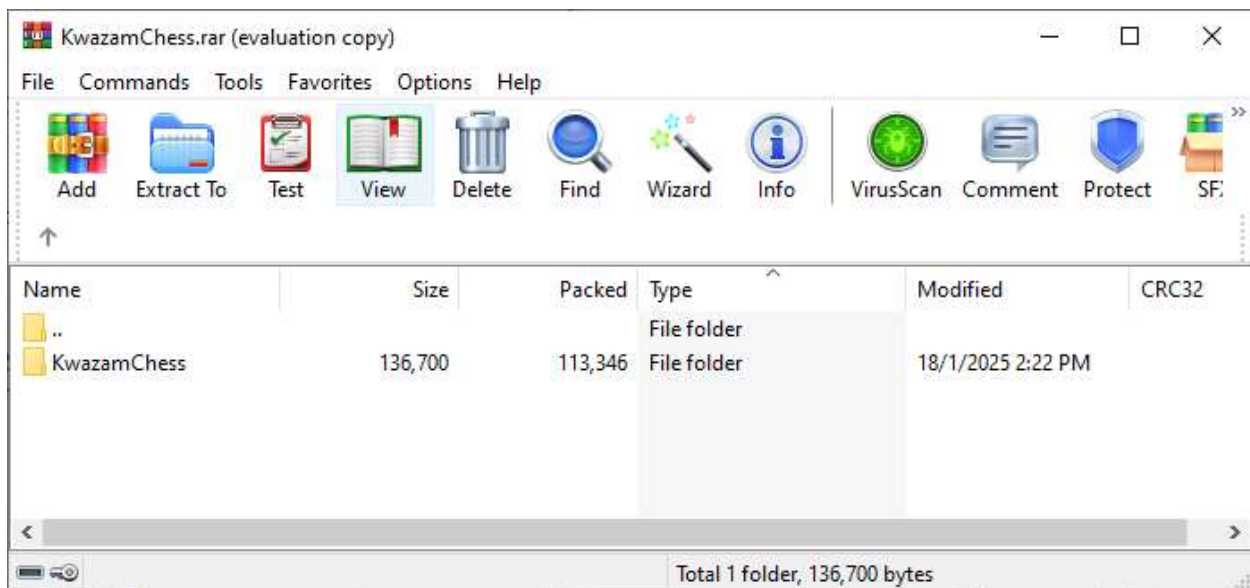
The following instructions are compatible with the Windows operating system. Different commands are needed in the terminal to run the main Java file for other operating systems.

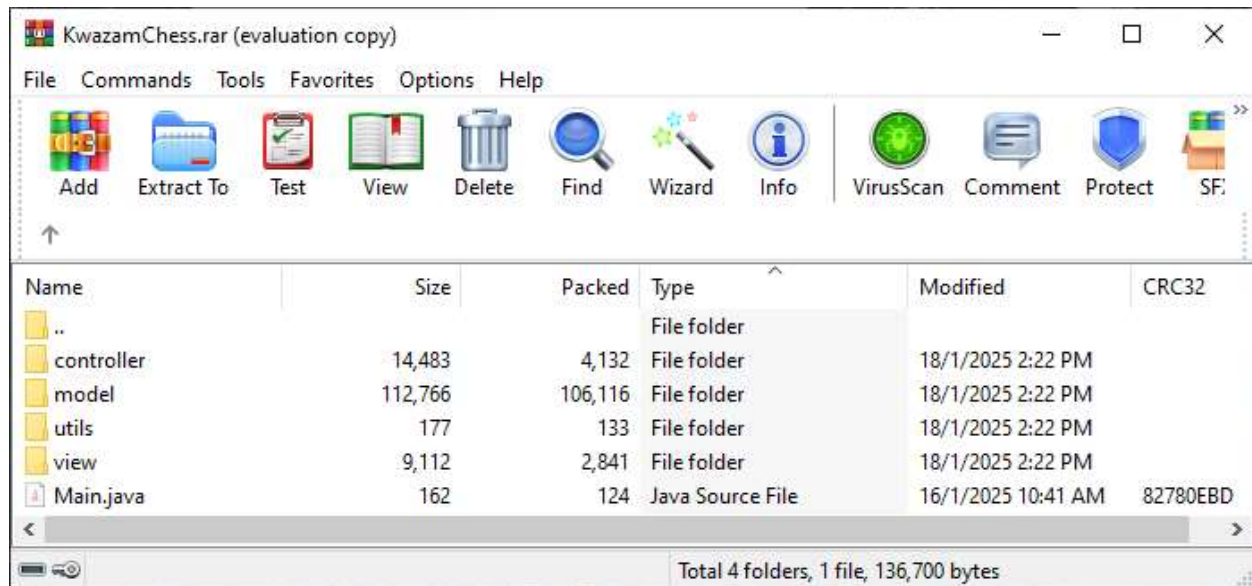
Step 1: File Extraction

Extract all the files in the zip file.

When the folder called “KwazamChess” is selected and opened, all the source code files can be seen. The source code's relative application file path is as follows:

...|KwazamChess





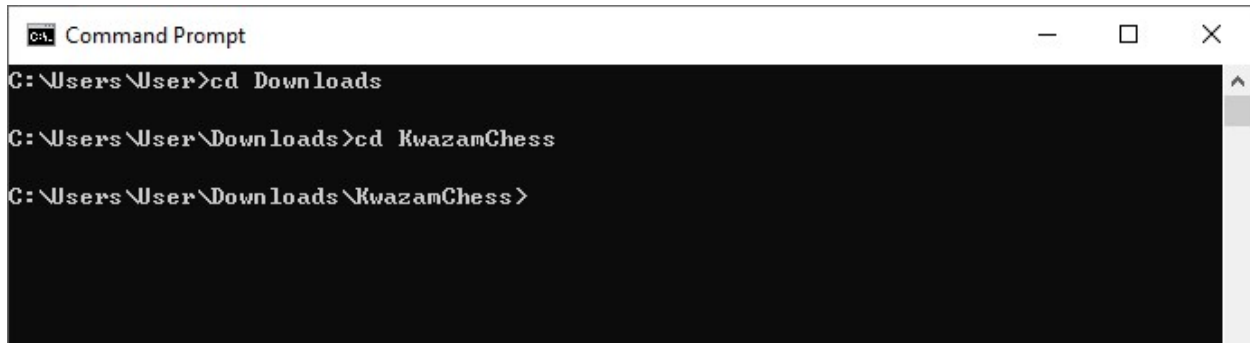
These are the following source code files inside the KwazamChess folder

Step 2: Open folder through command prompt

A. Open the command prompt



B. Navigate to the “KwazamChess” folder.



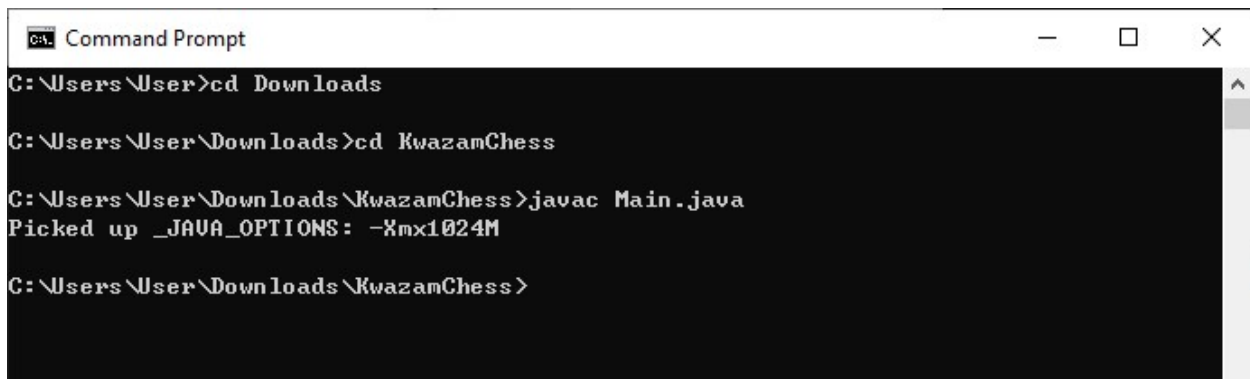
```
Command Prompt
C:\Users\User>cd Downloads
C:\Users\User\Downloads>cd KwazamChess
C:\Users\User\Downloads\KwazamChess>
```

The “KwazamChess” folder contains a file called “Main.java”. The “Main.java” file contains the main and is runnable.

Step 3: Compile the files

Once you have navigated to the “KwazamChess” folder as shown above. Type and run the following command:

javac Main.java



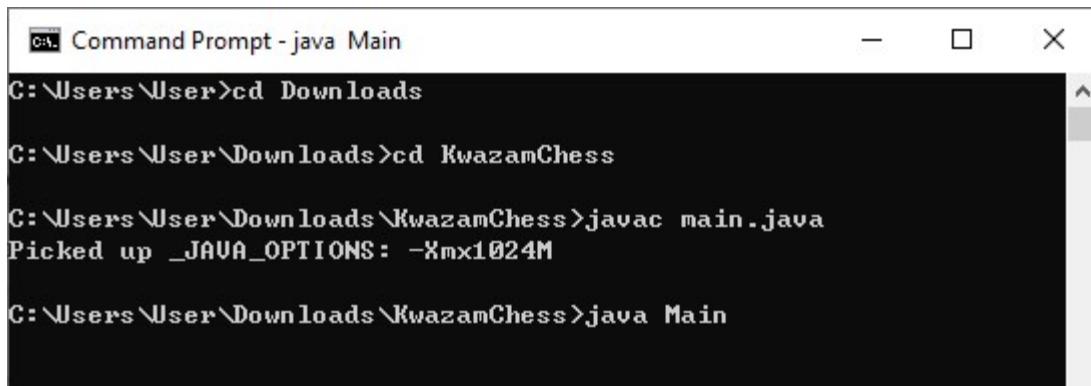
```
Command Prompt
C:\Users\User>cd Downloads
C:\Users\User\Downloads>cd KwazamChess
C:\Users\User\Downloads\KwazamChess>javac Main.java
Picked up _JAVA_OPTIONS: -Xmx1024M
C:\Users\User\Downloads\KwazamChess>
```

You will see that the CLASS Files are generated in the same folder. This shows that the compilation is successful.

Step 4: Execute the application

Type and run the following command:

java Main



```
Command Prompt - java Main
C:\Users\User>cd Downloads
C:\Users\User\Downloads>cd KwazamChess
C:\Users\User\Downloads\KwazamChess>javac main.java
Picked up _JAVA_OPTIONS: -Xmx1024M
C:\Users\User\Downloads\KwazamChess>java Main
```

The application will open, and you are ready to go.

Additional Method (using IDE)

Step 1: launch your preferred IDE (e.g., IntelliJ IDEA, Eclipse, BlueJ, or VS Code).

Step 2: navigate to the File menu and select Open Folder or Open Project, depending on the IDE.

Step 3: Locate your project folder in the file explorer that appears, select it, and click Open.

Step 4: The IDE will load the project, displaying its structure and files in the workspace for easy access and editing.

3. UML Class diagram

3.1 Class Table

Class Name	Class Description	Attributes/Operations
Model		
Piece (Abstract)	<p>The Piece class is an abstract base class designed as the foundation for all pieces. It provides essential attributes and methods to manage game pieces effectively.</p> <p>Attributes: col and row: Coordinate of the piece's position.</p> <p>xPos and yPos: Pixel coordinates for graphical rendering of the piece.</p> <p>isBlue: A boolean flag to determine the piece's team or side (e.g., blue or red).</p> <p>name: The name of the piece, used for identification and display purposes.</p> <p>sprite: A BufferedImage object representing the visual appearance of the piece.</p> <p>model: A static reference to the game model, allowing pieces to interact with the broader game state.</p> <p>Constructor:</p>	<div>Piece (Abstract)</div> <ul style="list-style-type: none"> - col: int; - row: int; - xPos: int - yPos: int - isBlue: boolean - name: string - sprite: BufferedImage - model: Model(Static) <ul style="list-style-type: none"> + Piece(col : int, row : int, isBlue : boolean, name : String) + loadSprite(path : String) : BufferedImage + setCol(col : int) : void + setRow(row : int) : void + setXPos(xPos : int) : void + setYPos(yPos : int) : void + setIsBlue(isBlue : boolean) : void + setName(name : String) : void + getSprite() : BufferedImage + getCol() : int + getRow() : int + getXPos() : int + getYPos() : int + getIsBlue() : boolean + getName() : String + canMoveTo(newCol : int, newRow : int) : boolean

	<p>Piece: Initializes the piece with its position, team color, and name. And calculates pixel coordinates based on the tile Size.</p> <p>Methods: loadSprite: Loads the sprite image from the specified file path.</p> <p>Getters and Setters: Provide access to and allow modifications of attributes.</p> <p>canMoveTo: An abstract method that enforces each subclass to define its specific movement rules.</p>	
Ram	<p>The Ram class represents a type of piece called "Ram" which extends the abstract Piece class. It represents the behavior of the Ram piece.</p> <p>Constructor: Ram: Initializes the Ram piece with its column, row, team, and name by base class constructor</p> <p>Attributes: ReachEnd: Indicates the direction of the Ram. If the Ram reaches the end of the panel, its movement direction is reversed, represented by a 180-degree rotation.</p> <p>Overridden Method:</p>	<div> <div>Ram</div> <div> - reachEnd: int; </div> <div> + Ram(col : int, row : int, isBlue : boolean) + canMoveTo(newCol : int, newRow : int) + setReachEnd(); + getReachEnd() </div> </div>

	<p>canMoveTo: Ram can only move forward in the direction determined by its team. Upon reaching the top or bottom edge of the board, its movement reverses, accompanied by a 180-degree rotation.</p> <p>Returns true if the move to the new position is valid; otherwise, false.</p> <p>Methods: setReachEnd: Updates the reachEnd value to toggle between 1 and -1.</p> <p>getReachEnd: Returns the current value of reachEnd, indicating the Ram's movement direction.</p>	
Biz	<p>The Biz class represents a type of piece called "Biz" which extends the abstract Piece class. It represents the behavior of the Biz piece.</p> <p>Constructor: Biz: Initializes the Biz piece with its column, row, team, and name by base class constructor</p> <p>Overridden Method: canMoveTo: Biz can only move in an L-shaped patten. Returns true if the move to the new position is valid; otherwise, false.</p>	<div> <div>Biz</div> <div> + Biz(col : int, row : int, isBlue : boolean) + canMoveTo(newCol : int, newRow : int) </div> </div>

Tor	<p>The Tor class represents a type of piece called "Tor" which extends the abstract Piece class. It represents the behavior of the Tor piece.</p> <p>Constructor: Tor: Initializes the Tor piece with its column, row, team, and name by base class constructor.</p> <p>Methods: isPathClear: A method that checks is there any pieces blocking the path. Check along the path between the current position and the target to ensure that all intermediate squares are empty. Returns true if the path is clear, false otherwise.</p> <p>Overridden Method: canMoveTo: Tor can only move in a straight line either vertically or horizontally. Returns true if the move to the new position is valid; otherwise, false.</p>	<div data-bbox="901 195 1414 510"> <div>Tor</div> <div> + Tor(col : int, row : int, isBlue : boolean) + isPathClear(newCol : int, newRow : int) + canMoveTo(newCol : int, newRow : int) </div> </div>
Xor	<p>The Xor class represents a type of piece called "Xor" which extends the abstract Piece class. It represents the behavior of the Xor piece.</p> <p>Constructor: Xor: Initializes the Xor piece with its column, row, team, and</p>	<div data-bbox="922 1497 1393 1795"> <div>Xor</div> <div> + Xor(col : int, row : int, isBlue : boolean) + isPathClear(newCol : int, newRow : int) + canMoveTo(newCol : int, newRow : int) </div> </div>

	<p>name by base class constructor.</p> <p>Methods: isPathClear: A method that checks is there any pieces blocking the path. Check along the path between the current position and the target to ensure that all intermediate squares are empty. Returns true if the path is clear, false otherwise.</p> <p>Overridden Method: canMoveTo: Xor can only move in an orthogonal patten. Returns true if the move to the new position is valid; otherwise, false</p>	
Sau	<p>The Sau class represents a type of piece called "Sau" which extends the abstract Piece class. It represents the behavior of the Sau piece.</p> <p>Constructor: Sau: Initializes the Sau piece with its column, row, team, and name by base class constructor.</p> <p>Overridden Method: canMoveTo: Xor can only move one step in any direction (up, down, left, right, or diagonally). Returns true if the move to the new position is valid; otherwise, false</p>	<div style="border: 1px solid black; padding: 5px; margin: 10px auto; width: fit-content;"> <p style="text-align: center;">Sau</p> <pre>+ Sau(col : int, row : int, isBlue : boolean) + canMoveTo(newCol : int, newRow : int)</pre> </div>

<p>PieceFactory</p>	<p>The PieceFactory class follows the Factory design pattern to create different types of chess pieces (Tor, Xor, Ram, Biz, Sau).</p> <p>Methods: getPiece: This method takes the name of a piece, its column and row coordinates, and a boolean indicating the team color (blue or red). It creates and returns an instance of the corresponding piece type. If the provided name does not match any of the predefined piece types, it returns null.</p>	<div data-bbox="906 231 1409 464"> <div>PieceFactory</div> <div>+ getPiece(name:String, col:int, row:int, isBlue:boolean):Piece</div> </div>
<p>Model</p>	<p>The Model class manages the collection of chess pieces in the game.</p> <p>Attribute: pieceList: An ArrayList that holds all the Piece objects in the game.</p> <p>Methods: initialize: Clears the pieceList and initializes the pieces by calling the addPieces method.</p> <p>addPieces: Creates an instance of the corresponding piece type and adds all the pieces to the pieceList</p> <p>getArrayList: Returns the pieceList containing all the pieces in the game.</p> <p>getPiece: Returns the piece located at the specified position on the panel. If no</p>	<div data-bbox="906 961 1409 1266"> <div>Model</div> <div>- pieceList: ArrayList<Piece></div> <div>+ initialize() : void - addPieces() : void + getArrayList() : ArrayList<Piece> + getPiece(col: int, row :</div> </div>

	piece exists at that position, it returns null.	
View		
Panel	<p>The Panel class display the panel, pieces, and valid move highlighting within the user interface.</p> <p>Attribute: cols: The number of columns on the panel (5 columns in this case). rows: The number of rows on the panel (8 rows in this case). selectedPiece: The currently selected piece in the game (when mouse click). validMoves: A list of valid move coordinates for the selected piece. view: A reference to the View object, which links the Panel to the overall game logic.</p> <p>Constructor: Panel: Initializes the panel with a reference to the View, sets the size of the panel based on the number of rows and columns, and initializes other attributes like selectedPiece and validMoves.</p> <p>Methods: calculateValidMoves: Calculate the valid move position for the currently selected piece by checking all possible moves and adding them to validMoves.</p>	<div>Panel</div> <div> - cols : int - rows : int # selectedPiece : Piece - validMoves : ArrayList<Point> - view : View </div> <div> + Panel(view : View) - calculateValidMoves() : void + paintValidMove() : void # paintComponent(g : Graphics) : void + getCols() : int + getRows() : int </div>


	<p>paintValidMove: Highlights the valid moves position for the selected piece by triggering a repaint.</p> <p>paintComponent: Rendering the panel, piece, and valid moves. It draw the panel, highlights the valid move areas, and draws each piece in its corresponding position. If the piece is ram it will rotate 180 degree when it reach the edge of the panel.</p> <p>getCols: Return the number of columns.</p> <p>getRows: Return the number of rows.</p>	
MenuPanel	<p>This MenuPanel class represents the menu bar for the game's user interface, allowing user to preform basic operation such as creating a new game, open save game, save a game, and exit the application.</p> <p>Attribute: fileMenu: A JMenu object representing the file menu, contains the file operation options.</p> <p>newMenuItem: A JMenuItem representing the "New" menu option, used to start a new game.</p> <p>saveMenuItem: A JMenuItem representing the "Save" menu</p>	<div> <div>MenuPanel</div> <div> - fileMenu : JMenu - newItem : JMenuItem - saveMenuItem : JMenuItem - openMenuItem : JMenuItem - exitMenuItem : JMenuItem </div> <div> + MenuPanel() + getNewMenuItem() : JMenuItem + getOpenMenuItem() : JMenuItem + getSaveMenuItem() : JMenuItem + getExitMenuItem() : JMenuItem </div> </div>

	<p>option, used to save the current game.</p> <p>openMenuItem: A JMenuItem representing the "Open" menu option, used to open a save game.</p> <p>exitMenuItem: A JMenuItem representing the "Exit" menu option, used to close the application.</p> <p>Constructor: MenuPanel: Initializes the menu panel by creating the file menu and the associated menu items (New, Open, Save, Exit). The menu items are added to the fileMenu, and the fileMenu is added to the menu bar</p> <p>Methods: getNewMenuItem(): Returns the JMenuItem corresponding to the "New" option.</p> <p>getOpenMenuItem(): Returns the JMenuItem corresponding to the "Open" option.</p> <p>getSaveMenuItem(): Returns the JMenuItem corresponding to the "Save" option.</p> <p>getExitMenuItem(): Returns the JMenuItem corresponding to the "Exit" option.</p>	
--	--	--

View	<p>This view class serves as the user interface layer of the application, contain game panel and menu panel, the overall frame display the kwazam-chess, interact with the controller class to handel user interaction.</p> <p>Attribute: Frame: A JFrame instance representing the window of the application, which contain all UI component.</p> <p>Panel: An instance of the panel class that serves as the central game panel for display and interacting with pieces.</p> <p>Controller: A Controller class instance that manage the game logic and handles user interaction.</p> <p>pieceList: A list of Piece that represent the current state of all chess pieces on the panel.</p> <p>mp: A MenuPanel class instance. providing the menu bar with options such as “New”, “Open”, “Save”, “Exit”.</p> <p>Constructor: View: Initialize the frame and add game panel, and menu panel to the frame. Set up window frame in the center and visible.</p> <p>Methods: getPanel: Return panel for rendering the game panel.</p>	<div>View</div> <pre> # frame : JFrame - panel : Panel + controller : Controller # pieceList : ArrayList<Piece> - mp : MenuPanel + View(controller : Controller) + getPanel() : Panel + getMp() : MenuPanel + getFrame() : JFrame + paint(pieceList : ArrayList<Piece>) : void + getCols() : int + getRows() : int + setSelectedPiece(selectedPiece : Piece) : void + paintValidMove() : void + printWinner(winner : int) : void + printSaulsInCheck(isBlue : boolean) : void + handleExit() : void </pre>
------	---	--

	<p>getMp: Return the mp for operation.</p> <p>getFrame: Return the frame.</p> <p>paint: Update the pieceList and repaint the game panel</p> <p>getCols: Return the number of columns on the game panel from the panel instance.</p> <p>getRows: Return the number of rows on the game panel from the panel instance.</p> <p>setSelectedPiece: Update the current selected piece in the game panel (when mouse click).</p> <p>paintValidMove: Triggers the game panel to calculate and highlight valid moves for the selected piece.</p> <p>printWinner: Displays a dialog announcing the game's winner. Then provides an option to start a new game or exit the application.</p> <p>printSaulsInCheck: Displays a message dialog if the "Sau" piece of either team is in check.</p> <p>handleExit: let the user select whether want to save the game or just close the application</p>	
Controller		
Input	This Input class is a controller component for handling user	

	<p>interactions such as mouse event, action events, and component events.</p> <p>Attributes: controller: A Controller class instance. Use to interact with View and Model class.</p> <p>Constructor: Initializes the Input class with a reference to the controller.</p> <p>Methods: mousePressed: Detects when the user clicks on the game board. Get the column and row of the clicked position. Select the piece at the clicked position if the piece color belongs to the current turn's player. Meanwhile highlights the valid move for the selected piece.</p> <p>mouseDragged: handles user interaction when dragging the piece. Dynamically updates the position of the dragged piece on the board during the drag operation. Meanwhile highlights valid moves for the selected piece.</p> <p>mouseReleased: Handle the release of a dragged piece. Validates the move and ensures the piece remains within the board boundaries. Updates the piece's position if the move is valid, or reverts it if invalid.</p>	<table><tr><th>Input</th></tr><tr><td>- controller: Controller</td></tr><tr><td>+ Input (controller: Controller) + mousePressed (e: MouseEvent); + mouseDragged (e: MouseEvent); + mouseReleased (e: MouseEvent); + actionPerformed (e: ActionEvent) + componentResized (e: ComponentEvent) + componentMoved (e: ComponentEvent) + componentShown (e: ComponentEvent) + componentHidden (e: ComponentEvent)</td></tr></table>	Input	- controller: Controller	+ Input (controller: Controller) + mousePressed (e: MouseEvent); + mouseDragged (e: MouseEvent); + mouseReleased (e: MouseEvent); + actionPerformed (e: ActionEvent) + componentResized (e: ComponentEvent) + componentMoved (e: ComponentEvent) + componentShown (e: ComponentEvent) + componentHidden (e: ComponentEvent)
Input					
- controller: Controller					
+ Input (controller: Controller) + mousePressed (e: MouseEvent); + mouseDragged (e: MouseEvent); + mouseReleased (e: MouseEvent); + actionPerformed (e: ActionEvent) + componentResized (e: ComponentEvent) + componentMoved (e: ComponentEvent) + componentShown (e: ComponentEvent) + componentHidden (e: ComponentEvent)					

	<p>ActionPerformed: respond to menu item selection. Start a new game if “new” is clicked. Save the current game state if “save” is clicked. Loads a previous saved game if “Open” is selected. Handel exit operation if “Exit” is clicked, offering an option to save the game before exiting</p> <p>componentResized: Adjusts the size of game board elements dynamically when the window is resized. Make the window responsive.</p> <p>componentMoved, componentShown, and componentHidden: did not have any operation.</p>	
Move	<p>This class storing the details of a piece's movement from its original position to a new position and tracks any piece that might be captured during this move.</p> <p>Attributes: oldCol: Contain the old column of piece. oldRow: Contain the old row of piece. newCol: Contain the new column of piece. newRow: Contain the new row of piece.</p>	 <pre> classDiagram class Move { # oldCol: int # oldRow: int # newCol: int # newRow: int # piece: Piece # capture: Piece + Move(controller: Controller, newCol: int, newRow: int) } </pre>

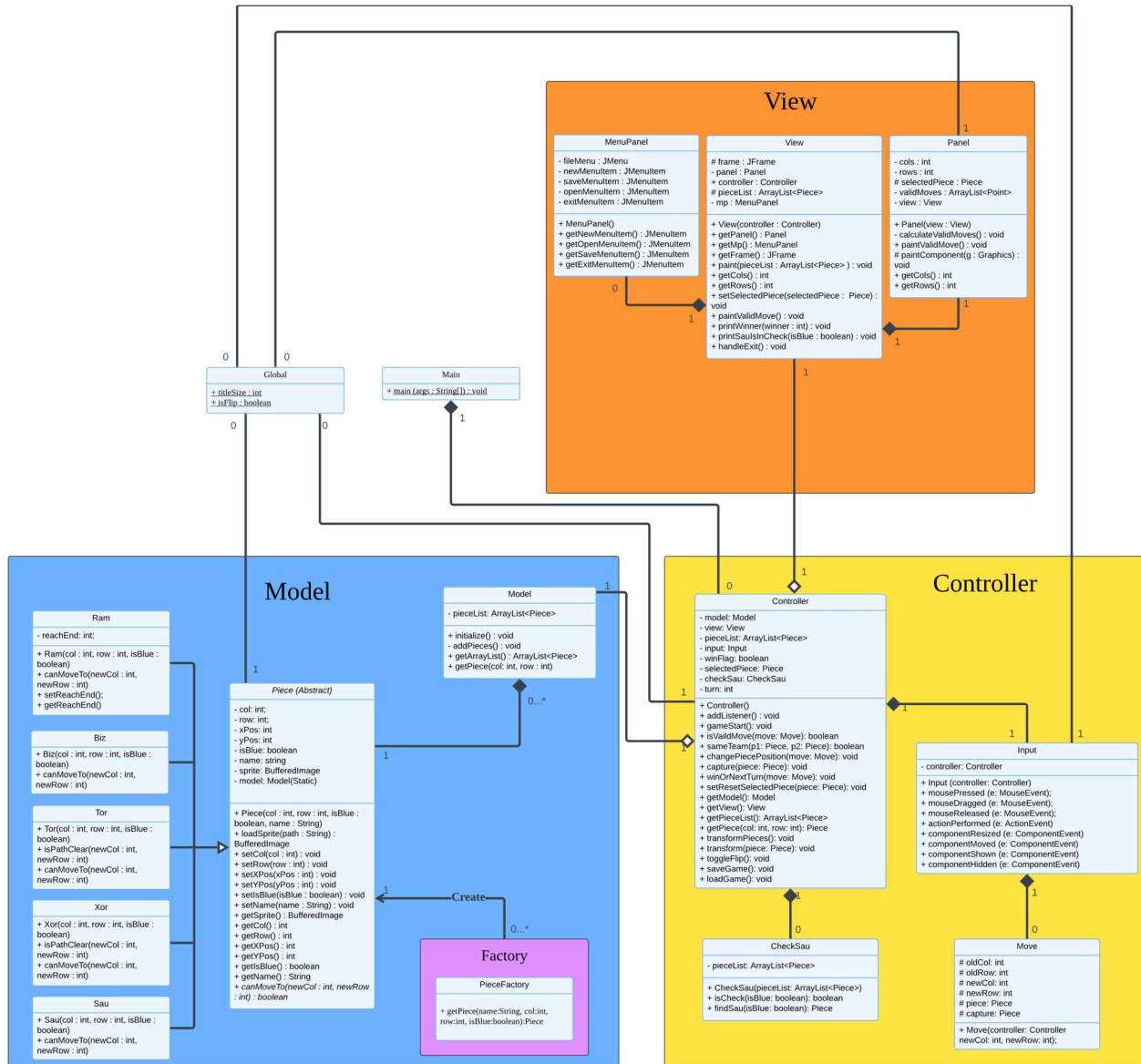
	<p>piece: An instance that represents the piece that is going to move.</p> <p>capture: An instance that represents the piece that is going to be killed.</p> <p>Constructor: Initializes the Move class with a reference to the new position and controller.</p>	
CheckSau	<p>This CheckSau class is a utility to determine whether the “Sau” piece is in check.</p> <p>Attributes: pieceList: A list of Piece that represent the current state of all pieces on the panel.</p> <p>Constructor: Initializes the CheckSau class with a reference to the pieceList.</p> <p>Methods: isCheck: Determine whether the Sau is in checked by looking the valid moves of all the pieces in panel coincide with current Sau position.</p> <p>findSau: Get Sau information from the pieceList.</p>	<div> <div>CheckSau</div> <div> - pieceList: ArrayList<Piece> </div> <div> + CheckSau(pieceList: ArrayList<Piece>) + isCheck(isBlue: boolean): boolean + findSau(isBlue: boolean): Piece </div> </div>

<p>Controller</p>	<p>The controller class acts as the mediator and observer for the whole program. It handles the events trigger by users from view class and perform actions using the algorithms provided in the model class.</p> <p>Attributes: model: an instance of Model class</p> <p>view: an instance of View class</p> <p>pieceList: an instance of pieceList copied from model</p> <p>input: an instance of Input class</p> <p>winFlag: a flag to indicate someone has won the game</p> <p>selectedPiece: to store the piece selected by player using mouse</p> <p>checkSau: an instance of CheckSau class</p> <p>Constructor: Initializes Controller class with an instance of View class and Model class, start the game and add listeners to the panel.</p> <p>Method: gameStart: Initialize everything to default value, initialize model and draw the view.</p> <p>isValidMove: return Boolean after checking if the piece being captured is not the same</p>	<div> <div>Controller</div> <div> - model: Model - view: View - pieceList: ArrayList<Piece> - input: Input - winFlag: boolean - selectedPiece: Piece - checkSau: CheckSau - turn: int </div> <div> + Controller() + gameStart(): void + isValidMove(move: Move): boolean + sameTeam(p1: Piece, p2: Piece): boolean + changePiecePosition(move: Move): void + capture(piece: Piece): void + winOrNextTurn(move: Move): void + setResetSelectedPiece(piece: Piece): void + getModel(): Model + getView(): View + getPiece(col: int, row: int): Piece + transformPieces(): void + transform(piece: Piece): void + toggleFlip(): void + saveGame(): void + loadGame(): void </div> </div>
-------------------	--	---

	<p>team and checking the valid path for the piece to move to.</p> <p>sameTeam: Return true if same team.</p> <p>changePiecePosition: move the piece to new location and update the pieceList using capture method.</p> <p>capture: set winFlag to true if Sau is captured, remove the old piece from the pieceList after moved.</p> <p>winOrNextTurn: printWinner and reset winFlag if someone won. If no one wins, add turn counter and flip the panel. If Sau is checked, printSaulsInCheck.</p> <p>setResetSelectedPiece: set the selected piece by user and inform view about it.</p> <p>getModel: get an instance of the Model class.</p> <p>getView: get an instance of the View class.</p> <p>getPieceList: to return the pieceList.</p> <p>getPiece: get the piece object in the arrayList using coordinate.</p> <p>transformPiece: find Xor and Tor every two turns from the arrayList.</p>	
--	--	--

	<p>transform: transform Xor to Tor and vice versa every two turns from the arrayList.</p> <p>toggleFlip: flip the panel and repaint it.</p> <p>saveGame: save the current state of game into a txt file.</p> <p>loadGame: load the previous state of game from a txt file.</p>	
Others		
Global	<p>Global class serves as a utility container for global variables used across the application. These variables can be access or modify by multiple class.</p> <p>Attributes: titleSize: An instance that decides the size of tiles and pieces. isFlip: An instance that decides whether that panel needs to be flipped or not.</p>	<div>Global</div> <div>+ tileSize : int (static) + isFlip : boolean (static)</div>
Main	Construct controller to start the game.	<div>Main</div> <div>+ main (args : String[]) : void (static)</div>

3.2 Class Diagram



3.3 Classes Utilizing Design Patterns

3.3.1 MVC Design Pattern

The system involves Graphical User Interface (GUI), therefore the foundational design pattern will definitely be the Mode-View-Controller Design. This can help make the program more reusable and avoid bugs.

```
public class Model {  
    private ArrayList<Piece> pieceList = new ArrayList<>();  
  
    public void initialize(){  
        pieceList.clear();  
        addPieces();  
    }  
}  
  
public Controller(){  
    this.view = new View(this);  
    this.model = new Model();  
    this.input = new Input(this);  
    turn = 0;  
    gameStart();  
    addListener();  
}  
  
public View(Controller controller){  
    this.controller = controller;  
    this.pieceList = new ArrayList<>();  
  
    frame = new JFrame(title:"Chess Game");  
    frame.setLayout(new GridBagLayout());  
    frame.getContentPane().setBackground(Color.BLACK);  
    frame.setSize(new Dimension(width:550, height:805));  
  
    panel = new Panel(this);  
    mp = new MenuPanel();  
  
    GridBagConstraints gbc = new GridBagConstraints();  
    gbc.gridx = 0;  
    gbc.gridy = 0;  
    gbc.weightx = 1.0;  
    gbc.weighty = 0.0; // Small vertical weight for the title  
    gbc.fill = GridBagConstraints.HORIZONTAL;  
    frame.add(mp, gbc);  
  
    // Add Panel in the center  
    gbc.gridx = 0;  
    gbc.gridy = 1;  
    gbc.weightx = 1.0;  
    gbc.weighty = 1.0; // Major vertical weight for the game panel  
    gbc.fill = GridBagConstraints.CENTER;  
    frame.add(panel, gbc);  
  
    frame.setLocationRelativeTo(c:null); //display in center of the screen  
    frame.setVisible(b:true);  
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
}
```

3.3.2 Singleton Design Pattern

The program involves Singleton Design Pattern, it is to ensure that any class has only one instance. Therefore, a single class is responsible for creating its own object to make sure only a single object gets created. This can clearly be seen in the model. Piece class as the model class is being constructed by the Model class itself. It is also static therefore one program can only have one model.

```

public abstract class Piece {

    private int col, row;
    private int xPos, yPos;

    private boolean isBlue;
    private String name;

    public static Model model;

    BufferedImage sprite;
}

```

3.3.3 Template Method Design Pattern

The template method design pattern is to let subclasses determine certain steps of algorithm without changing the algorithm's structure itself. This design pattern is used in the pieces class (Ram, Sau, Biz, etc.) as Piece is an abstract class. Method of Piece such as canMoveTo needs to be overridden according to the algorithm to find valid moves of each piece.

```

public abstract boolean canMoveTo(int newCol, int newRow);

@Override
public boolean canMoveTo(int newCol, int newRow) {
    int rowDiff = Math.abs(newRow - this.getRow());
    int colDiff = Math.abs(newCol - this.getCol());

    // One step in any direction
    if (rowDiff <= 1 && colDiff <= 1) {
        return true;
    }

    return false;
}

```

3.3.4 Factory Method Design Pattern

The factory pattern allows the type of object instantiated to be determined at run-time. The program uses this to spawn the pieces to the board when load game function is triggered.

```

public class PieceFactory {
    public Piece getPiece(String name, int col, int row, boolean isBlue){
        switch(name){
            case "Tor" :
                return new Tor(col, row, isBlue);
            case "Xor" :
                return new Xor(col, row, isBlue);
            case "Ram" :
                return new Ram(col, row, isBlue);
            case "Biz" :
                return new Biz(col, row, isBlue);
            case "Sau" :
                return new Sau(col, row, isBlue);
        }
        return null;
    }
}

public void loadGame(){
    String name;
    int col;
    int row;
    boolean isBlue;
    int loadTurn = 0;
    PieceFactory pieceFactory = new PieceFactory();

    try(BufferedReader br = new BufferedReader(new FileReader(fileName:"model\\src\\data.txt"))){
        pieceList.clear();
        String line;
        while((line = br.readLine())!= null){
            String[] parts = line.split(regex:",");

            if(parts.length == 4){
                name = parts[0];
                col = Integer.parseInt(parts[1]);
                row = Integer.parseInt(parts[2]);
                isBlue = Boolean.parseBoolean(parts[3]);
                pieceList.add(pieceFactory.getPiece(name,col,row,isBlue));
            }
        }
    }
}

```

3.3.5 Composite Design Pattern

The composite pattern is to treat individual objects and multiple, recursively composed objects uniformly through a common interface. The “getter” and “setter” in the program are using this design pattern. For example, Model.getPiece treats all pieces, no matter what kind of pieces they are, as the same.

```

public Piece getPiece (int col, int row){
    for (Piece piece: pieceList){
        if (piece.getCol() == col && piece.getRow() == row){
            return piece;
        }
    }
    return null;
}

```

3.3.6 Adapter Design Pattern

The adapter design is purposed to convert interface of legacy class into an interface to delegate request from the client and performs necessary conversion to allow communication. In the Model.getArrayList, it will return the whole ArrayList. The legacy ArrayList library from java will not do this, therefore this method is an adapter to return the ArrayList.

```

public ArrayList<Piece> getArrayList(){
    return pieceList;
}

```

3.3.7 Façade Design Pattern

The façade design provides a unified interface to a set of related objects in the subsystem. In this program which involves complicated algorithms to calculate the moves of the pieces, the usage of algorithms can be simplified using this design pattern. For example,

the Piece.canMoveTo method consists complicated algorithms and they are different for every pieces. However, in Panel.calculateValidMoves when calling canMoveTo, we do not need to give Piece.canMoveTo to know what piece we are having, the canMoveTo method will handle them itself.

```
private void calculateValidMoves(){
    validMoves.clear();

    if (selectedPiece != null){
        int currentRow = selectedPiece.getRow();
        int currentCol = selectedPiece.getCol();

        for (int row = 0; row < rows; row++){
            for (int col = 0; col < cols; col++) {

                if ( col == currentCol && row == currentRow){ //self position
                    continue;
                }

                boolean canMove = selectedPiece.canMoveTo(col, row);

                Piece thatPositionPiece = view.controller.getPiece(col, row);

                if (canMove && (thatPositionPiece == null || thatPositionPiece.getIsBlue() != selectedPiece.getIsBlue())){
                    validMoves.add(new Point(col, row));
                }
            }
        }
    }
}

@Override
public boolean canMoveTo(int newCol, int newRow) {

    int rowDiff = Math.abs(newRow - this.getRow());
    int colDiff = Math.abs(newCol - this.getCol());

    // Orthogonal movement for Tor
    if (this.getName().equals(anObject:"Tor") && (rowDiff == 0 || colDiff == 0)) {
        return isPathClear(newCol, newRow);
    }

    return false;
}
```

3.3.8 Observer Design Pattern

The observer Design Pattern is to provide a method to represent a one-to-many relationship between objects such that when one object is modified, dependent objects are notified automatically. This pattern is involved as MVC is used. The Controller class acts as an observer for the events triggered such as make moves, save game and load game. It will update model and view accordingly to the methods called.


```

public Controller(){
    this.view = new View(this);
    this.model = new Model();
    this.input = new Input(this);
    turn = 0;
    gameStart();
    addListener();
}

```

3.3.9 Mediator Design Pattern

The mediator design is used to reduce coupling between classes that communicate with each other. The controller is acting as the mediator between model and view. For example, when view need to get image of pieces from model, it will need to ask the controller to get the image from model to then be displayed using the view class.

```

public class Model {

    private ArrayList<Piece> pieceList = new ArrayList<>();

    public void initialize(){
        pieceList.clear();
        addPieces();
    }
}

```

Model class saves the images of pieces in the pieceList

```

public void gameStart(){

    winFlag = false;
    turn = 0;
    selectedPiece = null;
    Global.isFlip = false;

    model.initialize();
    pieceList = model.getArrayList(); // Get pieces from the model
    checkSau = new CheckSau(pieceList);
    view.paint(pieceList);
}

```

Controller class copy an instance of the pieceList

```

public View(Controller controller){

    this.controller = controller;
    this.pieceList = new ArrayList<>();
}

```

View class uses the pieceList to render the images to the screen

3.3.10 State Design Pattern

The state design pattern allows the class for an object to apparently change at run-time. In the Panel class, as the Ram will need to turnaround when reached the end of the board, it needs to downcast the Piece into Ram as only Ram has the isReachEnd attribute. It is all happening during the runtime.

```
// Draw the pieces
for (Piece piece : view.pieceList) {
    BufferedImage sprite = piece.getSprite();
    if (sprite != null) {
        int xPos = piece.getXPos();
        int yPos = piece.getYPos();

        // Check if the piece is a Ram and if it needs rotation
        if (piece instanceof Ram) {
            Ram ram = (Ram) piece; //downCasting
            if (ram.getReachEnd() == -1) { // Rotate inversely
                g2d.rotate(Math.toRadians(angdeg:180), xPos + Global.titleSize / 2, yPos + Global.titleSize / 2);
            }
        }

        g2d.drawImage(sprite, xPos, yPos, Global.titleSize, Global.titleSize, observer:null);

        // Reset rotation after drawing the Ram
        if (piece instanceof Ram && ((Ram) piece).getReachEnd() == -1) {
            g2d.rotate(Math.toRadians(-180), xPos + Global.titleSize / 2, yPos + Global.titleSize / 2);
        }
    }
}
```

3.3.11 Abstract Factory Design Pattern

The abstract factory pattern is used to provide a client with a set of related or dependent objects. In the program which need a lot of pieces with different types, abstract factory is a good design pattern to be used as no pieces can be constructed without having a type (subclass).

```
public abstract class Piece {
    private int col, row;
    private int xPos, yPos;

    private boolean isBlue;
    private String name;

    public static Model model;

    BufferedImage sprite;

    public Piece(int col, int row, boolean isBlue, String name){
        this.col = col;
        this.row = row;
        this.xPos = col * Global.titleSize;
        this.yPos = row * Global.titleSize;
        this.isBlue = isBlue;
        this.name = name;
    }
}

private void addPieces() {
    pieceList.add(new Tor(col:0, row:0, isBlue:false));
    pieceList.add(new Biz(col:1, row:0, isBlue:false));
    pieceList.add(new Sau(col:2, row:0, isBlue:false));
    pieceList.add(new Biz(col:3, row:0, isBlue:false));
    pieceList.add(new Xor(col:4, row:0, isBlue:false));

    pieceList.add(new Ram(col:0, row:1, isBlue:false));
    pieceList.add(new Ram(col:1, row:1, isBlue:false));
    pieceList.add(new Ram(col:2, row:1, isBlue:false));
    pieceList.add(new Ram(col:3, row:1, isBlue:false));
    pieceList.add(new Ram(col:4, row:1, isBlue:false));

    pieceList.add(new Tor(col:0, row:7, isBlue:true));
    pieceList.add(new Biz(col:1, row:7, isBlue:true));
    pieceList.add(new Sau(col:2, row:7, isBlue:true));
    pieceList.add(new Biz(col:3, row:7, isBlue:true));
    pieceList.add(new Xor(col:4, row:7, isBlue:true));

    pieceList.add(new Ram(col:0, row:6, isBlue:true));
    pieceList.add(new Ram(col:1, row:6, isBlue:true));
    pieceList.add(new Ram(col:2, row:6, isBlue:true));
    pieceList.add(new Ram(col:3, row:6, isBlue:true));
    pieceList.add(new Ram(col:4, row:6, isBlue:true));

    Piece.model = this;
}
```

4. Use Case diagram

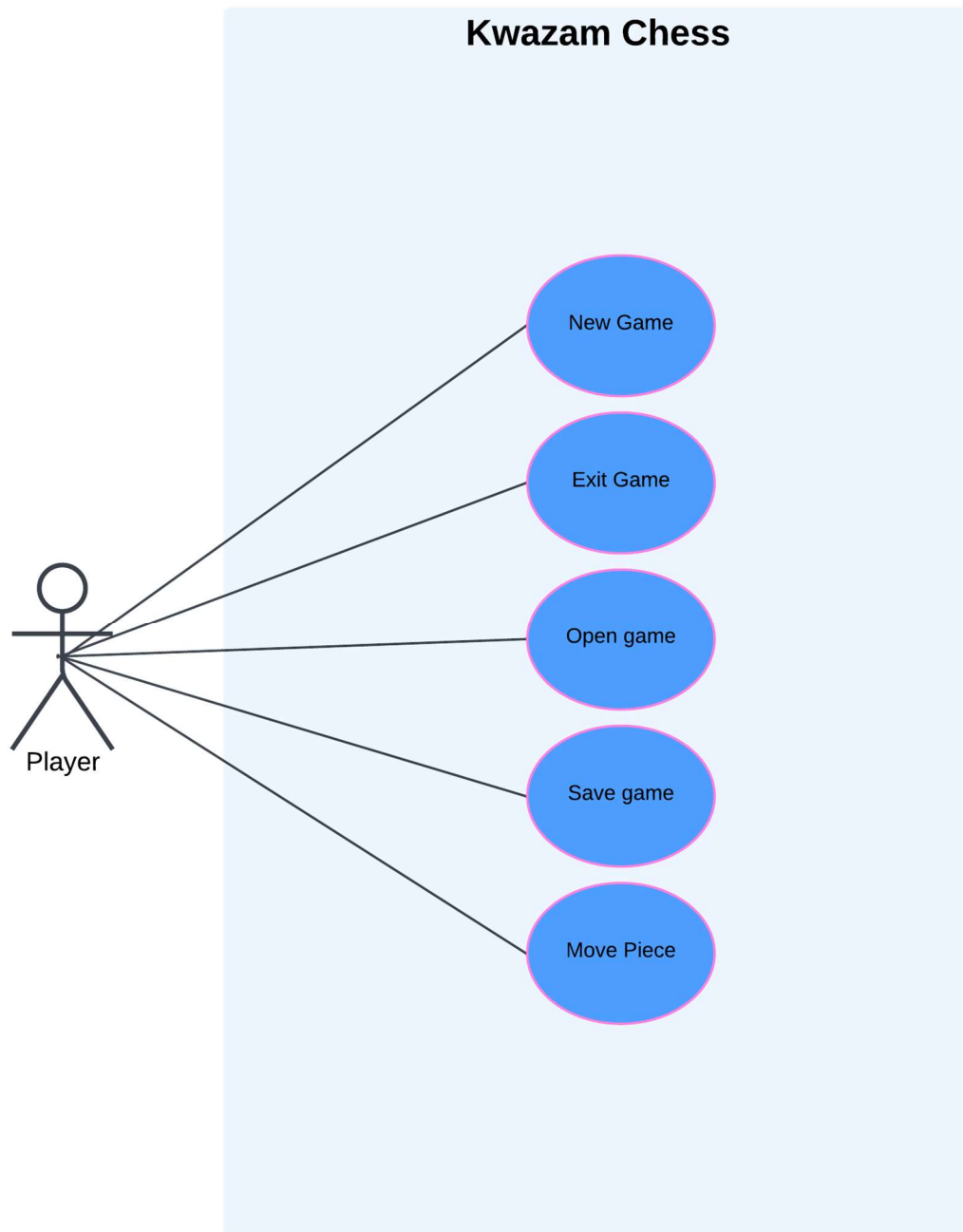
4.1 Use Case

The table below explains the use cases for the user of this software. There is only one actor, which is the Player.

Use Case	Description
New Game	The Player can start a new game where the board and pieces are in its initial state
Exit Game	The Player can exit a game whenever player want to
Open Game	The Player can open their game save files to continue games that they have not finished.
Save Game	The Player can save their current game progress if they wish to continue the game another time.
Move Piece	The Player must have already selected a piece belonging to the player. The Player can then move it to any empty tiles on the board. The action is called 'moving'.

4.2 Use case Diagram

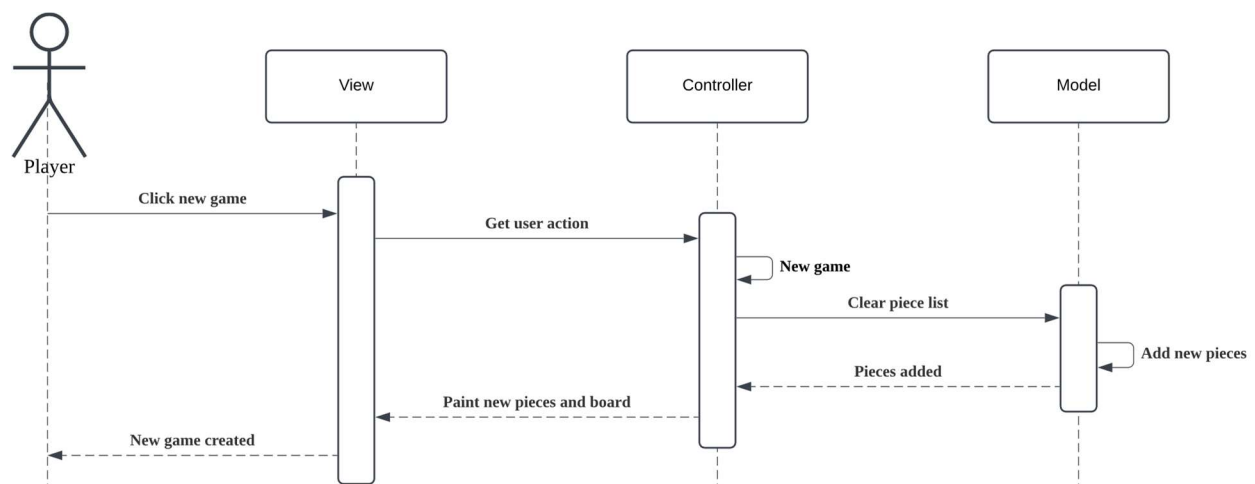
The Use Case Diagram below shows what the Player can do in our software. There are 9 use cases, and 3 of the use cases extend from the Select piece. The use cases are explained in detail in Section 3.1



5. Sequence diagrams

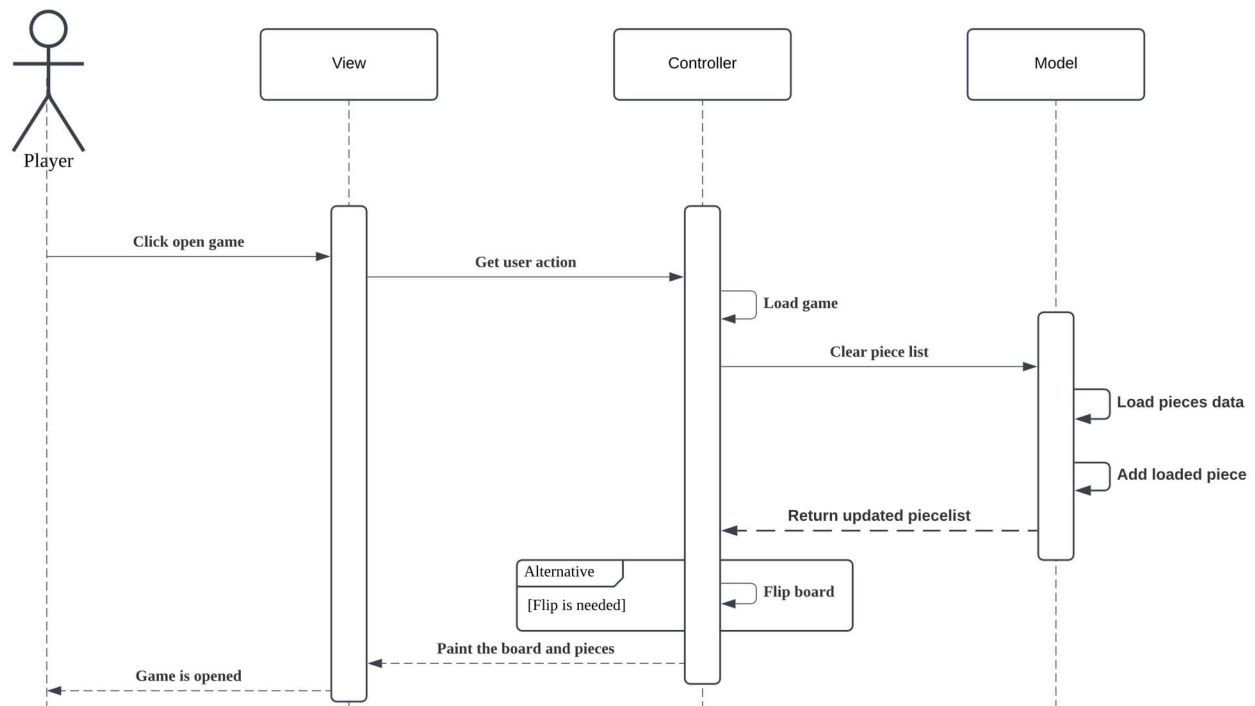
5.1 Create new game

The sequence diagram below shows that the flow of a player click “new game” option. When player clicks new game, the view will notify controller. The controller will reset the game state by clearing the old pieceList and add new pieces in model. Once the model updates the game state, controller will ask view to repaint the board and pieces. After repaint, the updated chessboard and pieces will be shown to the player.



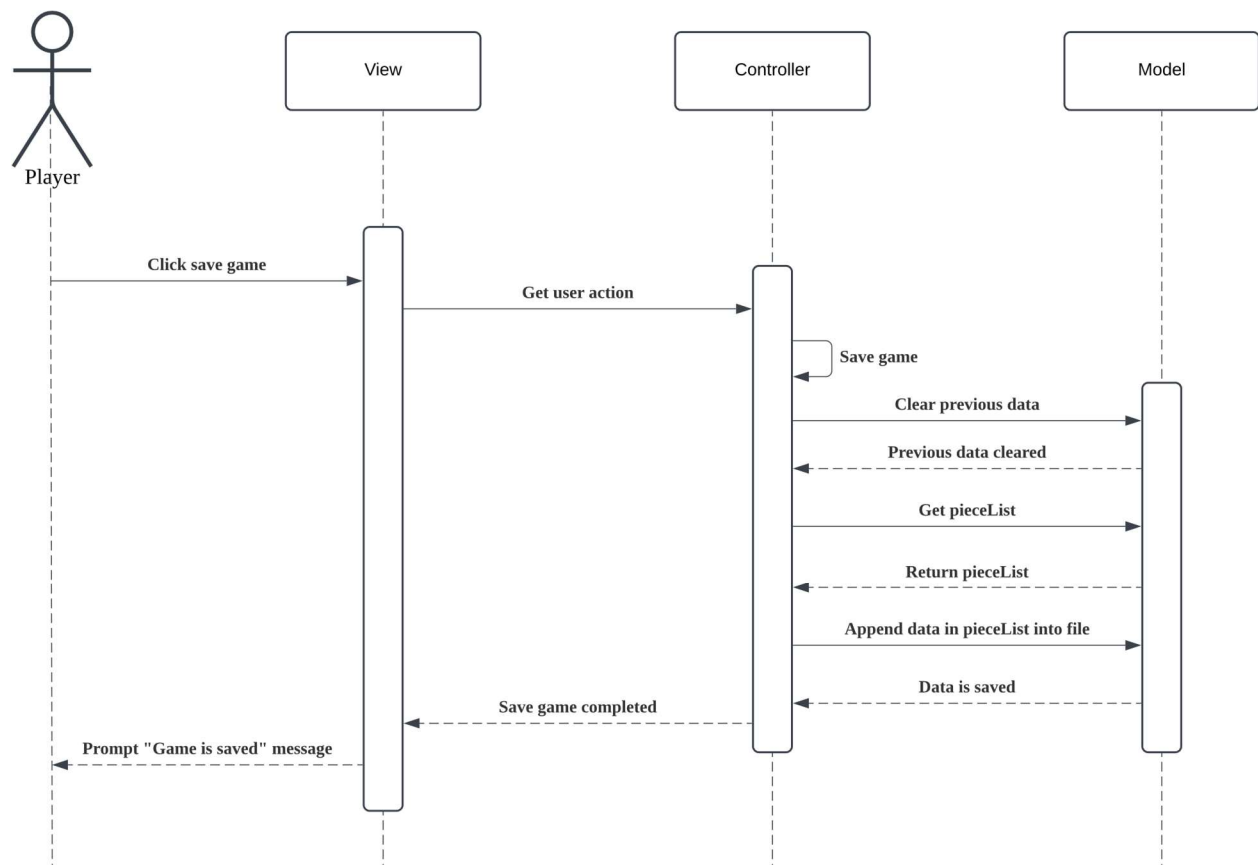
5.2 Open saved game (Load)

The sequence diagram below shows the flow of a player clicks open game option (load game). When the player clicks “open game”, the view will capture the user action and forward the action to the controller. The controller will ask the model to load the save game. Model clears the current pieceList, retrieves the saved game data which includes piece positions, turn count etc. from the file. The model will then add the saved piece into pieceList and check whether the board needs to be flipped based on the saved turn count. The controller will then update the view, ask it to repaint the chessboard and pieces to reflect the loaded game state. The loaded game will then be shown to the player.



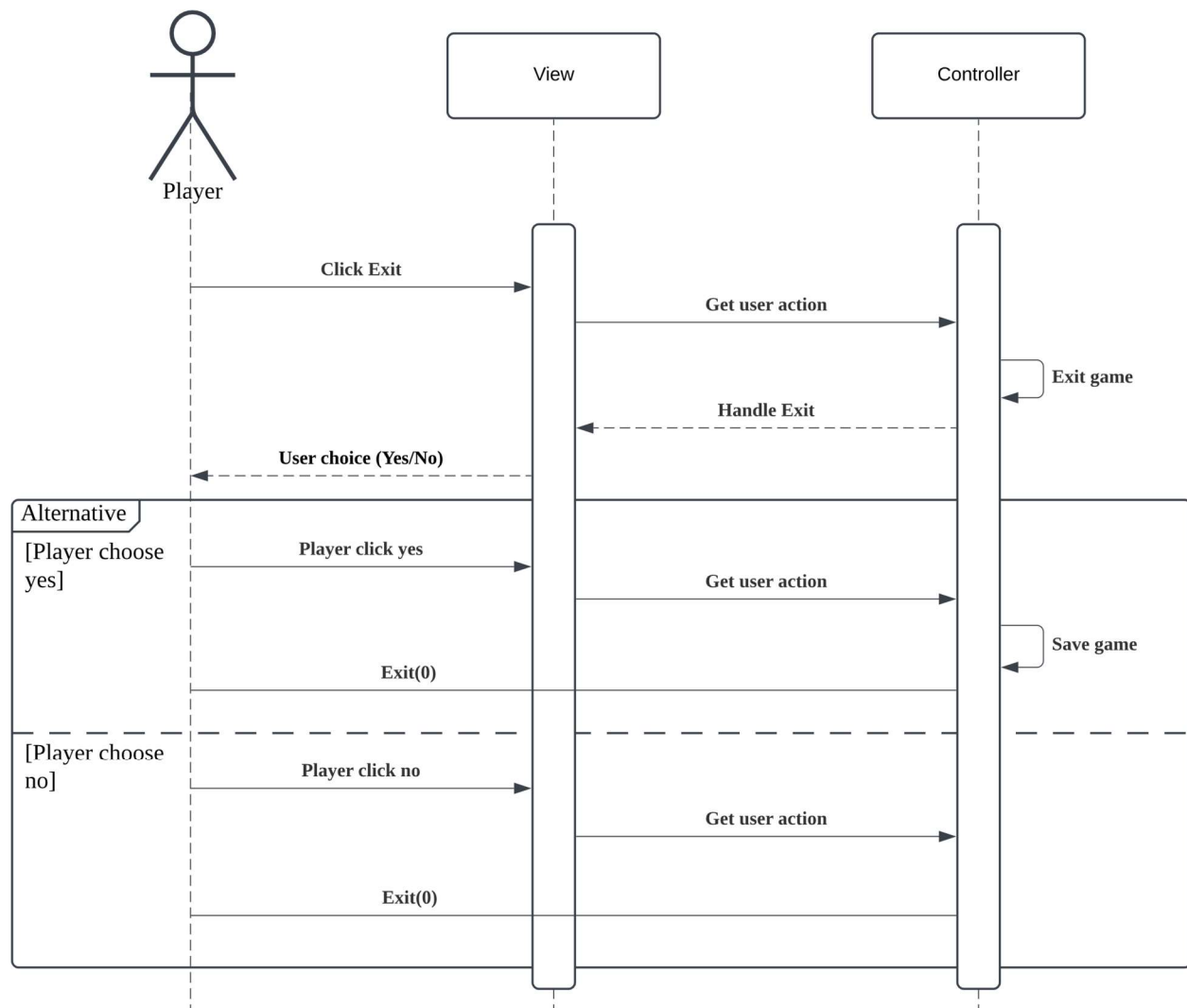
5.3 Save game

The sequence diagram below shows the flow of player clicks “save game” option. When the player clicks save game, the view will capture the user action and forward the action to controller. The controller will tell the model to clear the previous data in the file and retrieves the current data from pieceList. The data got from the pieceList will be appended into the file. Once all the data is saved into the file, the controller will display “Game is saved” message to the player through the view.



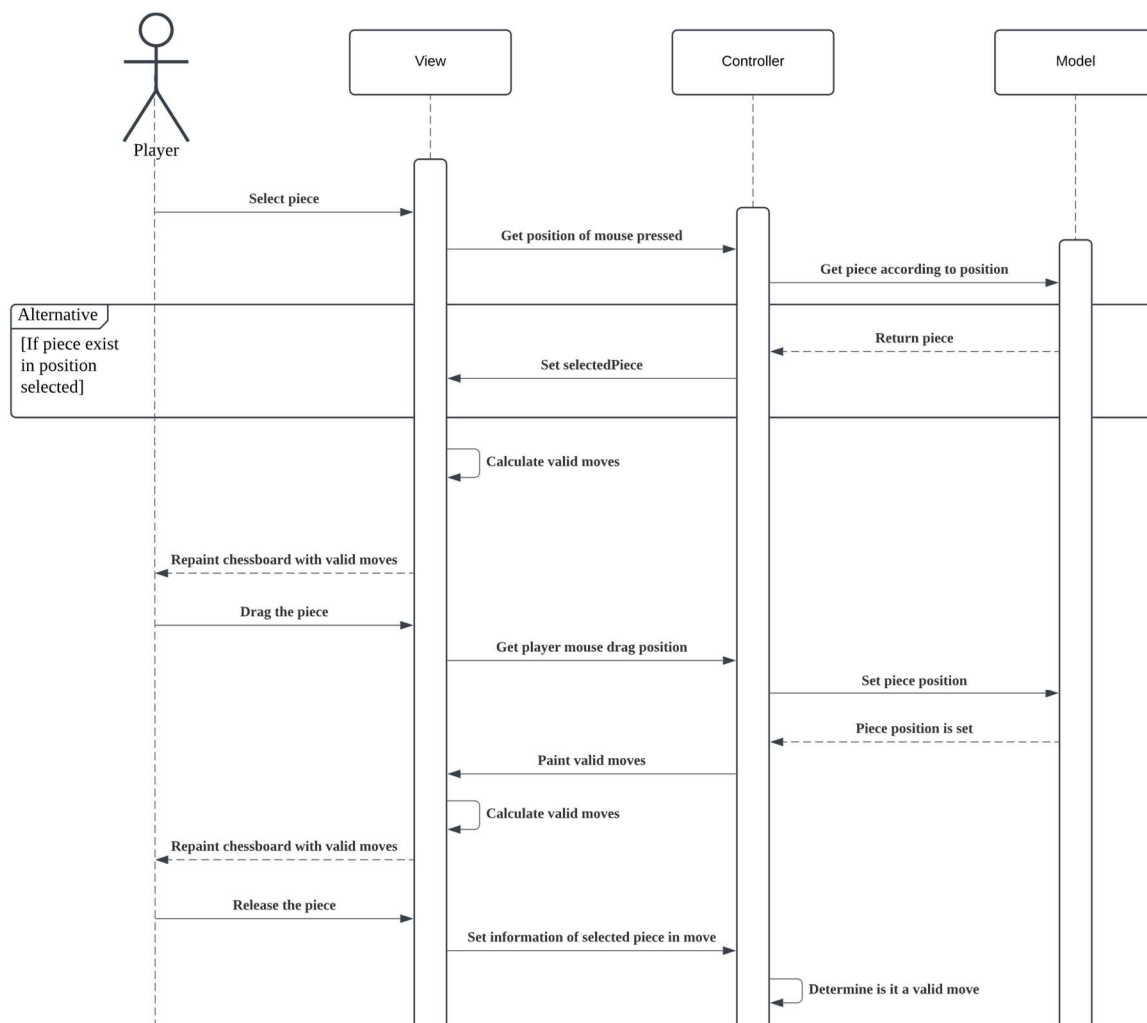
5.4 Exit game

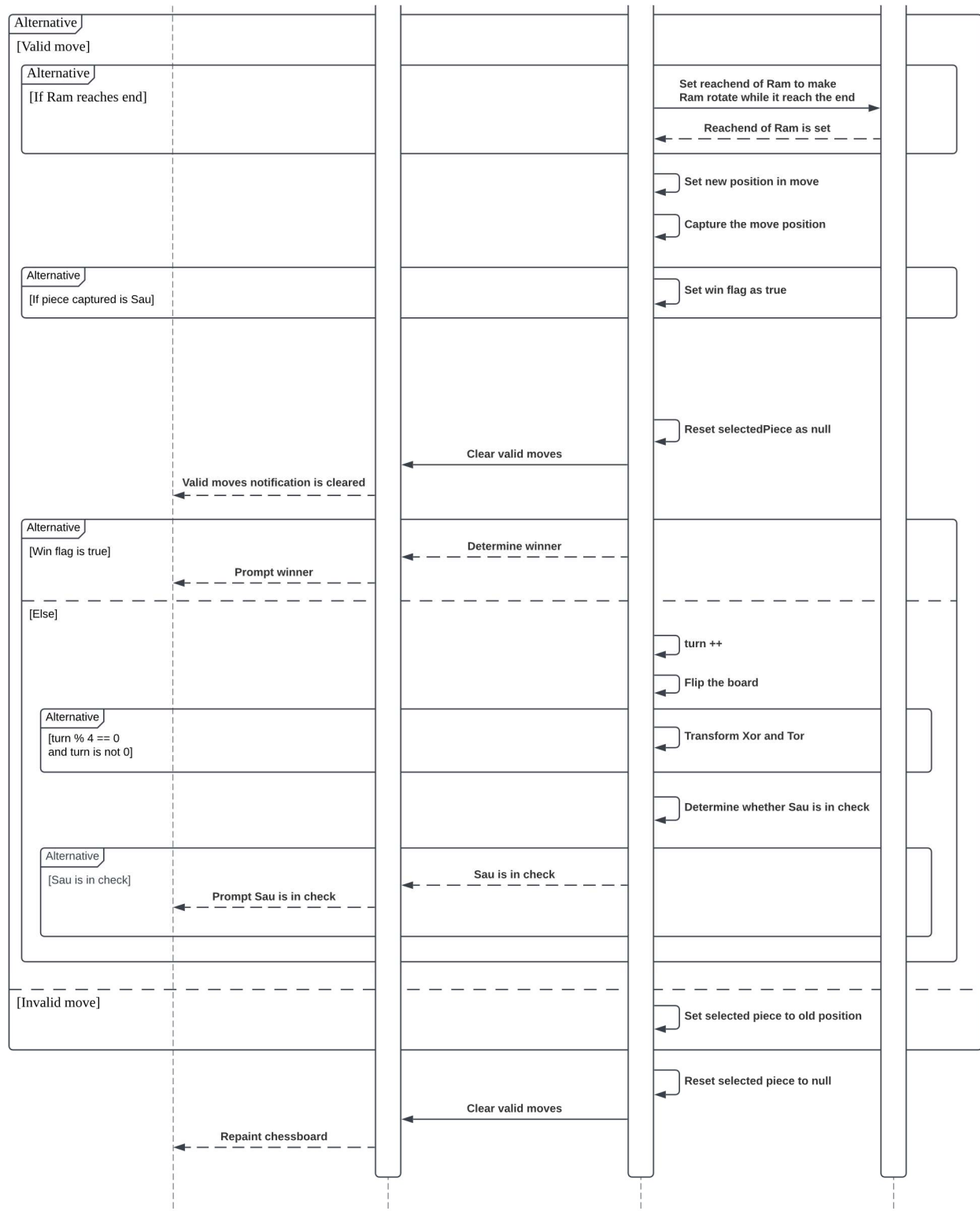
The sequence diagram below shows that the flow of player clicks exit game option. When the player clicks exit game, the view will capture the user action and forward the action to controller. The controller displays a confirmation dialog via view, ask if the player wants to save the game before exit or not. If the player chooses yes, the controller will proceed with save game and exit the application. If the player chooses no, the controller will directly exit the application without saving the game.



5.5 Move piece

The sequence diagram below shows the flow of the player doing move piece during the game. The view captures the player's mouse input, and controller processes the selected piece. The controller will then ask the view to calculate the valid moves and repaint the chessboard and pieces with valid moves for the player to see. The controller will always update the current piece position to model while player is dragging the selected piece. After the position of piece is updated, controller will ask view to repaint the chessboard and pieces with valid moves. When the player releases the piece in a certain tile, the controller will evaluate the move's validity. If the move is valid, the piece will be updated into the new position. The controller will then determine the special cases, triggering specific actions such as transforming pieces or flipping the board. The controller will also check if the Sau is in checks and determine the winner if the winning condition is met. The notification will be notified to the player through view. If the move is invalid, the controller will reset the piece to its original position.





6. User Documentation

6.1 User Interface Overview

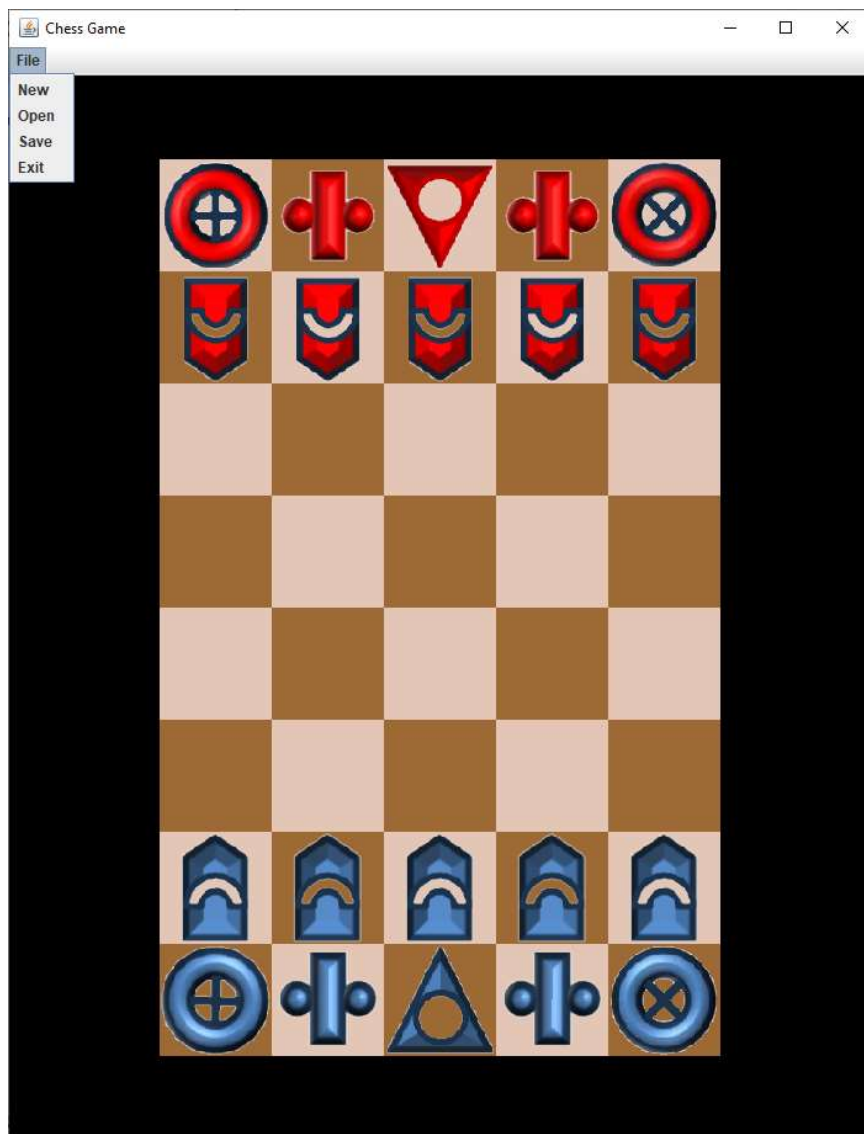
There is only 1 interface:

1. GamePage

The game page is the main interface where players interact with the chess game.

The player also can access various options through **Menu Bar** which is located at the top left of the screen.

The following image is the interface for the Game Page:



The Game Page contains 4 buttons in the **Menu**, Details are as below:

1. **New**

Players can instantly **start a new game**. The game created will set the pieces to their original position.

2. **Open**

Players can load a previously saved game.

3. **Save**

Players can save the current game progress. After pressing the Save Button, message window will pop up to tell the player the game file has been saved

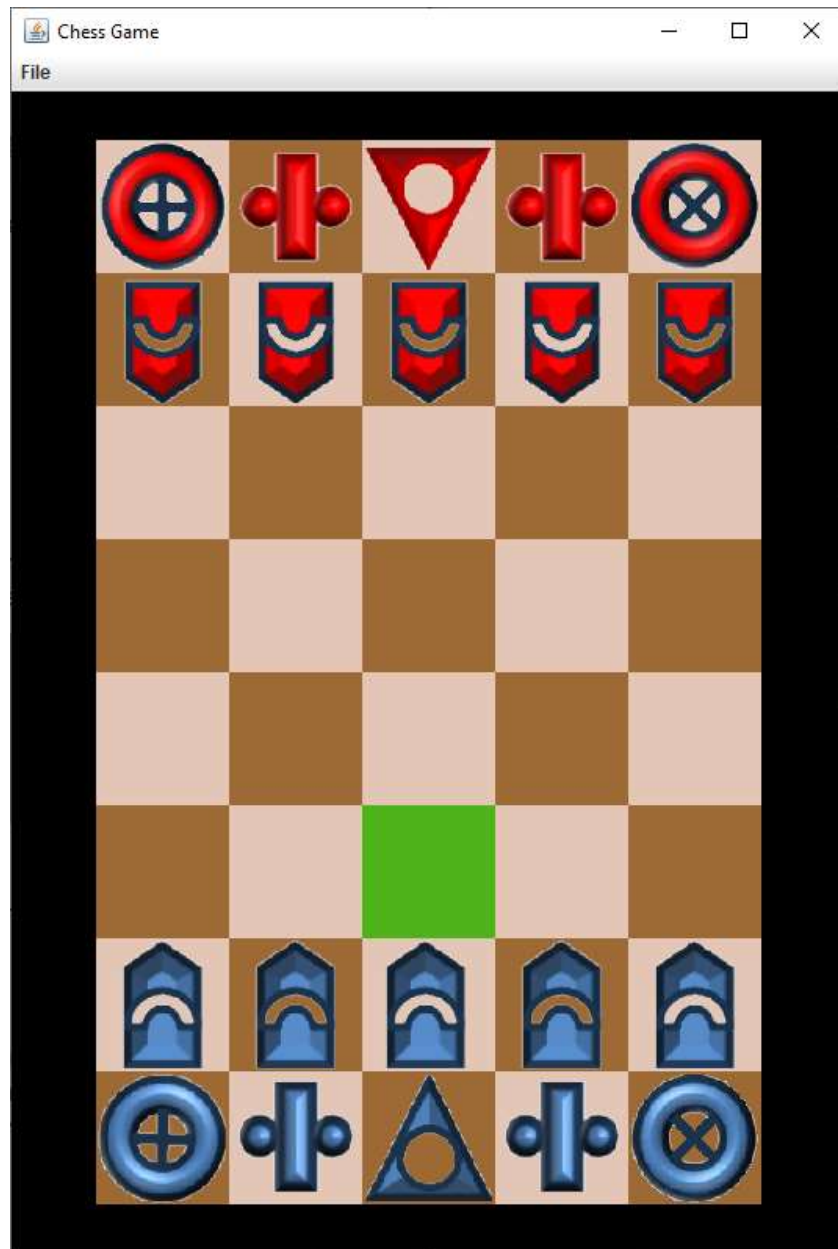


4. **Exit**

Players can close the application. After pressing the Exit Button, An Exit Confirmation popped up window will be called to confirm player action

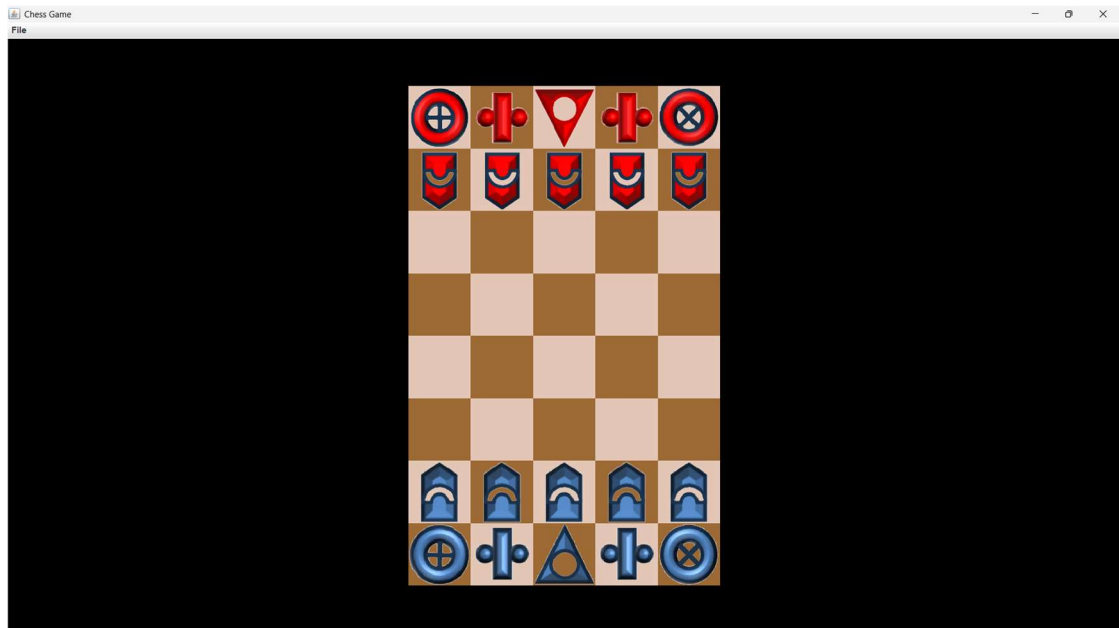
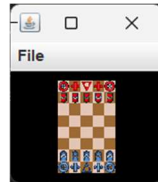


In addition, when a piece is selected, the tiles at which the piece can move to will be highlighted in green.



Resizable window

The game will be working in almost every resolution of the screen, as long as the screen is more than 1 pixel, it will actually work. If the board is too small, you can drag to resize the window bigger, if it is too big, you can drag to resize the window smaller.



6.2 User Roles and Permission

There is only one user type for this software application. The role of the user in this application is as a Player. There are 2 Players, one becomes the Blue Player and the other as the Red Player. The Blue Player always starts first. A Player's role is to play the game. Therefore, it has access to all the features provided in the Menu Bar in the Game Page interfaces.

6.3 Game Instructions

Objective

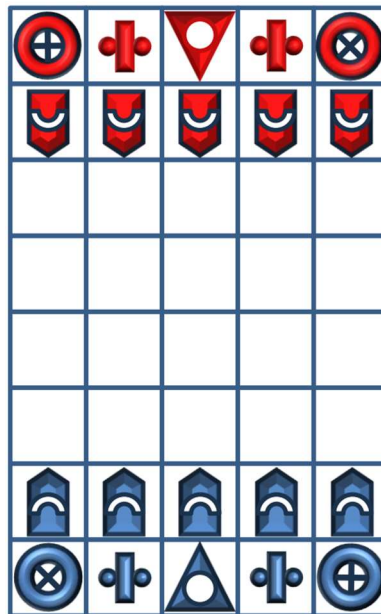
Any side who captures the opponent's Sau wins the game.

Game setup


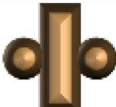



Board 5x8 grid

Two players (Blue and red)

Initial layout:



Piece movement rules

	The Ram piece can only move forward, 1 step. If it reaches the end of the board, it turns around and starts heading back the other way. It cannot skip over other pieces.
	The Biz piece moves in a 3x2 L shape in any orientation (kind of like the knight in standard chess.) This is the only piece that can skip over other pieces.
	The Tor piece can move orthogonally only but can go any distance. It cannot skip over other pieces. However, after 2 turns, it transforms into the Xor piece.
	The Xor piece can move diagonally only but can go any distance. It cannot skip over other pieces. However, after 2 turns, it transforms into the Tor piece.
	The Sau piece can move only one step in any direction. The game ends when the Sau is captured by the other side.

Note: 1 turn means 1 blue move and 1 red move.

How to play

Start: Blue moves first.

Select a piece to move: Click on the piece or drag the piece desired to move. Valid moves are highlighted in green.

Move the piece: Drag the piece to the desired box to make a move. Don't worry, invalid move will not be counted as a move.

Win the game: Protect your Sau against opponents from being captured and at the same time try to capture the opponent's Sau. You will be notified if your Sau is in danger (checked). Whoever captured the opponent's Sau wins the game.

Tips

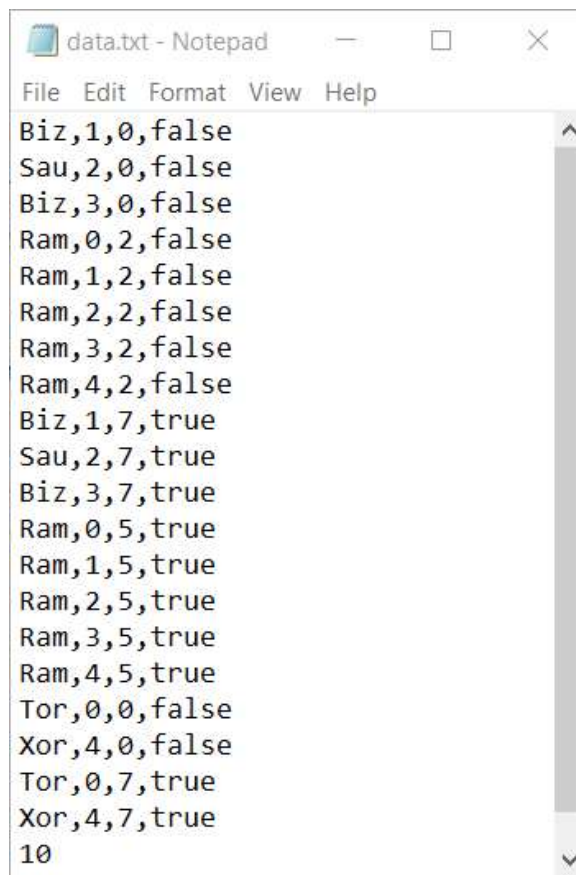
Use the special rule (transformation of Xor and Tor pieces every 2 turns, backward movement of the Ram) to attack opponent's Sau without noticing.

6.4 File Management Information

6.4.1 Save file

As section 5.1 mentioned, the saved game file is generated to save the current game state, including the position, attributes of all pieces, and the current turn, into a file **data.txt**. The save game method writes these attributes as a single line in the format. The player will be able to override the save file, the player is given the authority to override the file.

The example of a save file format is as below:



```
data.txt - Notepad
File Edit Format View Help
Biz,1,0,false
Sau,2,0,false
Biz,3,0,false
Ram,0,2,false
Ram,1,2,false
Ram,2,2,false
Ram,3,2,false
Ram,4,2,false
Biz,1,7,true
Sau,2,7,true
Biz,3,7,true
Ram,0,5,true
Ram,1,5,true
Ram,2,5,true
Ram,3,5,true
Ram,4,5,true
Tor,0,0,false
Xor,4,0,false
Tor,0,7,true
Xor,4,7,true
10
```

The following table below shows how the attributes mean in the saved file, we take two examples:

Piece Name	Column	Row	IsBlue
Biz	1	0	false
Ram	1	5	True

Biz,1,0, false: A Red Biz at column 1, row 0

Ram,1,5, true: A blue pawn at column 1, row 5.

10: Indicates it's the 10th turn.

6.4.1 Load file

As section 5.1 mentioned, Player can load the saved game state from **data.txt** by reconstructing the pieces and restoring the game turn. The loading of game file can be done through the Open Game button in Menu Panel Page which is located on the top of the game page.

