Homework 5 Recap

CSE 331

GUIs can be Gooey!

These issues presented here don't necessarily reflect grades.

We pulled these at random from your submissions.

Some of these are good, some are bad.

Models

Have ‘em. Not everything should live in the GUI

Model: Players

It’s good to have models behind the GUI

```
public final class Player {

    private String name;
    private int wins;
    private int losses;
    private String icon;
```

Model: Game State

A good way to track what state the GUI should be in.

```
public enum GameState {
   NOT_YET_BEGUN, BETWEEN_GAME, GAME_IN_PROGRESS;
}
```

Model: Tic-Tac-Toe Board

How do you represent the state of the board?

```
private String[][] gameBoard;

private int[][] gameBoard;

public enum Symbol {
   X, O, Neither;
}

private Symbol [][] board;
```

Model: Tic-Tac-Toe Board

How do you represent the state of the board?

```
public class Spot {
   private Player taken;

   private Spot() {
      taken = null;
   }
   public Player owner();
   public void capture(Player p);
   public boolean isEmpty();
}
```

Model: Winning

```
private Player checkWinner(){
        if (!board.get(Position.UPLEFT).isEmpty() &&
                        board.get(Position.UPLEFT).whosHere().equals(board.get(Position.UP).whosHere()) &&
                        board.get(Position.UPLEFT).whosHere().equals(board.get(Position.UPRIGHT).whosHere()))
                return board.get(Position.UP).whosHere();
        else if (!board.get(Position.CENTERLEFT).isEmpty() &&
                        board.get(Position.CENTERLEFT).whosHere().equals(board.get(Position.CENTER).whosHere()) &&
                        board.get(Position.CENTERLEFT).whosHere().equals(board.get(Position.CENTERRIGHT).whosHere()))
                return board.get(Position.CENTER).whosHere();
        else if (!board.get(Position.LOWLEFT).isEmpty() &&
                        board.get(Position.LOWLEFT).whosHere().equals(board.get(Position.LOW).whosHere()) &&
                        board.get(Position.LOWLEFT).whosHere().equals(board.get(Position.LOWRIGHT).whosHere()))
                return board.get(Position.LOW).whosHere();
        else if (!board.get(Position.LOWLEFT).isEmpty() &&
                        (board.get(Position.LOWLEFT).whosHere().equals(board.get(Position.CENTER).whosHere()) &&
                        board.get(Position.LOWLEFT).whosHere().equals(board.get(Position.UPRIGHT).whosHere())))
                return board.get(Position.CENTER).whosHere();
        else if (!board.get(Position.UPLEFT).isEmpty() &&
                        (board.get(Position.UPLEFT).whosHere().equals(board.get(Position.CENTER).whosHere()) &&
                        board.get(Position.UPLEFT).whosHere().equals(board.get(Position.LOWRIGHT).whosHere())))
                return board.get(Position.CENTER).whosHere();
        else if (!board.get(Position.LOW).isEmpty()&&
                        (board.get(Position.LOW).whosHere().equals(board.get(Position.CENTER).whosHere()) &&
                        board.get(Position.LOW).whosHere().equals(board.get(Position.UP).whosHere())))
                return board.get(Position.CENTER).whosHere();


        ...
```

This is scary! Lots of duplication.

Model: Winning

Use helper methods to simplify checking for the winner.

```
private boolean winRow(Symbol symbol, int row)
private boolean winColumn(Symbol symbol, int col) {
private boolean winLeftToRightDiagonal(Symbol symbol)
private boolean winRightToLeftDiagonal(Symbol symbol)
```

GUI Design

Keep it modular and clean.

GUI: Buttons

Don't forget Java basics…

```
private void disableMainGameButtons() {
    button1.setEnabled(false);
    button2.setEnabled(false);
    button3.setEnabled(false);
    button4.setEnabled(false);
    button5.setEnabled(false);
    button6.setEnabled(false);
    button7.setEnabled(false);
    button8.setEnabled(false);
    button9.setEnabled(false);
}
```

Or:

```
private JButton lt, t, rt, l, c, r, lb, b, rb;
```

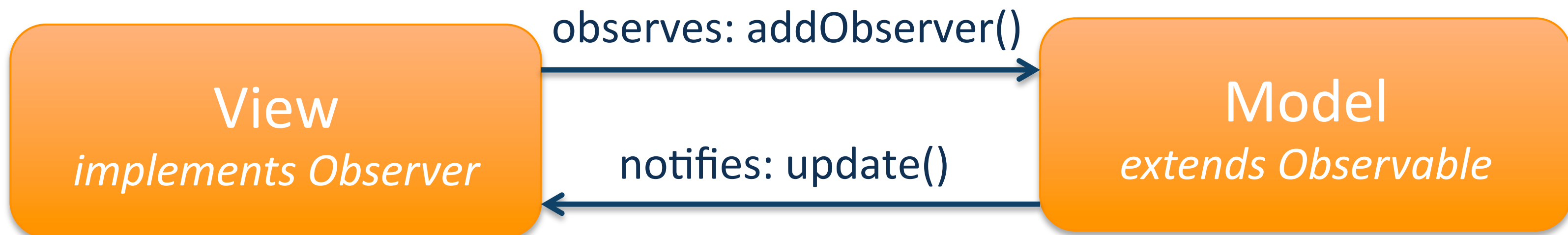
GUI: Buttons

Don't forget Java basics… use arrays!

```
private JButton[][] buttons;
```

GUI: Observer Pattern

A great way to simplify interactions between view and model.



GUI: Observer Pattern

Couldn't this view observe the model and change when it resets?

```
public void actionPerformed(ActionEvent event) {
    ...
    if(sure == JOptionPane.YES_OPTION) {
        resetButtons(false, true, true);
        player1.setText("Player 1");
        player2.setText("Player 2");
        p1wins.setText("0");
        p2wins.setText("0");
        p1losses.setText("0");
        p2losses.setText("0");
        status.setText("Welcome to Tic-Tac-Toe!");

        game.reset();
    }
```

GUI: Duplication

These do almost the same thing.

```
private Container setUpPlayer1Panel(){
private Container setUpPlayer2Panel(){
```

Option A: Parameterize it.

```
private Container setUpPlayerPanel(Player p) {
```

Option B: Move it into its own class.

```
public PlayerPanel(Player p) {
```

GUI: Setup Code

This is from a setupComponents() method:

```
mainFrame = new JFrame("CSE 331 Tic-Tac-Toe");
mainFrame.setLocation(300, 100);
mainFrame.setSize(500, 500);
mainFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

nameLabel1 = new JLabel("Name:");
nameLabel2 = new JLabel("Name:");
name1 = new JTextField(8);
name2 = new JTextField(8);
winsLabel1 = new JLabel("Wins:");
winsLabel2 = new JLabel("Wins:");
wins1 = new JLabel();
wins2 = new JLabel();
lossesLabel1 = new JLabel("Losses:");
lossesLabel2 = new JLabel("Losses:");
losses1 = new JLabel();
losses2 = new JLabel();

grid1 = new JButton();
grid2 = new JButton();
grid3 = new JButton();
grid4 = new JButton();
grid5 = new JButton();
grid6 = new JButton();
grid7 = new JButton();
grid8 = new JButton();
grid9 = new JButton();

buttonToIndex = new HashMap<JButton, Integer>();
buttonToClicked = new HashMap<JButton, Boolean>();

buttonToIndex.put(grid1, 0);
buttonToIndex.put(grid2, 1);
buttonToIndex.put(grid3, 2);
buttonToIndex.put(grid4, 3);
buttonToIndex.put(grid5, 4);
buttonToIndex.put(grid6, 5);
buttonToIndex.put(grid7, 6);
buttonToIndex.put(grid8, 7);
buttonToIndex.put(grid9, 8);
```

Don’t put everything in one method.

Split it up into separate methods.

For complicated panels create a JPanel subclass.

GUI: Setup Code

Better modularity:

```
public class TicTacToeGUI extends JFrame {

    private GamePanel gamePanel;
    private NotificationAreaPanel notificationAreaPanel;
    private StatsPanel statsPanel;
```

GUI: Setup Code

Better structure:

```
public class PlayerInfoPanel extends JPanel implements Observer {
    private JTextField nameField;
    private JLabel wins;
    private JLabel losses;
    private Symbol symbol;
    private Game game;

    public PlayerInfoPanel(Symbol symbol, Game game);
```

GUI: ActionListeners

Don't have a "hear everything" listener. Split it based on similar functionality.

```
if (event.getSource() == button1 || event.getSource() == button2
    || event.getSource() == button3 || event.getSource() == button4
    || event.getSource() == button5 || event.getSource() == button6

    || event.getSource() == button7 || event.getSource() == button8
    || event.getSource() == button9) {
    ...
} else if (event.getSource() == newGame) {
    ...
} else if (event.getSource() == reset) {
    ...
} else if (event.getSource() == exit) {
    ...
}
```

GUI: ActionListeners

Anonymous inner classes. Can be convenient.

```
ActionListener exitListener = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        System.exit(0);
    }
};

exit.addActionListener(exitListener);
```

The End!

Questions?