CS 2300 Lab MVC GUI – Part 3 Week 10

Control Panels and Programmer Panel

The goal of this project is to set up a set of basic classes implementing a minimal MVC (Model View Controller) GUI (Graphical User Interface). You can subsequently use it as a template for full featured GUI projects.

0. Discussion topics to read up on: `protected`, `abstract`.

1. Continue on with the project from the last lab. This lab will closely parallel the classroom development.

2. **Create a new class** called `PnlAbsCtrl.` As always, be sure the classes go into the mvc22f package. Adjust it so that it looks like this:

```
public abstract class PnlAbsCtrl extends JPanel
```

Notice you will have to add the keyword `abstract` by hand. This class is `abstract` because is used ONLY as the parent class for other panels. Obviously this class cannot also be NOT `final`, because it will be the parent of other classes!

Put a field in the new class:

```
    protected final Model model;
```

Note that this is `protected`, not `private` as usual. This is done to illustrate the concept of a protected field.

2.1 Automatically generate a constructor for this class: ALT+INS, choose c-tor, click the simplest parent c-tor (the last one), OK, then OK again. Add a statement:

```
public PnlAbsCtrl(Model model) {

    this.model = model;

    setUp();

}
```

This will cause a syntax error due to the method `setUp` not being found. Fix this by adding this method right after the c-tor:

```
private void setUp() {

    setBorder(BorderFactory.createEtchedBorder());

}
```

A good question is: Why not just put the `setBorder` method call inside the c-tor? Doing so will cause the following error:

“calling an **overridable** method from a constructor.”

Putting an overridable method (such as `setBorder`) in a c-tor can cause security problems, so you will get an error. We get around this issue with an “end run”, namely putting the overridable method in a `private` method named `setUp`, which eliminates any potential issues. Using a `setUp` method is also a good way to avoid cluttering up the c-tor.

The `PnlAbsCtrl` class will be used as a parent class for several (about 5) “mini panel” classes, defined next.

2.2. The purpose of the `PnlAbsCtrl` parent class is to:

2.2.1. give each mini panel a consistent look (with an etched edge border), and

2.2.2. define the `model` field in one place (in the `PnlAbsCtrl` class) rather than in all of the five **child classes**. Because the `model` is `protected` (not `private`), the child classes will be able use the `model`. This might or might not be better in any given situation. Here it is just an illustration of another way of doing things.

3. **Control panel classes**. We will start with skeleton classes so we can have a running program. Later we will add functionality to these classes.

3.1. Create a new class called `PnlProgr`, and make it extend the class `PnlAbsCtrl`. As before, be sure this and all classes go in the `MVC22F` package. Make this class look like this:

```
public final class PnlProgr extends PnlAbsCtrl {}.
```

This will cause an error, indicated by `PnlAbsCtrl` turning red. Click on red word, then ALT+ENTER. Choose “Create c-tor matching super” to fix the error by creating an appropriate c-tor for the class. OR, you might get a red balloon to click on that will offer to write the c-tor.

Mass production!!! Next do the **exact same thing** to create 5 more classes; `PnlColor, PnlSolid`, `PnlSizeX`, `PnlSizeY`, and `PnlTask` Each of these should extend `PnlAbsCtrl` and have the c-tor added. We will subsequently add code to make these panels form the controls for the application.

4. Go to the `Controller` class and enhance the c-tor with several statements, so that it looks like this:

```
public Controller(Model model) {

    this.model = model;

    setBorder(BorderFactory.createLineBorder(Color.GREEN));

    setLayout(new GridLayout(6, 1));

    add(new PnlProgr(model));
    add(new PnlTask(model));

    add(new PnlSizeX(model));

    add(new PnlSizeY(model));

    add(new PnlColor(model));

    add(new PnlSolid(model));

}
```

In this c-tor, we are instantiating (with the `new` keyword) instances of the five new classes. We do this (without giving them a name) right inside the `add` statement, exactly where they are needed. Instances like this (that do not have a name) are called **anonymous instances**. Being able to do this (because of **garbage collection**) greatly simplifies advanced programming.

This is a legal program, so run it and see how it looks. It should look almost the same as before. However, note that the control panel (on left side of the frame) is now divided up into six little boxes. The borders of these boxes have an "etched look". That is because of the type of border we used in the parent class, `PnlAbsCtrl`. This is an example of achieving a consistent "look and feel" by using inheritance.

Grab the corner of the frame with the mouse and make it bigger and smaller, and watch how these things change.

We will put some stuff in the six boxes to make the application look more like a GUI.

5. For the first control panel we will just put a title and the name of the programmer (That is YOU!). Go to the `PnlProgr` class. Add some code so that it looks exactly like the following, EXCEPT put in your name, NOT Ralph Kelsey (I am probably the only Ralph Kelsey involved in this project!):

```
public final class PnlProgr extends PnlAbsCtrl {

    private final JLabel jlProgrammer = new JLabel("Programmer:");

    private final JLabel jlSpacer = new JLabel(" ");

    private final JLabel jlYourName = new JLabel("Ralph Kelsey");


    public PnlProgr(Model model) {

        super(model);

        add(jlProgrammer);

        add(jlSpacer);

        add(jlYourName);

    }
}
```

Here we used the **Swing component** called `JLabel`. A `JLabel` is just a panel with some text (or pictures, whatever) on it. We are instantiating three labels (`jlProgrammer` , `jlSpacer`, and `jlYourName`) and `add`ing them to the `PnlProgr` we are constructing. The three `JLabel`s are placed in a row, going left to right, in the order they were added. This is because the default `FlowLayout` layout manager places them like that.

Put your own name in the `jlYourName` label (which is probably not `Ralph Kelsey`).

Notice we ALWAYS use a standard convention for naming the elements comprising our GUI. All `JLabel`s start with the letters `jl` and then the name of what they are. If you do this 100% of the time, it is a lot easier to keep track of what components are.

The `jlSpacer` label will be put in the middle between a label on the left and something on the right. This allows you to add or subtract some extra spaces and adjust how things line up.

Run the program. Now you should see the Control part of the GUI (left hand side) as a stack of five or six boxes. You see the etchedBorder we put in the superclass (`PnlAbsProgr`), so all the child classes get it automatically. This is how **inheritance** pays off big! We only added some “stuff” to the top box, the “Programmer Panel”. Note how the `GridLayout` manager figures out how much width it needs for that panel, and makes all the other ones the same size.

6. Next we will add another choice for the user to control: what should be drawn in the View. We will call this the `task`. We will retrofit things so that the ellipse drawn already will be just one of several possible choices (or, tasks). We will use an **enum** (discussed below) to keep track of what task to perform.

7. **Refactoring**. We will make the `paintComponent` method draw different possible things. It is a good idea to first clean the existing version up a bit to make this easier. Making some changes that do not change the operation of the program, but make it cleaner is called refactoring. Sometimes this is as simple as changing the name of a variable. In this case, we will do a little reorganizing.

It is a good habit to always be refactoring, because as you go along, you realize easier and cleaner ways to arrange your code. The main change here is to convert some ad hoc variables into class fields and create a method to initialize them. The advantage is that the new features (to be subsequently added) will be easier to implement.

7.1. Go to the `View` class. See the six `int`s in the `paintComponent` method, starting with `xCenter` and ending with `height`? Make each of them a **field**. This means write these up at the top of the `View` class, right after model, like this:

```
    private final Model model;

    private int xCenter;
```

…

```
    private int height;
```

This will make them **state variables**, or **fields**. The advantage is that they can be accessed anywhere within the class. This will also cause some errors to pop up where they are already defined. That will be fixed in a minute.

7.2. Create an additional method:

```
   private void setUp(){  }
```

right after the constructor.

7.3. Take the six lines of code, starting with

```
    int xCenter = getWidth() / 2;
```

and move them into the new method. Then remove the keyword `int` from the beginning of each of the six lines. This means we are no longer creating a new variable with that name on the stack, but are adjusting some fields of the class. This will prove advantageous.

The error messages should have gone away. This method will do the same work as before to set up the needed values for drawing a picture.

7.4. In the `paintComponents` method, right after the `super.paintComponents(g)` statement, put

```
setUp();
```

to make these calculations happen.

Run the program to be sure everything worked.

Here we have **refactored** to make the program better organized, and checked to be sure that everything was done correctly.

7.5. Next is another refactoring. Click on the opening brace `{` of the `setUp` method. It will turn blue. Look down a few lines, and notice that the matching closing brace `}` is also blue. This is an extremely useful trick to correctly find the end of a method.

Right after the blue brace marking the end of the `paintComponent` method, create a new method:

```
    private void paintEllipse(Graphics g){  }
```

7.6. Move the "if -else" structure out of `paintComponent` and into `paintEllipse`.

7.7. Put the method call

```
paintEllipse(g);
```

In `paintComponent`, right where the “if – else” had been before it got moved.

7.8. These two methods should now look like this:

```
protected void paintComponent(Graphics g){

    super.paintComponent(g);

    setUp();

    g.setColor(model.getColor());

    g.clearRect(0,0, getWidth(), getHeight());

    paintEllipse(g);

}

private void setUp(){ … }


private void paintEllipse(Graphics g){

    if (model.isSolid()) {

        g.fillOval(xStart, yStart, width, height);

    } else {

        g.drawOval(xStart, yStart, width, height);

    }

}
```

8. We are done refactoring (for now)!! But first run the program and be sure it works exactly like it did before.

What did this refactoring buy us?

A cleaner setup. The methods are smaller (always a good thing). Also, the original `paintComponent` method was doing two different tasks: (1) setting up some coordinates, and (2) drawing a picture. That was a bad combination, because we will always want to set up the coordinates, but we might want to draw a different picture.

It is usually a good idea to be sure that a method does **one thing only**. If you find out that a method is doing several things, break it up into smaller methods, each doing one thing, and call these new methods from the original method.

Before long, we will put in some other `paintXX` methods. This will be a lot easier because of the cleaner architecture obtained by the refactoring.

9. **Choose what picture to draw**. One more refactoring will set it up for easy addition of additional features. Go to the `paintComponent` method in the `View` class. Replace the statement:

```
     paintEllipse(g);
```

With this switch structure:

```
switch(model.getTask()){
    case ELLIPSE:
        paintEllipse(g);
        break;
    default:
        throw new RuntimeException("Bummer! Reached default case "
                    + "in View.paintComponent switch");
}
```

So far, there is only one case implemented (namely, `ELLIPSE`), and also that is the default task assigned in the Model, so the program should draw an ellipse.

Run the program. Everything should work exactly as before.

In subsequent labs, additional cases will be added to draw different stuff!

What to turn in.

1. Open file explorer

2. Find your project (likely C:\Users\<user>\IdeaProjects\MVC22F)

3. Right click on your project folder (this should contain src\mvc22f with all your .java files)

4. Select Send to > Compressed (zipped) folder

5. Submit MVC22F.zip to Blackboard