Sfwr Eng 2AA4 – Assignment 1

Group 18

Zichen Jiang
Danish Khan
Hasan Siddiqui

Dr. Maibaum
Tutorial 2
Due: Monday February 22, 2016

Design Overview

Modules

The application is decomposed into 3 components:

1. Model: handles data representation, data logic, e.g.:
   i. Classes to represent the game board, pieces, and nodes on the game board
   ii. a function to assign a piece to a node on the board
2. View: displays the graphical user interface, e.g.:
   i. display an empty game board at the start of a new game
   ii. display pieces on the board
3. Controller: handles user input, and accesses both view and model, e.g.:
   i. listeners are assigned to view to receive mouse click
   ii. buttons with listeners to clear the board

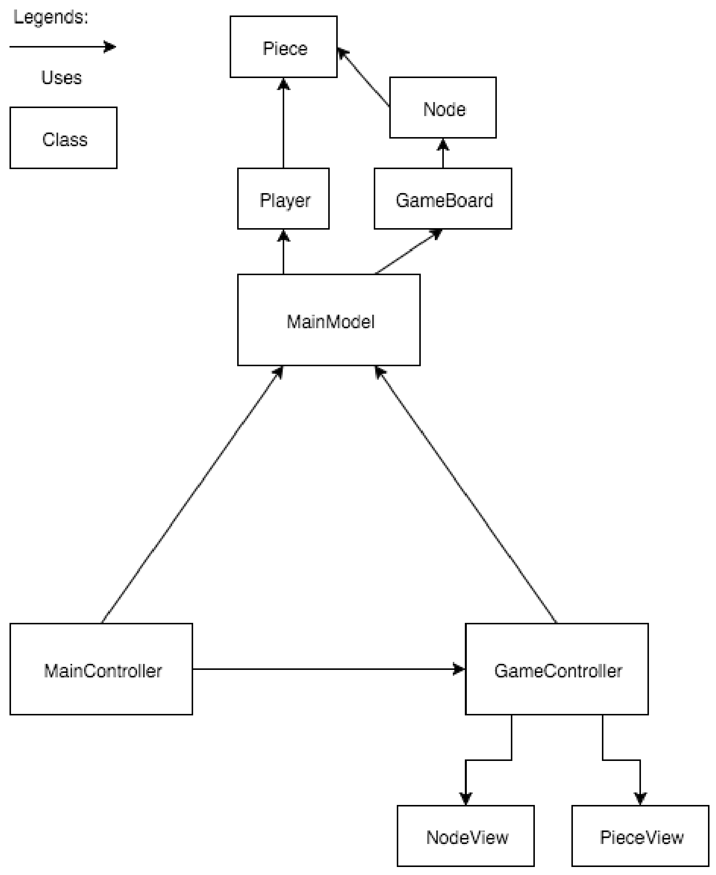
Justifications for MVC

Model-View-Controller (MVC) is a very popular design pattern used for computer applications implementing user interfaces. The advantages of MVC design paradigm are many:

1. Separation of Concerns
   - It provides an isolation of the application's presentation layer that displays the data in the user interface, from the way the data is actually processed. For example, in this project, if there needs to be a change for the underlying logic on the board, only Model part needs to be modified while the Controller and View can be left untouched.
2. Modularity
   - The way in which a system's components can be separated and recombined. Meaning that for a software, different components can be maintained separately without breaking the entire system. This allows separation of concerns to be implemented with little effort.
3. Expectancy for Change
   - This suggests that software should be able to support upgrades or changes on a small part without causing the entire system to break. Modules should communicate with each other on a very abstract level. They only deal with each other's inputs and outputs. With the MVC ideology, View will be updated by Model, and Model will be maintained by Controller. The internal logic within any of the modules can be changed without breaking other modules.

Highest level of Uses Relationship:

The following diagram showed the highest level of uses relationship in this application



Due to the group's inexperience in developing a program that strictly follows MVC design pattern, view and controller was not entirely separated.

The main view, or the parent class of `NodeView` and `PieceView` , is integrated with the controller The reason is that, the `paintComponent()` method needs to draw all of the `NodeView` and `PieceView` objects, the current player, the currently selected piece and the node being hovered over by the mouse. Meanwhile, the controller also needs access to the `PieceView` and `NodeView` objects as the user clicks in those objects and the user's clicks act directly on those objects, changing their display.

If given more time to experiment, or more experience developing in Java and follows MVC design pattern, view can be separated entirely from the controller. This will be discussed more in the Internal Review section.

Module

Model

Under Model module, there are several classes created to fully represent the data structures and logics.

MainModel.java

MainModel is the top level class in Model. It acts like a parent class for all the other classes in Model. MainModel will access all other classes under Model. This is the only class that can be accessed by Controller. If there were changes made to Model, one only needs to review the parts in Controller where only MainModel is used. Therefore the program is easy to adapt for change.

Uses

- **GameBoard.java**
- **Player.java**

Interface (MIS)

- **MainModel(): MainModel**
  Constructs a new MainModel object. Will call newGame().
- **getBoard(): GameBoard**
  Returns the GameBoard object of current MainModel.
- **getCurrentPlayer(): Player**

Returns the current Player object of current MainModel.
- **newGame(): void**
  Initialize the MainModel objects. Will be called in the constructor.

Implementation (MID)

Uses

None

Variables

- **currentPlayer: int**
  Stores an integer, 0 or 1, that indicates the current player.
- **gameBoard: GameBoard**
  Stores the GameBoard object for the current game board.
- **players: Player[]**
  An array that holds the two players of the game.

Access Programs

- **MainModel(): MainModel**
  Constructs a new MainModel object. Will call newGame().
- **getBoard(): GameBoard**
  Returns the GameBoard object, gameBoard, of the MainModel object.
- **getCurrentPlayer(): Player**
  Returns the current Player object, Player[currentPlayer], of the MainModel object.
- **newGame(): void**
  Initialize the MainModel objects. It will create and initialize a new GameBoard object, gameBoard and two new Player objects.

DIRECTION.java

It is a class of Enum type. It is used in Node.java to represent the relationship between nodes on a game board (see Node.java).

Uses

None

Interface (MIS)

None

Implementation (MID)

- **UP**
  Represent the up direction.
- **DOWN**
  Represent the down direction.
- **LEFT**
  Represent the left direction.
- **RIGHT**
  Represent the right direction.

Player.java

The Player class represent one player of the game. It has a bench and a variable to represent the number pieces the player has. The bench is a stack used to store the pieces that has not been placed on the game board. Every time a piece is placed, it will be first be popped out of the bench stack. When the length of the bench is 0, it indicates the end of Phase I. The variable that represents the number of pieces the player has will help to check the winning state.

Uses

- **Piece.java**

Interface (MIS)

- **Player(): Player**

Creates a new Player object.

- **getPiece(): Piece**
  Get an unplaced piece form the player to be placed on the game board.

Implementation (MID)

Variables

- **bench: Piece[]**
  This is a stack used to store the pieces that have not been placed on the board.
- **numOfPieces:int**
  This is a variable that represent

Access Programs

- **Player(ID: int): Player**
  It will create sixes pieces
- **getPiece(): Piece**

getNumOfPieces(): int

Piece.java

This class represent the pieces that each player has in the game.

Uses

None

Interface (MIS)

- **Piece(player: Player): Piece** Construct a new Piece object, and sets its owner
- **getOwner(): Player** Returns a Player object, the owner of the piece.

Implementation (MID)

Variables

- belongsTo: Player
  It represent the owner of the Piece. It is set by the constructer.

Access Programs

- **Piece(player: Player): Piece**
  Construct a new Piece object, and sets its owner
- **getOwner(): Player**
  Returns the value of `belongsTo` , a Player object, the owner of the piece.

GameBoard.java

This class contains information about the game board. It contains information about all the nodes on the board, and provides interface to manipulate each nodes on the board. It stores `Node` objects in an array, and each `Node` is represented by their index in the array.

Uses

- **Node.java**
- **Error.java**

Interface (MIS)

- **GameBoard(): GameBoard**
  Creates a new `GameBoard` object, with all nodes on it initialized.
- **setPiece(piece: Piece,node: Node)**
  Sets a Piece object on a given Node object
- **movePiece(origin: Node,dest: Node)**
  move a Piece from one Node object, `origin` , to the another one, `dest`

- **removePiece(piece: Piece)**
  remove a Piece object, `piece` , from the game board.
- **checkLegalMove(): Error**
  Check if the game board contained any illegal move. Returns an `Error` object.
- **checkWin(player: Player): boolean**
  Check if the given player, `player` , had won

Implementation (MID)

Variables

- **nodes: Node[]** An array of Node objects that contains all the nodes on the game board.

Access Programs

- **GameBoard(): GameBoard**
  Creates a new `GameBoard` object, and creates all the nodes on it. It will use two nested for loops to create the nodes on the outter square, and another one to create the inner square. It will also uses `Node` 's methods, `setNeighbours` to build up the relationships between the nodes.
- **setPiece(piece: Piece,node: Node)**
  Sets a Piece object on a given Node object. Use Node's method `setPiece` to achieve this.
- **movePiece(origin: Node,dest: Node)**
  Move a Piece from one Node object, `origin` , to the another one, `dest` . First, get the piece from `origin` using `getPiece` , then set the piece to `dest` using `setPiece` . After that, remove the piece from `origin` node using `removePiece`
- **removePiece(node: Node)**
  Remove the piece placed `node` from the game board. Use the `removePiece` method from `Node`
- **checkLegalMove(): int**
  Check if the game board contained any illegal move. Returns an `Error` object. Loop through all the nodes of the game board, it will check if the user is attempting to put one piece above another piece, or trying to click on an empty node without picking up a node.
- **checkWin(): void**
  Check if one player had won. Loop though all nodes on the game board, and call `checkV()` and `checkH()` on each node.
- **checkV(node: Node): boolean**
  Check the vertical connections of `node` . Use `Node` 's method, `getNeighbor` , to check the `UP` and `DOWN` directions of the node recursively, i.e. check the `UP` and `DOWN` directions of it's neighbour's, and neighbour's neighbor, if it had any. Return true if all nodes on the vertical direction of the `node` is occupied with the same player's piece.
- **checkH(node: Node): boolean**
  Check the horizontal connections of `node` . Use `Node` 's methode, `getNeighbor` , to check the `LEFT` and `RIGHT` directions of the node recursively, i.e. check the `LEFT` and `RIGHT` directions of it's neighbor's, and neighbour's neighbor, if it had any. Return true if all nodes on the vertical direction of the `node` is occupied with the same player's piece.

Node.java

This class represent each nodes on the game board. It contains information about the piece that may be placed on it. The nodes will be connected by each other. The neighbours of the cell are being represented by an EnumMap of DIRECTION and Node. Node is being implemented in such way, instead of having a 2D array, is that, the game board of Six Men's Morris has different number of nodes on each row. It would be hard to use a 2D array to store the nodes, as there would be fake nodes created to take up spaces, and detecting 3 pieces in a row would be hard, too.

Interface (MIS)

- **Node(): Node**
  Returns a new `Node` object
- **getPiece(): Piece**
  Return the `Piece` object that's being placed on the node. It returns `null` if the node has no piece.
- **removePiece(): void**
  Remove the `Piece` object that's being placed on the node.
- **setPiece(piece: Piece): void**
  Set the given piece, `piece` , on the node.
- **getNeighbor(dir: DIRECTION): Node**
  Return the neighbor of the node on the given direction, `dir` . It returns `null` if there is no neighbor on that direction.
- **setNeighbors(neighbors: EnumMap): void**
  Set the neighbors of the node to `neighbors`

Implementation (MID)

Variables

- piece: Piece
  Contains the piece that's being placed on the node. It's `null` if there's no piece.
- neighbors: EnumMap
  Contains the information about its neighbors on all four directions.

Access Programs

- **Node(): Node**
  [blank]
- **getPiece(): Piece**
  Return the field `piece` .
- **removePiece(): void**
  Set the field `piece` to `null` .
- **setPiece(piece: Piece): void**
  Set the field `this.piece` to `piece` .
- **getNeighbor(dir: DIRECTION): Node**
  Use `neighbors.get(dir)` to get the node on given direction.
- **setNeighbors(EnumMap): void**
  Set field `this.neighbors` to `neighbors`

Error.java

A class that contains error messages. Mostly the error messages are for illegal moves on the game board.

Uses

None

Interface (MIS)

- **Error(): Error**
  Create a new `Error` object.
- **getMsg(ID: int): string**
  Get the string message of a given message ID, `ID` .

Implementation (MID)

Variables

- **messages: string[]**
  Contains all the error messages that may appear in the game.

Access Programs

- **Error(): Error**
  Return a new `Error` object.
- **getMsg(ID: int): string**
  Return the `ID` 'th element of the `message` array.

View

The view is implemented using `Swing` . The nodes and pieces are `Ellipse2D.Double` 's. We decided to use Swing to implement the GUI because it is time-efficient to create, and is effortless to maintain or upgrade. The main view, or the parent class of `NodeView` and `PieceView` , however, are not strictly in the View module. It is integrated with the controller. The reason is that, the `paintComponent()` method needs to draw all of the `NodeView` and `PieceView` objects, the current player, the currently selected piece and the node being hovered over by the mouse. Moreover, the controller also needs access to the `PieceView` and `NodeView` objects as the user clicks in those objects and the user's clicks act directly on those objects, changing their display.

NodeView.java

This class is a subclass of `Ellipse2D.Double` , which describes an ellipse defined by a framing rectangle. It will display a filled ellipse with defined radius on the game board to represent a node.

Interface (MIS)

- **NodeView(x: int, y: int, node: Node): NodeView**
  It will create a new `NodeView` object centered at given coordinates, `x` and `y` . It will set the node object from Model to one of its field to identify itself.
- **moveTo(x: int, y: int): void**
  It will modifies the `NodeView` 's coordinates, and make it center at `x` , `y` .
- **contains(Point click): boolean**
  It overrides the default contains function inherit from `Ellipse2D.Double` . It takes a point on the screen and tells if the point is inside the shape `NodeView` .

- **getNode(): Node**
  It will return the `Node` object that the `NodeView` object is representing.

Implementation (MID)

Variables

- **serialVersionUID: long**
  It's generated by eclipse to uniquely identify the `Ellipse2D.Double` subclass.
- **RADIUS: int**
  Determines the display radius of one node. It is `static` and `final` .
- **CLICKRADIUS: int**
  Determines the clickable radius of one node. It is `static` and `final` .

Access Programs

- **NodeView(x: int, y: int, node: Node): NodeView**
  First it will call `super(x - RADIUS, y - RADIUS, 2 * RADIUS, 2 * RADIUS);` to create an ellipse with the corresponding coordinates and size. The first two parameters represent the coordinates of the top left corner of the object, so we need to subtract `RADIUS` from it to make the `NodeView` centered at the given coordinates, `x` and `y` . The later two arguments define the total size of the object being two times the `RADIUS` .
- **moveTo(x: int, y: int): void**
  It will modifies the `NodeView` 's coordinates to `x - RADIUS` and `y - RADIUS` . As explained above, the coordinates of the object represent the top left corner of the object, so we need to subtract `RADIUS` from it to make the `NodeView` centered at the given coordinates `x` and `y` .
- **contains(Point click): boolean**
  It overrides the default contains function inherit from `Ellipse2D.Double` . It will take the x and y coordinates of the clicked point on screen, and make sure that x is within the range between `this.getCenterX() - CLICKRADIUS` and `getCenterX() + CLICKRADIUS` , and that y is within the range between `getCenterY() - CLICKRADIUS` and `getCenterY() + CLICKRADIUS` .
- **getNode(): NodeView**
  It will return the field `node` .

PieceView.java

This class is a subclass of `Ellipse2D.Double` , which describes an ellipse defined by a framing rectangle. It will display a filled ellipse with defined radius on the game board, on top of a node, to represent a piece.

Interface (MIS)

- **PieceView(x: int, y: int, i: intd): PieceView**
  It will create a new `PieceView` object centered at given coordinates, `x` and `y` . It will also have the id indicated by the parameter `id` .
- **moveTo(x: int, y: int): void**
  It will modifies the `NodeView` 's coordinates, and make it center at `x` , `y` .
- **moveToNode(node: NodeView)" void**
  It will move the current `PieceView` to the position of the given `NodeView` , `node` .
- **contains(Point click): boolean**
  It overrides the default contains function inherit from `Ellipse2D.Double` . It takes a point on the screen and tells if the point is inside the shape `PieceView`
- **currentNode(): NodeView**
  It will return the `NodeView` the piece is currently on. Will return null if not on placed on any `NodeView`
- **getId(): int**
  It returns the `id` field of the `PieceView` .

Implementation (MID)

Variables

- **serialVersionUID: long**
  It's generated by eclipse to uniquely identify the `Ellipse2D.Double` subclass.
- **RADIUS: int**
  Determines the display radius of one piece. It is `static` and `final` .
- **CLICKRADIUS: int**
  Determines the clickable radius of one piece. It is `static` and `final` .

Access Programs

- **PieceView(x: int, y: int, i: intd): PieceView**
  First it will call `super(x - RADIUS, y - RADIUS, 2 * RADIUS, 2 * RADIUS);` to create an ellipse with the corresponding coordinates and size. The first two parameters represent the coordinates of the top left corner of the object, so we need to subtract `RADIUS` from it to make the `PieceView` centered at the given coordinates, `x` and `y` . The later two arguments define the total size of the object being two times the `RADIUS` .

- **moveTo(x: int, y: int): void**
  It will modifies the `PieceView` 's coordinates to `x - RADIUS` and `y - RADIUS` . As explained above, the coordinates of the object represent the top left corner of the object, so we need to subtract `RADIUS` from it to make the `PieceView` centered at the given coordinates `x` and `y` .
- **moveToNode(node: NodeView)" void**
  It will modifies the `x` and `y` values of `PieceView` to `node.getCenterX() - RADIUS` and `node.getCenterY() - RADIUS` respectively. The reason for subtracting `RADIUS` is discussed previously.
- **contains(Point click): boolean**
  It overrides the default contains function inherit from `Ellipse2D.Double` . It will take the x and y coordinates of the clicked point on screen, and make sure that x is within the range between `this.getCenterX() - CLICKRADIUS` and `getCenterX() + CLICKRADIUS` , and that y is within the range between `getCenterY() - CLICKRADIUS` and `getCenterY() + CLICKRADIUS` .
- **currentNode(): NodeView**
  Returns the `currentNode` field.
- **getId(): int**
  Returns the `id` field.

Controller

MainController.java

This is the class that creates the overall user interface, and hooks up listeners to clickable items in the window.

Uses

- **GameController.java**

Interface (MIS)

- **MainController(): MainController**
  Calls `initUI` method.
- **main(args: String[]): void**
  Display the game window to the user.

Implementation (MID)

Variables

- **GameController: board**
  Stores the GameController object.
- **controls: JPanel** A `JPanel` that contains the `JButtons`
- **newGameButton: JButton** Has a listener that reset the game
- **gameModeButton: JButton** Has a listener to change the game mode, either the user can place any color piece on the board, or the user has to place different color piece each turn.

Access Programs

- **MainController(): MainController**
  Calls `initUI` method.
- **initUI(): void**
  Creates the `JButton`s and put them on the `JPanel` . Assigns listeners to the `JButton`s
- **main(args: String[]): void**
  Creates a new JFrame window, and displays it to the user.

GameController.java

This class extends JPanel and holds all the View objects, using them to process user input through its own MouseAdapter. Validates and performs changes to the current Game object according to input, and displays corresponding changes on board.

Uses

- **Game.java**
- **Node.java**
- **Player.java**
- **NodeView.java**
- **PieceView.java**

Interface (MIS)

- **GameController(): GameController**
  Creates a new 'GameController' object which starts a new game and instantiates mouse listeners to process user input.
- **newGame(): void** Starts a new game with new objects.
- **switchMode(): int** Switches between turn-based game and setting up board state, and returns current mode.

Implementation (MID)

Variables

- **currentGame: Game** Model of current game, contains pieces and nodes, and performs actions on them.
- **nodes: NodeView[]** Array of NodeView objects for display corresponding to Nodes in Game model.
- **redPieces: PieceView Array of red PieceView objects for display corresponding to red Player's pieces.
- **bluePieces: PieceView Array of blue PieceView objects for display corresponding to blue Player's pieces.
- **selectedPiece: PieceView** Holds currently selected PieceView by user to be moved.
- **hoveredNode: NodeView** Last NodeView hovered over by mouse, highlighted for aesthetic purposes.
- **currentPlayer: int** Holds ID of current player, or -1 if game is in Set Board State mode
- **boardStartX: int** X coordinate for left edge of board.
- **boardStartY: int** Y coordinate for top edge of board.
- **boardCenterX: int** X coordinate for center of board.
- **boardCenterY: int** Y coordinate for center of board.
- **boardEndX: int** X coordinate for right edge of board.
- **boardEndY: int** Y coordinate for bottom edge of board.
- **redBenchX: int** X coordinate for red player's bench pieces on left of board.
- **blueBenchX: int** X coordinate for blue player's bench pieces on right of board.
- **boardSize: int** Board side length, distance from one edge to the other.
- **gridSize: int** Distance from one row or column to the next, one fourth of grid size.
- **serialVersionUID: long** It's generated by eclipse to uniquely identify the `JPanel` subclass.
- **N_NODES: int** Number of nodes.
- **N_PIECES: int* Number of pieces.

Access Programs

- **GameController(): GameController**
  Calls newGame() method to initialize new objects. Adds embedded Mouse Listener subclasses for processing user input.
- **newGame(): void** Starts a new game with new objects.
- **switchMode(): int** Switches between turn-based game and setting up board state, and returns current mode.
- **initNodes(): void** Define each node as a new node.
- **initPieces(): void** Define the pieces in each list according to their player ID.
- **calcBoard(): void** Calculate board dimensions and coordinates using panel dimensions.
- **calcCoordinates(): void** Calculate NodeView and PieceView coordinates.
- **paintComponent(): void** Display board changes.
- **finalizeMove(node: NodeView) Perform move.
- **selectPiece(piece: PieceView, pieceID: int) Select a piece.

Trace back

Requirement	Module
Set up a game board	GameField.java, Model.java
Game board include two different kind of discs	GameField.java, Player.java, Piece.java
Discs are placed on the side of board	GameField.java, Model.java
User is able to start a new game	GameController.java
User is able to choose a colour of disc	GameView.java
User is able to place disc on the board	GameView.java, Model.java, GameField.java
System can analyze whether the current state is possible or not	GameField.java, Node.java
Errors will be highlighted on the screen	GameView.java, Model.java, GameController.java

Internal Review

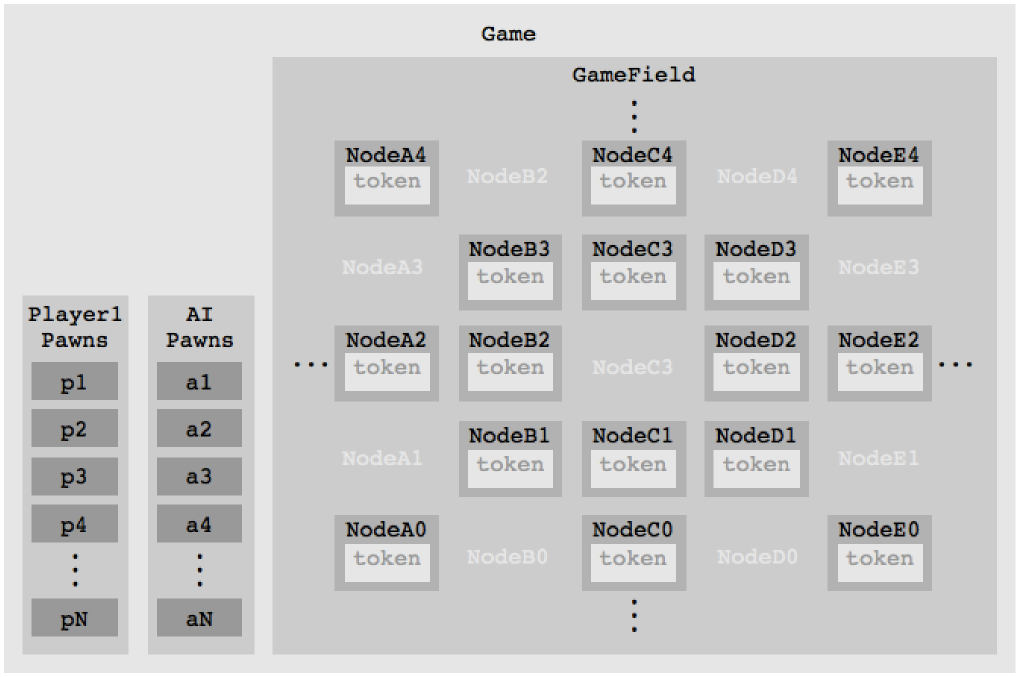
The final product was well recieved due to investing time in the designing stage. The thoroughness of the design ensured that the implementation required only one iteration.

During the implementation, if a consideration for deviating from the original design occured, arguments made for the deviation were analyzed and discussed with all members of the group and implemented only on undivided agreement. This ensured that all group members were aware of changes and bottlenecks did not appear in the development.

Were this project given another iteration of development, it is noted that the view implementation would then instead use picture elements instead of the g library to appeal to a broader user base for the Morris game.

Design Decisions

The Model was built to accommodate any future changes in board architecture and board rules. The current implementation supports rules for N-Men's Morris for any valid integer N. The linked Node board architecture was abandoned in favour of increasing the implementation's responsiveness to change in the future.



Change from a Linked-Node Structure to a Grid-Node Structure

The GameBoard.java class was implemented as GameField.java. The rules and board architecture of Six-Men's Morris in GameBoard.java was abstracted to an N-Men's Morris rule and board architecture in GameField.java. The new architecture relies on a grid based system of Nodes. Each Node is a container containing three labels;

1. Row (integer)
2. Column (integer)
3. Radius from origin.

Due to the nature of Morris board, there are always three Nodes in one row and three Nodes in one column. With the exception of the centre column, which always contains four Nodes for an N-Men's Morris rule set, the worst case search for a Node is at most 9 comparisons. The benefit of this architecture is that the implementation can accommodate 45-Men's Morris, or more, and still require just 9 comparisons.

Removal of DIRECTION.java

The DIRECTION enum class was abrogated during the implementation of the Model in favour of accommodating N-Men's Morris where N is any valid morris number. A valid morris number N is any integer N such that N >= 6 and N mod 3 = 0. The following classes were affected;

- GameField.java
- Game.java

Separation of Error.java

The Error class was split into three separate classes and packaged separately in a package.test;

- GameFieldTest.java
- GameTest.java
- PlayerTest.java

Further testing was done natively within vital class files in separate test clients until code reliability was secured.

Test Report

Test Cases:

Symbol Definition

- N = Valid Node
- FN = Full Valid Node
- EN = Empty Valid Node (i.e. the node is not assigned a player piece)
- IEN = Inalid Empty Node (i.e. the node exists and is not assigned a player piece)
- IFN = Invalid Full Node (i.e. will not exist)

Test Cases for Rule Implementation for Morris Number 6.

1. set piece to NE : true : passed
2. set piece to NF : true : passed
3. set piece to INE : false : passed
4. set piece to INF : false : passed
5. move piece from NF to NE : true : passed
6. move piece from NE to NF : false : passed
7. move piece from NF to NF : false : passed
8. move piece from NF to INE : false : passed
9. move piece from NF to INE : false : passed
10. move piece from NE to INF : false : passed
11. move piece from NF to INF : false : passed
12. move piece from INF to NE : false : passed
13. move piece from INF to INE : false : passed
14. move piece from INE to INF : false : passed
15. move piece from INF to INF : false : passed
16. move piece from adjacent to adjacent : true : passed
17. move piece from adjacent to nonAdjacent : false : passed
18. move piece from nonadjacent to nonadjacent : false : passed

Test Cases for Board Logistics.

1. remove piece from full player stack : returns Piece : passed
2. remove piece from stack with 1 Piece: returns Piece : passed
3. remove piece from empty stack : returns null : passed

View Functionality

The user-view interaction was tested informally through various trails.