**Video #3: Adding actions to an App in the Code View of MATLAB's App Designer**

Our final step in the process of building an interactive program with MATLAB's App Designer is to add code that captures the actions we want the program to perform when the user interacts with our graphical user interface. We do this in the Code View of App Designer. I'm going to write new code for this program from scratch, starting with the new version of the energyApp program that just has the visual layout completed. In the Code View, we can see a thumbnail of the visual layout in the lower left corner.

So the code file initially contains a lot of code that MATLAB generated automatically, based on our visual layout. It's hard to tell, but this code is actually all displayed on a light gray background, which means we can't edit this code - even if I try, by hitting my delete key or typing new text, I can't change it. But let's take a quick look at this code to see what's there.

At the top is a list of properties - you'll recognize the names, there's one for each GUI component that we created, for example, here's the sourceDropDown and plotButton. These properties are all stored in a structure named app, so we'll refer to them in the code we write, as app.componentName, like app.sourceDropDown. You'll also see these expressions (Access = public) and (Access = private). Anything that's private is just accessible inside this code file. Public things are accessible from outside this App, but we won't be accessing things from the outside, so everything we create will be private.

```
1   classdef energyAppnew < matlab.apps.AppBase
2
3       % Properties that correspond to app components
4       properties (Access = public)
5           UIFigure                matlab.ui.Figure
6           UIAxes                  matlab.ui.control.UIAxes
7           sourcesLabel            matlab.ui.control.Label
8           sourceDropDownLabel     matlab.ui.control.Label
9           sourceDropDown          matlab.ui.control.DropDown
10          widelineCheckBox        matlab.ui.control.CheckBox
11          plotButton              matlab.ui.control.Button
12          plottitleEditFieldLabel matlab.ui.control.Label
13          plottitleEditField      matlab.ui.control.EditField
14          closeButton             matlab.ui.control.Button
15          produceButton           matlab.ui.control.StateButton
16      end
17
18      % Component initialization
19      methods (Access = private)
20
```

Scrolling down, we see more code that refers to these GUI components and the properties that we set in the Inspector when we were working in the Design View. For example, for the Drop Down menu, the app.sourceDropDown, we see properties like the Items in the menu, font size, position of this menu on the canvas and its Value that's the item selected by the user. All the properties we set for our components are here.

```
% Create sourceDropDown
app.sourceDropDown = uidropdown(app.UIFigure);
app.sourceDropDown.Items = {'coal', 'gas', 'oil', 'nuclear', 'renewable', ''};
app.sourceDropDown.FontSize = 20;
app.sourceDropDown.Position = [439 371 146 26];
app.sourceDropDown.Value = 'coal';
```

So what do we want to add to this program for our particular application? The first thing to think about is, is there information that our program needs to access, that we might want to set up at the outset? This may be data that's stored in a separate file, that we want to load into the workspace for this program, so we can access this data while the program is running. In the case of our energyApp here, it's the data on production and consumption of various energy sources. There's another file in my Current Folder named setupEnergy - let's have a look at this code file.

```
function data = setupEnergy
% data = setupEnergy
% sets up the data for the energyApp program

% years in which data is available
data.years = [1960 1970 1980 1990 2000];

% tables of data on production and consumption of the five energy sources
% (coal, natural gas, petroleum, nuclear, renewable), over the five years
% shown above (in quadrillion BTU's). The 5 energy sources are represented
% in different columns, and the 5 years are represented in different rows
data.produce = [10.82  14.61  18.60  22.46  22.62;
                12.66  21.67  19.91  18.36  19.66;
                14.93  20.40  18.25  15.57  12.36;
                 0.01   0.24   2.74   6.16   7.86;
                 1.61   4.08   5.49   6.14   6.16];
data.consume = [9.84  12.26  15.42  19.25  22.58;
               12.39  21.79  20.39  19.30  23.92;
               19.92  29.52  34.20  33.55  38.40;
                0.01   0.24   2.74   6.16   7.86;
                1.66   4.10   5.71   6.25   6.16];
```

setupEnergy is a function with no inputs and one output called data. It creates three variables, a vector of 5 years and two matrices with production data for the 5 energy sources over those 5 years, and similar consumption data - each row of the matrix is a different energy source. Note the names of these variables, data.years, data.produce, data.consume - these values are stored in the fields of a structure named data, and that whole structure is returned by this function.

So where in our program, will we add code to set up this data? We're going to create a new property that's assigned to the data returned by the setupEnergy function. To add a new property, I'll go to this icon in the toolbar at the top, with +P and Property below it, and click on the icon. This opens up a place in the code file where I can add a new property - the code has a white background that allows me to edit it. Let's change the Property name to data. We'll assign it to the structure returned by setupEnergy, and add a comment. When we refer to this property in our code, we'll write app.data, because new properties we create here are automatically stored in the app structure as well.

```
  properties (Access = private)
      Property % Description
  end

properties (Access = private)
    data = setupEnergy;      % years, production & consumption data
end
```

Now we have the data set up for our program, what about the actions we want to perform when the user interacts with the GUI? There are three GUI components for which we want to add actions - the plot button, close button, and state button that toggles between produce and consume. I'll start with the simplest button, the close button, and end with the most elaborate, the plot button.

To add actions for a GUI component, we'll define one or more functions that are explicitly tied to that component. We refer to these functions as callback functions, and they are automatically executed when the user interacts with the associated GUI component. To add a callback function for the close button, I'll first select app.closeButton in the Component Browser, and then select the Callbacks tab below the list of components. The tool lists a kind of callback function that I can define for a button, ButtonPushedFcn, a function that's called when the user pushes the button. Let's click on the triangle to see our options, just one - <add ButtonPushedFcn callback>. Let's do that, and see what was added to the code file.

```
    % Callbacks that handle component events
    methods (Access = private)

        % Button pushed function: closeButton
        function closeButtonPushed(app, event)

        end
    end
```

Let's dissect this. Most of the code here is on a gray background, so we can't edit it. It's placed in a section labeled methods, which you can just think of as a different word for functions. Functions in this area are private, so they can only be called inside our App. The skeleton of a new function was added here with the name closeButtonPushed, and it has two inputs: the app structure and the event that occured. Actions like pressing a button are referred to as events. When the user presses the close button, this function will be called, and information about this particular event will be passed to this function through the second input. We won't be using this event information so we can ignore the second input. Inside the function is white space where we can add code. When we push the close button, we want to terminate the program, and we can do this with one line of code, delete(app). For any App that you create with a close or quit button, you'll create a button pushed callback function with this one statement, delete(app). Let's also add a comment.

```
% Button pushed function: closeButton
function closeButtonPushed(app, event)
    delete(app)       % terminate the program
end
```

The next callback function we'll define is one that will be called when we click on the produce button. We want the text on this button to change whenever we click on it, flipping between the words produce and consume. So let's select app.produceButton in the Component Browser, and we see there's also one type of callback function we can define, a ValueChangedFcn. The produceButton is a State Button that flips between two states, on and off, as we click on the button. The current state is the Value associated with the button. So clicking on this button changes its Value. So I'll click on the triangle and select <add ValueChangedFcn callback> and a skeleton of a new function appears in the code file.

```
% Value changed function: produceButton
function produceButtonValueChanged(app, event)
    value = app.produceButton.Value;

end
```

It has the name produceButtonValueChanged and it's a little different from the previous function we created, in that there's already a line of code there. It's an assignment statement that creates a simple variable named value, assigned to the current Value property of the produceButton. This Value is actually just 1 or 0, depending on whether the state of the button is on or off. If this button is in the on state, we want it to show the word produce, and if it's in the off state, we want to show the word consume. So we can just write an if statement, if value (this expression will be true if the Value property of the button is 1, which means on) then we want to change the text to produce. How do we refer to the text on the button? app.produceButton.Text, and we can just assign it directly to the string 'produce'. Otherwise we'll set it to the string 'consume'. That's it, we're now done with writing code to add an action when the user presses this button. Let's test the two bits of code I wrote so far. I'll save and Run the revised program, click on the produce button (see the changing text) and click on the close button to terminate the program.

```
% Value changed function: produceButton
function produceButtonValueChanged(app, event)
    % change text on the button when the user presses it
    value = app.produceButton.Value;
    if value
        app.produceButton.Text = 'produce';
    else
        app.produceButton.Text = 'consume';
    end
end
```

Finally let's implement the plot action - this is our most involved task. I'll click on app.plotButton, and see that my option for the callbacks is to <add ButtonPushedFcn callback>. A new function skeleton is inserted into my code with the name plotButtonPushed and a white area where I can add code. I'll walk through all the steps we need to do for the plot, one by one. Since we just talked about the produceButton, let's take care of that information first. Recall that we created a new property in the app that stores the data for the program <scroll up> - this data was created and returned by the setupEnergy function, and is placed in a structure that has a vector of years and two matrices with the production and consumption data. So as a start, we'll check what the user requested, production or consumption data, and set up the chosen data source for the plot. How do we access the state of the produceButton? We again refer to app.produceButton.Value. If true, I'll create a variable named dataSource assigned to app.data.produce (the name of the matrix with the production data), otherwise assign dataSource to app.data.consume. And I'll add a comment about what this does.

```
% Button pushed function: plotButton
function plotButtonPushed(app, event)
    % set dataSource to consumption or production data, depending
    % on the state of the produce button
    if app.produceButton.Value
        dataSource = app.data.produce;
    else
        dataSource = app.data.consume;
    end
```

Next, I need to figure out which energy source to use - there are 5 options listed in the Drop Down menu, coal/gas/oil/nuclear/renewable, and the data for these different sources is stored in the 5 rows of the dataSource matrix. So which item in the menu did the user select - which row of the data matrix should we use? To help us understand the strategy here, I'll go back to the MATLAB Command Window for a moment. Suppose I create a cell array of strings, nums = {'one' 'two' 'three' 'four'}; We've used strcmp in the past to compare two strings to see if they're equal. It turns out that we can compare a single string to an entire cell array of strings all at once. Let's do that, strcmp(nums, 'three'). This gives me a logical vector that's true in the place where the string 'three' appears in the cell array, the third location. Suppose I want to know the index where the true value appears, we can call the find function to get this index 3.

```
>> nums = {'one' 'two' 'three' 'four'};
>> strcmp(nums, 'three')
ans =
  1×4 logical array
   0   0   1   0
>> index = find(strcmp(nums, 'three'))
index =
     3
>>
```

We're going to use this same strategy to find the index of the item in the Drop Down menu that the user selected, in fact I'll start by copy/pasting this statement to our code file. Then let's replace the pieces. I'll change the variable name to sourceIndex. MATLAB stores the items of a Drop Down menu in a cell array, and we refer to this cell array as app.sourceDropDown.Items. The item that the user selected is the app.sourceDropDown.Value. And let's add a comment.

```
% index of the selected energy source in the drop down menu
sourceIndex = find(strcmp(app.sourceDropDown.Items, app.sourceDropDown.Value));
```

We're almost ready to plot, we just need to check whether the user wants the plot to be displayed with a wide line. We can do this with, if app.widelineCheckBox.Value (the checkbox is checked), I'll create a variable linewidth that I'll assign to a large width 4, otherwise assign to 1.

```
% use state of checkbox to determine linewidth
if app.widelineCheckBox.Value
    linewidth = 4;
else
    linewidth = 1;
end
```

Now, finally, we're ready to plot. One thing that's different about how we call plot in an App is that we're going to say explicitly, what is the name of the Axes where we want this plot to be displayed, because there may be multiple plotting areas on our GUI, so we need to be explicit about which one to use. Here we only have one - app.plotAxes. Next we enter the X coordinates, which are the years, app.data.years. The Y coordinates are our energy data, in dataSource(sourceIndex, :), and finally we specify the linewidth that we just set up. While I'm at it, I'll add X and Y labels to the plot, app.plotAxes.XLabel.String and app.plotAxes.YLabel.String. Our very last step is to take the text that the user entered for the plot title and display it as the title for the plot, so app.plotAxes.Title.String = app.titleEditField.Value. That completes the callback function that will be executed when the user clicks on the plot button. Let's save and test the new plot function.

```
% plot the requested data
plot(app.plotAxes, app.data.years, dataSource(sourceIndex,:), 'LineWidth', linewidth);
app.plotAxes.XLabel.String = 'years';
app.plotAxes.YLabel.String = 'quadrillion btu';
app.plotAxes.Title.String = app.titleEditField.Value;
```

We're now done with all the code we had to add to implement the actions we wanted for this App. The functions that we added are listed in the Code Browser on the left, so we can easily jump to a particular function to view or edit the function.

To recap, we first added a new property to store information that the App needs to access from the outside. In other applications, we might add properties here that need to be accessed by multiple callback functions in our file. Then, for any GUI component where we want to perform an action immediately that's special to our program, and not something that MATLAB just does automatically, we added a callback function. This callback function gets executed when the user

interacts with the GUI component, for example, when they push a certain button. We put code in the callback function that captures the additional actions that we want our program to perform.

There were a lot of little steps that we had to take in the process of building this App, and it may seem daunting at first, but as you practice going through this process on your own, it will become more intuitive with practice. If you look at the project gallery page at our course website, you can see that all the projects there had a graphical user interface - it will be a really powerful addition to your repertoire of MATLAB skills.