1. **Write the NAME of one of the controller classes (or class that contains a controller).**
   UndoButton.java - a class that creates an undo button that views and controls the model

   Copy and paste a code segment of the controller that calls the mutator of the model
   We decided to make the UndoButton its own class. This segment of code comes from the constructor where the button is created and an action listener is added to it.

```
this.addActionListener(new ActionListener() { // Adds an ActionListener for the button
	public void actionPerformed(ActionEvent e) {
		int previousModelSize = modelSize; // Holds the size of the board before the button was
		                                   // pressed
		updateModelSize(); // Updates the modelSize integer
		newTurn(previousModelSize); // Checks if it's a new turn, it's a new turn if the number of
		                            // elements in the model is larger than it was before the
		                            // button was pressed
		model.remove(); // Removes the last item in the model
		eligibility(); // Checks the eligibility of the button
		count++; // Increments count
		counter.push(count); // Adds count to the stack
		}
		});
```

2. **Write the NAME of the model class**
   Board.java - holds the data being stored

   Copy and paste a code segment of a mutator of the model that modifies data and also notifies view(s).
   This segment of code shows the remove() method of the Board class. Here the last element in the ArrayLists data and squares is removed.

```
	/*
	 * Removes the latest element from the ArrayLists
	 */
	public void remove() {
		data.remove(data.size() - 1);
		squares.remove(squares.size() - 1);
		setChanged();
		notifyObservers();
	}
```

   Give me the name of the mutator as well
   remove() - Removes the latest element from the ArrayLists and notifies observers

3. **Write the NAME of the view class**
   BoardViewer.java - This is an abstract class that implements the methods that are identical across each of the different themes. This takes advantage of the template method design pattern as well.

   Copy and paste a code the notification method of the view.

The update method is called when the model notifies its observers. Here we have BoardViewer.java call repaint() and updateSquares(). Repaint() redraws the board and updateSquares() essentially keeps track of what symbol (X or O) is in each square inorder for BoardViewer.java to check if it's in a winning state.

```
public void update(Observable o, Object arg) {
		if (o == model) {
			repaint(); // Redraws the panel
			updateSquares(); // Updates the squares
		}
	}
```

Show me how the notification method paints the view using the data from the model.
This segment of code comes from the paintComponent(Graphics g) method. Here we iterate through each element in the model and create the variables square and shape. These variables track which square the element is located in as well as what shape it is (X or O). Then we have a series of if statements that either draw X's or O's in the correct squares. This portion of code shows the first square as an example.

```
for (int i = 0; i < model.size(); i++) { // Iterates through each integer in the model
	String square = model.getSquare(i); // Gets the name of the square at index i in model
	int shape = model.getNumber(i); // Gets the integer at index i in the model

	// The first square
	if (square.equals("square1") && shape == 0) { // If the shape in the first square is equal to
							// zero, an X is drawn on the board
		Point xStart1 = new Point(10, 10);
		Point xEnd1 = new Point(90, 90);
		Point xStart2 = new Point(90, 10);
		Point xEnd2 = new Point(10, 90);
		Line2D.Double xLine = new Line2D.Double(xStart1, xEnd1);
		Line2D.Double xLine2 = new Line2D.Double(xStart2, xEnd2);
		g2.draw(xLine);
		g2.draw(xLine2);
	} else if (square.equals("square1") && shape == 1) { // If the shape in the first square is equal
							// to one, an O is drawn on the board
		Ellipse2D.Double circle = new Ellipse2D.Double(5, 5, 90, 90);
		g2.draw(circle);
	}
```

4. **Write the NAME of a strategy and copy the code**
   BoardStrategy.java - an interface that lists all the required methods for a BoardViewer

```
import java.awt.Graphics;
import java.util.Observable;
import javax.swing.JButton;
public interface BoardStrategy {

	/*
	 * Overrides the paintComponent method of the JPanel superclass.
	 * Creates the tic-tac-toe board and draws X's and O's as needed.
	 */
	public void paintComponent(Graphics g);

	/*
```

```
         * Creates a new square for each element in the model. Checks if the board is in
         * a winning state.
         */
        public void updateSquares();

        /*
         * Determines whether the board is in a winning state
         */
        public void winningState();

        /*
         * Disables the frame this panel is in
         */
        public void disableFrame();

        /*
         * Sets the visibility of the frame to true
         */
        public void visibility();

        /*
         * Enables the frame this panel is in
         */
        public void enableFrame();

        /*
         * Gets the classe's unique button
         * @return the classes button
         */
        public JButton getInitialButton();

        /*
         * Implements the update method of the Observer interface
         */
        public void update(Observable o, Object arg);

}
```

5. **Write the name of two concrete strategies**
   DarkTheme and RainbowTheme - these classes extend the BoardViewer abstract class.

6. **Copy and paste the code segment where you create a concrete strategy and plug-in into the context program.**
   This is where we create a concrete strategy and add it to the model. This takes place in TicTacToeTest.java

```
DarkTheme dt = new DarkTheme(model); // Creates a dark theme of the board(viewer and controller)
model.addObserver(dt); // Adds the theme to the model's list of observers
model.addStyle(dt); // Adds the theme to the model's list of styles
```

   This is how Board.java implements the strategy pattern. It holds a list of all the different styles, gets each style's button for the "selecting a style" frame, and adds each button to that frame. Then when the program starts, it displays this frame for the user to choose a style for the game.

```
public void initialScreen() {
	JFrame initialFrame = new JFrame(); // the initial screen that shows the different buttons for
					    // each of the styles

	Label title = new Label("Please select a style for the board");
	Font smallFont = new Font("Courier", Font.BOLD, 25);
	title.setFont(smallFont);

	createButtons(); // Creates a button for each style of the board

	for (int i = 0; i < buttons.size(); i++) { // Adds action listeners to the board
		int index = getRelatedStrategyIndex(buttons.get(i));
		buttons.get(i).addActionListener(event -> styles.get(index).visibility()); // When the
					    // button is pressed, the associated board becomes visible
		buttons.get(i).addActionListener(event -> initialFrame.setVisible(false)); // When the
					    // button is pressed, the initial screen is no longer visible
		buttons.get(i).addActionListener(event -> styles.get(index).enableFrame()); // When the
					    // button is pressed, the associated board is enabled
		}

		JPanel buttonPanel = new JPanel(); // A panel that holds all the buttons

		initialFrame.add(title, BorderLayout.NORTH); // Adds the title to the top of the frame
		for (JButton button : buttons) {
			buttonPanel.add(button); // Add each button to the panel
		}
		initialFrame.add(buttonPanel, BorderLayout.SOUTH); // Adds the panel to the bottom of the
								   // frame
		initialFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		initialFrame.pack();
		initialFrame.setVisible(true);
	}
```

7. **One page of paper that includes answers for the following questions:**
   Which materials/key concepts from this course did you apply on the project?
   Which topics did you have to learn through self-study in order to complete the project?

This Object-Oriented Design course taught us valuable lessons when it comes to preparing for a project. We learned about the different types of approaches to designing a program as well as how to communicate as a team to reach a goal. The design phase is a crucial procedure for all types of work and especially for our project. This phase is important because it helped guide and lay the framework for the final version of our product. First of all, as a team we were able to highlight the various requirements for the project by identifying the many nouns and verbs we came across as we read through the assignment for the first time. With this information, we were able to create CRC cards which helped us organize the various classes/responsibilities we needed, as well as solidify which classes would collaborate with one another. By identifying this vital information, we were then able to categorize our ideas into relevant diagrams such as Class Diagrams, Sequence Diagrams, and Use Cases. Each of these key diagrams held a heavy emphasis on the design phase and made writing the code ten times easier when it was finally time to implement our ideas. In terms of the class diagram, the CRC cards helped identify how classes would interact with each other so all we had to do was connect them with lines that showed the appropriate relationship. Then we worked on the use cases and used them to finish the sequence diagram. These two diagrams played an important role in

identifying how the code should run from start to finish. Using these concepts we learned in Chapter 2 allowed us to begin the coding phase with confidence that we had an understanding of how the overall program should run.

Coding is a diverse process, but the use of design patterns presents coders with the opportunity to follow a framework to achieve their goals. For our project, we identified the necessary patterns, such as the Model-Viewer-Controller Pattern as well as the Strategy Pattern. With the use of these two essential patterns, we were able to organize and produce our program concisely and coherently. For example, we implemented the MVC pattern in our program by creating a Board class which served as a model to store all our necessary data and an abstract BoardViewer class to observe the model as well as control it. BoardViewer represents our program visually and uses action listeners to listen to events and change the data in the model. When the model is changed, it notifies its observers to update which causes the board to be redrawn. In terms of the strategy pattern, we created an interface that held all the methods necessary for a BoardViewer. Then we created various classes that served as the different “themes” or styles of the board. These classes served as the concrete strategies because they implemented the strategy interface by extending the abstract BoardViewer class and implementing any abstract methods. Then we had our model hold a list of all the strategies, that way when we created the initial frame with buttons for each style, we could simply iterate through the list of styles and get their unique button. Overall, this Object-Oriented Design course is much more in depth than simply learning to create a plan before coding or learning to follow patterns to make coding simpler. It grants us the ability to think outside the box, which is exactly what we did when we also decided to implement the Template Method Design Pattern. This pattern wasn’t required for the project, but we decided to use it anyways to reinforce our code and make everything run more smoothly.

In terms of the topics we had to learn through self study in order to complete this project, there really wasn’t that much information we had to learn on our own. We felt like the previous assignment we completed, Homework Assignment #2, did a fantastic job preparing us for this project. We basically had a strong understanding of how to implement the MVC Pattern coming into this project, so the only thing we really didn’t have any experience with was the Strategy Pattern. As a result, we had to do a little bit of research on this topic, but it wasn’t too difficult to understand. The hard part was figuring out how to merge multiple design patterns in one program. Since we are working with multiple classes and objects, there were a lot of things to keep track of. Many times as a group, we had to take a step back and remember the ideas and implementation of the MVC and Strategy pattern before moving forward. Although we had to learn some aspects of this assignment on our own, the feeling of overcoming obstacles and solving problems throughout this project was invaluable as we improved our proficiency in problem-solving and tackling challenges as a team.