1

Hello. Welcome to this demo of “Building Diagram Displays for the Desktop" using IBM ILOG JViews Diagrammer.

2

Prerequisites for this demo are that you first watch two other demos. One is an overview of IBM ILOG JViews Diagrammer. The other is a tutorial on how to use the “tool chain:” the applications that come with JViews Diagrammer for producing data files – such as those we will use in this demo.

Those data files include a "nodes.jar" file which is a palette of symbols that will be used to populate the both the diagram and the dashboard displays.

The symbols were created in something called the Symbol Editor. It is described in the other presentation.

Then there is a file called AcmeNetwork.idpr – which also references two other files: a .css and a .xml file. These define the view and the data for the sample application we'll create in this presentation.

Those files were created using Designer for JViews Diagrammer. Designer is covered by the earlier demo and won't be covered again here.

The sample application also uses a dashboard definition. That is contained in the file, AcmeNetwork.idbd. It was created specifically for this presentation using the DashBoard Editor – another tool that comes with JViews Diagrammer.

The Dashboard Editor will not be covered in this presentation. However, it is easy to use for creating dashboards. This demo shows you how to embed those views in an application.

3

First you will learn about the architecture of JViews Diagrammer applications.

Then you will see a demonstration of the sample application that is the focus of this presentation.

Next, you will learn the main steps required to build such an application. Then you will see the code for creating a dashboard view, a diagram view, and all related controls.

The presentation finishes with an implementation of a popup menu on the diagram view.

Let’s get started.

The block diagram at the right of this chart shows the Diagrammer architecture. The most important block in this diagram is orange. It represents the “SDM Engine.” “SDM” stands for "Stylable Data" Mapper. The SDM Engine takes data objects from various sources – databases, XML files, or APIs – and generates graphics to represent the data objects. The rules about which graphics to generate are contained in a style sheet – a CSS file. The graphics are rendered in the views of the application.

4

Here is a diagram showing the key classes in the Diagrammer library.

An IlvDiagrammer and its extension, IlvDashboardDiagram are the central components for displaying diagrams and dashboards respectively. They are SWING components.

IlvDiagrammer has a method named, “getEngine” that returns the SDM engine associated with that diagrammer instance.

From the diagrammer, you can get the view – an instance of IlvSDMView that actually owns the graphics. The view is a SWING component. And the graphics it contains are not SWING components - an important point. They are all instances of a JViews class named, IlvGraphic.

The engine contains a data model of type IlvSDMModel. Programmers fill the model with non-graphical, data objects one way or another – we’ll see two ways in this presentation.

For each data object in the model, the engine adds a graphic to the view. As the data model changes over time, the view updates automatically.

For programmers – most of the work focuses on the data model.

5

Let’s see the sample application in action.

The application has six different panels. The panel in the top-center of the frame is a diagram – an instance of IlvDiagrammer. It displays nodes and links. Notice the variations of the nodes. Some of them are flashing; they have different background colors, different icons. This behavior was defined in the Symbol Editor. We simply loaded the palette to show it.

The diagram view in the top-center is surrounded by three toolbars. At the right is the palette bar. It enables users to create objects in the diagram -- create new nodes and new links. At the top of the diagram you can see an edit toolbar. It contains basic cut copy and paste operations; as well as graph layout operations. For our view, the graph layout is automatic. But sometimes you want to make it manual so that users can invoke it whenever they need it. And that's what these buttons are for.

The toolbar at the left is a view bar. It enables panning and zooming - a fundamental feature of JViews Diagrammer.

When I'm zoomed in, notice that the component at the upper-left of the frame continues to show the entire view. It also has a box in the middle that I can click and drag as a user to change the focus of the main view. This is an overview component.

When I select a node or link in the main view, notice that the property sheet in the upper-right of the frame is populated with data about that node. Different nodes - different data.

And when I select a node, the same node is selected in two other views. At the lower-right you'll see this tree view showing the same kinds of graphics as the main view. Whatever node is selected in each of the views is reflected in the others.

At the bottom we also see a view of the same data model. And this is a table view. It can be filtered or we can show everything. And also - selection is synchronized with the main view.

At the lower-left our sample application shows a dashboard. It was added to this application for demonstration purposes. So the gauge at the top is just showing random data of some kind. However, these other four gauges, the horizontal bars, are actually showing data about the data model behind the main view. So I can change properties in the main view with a popup And you'll notice the bars at the lower-left will change as well. They're showing statistical data about the main view's contents.

6

So now let's talk about the main steps needed to create an application like we've just seen.

Developers create symbols in the Symbol Editor that will be used to populate the diagram and the dashboard views. Those are saved as a palette [in] a jar file.

For diagram displays, the Designer is then used to assign symbols from the palette to data objects, based on the properties of those objects. The designer assigns data fields from the data model to parameters in the Symbols to make them dynamic.

This presentation shows you how to create a view from the definition produced by the Designer. Previous presentations show you how to create those views in the Designer and the Symbol Editor.

The process is similar for Dashboards but a Dashboard Editor is used instead of the Designer. The Dashboard Editor produces a dashboard definition file that statically defines the contents of a view. For this presentation, we load a dashboard with Java code and animate it.

7

Let's start with a look at the dashboard view code. The code you see here comes from the sample you saw earlier.

First we must create something called a “dashboard context” with its default constructor. Then we create the dashboard view itself – passing in the context.

For this application, we choose to turn off editing so the user will see, but will have no ability to edit the controls.

Now we tell the dashboard view to load its definition from an idbd file – produced from the Dashboard Editor.

After this, the dashboard is added to a SWING container, so that it will appear on the screen.

8

To animate the dashboard we must manipulate its data model. It has a data model in memory. For this sample, that code is contained in a class named, DashboardUpdater. It updates the data for each control either in response to a timer for the gauge at the top or in response to some other event as we'll see later.

To update a control, we first get the control using its ID. An ID is a string that was assigned to the control in the Dashboard Editor. With a reference to the control, we can then update its properties using the setObjectProperty method of the dashboard control. In this case, the property is named value. But it could be named anything. Based on our knowledge of the dashboard definition, we know that value is a Float. So we pass it a float value.

This code pattern is used for all control updates. You can see more example in the [source code for the] sample associated with this demo.

9

Now let's see how to create a diagram view.

First, we create an instance of IlvDiagrammer. It's a SWING component. Next, we call a custom method to load the data and style sheet either from a single project file or from the model API's. The method is named, loadDataAndStylesFromFile

At the bottom of this chart, you see the method that loads the IDPR file that was produced by Designer for JViews Diagrammer. So that's the first method that I've described: we're loading both data and stylesheets at once by loading this idpr file.

Once the diagrammer is added to a SWING contains and set visible, we have a live diagram.

10

Now, most applications do not want to use static data such as we had in the XML file shown in the previous slide that was loaded by the IDPR file shown in the previous slide. So this chart shows how the data model can be populated or updated through API calls to the SDM Engine instead.

In this case, we create an SDM Model using IlvDefaultSDMModel. This is one implementation of the interface, IlvSDMModel. There are many other implementations available, but this one is used in most applications.

We then call "createNode" on the model to create a new node. THere are a lot of ways to do this, but this is a convenience method on the model and it's an easy way to approach it. We use the model to set its ID, alarm state, status, name, and type. All of these data model properties are connected to parameters in the symbols by the Designer stylesheet. At runtime that will produce animation.

Next we set the model on the diagram with diagrammer.getEngine().setModel(). And then we set the stylesheet. This was one of the files that was created in Designer. We only need the stylesheet - we don't need the data - so we simply call setStylesheet().

This is in contrast to what was shown on the previous chart.

11

Now that we have a diagram on the screen, we can add some toolbars. These three lines of code each add one toolbar to the application. The first one adds a view bar - the toolbar at the left of our diagram - for pan and zoom capabilities.

The second one adds a palette bar -- the one at the right of the diagram - for adding new nodes and links -- the user is enabled to do so with this toolbar.

The last line adds an edit bar for cut and paste and graph layout operations. Also label editing.

By now you may have noticed that none of these toolbars specifies a diagram component. How does the toolbar know what diagram it should act on. With JViews Diagrammer, this happens automatically. The toolbars walk the SWING hierarchy in which they are shown looking for an instance of IlvDiagrammer. When one is found, it becomes the diagram for the toolbar. But this can also be specified explicitly if you find the need to do so.

12

Here we add more SWING components to provide other views of the data model from a diagram. The first is the table component. It includes a combo box for filtering the contents.

Next is an overview with the pan control we mentioned in the demo.

The third component is a property sheet. It shows the data model properties for the currently-selected node or link in a diagram.

The last item is a tree view that lists the nodes and links in the diagram using the same graphics as the diagram.

All of these find the diagram automatically as was described in the previous chart.

13

Now let's look at some of the code for the popup menu. It shows menu items that depend on the item being clicked.

When the symbols were defined for this diagram using the Designer, a popup manager was specified and given the name, "SitePopup" as shown in the screenshot in the upper-right of this chart. That makes it easy to associate popups with specific types of nodes or links as shown in the second line of code on this chart. In this case we use it for all nodes. In the third line of code, we use a more conventional approach to set the popup for links and blank areas of this diagram.

The next chart shows some of the code used to create the popup menus.

14

This is a partial listing. A complete listing of the code for the popups is available in the sample code associated with this demo.

Popup menus are instructive because they need to know which node or link has been clicked. This is a common requirement for all user interactions on a diagram. So the code on this chart is a common pattern in JViews programming.

Inside the popup listener, we first get the menu item that was clicked by the user.

Then we get something called a popup menu context -- a convenience class provided by JViews Diagrammer.

We use the context to get the graphic object that was clicked. This is necessary because graphics reside in a SWING object but are not themselves SWING objects.

With a reference to the graphic object, it is easy to get the data model object that it represents.

In this case we simply toggle the status property of that data object.

You will use code similar to the code on this chart for all user interactions, mouse listeners, and data property manipulations in your application.

15

The last thing we do in this example is to change the gauges of the dashboard [as] the properties change in the diagram. This gives us a chance to show you another important pattern in JViews Diagrammer programming: data model listeners.

There are two kinds of listeners for the data models in JViews Diagrammer: SDMModelListener's and SDMPropertyChangeListener's. Data model listeners provide notifications of model changes such as nodes being added or removed. The class [DashboardUpdater] in our sample is an SDMPropertyChangeListener. It is notified of changes to the properties of the nodes and links in the data model.

When the user changes the status or alarm on a node in the main diagram, this listener updates the bar gauges in the dashboard.

16

This video has shown you how to create an application with an interactive diagram view and an animated dashboard. It includes many other views of the data model behind the diagram.

In a video to follow this one, we will show you how to leverage code this sample to build a web application as well.

17

Sample source for this [demo] is available from the page that hosts this video demo.

Good luck with your JViews Diagrammer project!