Write up for Design Pattern Assessment:

[NOTE: The following are the small code segments of our code. Please view and test the full code in our Java files.]
Github: https://github.com/Group19s/CS151Project

1. **Write the NAME of one of the controller classes (or class that contains a controller). Copy and paste a code segment of the controller that calls the mutator of the model.**

   One of the classes that contains a controller in our code implementation is the CellButton class. We have two methods buttonActionListener(JButton btn) and undo(), which call the mutator of the model.
   Here are the code snippets of the two methods that call the mutator of the model:

```
public void buttonActionListener(JButton btn) {
      btn.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent e) {
            // fetching the original text of the button on the board
            String buttonText = ((JButton) e.getSource()).getText();
            int d = Integer.parseInt(buttonText.substring(0, 1));
            int f = Integer.parseInt(buttonText.substring(1, 2));

            currentPlayer = nextPlayer();
            data.changeData(d, f, currentPlayer);  // calling the changeData method of
                                                   // BoardData class
            arrayBtn[d][f].setEnabled(false);//update itself

            if (data.CheckWinner()) {  // if we have a winner, disable all buttons
               buttonDisable();
               undoButton.setEnabled(false);
               return;
            }
            if (data.CheckGameOver()) {
               undoButton.setEnabled(false);
               JOptionPane.showMessageDialog(new JFrame("message"), "Game Over",
"Backup Problem", JOptionPane.ERROR_MESSAGE);
            }
         }
   });
```

```
}

public void undo() {
      if (currentPlayer == p0 && p0.undoCheck == 0 && p0.undo < 3) {
         p0.undo++;
         p0.undoCheck = 1;
         p1.undoCheck = 0;
         L.removeLast();  // remove the player who played last

         int[] position = data.setPrevious();  // calling the setPrevious method of
                                               // BoardData class
         arrayBtn[position[0]][position[1]].setEnabled(true);
         arrayBtn[position[0]][position[1]].setText(String.valueOf(position[0]) +
position[1]);

      } else if (currentPlayer == p1 && p1.undoCheck == 0 && p1.undo < 3) {
         p1.undo++;
         p1.undoCheck = 1;
         p0.undoCheck = 0;
         L.removeLast();   // remove the player who played last

         int[] position = data.setPrevious();
         arrayBtn[position[0]][position[1]].setEnabled(true);
         arrayBtn[position[0]][position[1]].setText(String.valueOf(position[0]) +
position[1]);

      } else {
         JOptionPane.showMessageDialog(new JFrame("message"), "You can't undo",
"Inane error", JOptionPane.ERROR_MESSAGE);
      }
}
```

2. **Write the NAME of the model class. Copy and paste a code segment of a mutator of the model that modifies data and also notifies view(s). Give me the name of the mutator as well.**

The name of the model class is the BoardData class. We have two mutator methods setPrevious() and changeData(int xp, int yp, Player event), which modify data and also notify views.

Here are the code snippets of the two methods that modify data and also notify views:

```
public int[] setPrevious() {
    int[] position = new int[2];  // making a new int array of size 2
    cellValue[pxp][pyp] = null;
    position[0] = pxp;
    position[1] = pyp;
    setChanged();
    notifyObservers();
    return position;
}

public void changeData(int xp, int yp, Player event) {
    cellValue[xp][yp] = event;  // sets the player in that cellValue's row and column
    pxp = xp;
    pyp = yp;
    setChanged();
    notifyObservers();
}
```

3. **Write the NAME of the view class. Copy and paste a code of the notification method of the view and show me how the notification method paints the view using the data from the model.**

Our view classes are the CellButton class and the AnimatedCellButton. The update(Observable o, Object arg) methods from each view class paint the view using the data from the model by calling getData() method from the BoardData class. If it is Player 1's turn, then the view class will display a green-colored button for that player. If it is Player 2's turn, then the view class will display a red-colored button respectively. The only difference in the update(Observable o, Object arg) methods between these two view classes is that the AnimatedCellButton also calls the getButton() method from the Button class.

Here are the code snippets of the methods that paint the view using the data from the model:

CellButton class:

```
public void update(Observable o, Object arg) {
    data = (BoardData) o;
    for (int n = 0; n < 3; n++) {
      for (int i = 0; i < 3; i++) {
        if (data.getData()[n][i] != null) {
          if (data.getData()[n][i].id == 1) {
            arrayBtn[n][i].setBackground(Color.GREEN);
            arrayBtn[n][i].setOpaque(true);
            arrayBtn[n][i].setBorderPainted(true);
            arrayBtn[n][i].setText(p0.name);
          }
          if (data.getData()[n][i].id == 2) {
            arrayBtn[n][i].setBackground(Color.RED);
            arrayBtn[n][i].setOpaque(true);
            arrayBtn[n][i].setBorderPainted(true);
            arrayBtn[n][i].setText(p1.name);
          }

        } else {
          arrayBtn[n][i].setForeground(Color.GRAY);
          arrayBtn[n][i].setOpaque(false);
          arrayBtn[n][i].setBackground(Color.BLACK);
```

```
            }

        }
    }
}
```

AnimatedCellButton class:

```
public void update(Observable o, Object arg) {
   data = (BoardData) o;
   for (int n = 0; n < 3; n++) {
      for (int i = 0; i < 3; i++) {
         if (data.getData()[n][i] != null) {
            if (data.getData()[n][i].id == 1) {
               arrayBtn[n][i].getButton().setBackground(Color.GREEN);
               arrayBtn[n][i].getButton().setOpaque(true);
               arrayBtn[n][i].getButton().setBorderPainted(true);
               arrayBtn[n][i].getButton().setText(p0.name);
            }
            if (data.getData()[n][i].id == 2) {
               arrayBtn[n][i].getButton().setBackground(Color.RED);
               arrayBtn[n][i].getButton().setOpaque(true);
               arrayBtn[n][i].getButton().setBorderPainted(true);
               arrayBtn[n][i].getButton().setText(p1.name);
            }

         } else {
            arrayBtn[n][i].getButton().setForeground(Color.GRAY);
            arrayBtn[n][i].getButton().setOpaque(false);
            arrayBtn[n][i].getButton().setBackground(Color.BLACK);
         }
      }
   }
}
```

4. **Write the NAME of a strategy and copy the code.**

We have the Strategy Interface called AnimatableButtonCreator.

Here is the code snippet of the Strategy Interface:

```
public interface AnimatableButtonCreator {
   JButton createButton(int xp, int yp, String value);
}
```

5. **Write the name of two concrete strategies. (Just names required).**

The names of two concrete strategy classes that implement the AnimatableButtonCreator Interface are NoAnimationButtonCreator and AnimatedButtonCreator classes.

6. **Copy and paste the code segment where you create a concrete strategy and plug-in into the context program.**

We have two concrete strategies: NoAnimationButtonCreator and AnimatedButtonCreator classes.

We have the superclass GameView which is an abstract class. The two subclasses CellButton and AnimatedCellButton extend the superclass GameView. The superclass GameView and subclasses CellButton and AnimatedCellButton use the AnimatableButtonCreator Interface to call the algorithm defined by NoAnimationButtonCreator and AnimatedButtonCreator concrete strategy classes, so they are Context classes.

Here are the code snippets of the two Concrete strategy classes:

```
class NoAnimationButtonCreator implements AnimatableButtonCreator {
   public JButton createButton(int xp, int yp, String value) {
      JButton btn = new JButton(xp + Integer.toString(yp) + value);
      btn.setName("n+Integer.toString(i)+value");
      btn.setBackground(Color.GRAY);
      btn.setLocation(xp, yp);
      btn.setPreferredSize(new Dimension(100, 100));
      return btn;
   }
}
```

```
class AnimatedButtonCreator implements AnimatableButtonCreator {
   public JButton createButton(int xp, int yp, String value) {
      Button btn = new Button(xp + Integer.toString(yp) + value);
      btn.getButton().setName("n+Integer.toString(i)+value");
      btn.getButton().setBackground(Color.GRAY);
      btn.getButton().setLocation(xp, yp);
      btn.getButton().setPreferredSize(new Dimension(100, 100));
      return btn.getButton();
   }
}
```

Here are the code snippets of the Concrete Strategies plug-in into the Context program:

GameView class:

-the relation between GameView(Context) and AnimatableButtonCreator (Strategies) is dependency this is why we create this method

```
public AnimatableButtonCreator executeStrategy(AnimatableButtonCreator strategy) {
   return strategy;
}
```

CellButton class:

-we use super heir because it the context class is inherited

```
public JButton createButton(int xp, int yp) {
      arrayBtn[xp][yp] =super.executeStrategy(new
NoAnimationButtonCreator()).createButton(xp, yp, value);
      return arrayBtn[xp][yp];
}
```

AnimatedCellButton class:

-we use super heir because it the context class is inherited

```
public JButton createButton(int xp, int yp) {
      JButton btn =super.executeStrategy(new
AnimatedButtonCreator()).createButton(xp, yp, value);
      arrayBtn[xp][yp] = new Button(btn);
      return btn;
}
```