Anh Thu Pham
Ashton Headley

1. Write Up Design Pattern Assessment
   - Class that contains a controller: MancalaGame

2.
```
/**
 * Watch for when the player selects a pit and perform the turn
* @param e the mouse click event
 */
@Override
public void mouseClicked(MouseEvent e) {
 if (turnAvailable) {
 int x, y;
 int mx = e.getX();
 int my = e.getY();

 // loop through all pits in the bottom row
 for (int pit = 0; pit < 6; ++pit) {
 x = boardStrategy.getPitX(pit);
 y = boardStrategy.getPitY(pit);

 // check if the click was inside the pit area. if (mx > x && mx < x +
boardStrategy.pitWidth && my > y && my < y +
boardStrategy.pitHeight + 50) {
 backup.makeBackup();
 model.doPlayerTurn(pit);
                    repaint();
 turnAvailable = false;
 }
 }
 }
}
```

   - Model: Model

3.
```
/**
 * Set the number of stones per pit at the beginning of the game
* default number of stones is four
 * @param initialStones
 */
public void setInitialStonesPerPit(int initialStones) {

 // bottom row
 for (int i = 0; i < 6; i++)
```

```
{
pitStones[i] = initialStones;
}
// top row
for (int i = 7; i < 13; i++)
{
pitStones[i] = initialStones;
}
// player 1 mancala
pitStones[6] = 0;

// player 2 mancala
pitStones[13] = 0;

ChangeEvent event = new ChangeEvent(this);
for (ChangeListener listener: listeners)
{
listener.stateChanged(event);
}
}
```

- View: MancalaGame

4.
```
/**
 * Paint information on all the players
 * @param g Graphics object
 */
protected void paintPlayerInfo(Graphics g) {
 Graphics2D g2 = (Graphics2D) g;
 if ( model.getWinningPlayer() < 0 ) {
 g2.drawString("Player " + model.getCurrentPlayer() + "'s  turn",
20, 20);

 // labeling the player on which side
 g2.drawString("Player " + model.getCurrentPlayer() + " ->",  350,
 350);
 g2.drawString("<- "+"Player " + model.getOtherPlayer(), 350,  30);
 } else {
 if (model.getWinningPlayer() == 0) {
 g2.drawString("Draw!", 20, 20);
 } else {
 g2.drawString("Player " + model.getWinningPlayer() + "  wins!",
20, 20);
```

```
}
}
```

- Strategy: BoardStrategy

5.
```
/**
 * Draw a Mancala board
 * @author Anh Thu, & Ashton Headley
 */
public abstract class BoardStrategy {

 protected final int outerPadding, innerPadding;
 protected final int paddingFromTop;
 protected final int pitWidth, pitHeight;
 protected final int storeWidth, storeHeight;
 protected final Color playerOneColor, playerTwoColor;
protected final Color backgroundColor;
 // private MancalaGame game;
 protected Model game;

 /**
 * Initialize the class
 *
 * @param game instance of MancalaGame  * @param
playerOneColor color to draw player one's pits in  * @param
playerTwoColor color to draw player two's pits in  * @param
backgroundColor
 */
 protected BoardStrategy(Model game, Color playerOneColor,
Color playerTwoColor, int outerPadding, int innerPadding, int
paddingFromTop, int pitWidth, int pitHeight,
 int storeWidth, int storeHeight, Color backgroundColor) {
 this.game = game;
 this.playerOneColor = playerOneColor;
 this.playerTwoColor = playerTwoColor;
 this.outerPadding = outerPadding;
 this.innerPadding = innerPadding;
 this.paddingFromTop = paddingFromTop;
 this.pitWidth = pitWidth;
 this.pitHeight = pitHeight;
 this.storeHeight = storeHeight;
```

```
this.storeWidth = storeWidth;
this.backgroundColor = backgroundColor;
}

/**
* Copy constructor
* @param other - object, from which we deep-copy values
*/
protected BoardStrategy(BoardStrategy other) {
this.game = new Model(other.game);
this.playerOneColor = other.playerOneColor;
this.playerTwoColor = other.playerTwoColor;
this.outerPadding = other.outerPadding;
this.innerPadding = other.innerPadding;
this.paddingFromTop = other.paddingFromTop;
this.pitWidth = other.pitWidth;
this.pitHeight = other.pitHeight;
this.storeHeight = other.storeHeight;
this.storeWidth = other.storeWidth;
this.backgroundColor = other.backgroundColor;  }

/**
* Draw one pit depending on x and y coordinates  * @param g2 -
2d Graphics, which is responsible for rendering 2D  figures, text and
images
* @param x - initial coordinate x
* @param y - initial coordinate y
*/
protected abstract void drawPit(Graphics2D g2, int x, int y);

/**
* Draw the storage spaces
* @param g Graphics object
*/
protected abstract void drawStores(Graphics g);

/**
* @return a deep copy of a specific board strategy  */
protected abstract BoardStrategy getCopy();

/**
* Get the player color for the current player
* @return the player's color
```

```
 */
 protected Color getCurrentPlayerColor() {
 return game.getCurrentPlayer() == 1 ? playerOneColor :
playerTwoColor;
 }

 /**
 * Get the player color for the other player
 * @return the player's color
 */
 protected Color getOtherPlayerColor() {
 return game.getOtherPlayer() == 1 ? playerOneColor :
playerTwoColor;
 }

 /**
 * Get the size of the board as a Dimension object  *
@return size of the board
 */
 public Dimension getSize() {
 int height = 3 * (outerPadding + pitHeight) + innerPadding  +20;
 int width = 6 * (pitWidth + innerPadding ) + 2 * (storeWidth  +
outerPadding);
 return new Dimension(width, height);
 }

 /**
 * Draw a row of pits
 * @param g Graphics object
 * @param x Beginning X position of row
 * @param y Beginning Y position of row
 */
 protected void drawRow(Graphics g, int x, int y) {
Graphics2D g2 = (Graphics2D) g;
 for (int i = 0; i < 6; ++i ) {
 drawPit(g2, x, y);
 x += pitWidth + outerPadding;
 }
 }

 /**
 * Draw the board pits and stores
```

```
 * @param g Graphics object
 */
 public void drawBoard(Graphics g) {
 Graphics2D g2 = (Graphics2D) g;
 drawStores(g2);

 int rowX = storeWidth + innerPadding * 2;

 g2.setColor(getCurrentPlayerColor());
 drawRow(g2, rowX, outerPadding);

 g2.setColor(getOtherPlayerColor());
 drawRow(g2, rowX, outerPadding + pitHeight + innerPadding );

 g2.setColor(Color.BLACK);
 drawPitLabels(g2);
 }

 /**
 * Draw the labels for each pit
 * @param g Graphics object
 */
 public void drawPitLabels(Graphics g) {
 Graphics2D g2 = (Graphics2D) g;
 if (game.getCurrentPlayer() == 1)
 {
 for (int i = 0; i < 6; ++i ) {
 g2.drawString("B" + i, getPitCenterX(i), storeWidth -  innerPadding);
 g2.drawString("A" + i, getPitCenterX(i),
getPitY(i)+(pitHeight*2)+ innerPadding-10);
 }
 }

 if (game.getCurrentPlayer() == 2)
 {
 for (int i = 0; i < 6; ++i ) {
 g2.drawString("A" + i, getPitCenterX(i), storeWidth -  innerPadding);
 g2.drawString("B" + i, getPitCenterX(i),
getPitY(i)+(pitHeight*2) +innerPadding-10);
 }
 }
 }
```

```
/**
* Retrieve the X position of a pit
* @param pit a pit number
* @return the pit's X position
*/
public int getPitX(int pit) {
int x;

// check if pit is a store
if ( pit == 6 || pit == 13 ) {
x = outerPadding + storeWidth / 2;

// subtract pit x from screen width
x = (pit == 6) ? getSize().width - x : x;  } else {

// reverse the top row numbers
if (pit > 6) pit = -pit + 12;

// begin with outside padding + mancala
x = outerPadding + storeWidth;

// add padding for each box
x += outerPadding * (pit+1);

// add boxes
x += pit * pitWidth;
}

return x;
}

/**
* Retrieve the Y position of a pit
* @param pit a pit number
* @return the pit's Y position
*/
public int getPitY(int pit) {

// check if a pit is a store or in the second row  if ( pit <=
6 || pit == 13 ) {
return outerPadding * 2 + pitHeight;
}

return outerPadding;
```

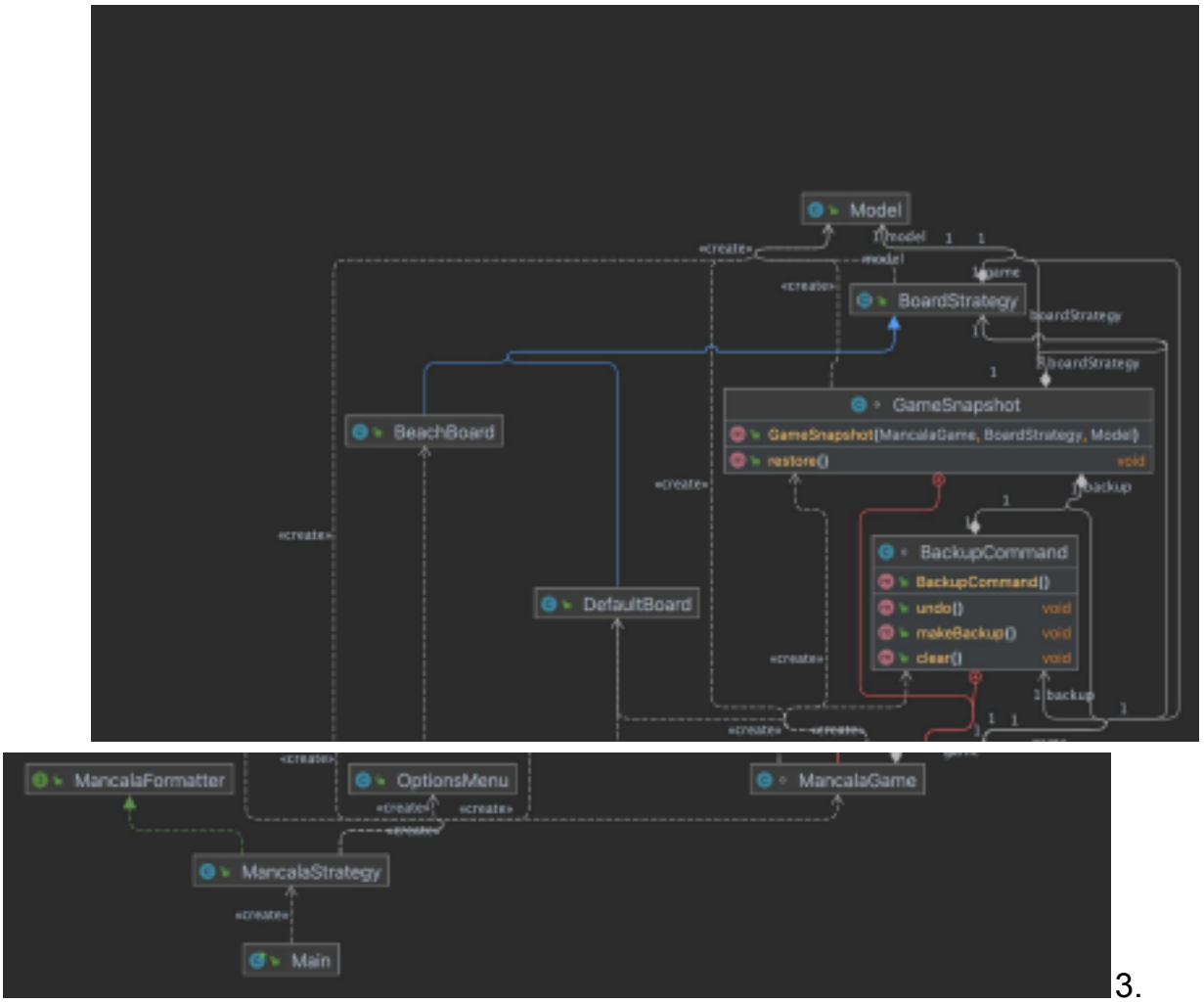
```
}

/**
* Get the X coordinate in the center of a pit  *
@param pit a pit number
* @return X position
*/
public int getPitCenterX(int pit) {
int x = getPitX(pit);

if (pit != 6 && pit != 13) {
x += pitWidth/2;
}

return x;
}

/**
* Get the Y coordinate in the center of a pit  *
@param pit a pit number
* @return Y position
*/
public int getPitCenterY(int pit) {
int y = getPitY(pit);

if (pit != 6 && pit != 13) {
y += pitHeight/2;
}

return y;
}

public void setGame(Model game) {
this.game = game;
}
}
```

- Concrete strategies: BeachBoard, DefaultBoard
- Creation of the concrete strategy:

6.
```
Model model = new Model();
BoardStrategy chosenBoard;

String[] boardOptionArray = {"Default", "Beach"};
```

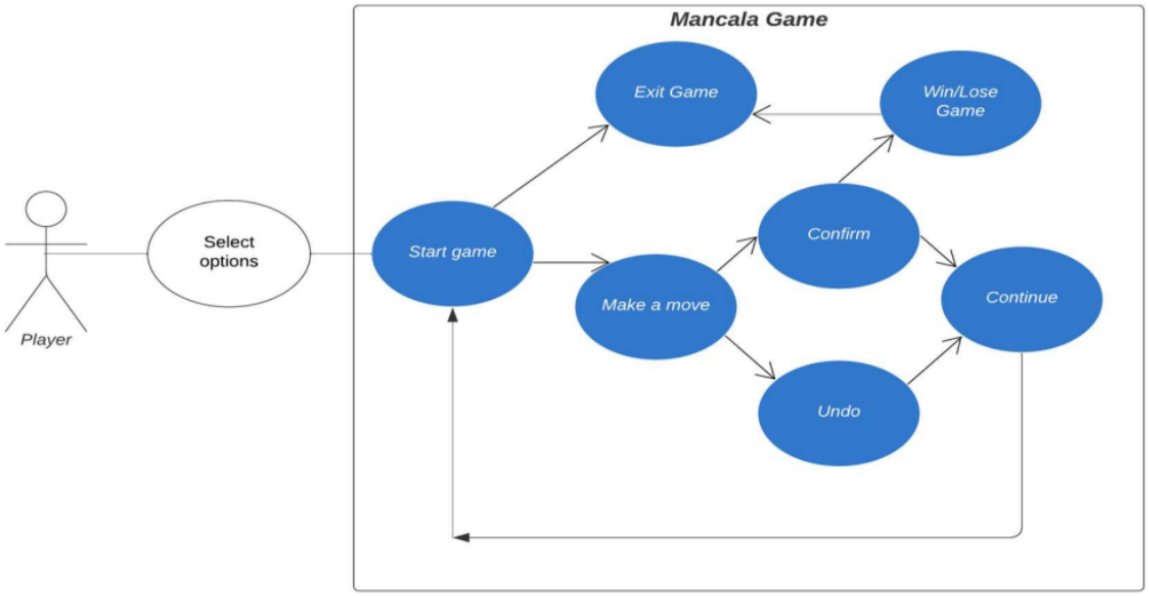
```
int boardOption = JOptionPane.showOptionDialog(null, "Select
the board style.", "Options Menu",
JOptionPane.DEFAULT_OPTION,
JOptionPane.INFORMATION_MESSAGE, null,
boardOptionArray, boardOptionArray[0]);
if (boardOption == 1)
{
 chosenBoard = new BeachBoard(model);
} else {
 chosenBoard = new DefaultBoard(model);
}

MancalaGame game = new MancalaGame(model, chosenBoard);
```

2. Class diagram:



3.

Use cases



Use case 1: **Start a game**

Step	User's Action	System's Response
1	User opens the application and clicks "start game" button	
2		System shows an empty mancala board.
3		System prompts "Enter the number of stones in each pit?"
4	User enters a number 3 or 4 to represent the number of stones in each pit.	
5		System populates a mancala board with the given number of stones entered by the user.

6	User number 1 clicks on the board to start the game.	

Use Case 2: **Undo one turn**

Step	User's Action	System's Response
1	User **makes a move** on their side of the board.	
2	User clicks on the undo button.	
3		System displays the number of undos this turn.
4		System subtracts one from the undo counter.
5		System finds out the number of stones in the pit the user selected before and removes one from each pit.
6		System restores the board state to once it was before the user makes a move.

Variation 1:

1) Start at 2.
2) User clicks the undo button again.
3) System does not change the board.

Variation 2:

1) Start at 2.

2) User clicks on a pit to **make a move**.
3) User presses the undo button again, but the user runs out of undo tries before the next player's turn.
4) System does not change the board.

Use Case 3: **Make a turn**

Step	User's Action	System's Response
1	User clicks on a pit on their side.	
2		System finds out how many stones are in that pit.
3		System adds one stone to a pit in a counter-clockwise direction until it runs out of stones.
4		System updates the board display.
5	User drops the last stone in an empty pit not on their side.	
6	User clicks the "end turn button" and allows the other player to select a pit.	

Variation 1:

1) Start at Step 1.
2) User clicks the "undo turn" button and carries out **Undo one turn**.
3) Continue with Step 1 until the undo counter runs out or User does Step 6.

Variation 2: Last stone dropped is in an empty pit on your side

1) Step 1 - 4
2) User drops the last stone on an empty pit on their side.
3) System finds that pit's location.

4) System adds that last stone and all the stones from the opposite pit into the user's Mancala.
5) Continues to Step 4

Variation 3: Last stone dropped in your mancala

1) Steps 1 - 4
2) System finds that the last stone dropped is in your mancala.
3) Repeat **Make a turn** again.

Variation 4: Dropped last stone in a non-empty pit

1) Start at Step 5. User dropped last stone in a non-empty pit
2) User repeats **Make a turn** again

Use Case 4: **End the game**

Step	User's Action	System's Response
1	Player A (user) clears their side of the board.	
2		System places all the stones on Player B (the other user)'s side that still has stones into Player B's mancala.
3		System adds up the total of stones in each player's mancala.
4		System displays which player has the most stones in their Mancala and who won the game.
5	User clicks the "end game" button.	

Variation 1:

1) User quits the entire program.

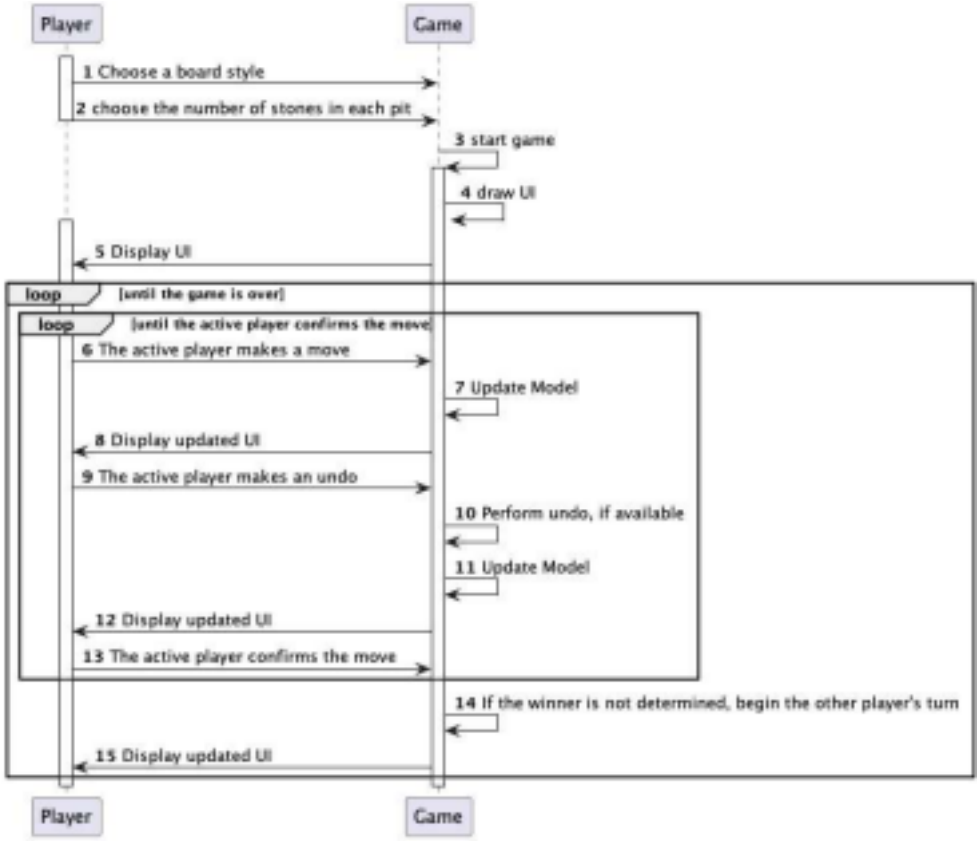
Variation 2:

1) User clicks the “forfeit” button.
2) System ends the game automatically.
3) System displays that the other user wins the game.

Variation 3:

1) User clicks the “end game” button before one side is cleared.
2) System continues with Step 3 & Step 4.

4. Sequence diagram:



1. During the first stages of the project, we used requirement analysis to understand the functional requirements our code needed to have. Based on the description of the mancala rules, we created a set of use cases to map out user actions and system responses for each rule in mancala.

We applied concepts from the graphical programming lecture to create the look and feel of our mancala board game. We created JPanels that had JButtons, and used the graphics package to draw stones with Ellipse2D.Double. Our graphical interface also had event handling mechanisms to add listeners, such as actionListener and mouseListener. In order to attach the listeners, we used lambda expressions and anonymous classes to create concrete classes for those listener functional interfaces.

As for the design portion of our project, we used the strategy design pattern to create two styles of our mancala board. We had an abstract class that implemented a strategy interface. The two board styles inherited its methods from that abstract class. We also needed to have a strong grasp of the MVC. At first our model class containing the data had graphical components, so we fixed our mistake and made our model purely functional. We also learned the difference between view and controller while implementing our project.

2. Firstly, we had to have a solid understanding of game theory, which helped us to better understand the strategic decision-making and planning that is required in order to play the game effectively. Additionally,we had to have a strong grasp of algorithmic thinking, which allowed us to simulate the game and execute the rules correctly.

We also needed to utilize our knowledge of logic and reasoning, as Mancala requires players to make logical deductions and predictions in order to develop effective winning strategies. And having been newly introduced to the MVC design pattern, it helped us better understand the way we wanted to go about our project.

We wouldn’t say there was much that we had learned to complete this project but rather go back and jog our memory about certain things we may have forgotten or needed some work on.

All in all, creating the Mancala game required a combination of knowledge from various fields, including game theory, algorithmic thinking and logic.