Informatics 122

Board Game Server: Final Design Document

Team 3

Bryan Linebaugh

Cassie Jeansonne

Sophia Chan

Jiahao Chen

Roy Lee

Joshwin Greene

Luke Raus

Overview

The goal of this project is to implement an extensible Board Game Server.

Requirements

- The Board Game Server will accommodate board games that involves a grid layout and game elements on this layout.
    - The games being implemented will be:
        - Chess
        - Checkers
        - Tic Tac Toe
- The Board Game Server will provide a defined interface that all game plugins must follow.
- The Board Game Server should be client-server, not Web-based. However, the server can be a "simulated" server and have the different players all on one machine.
- The Board Game Server should provide one or more ways for people to find other players.
- The Board Game Server should support personal player profiles. Login can be very simple, and does not have to be secure.
- The Board Game Server will support 2-player games.
- The Board Game Server should work by providing a player with a list of games they can play and allow them to choose which one to start.

Scope of Work

Milestones

March 1: “Plan of Attack”

- Get up-to-speed about JavaFX

March 3: Preliminary Design

March 8: Detailed Design Document and Presentation

March 10: Demo #1

- Have “server” complete & plugins launchable

March 15: Game plugins complete

March 16: Integrate plugins to the “server”

March 17: Final Presentations & Retrospective

- Final Demo
- Final Design Document
- Peer Evaluations

Assumptions

- The Board Game Server should be client-server, not Web-based. However, the server can be a "simulated" server and have the different players all on one machine.
- Users can “host” games and currently hosted games will be displayed in a list to other connected clients
- Users are paired upon a secondary user joining a hosted game, otherwise users are paired and a game doesn’t commence.

Constraints

- Scheduling conflicts of group members limits communication of design ideas and decisions.
- Remote communication of complex design decisions is difficult.

CM Repository

https://github.com/blinebau/inf122-game-server

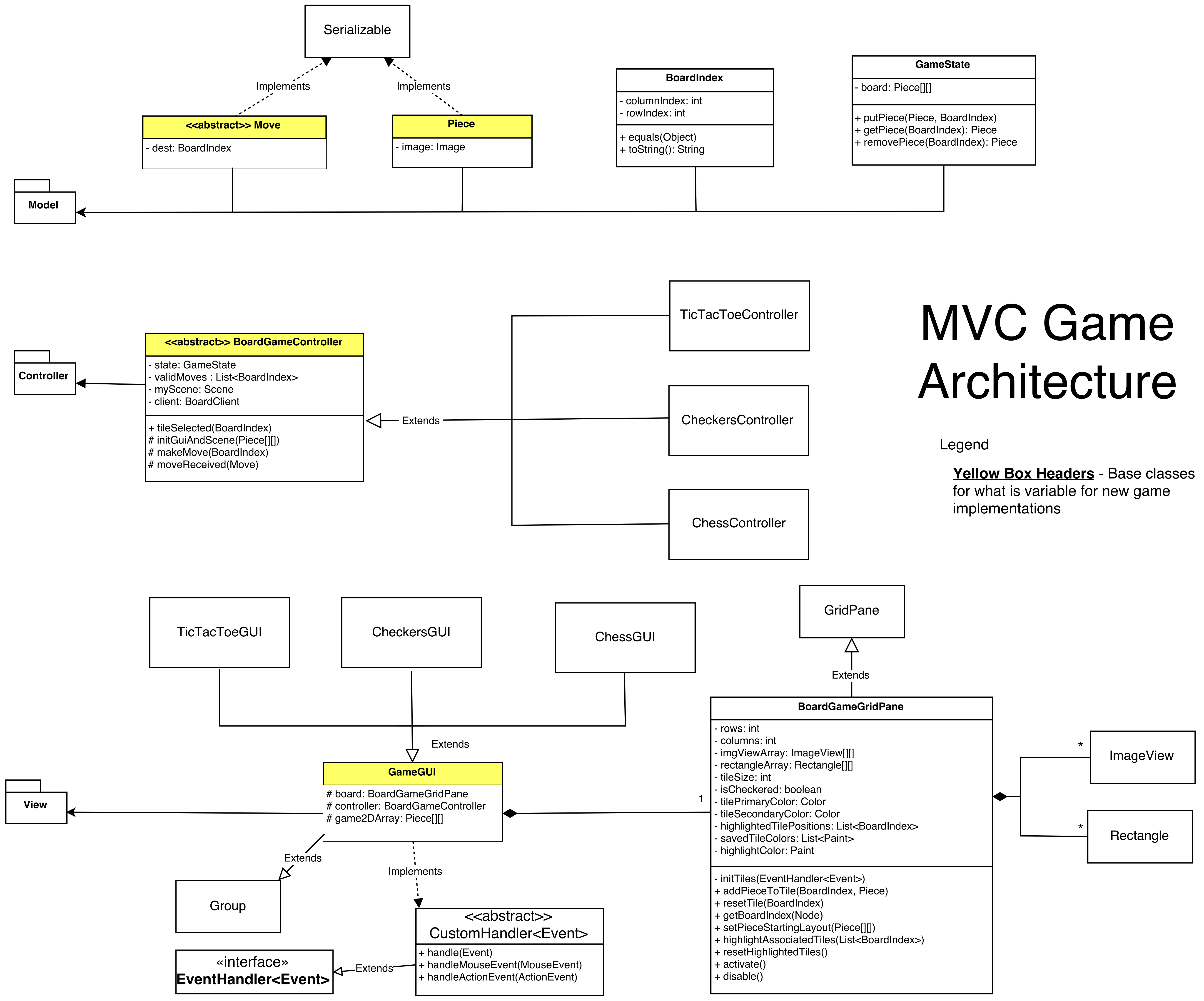
Instructions for how to run BoardGameServer

1. Run ServerGUI.jar, choose a port, and press start in order to start the server
2. Run the ClientGUI_1.jar and ClientGUI_2.jar
    a. A username should be entered into the login screens for ClientGUI instances.
3. Choose a game on one of the clients and have the other client join the same game
4. Notes
    a. If a client is exited via the x button, you should restart the server before trying to get two instances of ClientGUI to communicate with each other. It may be safer to just re-execute each jar.

MVC Game Architecture

Legend

Yellow Box Headers - Base classes for what is variable for new game implementations


Serializable
Implements
Implements
<<abstract>> Move
- dest: BoardIndex
Piece
- image: Image
BoardIndex
- columnIndex: int
- rowIndex: int
+ equals(Object)
+ toString(): String
GameState
- board: Piece[][]
+ putPiece(Piece, BoardIndex)
+ getPiece(BoardIndex): Piece
+ removePiece(BoardIndex): Piece
Model
Controller
<<abstract>> BoardGameController
- state: GameState
- validMoves : List<BoardIndex>
- myScene: Scene
- client: BoardClient
+ tileSelected(BoardIndex)
# initGuiAndScene(Piece[][])
# makeMove(BoardIndex)
# moveReceived(Move)
Extends
TicTacToeController
CheckersController
ChessController
TicTacToeGUI
CheckersGUI
ChessGUI
GridPane
Extends
Extends
GameGUI
# board: BoardGameGridPane
# controller: BoardGameController
# game2DArray: Piece[][]
View
Extends
Group
Implements
<<abstract>> CustomHandler<Event>
+ handle(Event)
+ handleMouseEvent(MouseEvent)
+ handleActionEvent(ActionEvent)
Extends
«interface» EventHandler<Event>
1
BoardGameGridPane
- rows: int
- columns: int
- imgViewArray: ImageView[][]
- rectangleArray: Rectangle[][]
- tileSize: int
- isCheckered: boolean
- tilePrimaryColor: Color
- tileSecondaryColor: Color
- highlightedTilePositions: List<BoardIndex>
- savedTileColors: List<Paint>
- highlightColor: Paint
- initTiles(EventHandler<Event>)
+ addPieceToTile(BoardIndex, Piece)
+ resetTile(BoardIndex)
+ getBoardIndex(Node)
+ setPieceStartingLayout(Piece[][])
+ highlightAssociatedTiles(List<BoardIndex>)
+ resetHighlightedTiles()
+ activate()
+ disable()
*
ImageView
*
Rectangle

General MVC Game Architecture

Class Descriptions
This section will cover the descriptions for the classes and interfaces that are a part of the MVC game architecture.

<u>Model</u>

Class: BoardIndex
Description: Implements Serializable. It functions like a Pair Class in C++, to enable easy passing of tuples. It is used to record the column and row index of objects that are interacted with on the board, which is mirrored by each game's 2D array.
Attributes:
- columnIndex: int
- rowIndex: int

Methods:
- equals(Object) - Overridden
- toString(): String - Overridden in order to print both the column and row in parentheses

Class: Move
Description: Implements Serializable. The Move class should be extended by the game plug-ins if they need to track more than just the destination, like the source for Chess and Checkers.
Attributes:
- dest: BoardIndex - used to track the destination for a given player's move

Abstract Class: Piece
Description: Superclass for all of the game pieces. It should be extended by the different game pieces.
Attributes:
- img - Image - represents the visual appearance

Class: GameState
Description: Used to manage a game's state, i.e. their underlying 2D array of Piece objects.
Attributes:
- Board: Piece[][]

Methods:
- putPiece(Piece, BoardIndex)
- getPiece(BoardIndex): Piece
- removePiece(BoardIndex): Piece

<u>View</u>

Interface: CustomEventHandler
Description: Used to give the GameGUI class the ability to handle general gui events, like MouseEvents and ActionEvents.
Abstract Methods:
- handle(Event) - Should call handleMouseEvent() or handleActionEvent() depending on the event that was fired
- handleMouseEvent()
    - Identify the object that was clicked and notify the controller
    - Implemented to support ActionEvents that are fired from custom panes and groups that have been added alongside the board in GameGUI

Abstract Class: GameGUI
Description: Implements CustomHandler. Used to give each game an interface to create their own custom gui. It contains and manages an instance of BoardGameGridPane, a reference to an instance of BoardGameController, and the game's 2D array of Piece objects. Since it is a JavaFX group, each game can add other panes and groups to GameGUI besides the actual game board, which is represented by BoardGameGridPane.
Attributes:
- board: BoardGameGridPane - the game board itself

- controller: BoardGameController - used to notify the controller
- game2DArray: Piece[][] - used to construct the board

Implemented Methods (only includes methods from interface):

- handle(Event) - does what the above contract states
- handleMouseEvent(MouseEvent)
    - Determines the type of object that the user interacted with and notifies the referenced BoardGameController if a tile has been clicked using BoardGameController's tileSelected method. Games that select pieces should override this, like Chess and Checkers.

Controller

Abstract Class: BoardGameController
Description: Used to give each game an interface to set up and manage their model and view, in addition to the ability to communicate with clients that it is paired with for game play.
Attributes:

- state: GameState
- client: BoardClient
- validMoves: List<BoardIndex>
- myScene: Scene - Used by BoardClient to set ClientGUI's scene

Abstract Methods:

- tileSelected(BoardIndex) - Should include game logic for when a tile is selected.
- moveReceived(BoardIndex) - Should include game logic for when a Move is received from the opponent
- makeMove(BoardIndex) - Should include game logic for when a move should be made. When a move is ready to be sent, it should be sent to the other player using client's sendMessage(Move) method.
- initGuiAndScene(Piece[][] game2DArray)
    - Should be used to initialize the game's gui and use it so set myScene. Other initialization modifications to the constructed gui object can be added here

How the components work together

The game architecture utilizes the Model-View-Controller architectural pattern. This architectural pattern was chosen to separate the way information is presented to and accepted bythe user. The Model section consists of classes Move, GameState, BoardIndex, and the abstract class Piece. These are all classes that are used by the game plugins. A Move object will be used to allow the games to bundle a user's move to send the moves back and forth in between clients. BoardIndex will be used to specify the location of a specific game piece, which will include the destination as an attribute specified by the row and column index. GameState keeps track of the state of the game, where pieces are moved and the abstract class Piece is available for the game plug-ins to extend to their specific game pieces.

The View section contains the interface CustomEventHandler, the class BoardGameGridPane, and the abstract class GameGUI. Each of the game GUIs will extend the abstract class GameGUI which gives an interface to each of the game GUIs to utilize and place their specialized interface for the specific game. The interface CustomEventHandler allows GameGUI to handle both MouseEvents and ActionEvents. The class BoardGameGridPane will be used by the GameGUI in order to create a grid-layout for each of the game plug-ins. GameGUI and subclasses of GameGUI are used to notify the controller when objects on the board and encompassing gui are interacted with, like tiles, pieces, or buttons on a side panel and display the board to users.
The Controller section contains an abstract class BoardGameController. All the specific games will extend this abstract class which allows for them to update the board through implementing the following abstract methods: initGuiAndScene, tileSelected(), moveRecieved, and makeMove().

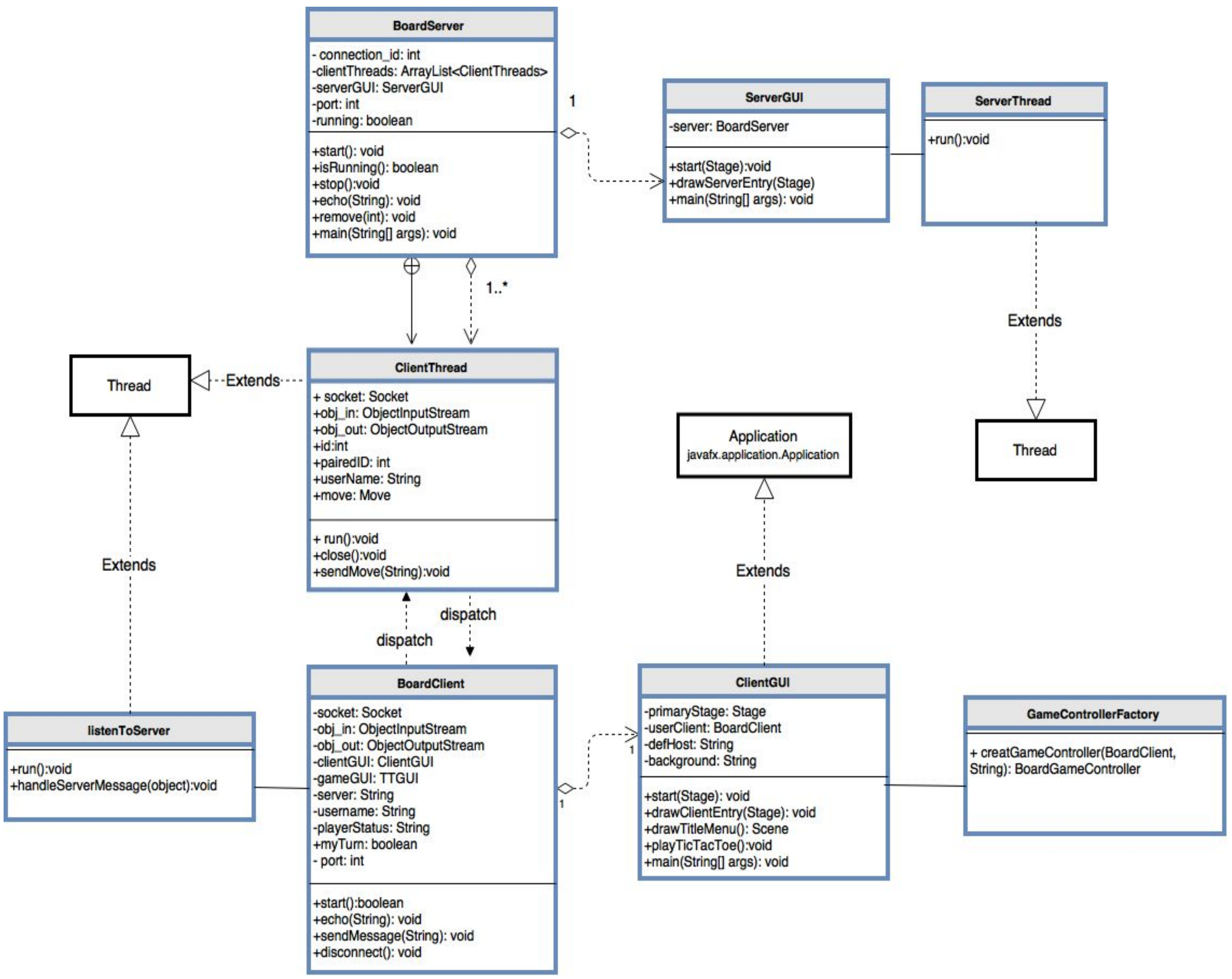
Fixed / Variable Classes

- ***Fixed***
    - **Model**
        - Move

        - BoardIndex
        - Piece
        - GameState
    - **Controller**
        - BoardGameController (including the controllers for each built-in game)
    - **View**
        - GameGUI (including the GUIs for each built-in game)
        - BoardGameGridPane
        - CustomHandler

- ***Variable*** (used to develop a new game)
    - A subclass of GameGUI
    - A subclass of BoardGameController
    - Subclass(s) of Piece
    - Subclass of Move (if needed)

Networking UML Diagram



Networking UML Diagram Description

The networking aspect of the program uses a Client-Server architecture model as per requirements of the design prompt. This architecture is facilitated using the networking capabilities of the Java.net package for implementing the functionality of a local network application. The design utilizes a MultiThreaded Server class implementation to allow for better connection requests and communication of Clients requesting connections to the server or sending messages to their counterparts. The communication between these networking components is handled through a series of endpoints in the form of Sockets existing for each of these components. In our program's design, the Server upon initialization creates a ServerSocket that Clients can connect to using their own Sockets. After the initial connection is made between the Client and the Server, the responsibility of listening and conversing via that Socket is delegated to a new Thread. So, for each new Client connection made the Server creates a new ClientThread to continuously listen to and dispatch messages between paired Clients using the Socket's stream. Each of these Clients are connected to their ClientThread on the server through their Sockets, thus a Multi-Client application requires a Multithreaded solution. Using the JavaFX library, the Client and Server will have accompanying GUIs that will be graphical representations of their statuses. In the case of the ClientGUI, it will be responsible for handling user interaction with the interface outside the scope of the game which be handled by each of the games' GUI representation.

The ClientGUI class is the main Application thread of the program utilizing the interface elements provided by the JavaFX library. Any changes or updates to the state of the GUI must be done on the same thread it is being rendered on. Additionally, in order to perform other tasks and processes the BoardClient class must be able to listen to it's Socket on a separate thread for communications from the server. This structure allows the Client to be consistently listening to any incoming message from the server and then act on them accordingly without interruption. However, since these message occur on a separate thread there is no way to affect the information and processes done by the Client onto the interface controlled by ClientGUI's Application thread. Fortunately, JavaFX provides a excellent solution to this dilemma, allowing concurrency between the Application thread of the interface and the background tasks demanded by the client. These helpful features are found within JavaFX's Concurrent package and are apart of a hierarchy of entities that provide awareness of background tasks for the JavaFX Application thread. Existing as the parent of the other classes of the package is the Worker<V> interface which provides an object representation of a background task in a program. The progress and state of a background task encapsulated within worker is observable and usable from the JavaFX Application thread. In our current design we include the implementing class Task to perform the various background tasks of writing to the server and then modifying the JavaFX Scene graph on success of that task. Using Task allows us a implementers to have background tasks run asynchronously to the Application thread allowing the Client to communicate to the Server and receive information before it is time to modify the current Scene graph. As mentioned previously, the JavaFX GUI is modified dependent upon the success of the process that is run within the Task object. The result driven programming is a helpful feature of Task class allowing implementers to set certain code to be executed based on the success, failure or cancellation of a task.

When a user wishes to play a game they have the option of deciding between hosting and joining an existing game hosted by another Client. If they wish to host a game choose the option to create a game ands select which game they would like to host. When the selection is made a communication is made to the Server that this client wishes to host a certain type of game. At this time the Server is aware of the currently connected Clients and can modify their status as hosts, broadcasting there open game to the other connected Clients. After being listed, the hosting Client is left waiting until another Client attempt to the join their game. When a separate Client attempts to join a hosted game from the listing a message is sent to the Server with message stating they wish to join the hosting Client's game. The Server then changes the ClientThread pairedID attribute to their partner's id attribute. Afterwards, a message is sent to both saying the game is ready, and then each of the Client's Scene is altered to Game's views beginning the game.

Networking Class Descriptions

This section will detail the classes involved in the Client-Server architecture of the application.

Server

- ***Class:*** BoardServer
- ***Description:*** BoardServer is the class containing the server implementation of the architecture. It contains the necessary implementation for setting up a server and the ServerSocket on a predetermined connection and port. As the server in a Socket programming application it is responsible for continually listening to new incoming connection requests and then connecting them if the request contains the correct identifying information. For each new connection BoardServer constructs a new Thread type object ClientThread that will exist to listen to it's Socket for messages from its partner Socket on the Client side of the program. Each time a new ClientThread is constructed the BoardServer add the object to a collection of currently connected threads.
- ***Attributes:***
    - connection_id: unique id for the connection
    - clientThreads: list of current connections
    - serverGUI: a ServerGUI object
    - port: port listened to in order for server to connect
    - running: boolean to determine if server is running
- ***Methods:***
    - start(): checks if the server is running, creates a server side socket, creates a client thread and saves to the ClientThread's arraylist. Closes the server once the activity is complete.
    - isRunning(): checks if the server is running
    - stop(): checks if the server is running, if false stops the server.
    - echo(message): echos message from the server
    - remove(connection_id): checks if the thread is in the arraylist and removes it
- ***Inner Class:*** ClientThread
- ***Description:*** ClientThread is an inner class extending Thread located within the outer class BoardServer. Instances of ClientThread are instantiated by BoardServer upon acceptance of a new incoming connection from a Client. Upon construction BoardServer passes in the Socket initialized by accepting connections from the ServerSocket to the ClientThread constructor. Each ClientThread has a unique Socket that will read to and write from over the course of its life cycle. Utilizing the I/O of the Socket's ObjectStream the ClientThread will perform various modifications to its attributes and send messages to their Client

based on the messages read from the stream. This Class paired with the BoardClient's inner class ListenToServer are the primary paths of communication in our application's Client-Server architecture.

- ***Attributes:***
    - socket: Socket Object
    - Obj_in: ObjectInputStream Object that allows socket to listen
    - obj_out: ObjectOutputStream Object that allows socket to write
    - id: unique id for a particular thread, allows for sending of messages and disconnecting
    - pairedID: unique id for a particular client's paired client
    - userName: user of the client
    - move: Move Object which captures a move made by a user
- ***Methods:***
    - run(): keeps the server running indefinitely, checks message coming from server and activates the appropriate thread for a particular game. Once complete closes everything.
    - close(): close all connections
    - sendMove(): if client is connected, send message and write message to stream. Writes all active threads to the stream.
- **Class:** ServerGUI
- **Description:** ServerGUI is a graphical interface representation of the BoardServer's status using the JavaFX library. The interface will include logging of status and connections, buttons for starting and stopping the server on a particular port.
- ***Attributes:***
    - server: BoardServer Object
    - eventLog: TextArea Object
- ***Methods:***
    - start(): creates a server entry stage
    - drawServerEntry(): creates and displays the stage title, image, and other labels for the server entry GUI.
- **Inner Class:** ServerThread
- ***Description:*** ServerThread is a simple class containing implementation for starting a BoardServer object on a thread separate to the ServerGUI Application thread using BoardServer's start() method.
- ***Attributes:***
    - None
- ***Methods:***
    - run(): starts the server

Client

- ***Class:*** BoardClient
- ***Description:*** BoardClient exists as the communication medium between the Server and what is happening when a user interacts with the application interface. The BoardClient class contains the implementation for creating the Socket that will request a connection to the ServerSocket in the BoardServer class. This process doesn't occur until a user signs into the application. The BoardClient object is "started" when an event indicating the user has clicked on the signIn Button located in the ClientGUI class. ClientGUI then attempts to "start" the BoardClient object by invoking its start() method. The start() method sends the connection request to ServerSocket from the BoardClient's Socket. Upon success of the connection the start() method returns true to ClientGUI indicating that BoardClient did indeed connect to the BoardServer. During this connection the BoardClient creates a Thread type object ListenToServer that is very similar to the ClientThread found within BoardServer. ListenToServer does exactly what it is named for, it listens to the BoardClient's Socket on a separate thread and interprets the message sent. Based on the messages sent from the BoardServer, ListenToServer using BoardClient message sending method to converse with the BoardServer or modifies various attributes of the BoardClient.
- ***Attributes:***
    - socket: Socket Object
    - Obj_in: ObjectInputStream Object that allows socket to listen
    - obj_out: ObjectOutputStream Object that allows socket to write
    - clientGUI: clientGUI Object
    - boardGameController: BoardGameController Object
    - server: server string

- username: user of server
    - port: port listened to in order for server to connect
    - playerStatus: state of player in game
    - myTurn: boolean that holds player term
    - hostClients: list of host clients
- ***Methods:***
    - start(): starts the client dialog
    - echo(message): echos message from the server
    - sendMessage(message): sends message with server status
    - sendMessage(move): sends a message with a move
    - disconnect(): close all connections
- ***Inner Class:*** ListenToServer
- ***Description:*** ListenToServer is an inner class extending Thread located within the outer class BoardClient. An instance of ListenToServer is instantiated by BoardClient upon successfully connecting to the BoardServer. ListenToServer will use BoardClient's unique Socket to read from and write to over the course of its life cycle. Utilizing the I/O of the Socket's ObjectStream the ListenToServer will perform various modifications to BoardClient's attributes and send messages to their Server based on the messages read from the stream. This Class paired with the BoardServer's inner class ClientThread are the primary paths of communication in our application's Client-Server architecture.
- ***Attributes:***
    - None
- ***Methods:***
    - run(): starts the server
    - handleServerMessage: handles server messages to track user moves
- ***Class:*** ClientGUI
- ***Description:*** ClientGUI is the graphical interface for the main application window and the various views outside of various games. The ClientGUI uses the JavaFX library for its graphical interface and is the Application thread for the Client side of the program. The various views it is responsible for drawing include the sign in and menu screens. The class also includes the implementation for handling events related to the interactive elements of those screens. It is responsible for interpreting a user's game selection on the main menu and creating the appropriate game controller based on their selection. During its construction the ClientGUI constructs a BoardClient object using itself and networking elements as arguments for it's constructor. During the initializing of a game the ClientGUI uses the communication methods of the BoardClient class to send messages to the Server indicating that the user has chosen to play a particular game.
- ***Attributes:***
    - stage: Stage Object
    - entryScene: SceneObject
    - userClient: BoardClientObject
    - defHost: contains host definition
    - defPort: contains port definition
    - lobby: a list of users in the lobby
    - hostClients: a list of clients being hosted
    - ROW_HEIGHT: final variable containing row height
    - Background: contains brackground image
- ***Methods:***
    - start(): creates a server entry stage
    - drawClientEntry(): creates and displays the stage title, image, and other labels for the client entry GUI.
    - drawTitleMenu: creates and displays the menu with relevant information
    - playTicTacToe: allows user to play tic tac toe
    - playChess: allows user to play chess
    - playCheckers: allows user to play checkers
    - readFile: Reads from a text file containing a listing of available games on the Board Server
- ***Inner Class:*** HostCell
- ***Description:***
- ***Attributes:***

    - cell: HBox Object
    - label: Label Object
    - pane: Pane Obejct
    - button: Button Object
    - lastCell:
- ***Methods:***
- updateItem(hostName, empty):

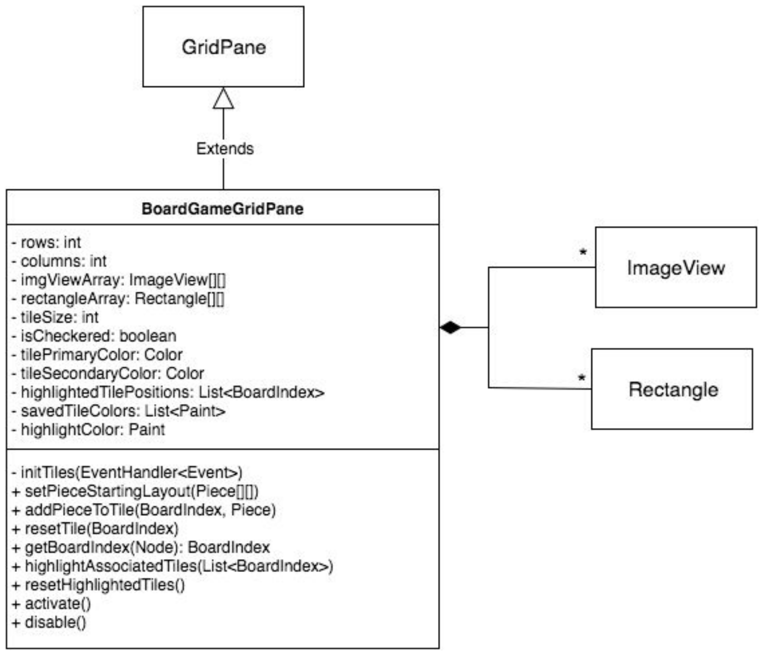
<u>Factory</u>

- ***Class:*** GameControllerFactory
- ***Description:*** Creates the controller for the appropriate game trying to be played.
- ***Attributes:***
    - none
- ***Methods:***
    - createGameController(client, gameName): uses conditional to choose correct game client based on the game name selected by user.

Construction of Game Controllers

The controllers for each of the game's will be constructed within the ClientGUI class. The construction of a particular game's controller occurs after the source of the MouseEvent is interpreted by the Application Thread. The application captures the event by using a list of buttons as a selection for what game the user wants to play. Upon interacting with one of the buttons the Application thread will utilize a ButtonClicked() method. The ButtonClicked() method is called by default whenever a Button existing in the JavaFX Scene graphs is clicked. Using the capabilities of JavaFX, an implementer is able to use the source of the click, for comparison, to determine which game button was selected by the user. Upon determination, the executed code uses a factory method pattern to construct the game's controller. Upon construction, the Application thread's Stage object is passed to be modified using the game's Scene. After the process is completed, the game is ready and the user can begin playing the game.

BoardGameGridPane UML Diagram


GridPane
Extends
BoardGameGridPane
- rows: int
- columns: int
- imgViewArray: ImageView[][]
- rectangleArray: Rectangle[][]
- tileSize: int
- isCheckered: boolean
- tilePrimaryColor: Color
- tileSecondaryColor: Color
- highlightedTilePositions: List<BoardIndex>
- savedTileColors: List<Paint>
- highlightColor: Paint
- initTiles(EventHandler<Event>)
+ setPieceStartingLayout(Piece[][])
+ addPieceToTile(BoardIndex, Piece)
+ resetTile(BoardIndex)
+ getBoardIndex(Node): BoardIndex
+ highlightAssociatedTiles(List<BoardIndex>)
+ resetHighlightedTiles()
+ activate()
+ disable()
*
ImageView
*
Rectangle

<u>BoardGameGridPane Documentation</u>

Class: BoardGameGridPane
Description: Offers a standard way to create a grid-based game board. Tiles are represented by Rectangles whereas game pieces are represented by ImageViews. In each space of the grid, you will find an ImageView on top of a Rectangle. In its constructor, hgap and vgap refer to the amount of space in between the columns (hgap) and rows (vgap).
Attributes:

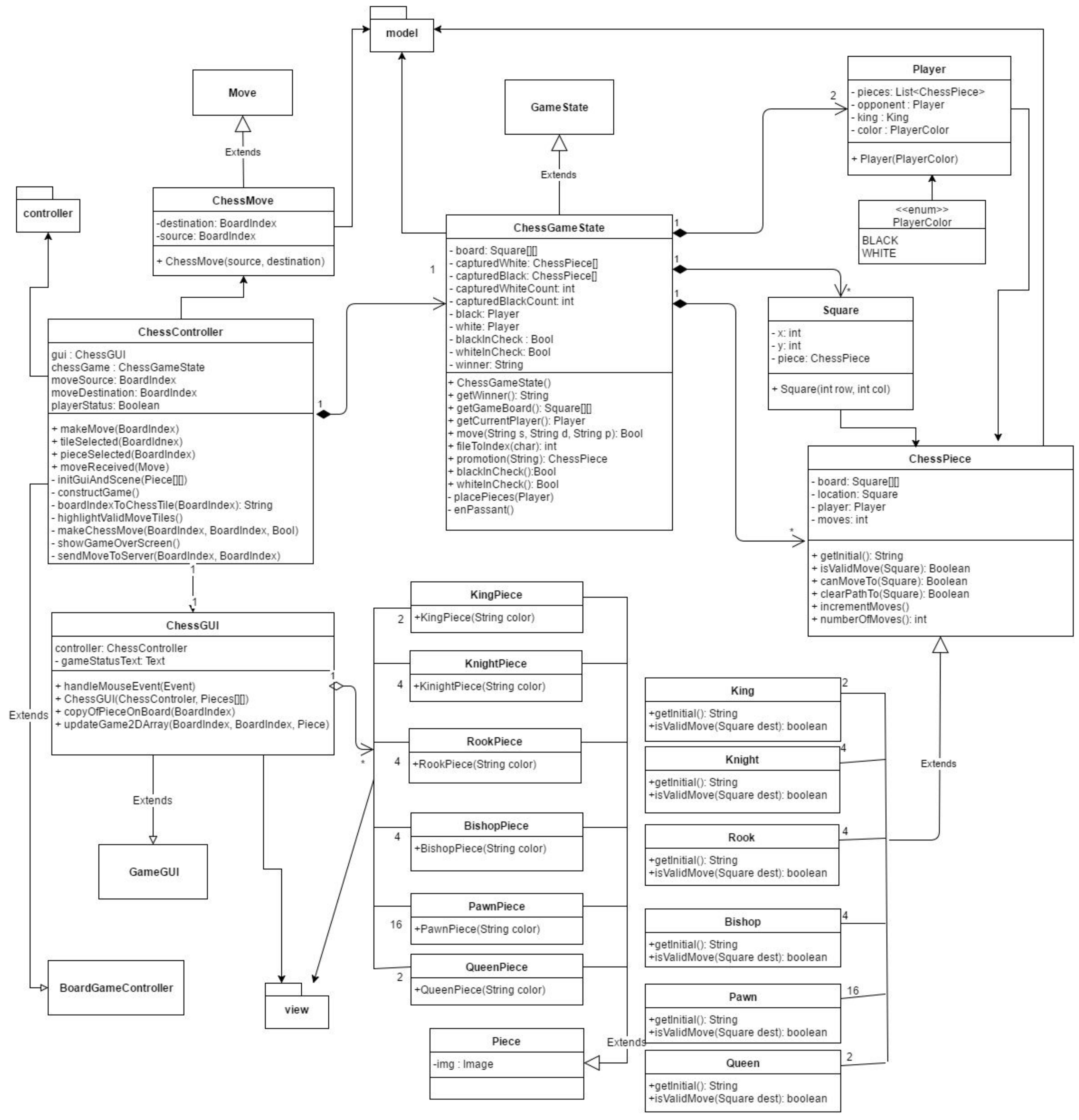
- rows: int - # of rows
- columns: int - # of columns
- imageViewArray: ImageView[][]
    - An ImageView's image will be either null (empty tile) or the image of a particular piece.
- rectangeArray: Rectange[][]
- tileSize: int
- isCheckered: boolean - Used to determine if the board should be checkered or a constant color
- tilePrimaryColor: Color
- tileSecondaryColor: Color
- highlightedTilePositions: List<BoardIndex>
- savedTileColors: List<Paint> - Used to save the color of tiles when a checkered board is being used
- highlightColor: Paint

Methods:

- initTiles(EventHandler<Event>)
    - Initializes the board's Rectangles and ImageViews that fill each space of the grid. They are recorded to their associated array so that they can be altered easily. When adding Rectangles, it will set their color / fill based on whether isCheckered is true. If it is true, it will fill the board using both the primary and secondary color in a checkered arrangement whereas it will only use the primary color if isCheckered is false. It also sets the rectangle width and height using the tileSize. It will also add an event handler to each rectangle/imageview that is added. This is done in order for each game's gui to be able to serve as the handler for events that are fired by these objects when they are interacted with by the player.
- setPieceStartingLayout(Piece[][])
    - Use the array's pieces to set the images of their corresponding ImageViews
- addPieceToTile(BoardIndex, Piece) - Set the corresponding Imageview's image to the Piece's image
- resetTile(BoardIndex) - Set the corresponding ImageView's image to null (default state)
- getBoardIndex(Node): BoardIndex - Returns a BoardIndex when given a Node that is on the board

- highlightAssociatedTiles(List<BoardIndex>)
    - Highlight the tiles that correspond to the given BoardIndexes
- resetHighlightedTiles() - Reset the highlighted tiles
- activate() - Activates the board so that users can interact with it
- disable() - Disables the board (ex. use case - it isn't the player's turn)

Chess UML Diagram



Chess Documentation

Class: ChessController extends BoardGameController
Description: The main controller for the chess game. It holds the games GUI and game state. It also makes moves updating both the GUI, State, and sending the move to the client socket accordingly.
Attributes:

- gui : ChessGUI
- chessGame : ChessGameState
- moveSource: BoardIndex
- moveDestination: BoardIndex
- playerStatus: Boolean

Methods:

- makeMove(BoardIndex)
- tileSelected(BoardIdnex)
- pieceSelected(BoardIndex)
- moveReceived(Move)
- constructGame()
- initGuiAndScene(Piece[][]): this initializes the games GUI and scene based on the playerStatus
- boardIndexToChessTile(BoardIndex): This method converts the BoardIndex and converts it to chess official locations for the ChessGameState (4,6) = e2
- highlightValidMoveTiles()
- makeChessMove(BoardIndex, BoardIndex, Bool)
- showGameOverScreen()
- sendMoveToServer(BoardIndex, BoardIndex)

Class: ChessGUI extends GameGUI
Description: This creates and stores the JavaFX scene for Chess based on the Pieces[][] that are passed to it. It also handles mouse events and updates the BoadGameGridPane that is displaying the pieces to the user whenever a move happens.
Attributes:

- controller: ChessController
- gameStatusText: Text

Methods:

- handleMouseEvent(Event)
- ChessGUI(ChessControler, Pieces[][])
- copyOfPieceOnBoard(BoardIndex)
- updateGame2DArray(BoardIndex, BoardIndex, Piece)

Class(es): KingPiece, KnightPiece, RookPiece, BishopPiece, PawnPiece, QueenPiece
Description: These classes all extend Piece (from our overall MVC classes). This is used to set the ChessGUI's game2DArray with Piece objects that all have a set image based on the class type and the color passed to them in the constructor. Whenever updateGame2DArray(...) is called in ChessGUI these are the classes that are moved around and updated int game2DArray which are then drawn on the user's scene with BoadGameGridPane.
Attributes:

- none

Methods:

- Piece(String color)

Class: ChessGameState extends GameState
Description: This holds the state or model of the game. The main method used by ChessController is the move(String s, String d, String p) which takes a chess define source and destination (e2 -> e4); it checks if the move is valid then updates the board model which is Square[][]. Other methods and attributes are not really used by the ChessController except getWinner(). This model was found on GitHub and was easily integrated into our design with little change.
Attributes:

- board: Square[][]
- capturedWhite: ChessPiece[]
- capturedBlack: ChessPiece[]
- capturedWhiteCount: int
- capturedBlackCount: int
- black: Player
- white: Player
- blackInCheck : Bool
- whiteInCheck: Bool
- winner: String

Methods:
- ChessGameState()
- getWinner(): String
- getGameBoard(): Square[][]
- getCurrentPlayer(): Player
- move(String s, String d, String p): Bool
- fileToIndex(char): int
- promotion(String): ChessPiece
- blackInCheck():Bool
- whiteInCheck(): Bool
- placePieces(Player)
- enPassant()

Class: ChessMove extends Move
Description: This is the actually move object that is sent over the client socket. This is how games can talk to each other and inform each other of the moves made by the other player. The game only needs a source and destination coordinate to move
Attributes:
- destination: BoardIndex
- source: BoardIndex

Methods:
- ChessMove(source, destination)

Class: Square
Description: This represents a tile in the ChessGameState model and contains and x and y location and a ChessPiece if it is occupied
Attributes:
- x: int
- y: int
- piece: ChessPiece

Methods:
- Square(int row, int col)

Class: ChessPiece
Description: This is the base class for all chess pieces that are modeled in the game. It has an abstract isValidMove(Square dest) and getInitial() class which the extending classes override. This is so when that which move checks if a destination for a piece is valid it can always call isValidMove(Square dest) on the ChessPiece object.
Attributes:
- board: Square[][]
- location: Square
- player: Player
- moves: int

Methods:

- getInitial(): Returns the letter representation of the ChessPiece (K, Q, R….)
- isValidMove(Square): Boolean
- canMoveTo(Square): Boolean
- clearPathTo(Square): Boolean
- incrementMoves()
- numberOfMoves(): int

Class(es): King, Knight, Rook, Bishop, Pawn, Queen
Description: These classes all extend ChessPiece and override the important isValidMove(Square dest) and getInitial() methods. They all have their own logic in isValidMove that complies with the rules of chess for each pieces. If the move is not valid it returns false.
Attributes:

- none

Methods:

- getInitial(): String
- isValidMove(Square dest): boolean

Class: Player
Description: Logic defining a player which is used to help check if moves are valid in ChessPiece. For example, chessPieces know what player they belong to and know if they can take another piece if it is a different color
Attributes:

- pieces: List<ChessPiece>
- opponent : Player
- king : King
- color : PlayerColor

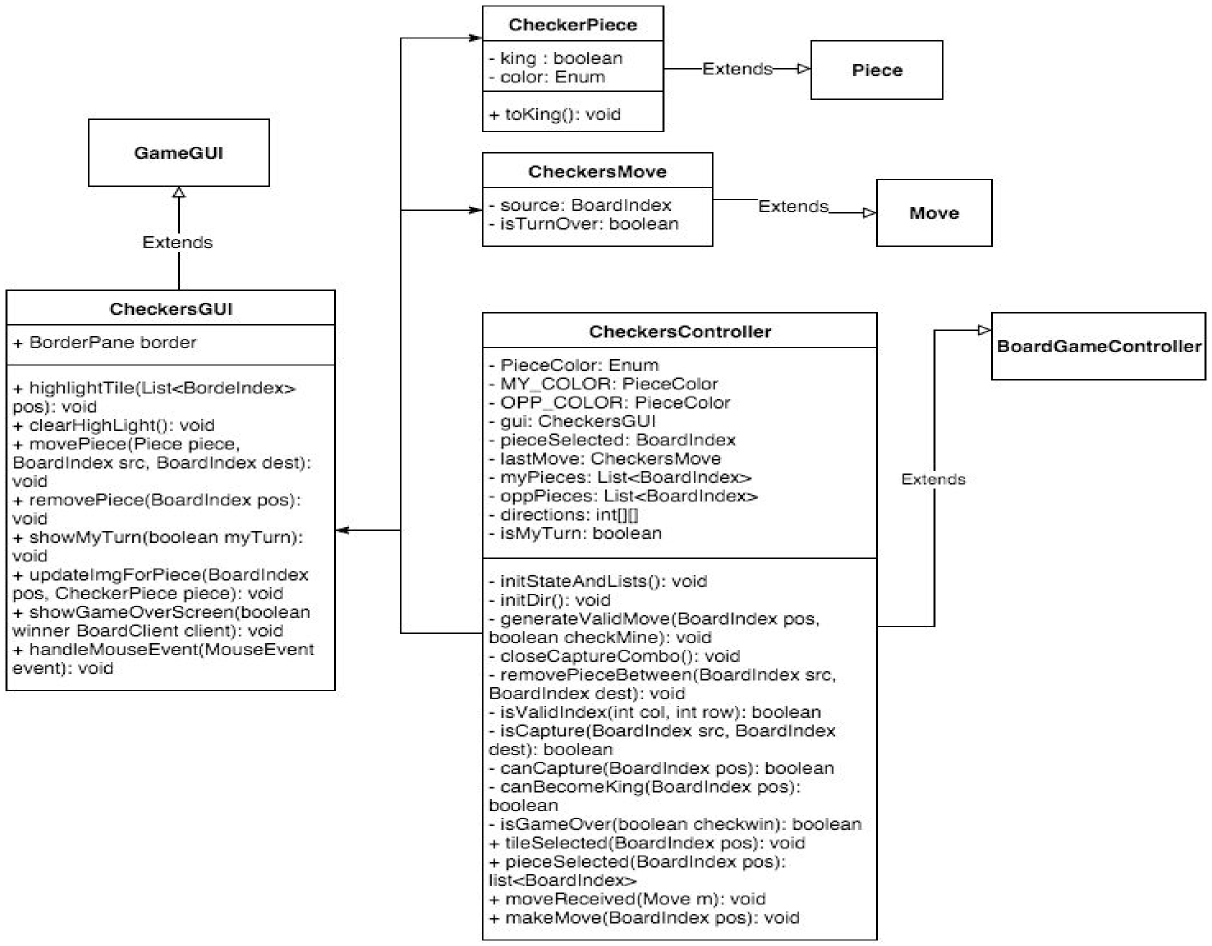
Methods:

- Player(PlayerColor)

Class: <enum> PlayerColor
Description: This enum is used to set the Player's PlayerColor attribute as either WHITE or BLACK
Attributes:

- WHITE
- BLACK

Checkers UML Diagram



CheckerPiece
- king : boolean
- color: Enum
+ toKing(): void
Extends
Piece
GameGUI
Extends
CheckersMove
- source: BoardIndex
- isTurnOver: boolean
Extends
Move
CheckersGUI
+ BorderPane border
+ highlightTile(List<BordeIndex> pos): void
+ clearHighLight(): void
+ movePiece(Piece piece, BoardIndex src, BoardIndex dest): void
+ removePiece(BoardIndex pos): void
+ showMyTurn(boolean myTurn): void
+ updateImgForPiece(BoardIndex pos, CheckerPiece piece): void
+ showGameOverScreen(boolean winner BoardClient client): void
+ handleMouseEvent(MouseEvent event): void
CheckersController
- PieceColor: Enum
- MY_COLOR: PieceColor
- OPP_COLOR: PieceColor
- gui: CheckersGUI
- pieceSelected: BoardIndex
- lastMove: CheckersMove
- myPieces: List<BoardIndex>
- oppPieces: List<BoardIndex>
- directions: int[][]
- isMyTurn: boolean
- initStateAndLists(): void
- initDir(): void
- generateValidMove(BoardIndex pos, boolean checkMine): void
- closeCaptureCombo(): void
- removePieceBetween(BoardIndex src, BoardIndex dest): void
- isValidIndex(int col, int row): boolean
- isCapture(BoardIndex src, BoardIndex dest): boolean
- canCapture(BoardIndex pos): boolean
- canBecomeKing(BoardIndex pos): boolean
- isGameOver(boolean checkwin): boolean
+ tileSelected(BoardIndex pos): void
+ pieceSelected(BoardIndex pos): list<BoardIndex>
+ moveReceived(Move m): void
+ makeMove(BoardIndex pos): void
Extends
BoardGameController

<u>Checkers Documentation</u>

Class: CheckersController extends BoardGameController
Description: The controller of the checkers game that facilitates all action and response in the game. It contains the gui, the logic, and models for the game.

Attributes:

- PieceColor: Enum
- MY_COLOR: PieceColor
- OPP_COLOR: PieceColor
- gui: CheckersGUI
- pieceSelected: BoardIndex
- lastMove: CheckersMove
- myPieces: List<BoardIndex>
- oppPieces: List<BoardIndex>
- directions: int[][]
- isMyTurn: boolean

Methods:

- initStateAndLists(): void
- initDir(): void
- generateValidMove(BoardIndex pos, boolean checkMine): void
- closeCaptureCombo(): void

- removePieceBetween(BoardIndex src, BoardIndex dest): void
- isValidIndex(int col, int row): boolean
- isCapture(BoardIndex src, BoardIndex dest): boolean
- canCapture(BoardIndex pos): boolean
- canBecomeKing(BoardIndex pos): boolean
- isGameOver(boolean checkwin): boolean
- tileSelected(BoardIndex pos): void
- pieceSelected(BoardIndex pos): list<BoardIndex>
- moveReceived(Move m): void
- makeMove(BoardIndex pos): void

Class: CheckersGUI extends GameGUI
Description: The view component of the Checkers game. It is updated by the CheckersController to reflect the changes of the game state.
Attributes:

- border: BorderPane

Methods:

- highlightTile(List<BordeIndex> pos): void
- clearHighLight(): void
- movePiece(Piece piece, BoardIndex src, BoardIndex dest): void
- removePiece(BoardIndex pos): void
- showMyTurn(boolean myTurn): void
- updateImgForPiece(BoardIndex pos, CheckerPiece piece): void
- showGameOverScreen(boolean winner BoardClient client): void
- handleMouseEvent(MouseEvent event): void

Class: CheckerPiece
Description: This model represents each checkers pieces on the board. It is used to create 8 x 8 piece array in the game state.
Attributes:

- king: boolean
- color: Enum

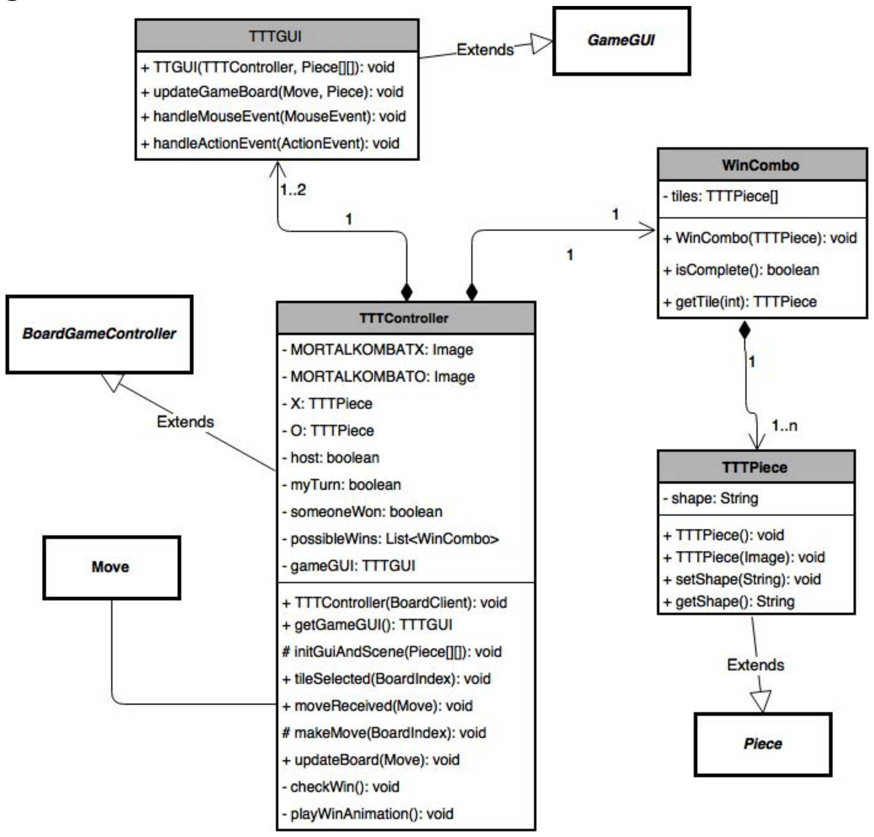
Methods:

- toKing(): void

Class: CheckersMove
Description: Checkers move is sent over to the server to reflect each player's move.
Attributes:

- source: BoardIndex
- isTurnOver: boolean

TicTacToe UML Diagram



<u>TicTacToe Documentation</u>

Class: TTTController extends BoardGameController
Description: The main controller for the tic-tac-toe game. It holds the game's GUI and game state. It also makes moves updating both the GUI, State, and sending the move to the client socket accordingly.

Attributes:

- MORTALKOMBATX: Image
- MORTALKOMBATO: image
- X: TTTPiece
- O: TTTPiece
- host: boolean
- myTurn: boolean
- someoneWon: boolean
- possibleWins: List<WinCombo>
- gameGUI : TTTGUI

Methods:

- initGuiAndScene(Piece[][]): Initializes the gameGUI and the scene based on the gameGUI
- makeMove(BoardIndex): Sends move to BoardClient
- tileSelected(BoardIdnex): Updates state based on the player's move
- moveReceived(Move): Updates state based on received move
- updateBoard(Move): Updates game gui of the client
- checkWin(): Checks if one of the possible winning combos has been completed
- playWinAnimation(): Plays the winning animation

Class: TTTGUI extends GameGUI
Description: Sets up the GUI for the TicTacToe game plug-in and extends the abstract class GameGUI.

Attributes:
- none

Methods:
- updateGameBoard(Move, Piece): Updates the gui game board given a Move and Piece
- handleMouseEvent(MouseEvent): Handles all mouse event
- handleActionEvent(ActionEvent): Handles any action events (TicTacToe does not use this)

Class: TTTPiece extends Piece
Description: Extends the abstract class Piece, adds a shape attribute.

Attributes:
- shape: String

Methods:
- TTTPiece(): Constructor that takes in no arguments
- TTTPiece(Image): Constructor that takes in an Image
- setShape(String): Sets the attribute shape (corresponding to an X or O)
- getShape(): Returns the attribute shape

Class: WinCombo
Description: Checks if a combo is a winning combo

Attributes:
- Tiles: TTTPiece[]

Methods:
- WinCombo(TTTPiece…): Constructor that takes in TTTPieces
- isComplete(): Checks if a winning combo has been completed
- getTile(int): Returns the Piece at that index

How to Plug in a Game

<u>Steps on how to create a new game implementation</u>

1. Create a controller for your game by extending BoardGameController and implementing its abstract methods. Use GameState and the controller to manage the game's underlying 2D array of objects.
2. Create a custom gui for your game by extending GameGUI and overriding handleMouseEvent and implementing handleActionEvent if it is needed
3. Create the pieces for your game by having each piece extend Piece and specify the images for each type of piece
4. (if needed) Create a move for your game by extending Move
5. Depends on if you are a game developer or the TA for this class
    a. Game Developer
        i. After your game implementation is complete (steps 1 - 4 and any additions), submit your game implementation to us. By doing this, we can make your game available to everyone else.
    b. TA
        i. Add the name of your game to the ListofGame.txt file
        ii. Add an additional else block to GameControllerFactory that constructs your game controller for your game

6. Additional Reference
    a. Refer to one of the built-in game implementations

Things that Changed

- BoardGameController
    - Removed the setUpModelAndView, validateMove, updateModel, updateView abstract methods
        - <u>Reason</u>: We realized that each game was updating the model and view differently and in various places. So, we decided to remove these methods.
    - Added the initGuiAndScene abstract method
        - <u>Reason</u>: Added to ensure that game implementers set the scene using their respective subclass of GameGUI
- Move
    - No longer include the myTurn attribute and isMyTurn() method
        - <u>Reason</u>: In order to standardize our approach for determining who is the first player and whose turn it is, we chose to use the hostStatus attribute of BoardClient in order to always make the host go first and we realized that the actions of sending and receiving moves already gives us the ability of knowing whether it is the player's turn or not.
- ChessController / CheckersController
    - selectPiece was removed
        - <u>Reason</u>: It didn't make sense to have them be controller methods. Instead, getPiece and removePiece were added to GameState.
- Networking
    - We tried a Single Thread solution. It had its own issues of interrupting the UI. While listening to the Socket, the Application thread was blocked and the interface would become unresponsive.
    - JavaFX Concurrency to the rescue! On a multithreaded client we can use concurrency to allow background tasks to complete and only then affect changes to UI
    - Utilizing Platform.RunLater() to initiate Networking communication at an unspecified time.
    - This allows the Application thread to continue without interruption
    - Since the previous design a Game Lobby has been implemented allowing Client's to host a particular game
    - Other connected Client's can view the hosted games available and join them at the touch of a button
    - The Server pairs these ClientThreads on the Server-side and then sends messages to the paired Client's stating the game is ready.

- BoardGameGridPane
    - highlightAssociatedTiles and resetHighlightedTiles methods(and associated attributes) were added in place of showValidMovesWithHighlight
        - <u>Reason</u>: It order to make the action of highlighted tiles more general besides only using it it to show valid valid moves, the methods name was changed.
    - activate and disable methods were added
        - <u>Reason</u>: Added in order to give games a way to restrict players from being able to interact with the board when it isn't their turn.

- Chess
    - Adding logic from a command line chess application found on GitHub. The command line version also followed a Model View Controller design which made it easy to extract the model part from it and it it to our own design.
    - ChessController got updated and simplified as classes became more defined and parts separated into more functions.
    - Highlighting tiles was added to the ChessController which called BoadGameGridPane's showValidMovesWithHighlight method
    - GameStatus text was added to ChessGUI which displays who's turn it is and whether or not a player is in check.
    - ChessMove (extends Move) was created in order to pass moves through the client's socket to the matched opponent

- Checkers
    - No major changes were made, besides the removal of Piece, to the checks because the implementation was delayed until the design was finalized.

- Tic-Tac-Toe
    - The first version of Tic Tac Toe was found on gitHub. It was used to test the network server implementation and allow us to understand the game logic. Once the overall design was complete, Tic Tac Toe was refactored to fit the overall design.
    - Tic Tac Toe now extends BoardGameController and GameGUI.