

# HOW TO CREATE AN ACTIVEX COMPONENT

## Purpose and Scope

The purpose of this document is to show you how to create an ActiveX wrapper for the Java Components within the Java SDK. This will allow a Developer to create an application within a C Basic environment rather than Java.

You will see how to create and register wrapped beans on your development system, and provide guidance for deploying the wrapped beans to your end users.

It will not discuss ActiveX or Javabeans technologies, and it will not cover the usage of the wrapped Javabeans.

## Prerequisites

- It is advised that you have a good understanding of Java and Visual C++ to ensure the following process is carried out correctly.
- Sun Microsystems Java SDK V1.4.2 or greater
- Microsoft Visual C++ V6 or greater
- Java Client SDK V3.0.0 or greater

## References

- ActiveX Bridge Developer Guide  
(<http://java.sun.com/j2se/1.4.2/docs/guide/beans/activex/developerguide/index.html>)

## Introduction

This section will guide you through the process of wrapping a Java component with an ActiveX wrapper using Sun's ActiveX bridging tools, it will:

- Explain how to prepare your environment to perform the wrapping
- Take you through the wrapping step by step
- Explain what process must be followed when deploying the wrapped Java component

## Prepare your environment

In order for the Sun's ActiveX packager to work correctly, you must first run the **VCVARS32.BAT** batch file for your Visual C++ development environment in the command/MS-DOS prompt window that you will be using to perform the wrapping.

The batch file is usually found in the "bin" subdirectory of the Visual C++ installation e.g. C:\Program Files\Microsoft Visual Studio\VC98\Bin

## Packaging the Java Component

The initial process you should carry out is to create a unique class-id for your wrapped bean.

**Note** *When wrapping the same bean, always use the same class-id to avoid having conflicting copies of the same component installed simultaneously.*

To create a class-id that you can use, you need the GUID generator from Visual Studio. This is usually found in the "common\tools" subdirectory e.g. c:\Program Files\Microsoft Visual Studio\Common\Tools\GUIDGEN.exe

Launch the GUIDGEN tool and select registry format from the options,

Click the "New GUID" button,

Locate the directory where the netvu.jar file from the Java SDK is stored.

At this stage you need to create a simple batch file that will automatically generate and register the wrapped bean for you.

**Note** *You can then use this batch file each time you want to wrap the same Java component and it will automatically use the same class-id.*

Type;

*notepad wrap\_<name of Java component>.bat*

and press Enter. (Replace <name of Java component> with the name of the Java component you want to wrap, i.e. if you were wrapping the Viewer Java component, you would type notepad wrap\_Viewer.bat).

Once the editor appears you want to add a single line to the text file using the following syntax:

```
c:\<Java SDK dir>\bin\packager -clsid {<class-id from GUIDGEN>} -out "<Java JRE dir>\axbridge\bin" -reg <NetVu JAR file><Java component to wrap>
```

For example, if you were wrapping the Viewer Java component, you would enter a command similar to the one below:

```
c:\j2sdk1.4.2\bin\packager -clsid {915ABFB5-D7F3-420e-89CC-2F94EEEF8E9F} -out "c:\Program Files\Java\j2re1.4.2_03\axbridge\bin" -reg netvu.jar uk.org.netvu.awt.Viewer
```

Save this file using "**File -> Save**"; then quit the editor using "**File -> Exit**".

**Note** *It is advisable to add the call to VCVARS32.BAT before the call to the packager to ensure that your environment is always correctly configured before wrapping the Java component e.g. call "c:\program files\microsoft visual studio .net 2003\vc7\bin\vcvars32.bat"*

The next process is to run the batch file that has just been created, type: "wrap\_<name of Java component>" and the Java component will automatically be wrapped for you and registered on your system using the class-id you generated earlier.

If the wrapping process was successful, there will be a DLL file with the same name as the Java component you have wrapped in the <JRE home>\axbridge\bin directory, and the netvu.jar file will be in the <JRE home>\axbridge\lib directory.

### **Testing your ActiveX component**

The simplest way to test your new ActiveX component is in the ActiveX Control Test Container application that ships with Microsoft Visual Studio.

Launch the application, usually found at "C:\Program Files\microsoft visual studio\common\tools\TstCon32.exe" and select Edit->Insert New Control.

In the list that appears, look for the name of the Java control that you wrapped with "Bean Control" appended to it e.g. "Viewer Bean Control".

Highlight the ActiveX control that you want to test and click OK. This will place the control in the test container and activate the ActiveX bridge.

From here you can manipulate the public interface of the ActiveX control using the Control->Invoke Methods... option from the drop down menus.

If you wrapped the Viewer Java component for example, you could invoke the "setHost()" method passing the address of a video server, then invoke the "live" method to start receiving live images from that video server.

### **Deploying your ActiveX component**

When you deploy the wrapped Java component(s), you need to ensure the user is using the Sun Microsystems JRE v1.4.2 or greater

During installation you should copy the DLL file to their <JRE home>\axbridge\bin directory (your installation procedure should create this directory if it does not already exist), and register it on their system. Some installation packages will handle this for you, otherwise you will need to call **REGSVR32.EXE** and pass the DLL file as a parameter.

You should follow the same instructions if you want to install the wrapped beans on to a system to be used for development, but do not want to follow the wrapping process on that system.

### **Summary**

You should now be able to create a batch file which can wrap any Java component from the Java SDK as an ActiveX component, test that component in the ActiveX test container, and deploy it to other developers and your end users.

## HOW TO CONVERT A WRAPPED JAVA COMPONENT TO AN OCX FILE

### Purpose and Scope

The purpose of this document is to explain why and when you might need to perform additional work on a Java component that has been wrapped using Sun Microsystem's ActiveX bridge as detailed in the previous section.

It will explain how to convert the ActiveX component to an OCX file using Microsoft Visual Basic and give some guidelines about deploying Java components that are 'do-able-wrapped' in this way.

It will not cover the use of the OCX file once it has been created.

### Prerequisites

It is recommended the Developer have a good understanding of Java and Visual Basic to ensure the following process is carried out correctly.

Microsoft Visual Basic V6.0 or greater

### References

Dedicated Micros Java SDK: How to create an ActiveX Component

### Introduction

Unfortunately there is a flaw in Sun Microsystems ActiveX bridge and the beans that it wraps do not work correctly in all ActiveX containers. There are known issues with Microsoft Visual C++ V6.0 and V7.0 (.NET), as well as Visual Basic V7.0 (.NET).

The solution for the problem which is detailed within this section, as a temporary measure only until the issue is resolved.

### Starting a new ActiveX Control project

Launch Visual Basic and create a new ActiveX Control project.

If your wrapped Java component ActiveX control is not already on the component palette, add it now by right clicking on the palette and selecting "**Components...**" from the pop-up menu.

Locate the Java component control that you want to convert to an OCX file in the dialog box

Put a check next to it and click on the "**OK**" button

### Adding the wrapped Java component

Drop an instance of the Java component control on to the form and rename using a meaningful name.

**Note** *Ensure you allocate the project and the user control with meaningful names as these will be used to create the OCX file later.*

Set the "**Top**" and "**Left**" properties of the Java component control to 0 so that it is placed in the top left corner of the user control.

Create an event handler for when the user control is resized.

Add code such that the Java component control changes its size to match the user control whenever the user control is resized. i.e:

```
Private Sub UserControl_Resize()  
    ViewerBean.Width = UserControl.Width  
    ViewerBean.Height = UserControl.Height  
End Sub
```

Mapping the properties

For each of the properties of the Java component control, you need to add a property delegation method to your user control i.e.

```
Public Property Get Host() As String  
    Host = ViewerBean.Host  
End Property
```

```
Public Property Let Host(ByVal newHost As String)
    ViewerBean.Host = newHost
End Property
```

### Mapping the methods

For each of the methods of the Java component control, you need to add a delegation method to your user control i.e.

```
Public Sub live()
    Call ViewerBean.live()
End Sub
```

### Creating the OCX file

To create the OCX file, save the project, then select "**File -> Make <project name>.ocx...**" from the drop down menu. Choose where you want to save the OCX file from the dialog box and click on the "**OK**" button.

### Testing the OCX file

To test the OCX file, launch the ActiveX Control Test Container.

Select "**File -> Register Controls...**" from the drop down menu and click the "**Register**" button on the dialog box that appears.

Using the file chooser dialog find the OCX file you created earlier, select it and click on the OK button. Click on the "**Close**" button of the Register Controls dialog box.

Select "**Edit -> Insert New Control....**" from the drop down menus.

Find the registered OCX control (don't confuse it with the original Java component you registered), highlight it and click on the OK button. This will insert the control into the container for you.

You can now use the "**Control -> Invoke Methods**" option from the drop down menus to test the functions of your newly created OCX control.

### Deploying this type of solution

When distributing this it is essential that you also distribute the original DLL file as described in the previous section, "*How to create an ActiveX Component*".

In addition you will also need to distribute the OCX file and this will need to be registered on the system.

As with the DLL file, your installation system may be able to do this for you automatically, otherwise you will need to call **REGSVR32.EXE** and pass the OCX file as a parameter.

**Note** *The OCX file can be installed anywhere; it does not need to be in the <JRE\_HOME>\axbridge\bin directory, but does require that the DLL file is stored there.*

### Summary

This document has shown you how to create an OCX file from a Java component that has been wrapped using Sun Microsystems ActiveX bridge. It has explained why this may be necessary and also provided advice about deploying such a solution. You should now be able to leverage Java components from the Dedicated Micros Java SDK into your native applications developed using Visual Basic V7.0 and greater or Visual C++ V6.0 and greater.

# DV-IP CONFIGURATION

## REMOTE CONFIGURATION AND SYSTEM VARIABLES

The configuration settings of the Image server are normally changed using a standard web-browser and the suite of web pages that are stored on the unit when shipped from the factory. These web pages can be used to change many aspects of the unit's operation

- Record rates
- Camera setup
- Text – In Images
- Alarm setup
- Database Configuration
- Network Settings
- Serial Port Setup
- Telemetry
- Audio Setup
- Automatic event export
- System log configuration

To access the configuration pages, enter the URL "*http://<your\_server\_name>/webpages/index.shtml*" and click the '**Configuration Options**' button

### Custom configuration using application code

The chapter "CGI's - Common Gateway Interface" introduced the concept of 'System Variables'. These are dynamic variables that can be modified; however their contents are stored to non-volatile ram or to the unit's internal hard disk. System variables hold the settings for every configurable aspect of the Image server.

Some variables can be considered read only while others are read/write and can be set by the user / developer.

### Reading a system variable

System variables are accessed using the **sys\_var.cgi** command. This can be called directly from a web browser, embedded into a web page (using JavaScript) or more usefully in the SDK environment using a Java class. The three techniques are illustrated below.

example:

The '**sys\_cams**' variable holds the number of cameras supported on the unit and will be the focus of our example.

1. Entering the following into your web browser should return a full web page response with the number of cameras displayed

*http://<your\_server\_name>/sys\_var.cgi?variable=sys\_cams&type=http*

2. It is more useful to the web page designer to have access to the value (16) so it can be used in different ways within the page. This is achieved by using a Server Side Include (SSI). This a line of script that can be embedded into a web page and looks like this

```
<!--#echo var="sys_cams"-->
```

When a web page containing this construct is requested, the web server replaces the construct with the value stored in the System Variable '**sys\_cams**', in this case 16.

The code snippet below shows the SSI in a complete HTML web page

```
<html>
<head>
<title>SYS_CAMS</title>
<script LANGUAGE="JavaScript"> // Start JavaScript section
var Cameras = '<!--#echo var="sys_cams"-->'; // Set variable Cameras to 16
```

```

document.write(Cameras);           // Display in document
</script>                          // End JavaScript section
<body>
</body>
</html>

```

The variable Cameras above now contains the value 16. In this example it is simply 'written' into the body of the HTML page but can be used in calculations or procedural code. Unfortunately the scope of this document does not cover JavaScript programming but time spent learning the basic syntax (similar to C) is time well spent. JavaScript can be used to great effect to create dynamic web pages around which the Image server interface is built. JavaScript should not be confused with Java.

3. Accessing a system variable using the Java toolkit can be achieved using the SDK 'HTTPRequestHandler' class. The example below requests the contents of the 'sys\_cams' system variable and sets the caption of a label component to this value.

```

void GetNumberOfCameras {
    HTTPRequestHandler HQL = new HTTPRequestHandler();
    HttpQuery query = new HttpQuery("http://172.16.85.10/variable.cgi?variable=sys_cams",
HQL);
    query.run();
    jLabel1.setText("sys_cams = " + HQL.ResultString[0]);
}

```

### Writing a system variable

Writing a system variable can also be done from a web page and from the Java SDK class **HTTPRequestHandler**, both methods are outlined below.

1. Since sys\_cams is a read only variable the c\_title\_1 variable which holds the camera title for camera 1 will be used here.

The setting of a variable relies on the 'Form Submit' technology which web servers support. Most interactive web pages involve the user completing a form and then hitting a button at the bottom to 'submit' the data to the server. The server will then process the data and based on this 'submitted' data will return a new web page with relevant information (eg: motor insurance premium).

The Image server supports this technology and custom web pages can be created to utilise it. The built in configuration pages use this technology to get and set system variable values.

The default configuration pages can be found in the *\frmpages* directory on the DV\_IP. A good way to learn about this is to actually download the files and study them.

exercise:

Each web page in the *\frmpages* directory contains HTML objects eg: text box, drop down list etc. The example form contains a text entry box that holds the camera title for camera 1.

When the form is requested, the current value needs to be displayed in the text box. To do a Server Side Include (SSI) request can be used to set up the form. Once this is in place, every time the form is displayed the current value will be displayed. When values are changed, the server is configured to re-display the same page with the updated values.

The camera title is retrieved using the SSI on line 7 of the HTML head section, when the body section is processed the function BuildHTML is called. This function builds up a string that is essentially the HTML tags for the body section. Using script to do this allows the IP address of the unit to be generated automatically, essential as the page can be hosted on any Image server without modification. Line 13 defines the button that is shown next to the text box. The button's TYPE parameter is set to 'submit'.

When a 'submit' button is pressed the web browser builds a new URL based on the HTML objects within the <FORM> / </FORM> tags. The URL is built from the ACTION parameter of the form and the NAME's of all the objects contained in the form. Each HTML object has a NAME parameter and obviously a value.

The URL built in this example, assuming the user entered XXX as the camera title, would be:

```
http://<your_server_name>.10/cam_example.frm?c_title_1=XXX
```

The system variable c\_title\_1 will be set to XXX.

The ACTION parameter contains the form that will be returned after submission with the extension replaced by '.frm' The actual form returned from the server would be cam\_example.shtml from the frmpages directory.

When the Image server is reset, the current values for the system variables are read from '.sfm' files held in the \FRM\_DATA directory on the Image server. Each web page in \frmpages will have a corresponding '.sfm' file. If custom web pages are uploaded the '.sfm' files should be removed. New files will be automatically generated that match the structure of the new web pages.

\frmpages\cam\_example.shtml

```
<html>
<head>
<title>Camera Setup Example</title>
<META HTTP-EQUIV="Pragma" CONTENT="no-cache">
<script LANGUAGE="JavaScript">
<!--
var cam_title1 = "<!--#echo var="c_title_1"-->";
function BuildHTML()
{
var HTMLData = "";
HTMLData += '<FORM NAME="SUBMITFORM" ACTION="" + location.protocol
//' + location.host + '/cam_example.frm" METHOD="GET">';
HTMLData += '<INPUT TYPE="text" NAME="c_title_1" SIZE="12" MAXLENGTH="12"
VALUE="" + cam_title1 + "">';
HTMLData += '<input TYPE="submit" VALUE="Submit">';
HTMLData += '</FORM>';
document.write(HTMLData);
}
//-->
</script>
</head>
<body bgcolor="#FFFFFF">
<script>
BuildHTML();
</script>
</body>
</html>
```

The URL from above can be replicated and used with the Java class HTTPRequestHandler to set a system variable. The example below uses a dummy form but if a valid form is used the HTML will be returned in the ResultString.

```
void jButton2_actionPerformed(ActionEvent e)
{
HTTPRequestHandler HQL = new HTTPRequestHandler();
HttpQuery query = new HttpQuery(
"http://172.16.85.124/dummy.frm?c_title_1=Camera1", HQL);
query.run();
jTextArea1.append(HQL.ResultString[0]);
}
```

## CREATING A CUSTOM CONTROL

The SDK provides many useful components for the GUI developer to use. Most of the time these components provide all the required functionality and an acceptable look and feel. Occasionally there maybe a case where custom appearance or in complex GUI's where screen space is at a premium it may be necessary to create a custom component.

**Note** *Developers using the Foundation version of JBuilder may experience problems during the advanced tutorials, due to the limited nature of this free software package. It is recommended that all advanced development work be done in the Developer Edition of JBuilder.*

The SDK makes this a simple process by separating the GUI code from the functional code that sits behind the component.

The SDK is based around the following architecture.

This tutorial will demonstate how to create a custom GUI panel that fits this architecture and will behave like CameraList component but will display camera buttons instead of a list.

The first step is to create a new class which we will call CameraButtonPanel

```
public class CameraButtonPanel
{
}
```

This class will act as a container for the buttons so we will inherit the functionality of a Java 'swing' panel by making our class extend javax.swing.JPanel.

```
import javax.swing.*;
public class CameraButtonPanel extends JPanel
{
}
```

The SDK provides a Java interface CameraControl to ensure the developer implements the necessary methods to create a custom camera selection component.

```
public interface CameraControl extends NetVuControl
{
    public abstract void changeCamera(int i);
    public abstract void onCameraChange(int i);
    public abstract void onHostChange(Camera acamera[]);
}
public interface NetVuControl
{
    public abstract void setControlSet(ControlSet controlset);
    public abstract ControlSet getControlSet();
}
```

By forcing the new component to 'implement' this interface the developer must provide the following methods.

```
public void setControlSet(ControlSet controlset);
public ControlSet getControlSet();
public void changeCamera(int i);
public void onCameraChange(int i);
public void onHostChange(Camera acamera[]);
```

The code now looks like this (imports omitted)

```
public class CameraButtonPanel extends JPanel implements CameraControl
{
    public void setControlSet(ControlSet controlset)
    {
    }
    public ControlSet getControlSet()
    {
    }
    public void changeCamera(int i)
    {
    }
    public void onCameraChange(int i)
    {
    }
    public void onHostChange(Camera acamera[])
    {
    }
}
```

The next step is to fill in the code for these methods. To make things easy some default behaviour is provided by the SDK in the CameraControlSupport class. It is a simple task to create an instance of this class and map the above methods onto the default behaviour.

```
public class CameraButtonPanel extends JPanel implements CameraControl
{
    // Create instance of support class
    CameraControlSupport ccs = new CameraControlSupport(this);

    // Map methods onto support class
    public void setControlSet(ControlSet controlset)
    {
        ccs.setControlSet(cs);
    }

    public ControlSet getControlSet()
    {
        return ccs.getControlSet();
    }

    public void changeCamera(int i)
    {
        ccs.changeCamera(camera);
    }

    public void onCameraChange(int i)
    {
    }

    public void onHostChange(Camera cams[])
    {
    }
}
```

The setControlSet must be called to connect an instance of CameraButtonPanel into the architecture outlined above. Once connected the onCameraChange and onHostChange 'callback' methods will be called to inform the CameraButtonPanel of changes made to the Viewer component.

Once connected to the control set all the GUI is required to do is call changeCamera passing the new camera number when the user clicks on a GUI camera button.

The onHostChange() method will be called when the Viewer is connected to a new unit and the camera details will be passed in as parameters. The array passed in will have an entry for each camera connected. Obtaining the size of this array will therefore indicate the number of camera (this can vary from unit to unit). It is therefore possible to use cams.length to determine the number of buttons to display for the connected unit. The camera titles can also be retrieved from this array and the button text could be set to reflect the camera titles.

onCameraChange will be called when the viewer is requested to display a different camera. This would normally be used to update the CameraButtonPanel GUI to reflect the changes.

For example if the viewer control was set to display camera 5 the onCameraChange() method would be called passing in 5 via the parameter list. The code added to the onCameraChange method would then show camera button 5 depressed and raise the previously selected button.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.border.*;
import uk.org.netvu.control.*;

public class CameraButtonPanel extends JPanel implements CameraControl
{
    final static int MAX_NUM_OF_CAMERAS = 16;

    CameraControlSupport ccs = new CameraControlSupport(this);

    JToggleButton[] camButtons = new JToggleButton[this.MAX_NUM_OF_CAMERAS];

    ButtonGroup bg = new ButtonGroup();
    GridLayout gridLayout1 = new GridLayout();

    public CameraButtonPanel()
    {
        try
        {
            jblnit();
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
    private void jblnit() throws Exception
    {
        gridLayout1.setRows(2);
        this.setLayout(gridLayout1);
        for(int i=0; i<MainFrame.MAX_NUM_OF_CAMERAS; i++)
        {
            camButtons[i] = new JToggleButton("" + (i+1));
            camButtons[i].setBorder(BorderFactory.createBevelBorder(BevelBorder.RAISED,Color.white,Color.white,new Color(103, 101, 98),new Color(148, 145, 140)));
            camButtons[i].addMouseListener(new CameraButtonMouseListener());
            camButtons[i].setVisible(false);
        }
    }
}
```

```

this.add(camButtons[i]);
bg.add(camButtons[i]);
}
}
private class CameraButtonMouseHandler extends MouseAdapter
{
public void mousePressed(MouseEvent e)
{
    for(int i=0; i<MainFrame.MAX_NUM_OF_CAMERAS; i++)
    {
        if (e.getSource() == camButtons[i])
        {
            CameraButtonPanel.this.changeCamera(i+1);
            break;
        }
    }
}
}
public void setCameraButtons(Camera[] cams)
{
for(int i=0; i<MainFrame.MAX_NUM_OF_CAMERAS; i++)
{
    camButtons[i].setVisible(false);
}
if(cams != null)
{
    for (int i = 0; i < cams.length; i++)
    {
        camButtons[i].setVisible(true);
        camButtons[i].setToolTipText(cams[i].getTitle());
    }
}
}
public void setControlSet(ControlSet cs)
{
    ccs.setControlSet(cs);
}
public ControlSet getControlSet()
{
    return ccs.getControlSet();
}
public void changeCamera(int camera)
{
    ccs.changeCamera(camera);
}
public void onCameraChange(int newCamera)
{
    // update GUI
camButtons[newCamera-1].setSelected(true);
}
public void onHostChange(Camera[] cams)
{
    if(cams != null)
    {
        setCameraButtons(cams);
    }
}
}
}

```

# LAYOUT PLUGIN TUTORIAL

## INTRODUCTION

The LayoutPlugin technology is a feature in the SDK that allows an application designer to control the organisation of the video panels within their GUI. Any Image server SDK application that you may have already developed, like the viewer in the Simple Tutorial, uses a LayoutPlugin to control its video display.

### Prerequisites

To have completed the Simple Tutorial and be familiar with the basic concepts of the SDK. Have access to the SDK version 4.2.0 or later.

**Note** *Developers using the Foundation version of JBuilder may experience problems during the advanced tutorials, due to the limited nature of this free software package. It is recommended that all advanced development work be done in the Developer Edition of JBuilder.*

### Theory

A LayoutPlugin provides a set of layouts (arrangements of the video panels) and any LayoutPlugin can be easily swapped for another, thus giving access to a different set of layouts (i.e. the layouts are pluggable).

To understand a little more about how this functionality is achieved, we should first get a handle on the classes and interfaces that are involved.

- JViewer/Viewer which derive from ViewerControl.  
A JViewer is the primary class that the SDK app is based around and is where your Video would have been displayed should you have built the application described in the Simple Tutorial. As part of this tutorial you will see that, by using the power of a Layout Plugin, we are able to display video in places other than this Container.
- ViewerPanel  
This interface is implemented by the classes in the SDK that display a video stream in the GUI. ViewerPanels are the objects that your LayoutPlugin will position in your GUI to achieve the desired layout.
- LayoutPlugin  
The Layout Plugin that you will write will implement this interface.
- DefaultLayoutPlugin  
This class, provided by the SDK, will be used in the absence of any user specified LayoutPlugin. Not surprisingly, it implements the LayoutPlugin interface.
- JLayoutComboBox  
This is a standard component provided by the SDK for selecting Layouts. A JLayoutComboBox will automatically be able to work with any simple LayoutPlugin you write.

### Step 1 - Adding the default layout functionality to the Simple tutorial.

- Work through or load your completed version of the JBuilder project from the Simple Demo.
- Select the GridLayout sub-component of the JPanel and change the rows property on the inspector to 5 - to allow room for the new control component we will be adding.
- Select the JViewer in the structure tree and change its **selectActivePosition** property in the inspector and set it to **"True"**. Turning this property on will allow you to use the mouse to select the ViewerPanels when the application is running.
- Locate the JLayoutComboBox component in the component palette and click it once.



- Click once just under the JCameraButtonGrid component in the frame displayed in the content pane.
- Once you have added it change the ControlSet property in the inspector to the ControlSet

object we created earlier, which should be called "**controlSet1**".

- Compile and run the application.

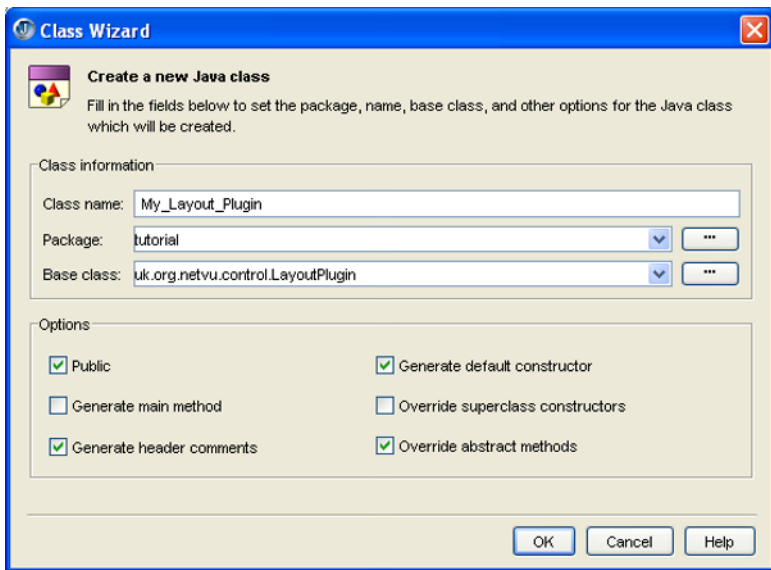
**Note** *There is no need to specify a `LayoutPlugin` to build this demo. The default `LayoutPlugin` will be used automatically.*

You will see your new Viewer appear with the **JLayoutComboBox** at the bottom of the screen. If you select Four Way from this list, the screen will divide into four quadrants and display images from the first four cameras from your server. Selecting a quadrant will highlight it with a red bounding box, then selecting a camera from the **JCameraButtonGrid** will display it in your selected segment.

There are many different layouts available with the default `LayoutPlugin`. Try using some to familiarise yourself with what can be achieved.

## Step 2 - Writing your own `LayoutPlugin`.

- Select "**File->New Class...**".
- Set the Class name to **MyLayoutPlugin**.
- Leave the package as **sdk\_tutorial**.
- Set the base class to be **uk.org.netvu.control.LayoutPlugin**.
- Selecting the options as they are in the image will help with producing the source file you are creating.



- Press OK and a source file for the class **MyLayoutPlugin** will appear. Edit the code so it matches the following:

```
package sdk_tutorial;

import java.beans.PropertyChangeListener;
import uk.org.netvu.control.*;
import uk.org.netvu.awt.*;
import java.awt.*;
import java.beans.*;

public class MyLayoutPlugin implements LayoutPlugin
{
    /** String to represent the Single layout. */
    private static final String SINGLE = "Single";
```

```

/** Icon to represent the Single layout. NB, we borrow the icon from the
    DefaultLayoutPlugin. */
private static final Image SINGLE_ICON =
Toolkit.getDefaultToolkit().getImage(DefaultLayoutPlugin.class.getResource("single.gif"));

/** String to represent the Four way layout. */
private static final String FOUR_WAY = "Four Way";

/** Icon to represent the Four way layout. NB, we borrow the icon from the
    DefaultLayoutPlugin. */
private static final Image FOUR_WAY_ICON =
Toolkit.getDefaultToolkit().getImage(DefaultLayoutPlugin.class.getResource("four_way.gif"));

/** A private field which used to rememeber which layout is current. */
private String layout = SINGLE;

/** The Viewer or JViewer that this layoutPlugin controls the Layout for. */
private ViewerControl viewerControl = null;

/** The LightweightViewer that the viewerControl uses to delegate jobs to. */
private LightweightViewer lightweightViewer = null;

/** The ViewerPanels that this LayoutPlugin will arrange. */
private ViewerPanel[] viewerPanels = null;

/** The largest number of ViewerPanels being displayed in the current Layout. */
private int maxPosition = 1;

private PropertyChangeSupport changeSupport = new PropertyChangeSupport(this);

/* Constructor. */
public MyLayoutPlugin(ViewerControl viewerControl)
{
    this.viewerControl = viewerControl;
    lightweightViewer = findLightweightViewer(viewerControl);
}

/** This private method will attempt to find the LightweightViewer that a
    ViewerControl should have.
    @param viewerControl This is the JViewer/Viewer for which to find the LightweightViewer.
    @returns A reference to the LightweightViewer should there be one, otherwise null. */
private LightweightViewer findLightweightViewer(ViewerControl viewerControl)
{
    // If the ViewerControl is a LightweightViewer, then we're done.
    if (viewerControl instanceof LightweightViewer)
    {
        return ((LightweightViewer)viewerControl);
    }

    LightweightViewer lightweightViewer = null;

    // Otherwise, get the components of the ViewerControl (which will be a
    // JViewer or Viewer)...
    Component[] components = ((Container)viewerControl).getComponents();

    // ...and search through them for the real LightweightViewer.
    for (int c = 0; c < components.length; c++)
    {
        if (components[c] instanceof LightweightViewer)

```

```

    lightweightViewer = (LightweightViewer)components[c];
}

return lightweightViewer;
}

public void addPropertyChangeListener(PropertyChangeListener pcl)
{
    changeSupport.addPropertyChangeListener(pcl);
}

public void removePropertyChangeListener(PropertyChangeListener pcl)
{
    changeSupport.removePropertyChangeListener(pcl);
}

public int getMaxPosition()
{
    return maxPosition;
}

public ViewerPanel getPosition(int position)
{
    return viewerPanels[position];
}

/** This is the method that is called to ask for a particular layout to be set up. */
public void setLayout(String layout)
{
    String oldLayout = this.layout;
    this.layout = layout;

    if (layout.equals(SINGLE))
        setGrid(1, 1);
    else if (layout.equals(FOUR_WAY))
        setGrid(2, 2);
    else
    {
        this.layout = SINGLE;
        setGrid(1, 1);
    }
    changeSupport.firePropertyChange(LAYOUT_PROPERTY, oldLayout, layout);
    lightweightViewer.validate();
}

/** Handy method to centralize our setting of the layout. */
private void setGrid(int rows, int columns)
{
    // Calculate the number of ViewerPanels required for this layout.
    maxPosition = rows * columns;
    // Remove the ViewerPanels that made up the previous layout.
    lightweightViewer.removeAll();

    // Set the layout to the appropriate grid for the new layout.
    lightweightViewer.setLayout(new GridLayout(rows, columns));

    // Create a new set of ViewerPanels for the new layout.
    viewerPanels = viewerControl.allocateViewerPanels(maxPosition);
}

```

```

// Add the ViewerPanels as child Components to the LightweightViewer.
for (int i = 0; i < maxPosition; i++)
{
    lightweightViewer.add((Component)viewerPanels[i]);
}
}

public String getLayout()
{
    return layout;
}

public String[] getLayouts()
{
    String[] layouts = { SINGLE, FOUR_WAY };
    return layouts;
}

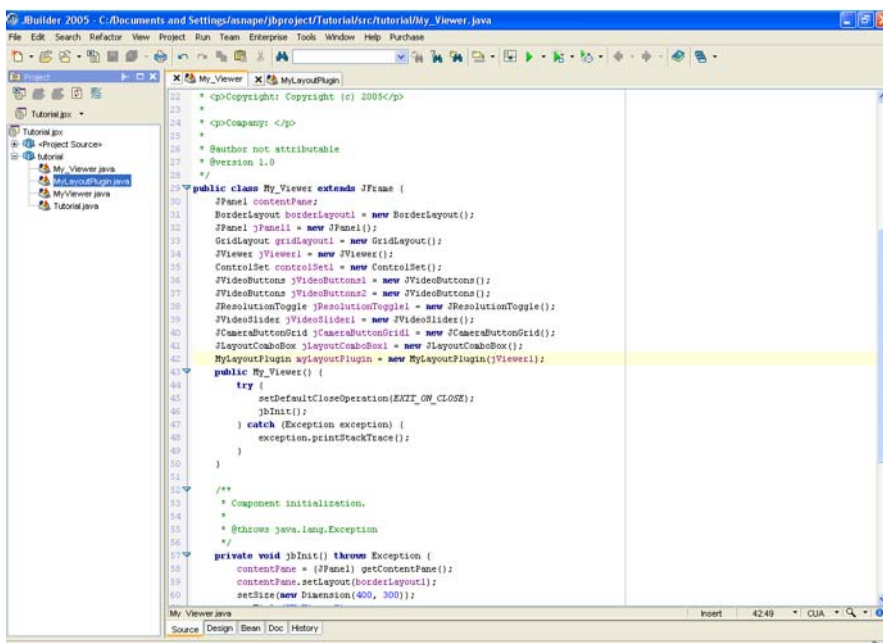
public Image[] getLayoutImages()
{
    Image[] layouts = { SINGLE_ICON, FOUR_WAY_ICON };
    return layouts;
}

public void setDefaultLayout(String string)
{
    setLayout(string);
}

public String getDefaultLayout()
{
    return getLayout();
}
}

```

- Change to the source file **MyViewer** using the Tab at the top of the editing pane. Add an entry to the bottom of the field declarations of the **MyViewer** class for an instance of **MyLayoutPlugin** called **myLayoutPlugin** and construct an instance, passing the **JViewer** that we are going to display in as the parameter to the constructor.



- Add the following highlighted code to the `jblnit()` method of the `MyViewer` class;

```
private void jblnit() throws Exception {
    contentPane = (JPanel) this.getContentPane();
    contentPane.setLayout(borderLayout1);
    this.setSize(new Dimension(400, 300));
    this.setTitle("My Viewer");
    jPanel1.setLayout(gridLayout1);
    gridLayout1.setRows(5);
    jViewer1.setMaximumSize(new Dimension(2147483647, 2147483647));
    jViewer1.setHost("172.16.85.8");

    jViewer1.setDefaultLayoutPlugin(myLayoutPlugin);

    controlSet1.setViewerControl(jViewer1);
    jResolutionToggle1.setControlSet(controlSet1);
    jVideoButtons1.setControlSet(controlSet1);
    jVideoSlider1.setControlSet(controlSet1);
    jCameraButtonGrid1.setControlSet(controlSet1);
    jLayoutComboBox1.setControlSet(controlSet1);
    contentPane.add(jPanel1, BorderLayout.SOUTH);
    jPanel1.add(jVideoButtons1, null);
    jPanel1.add(jResolutionToggle1, null);
    jPanel1.add(jVideoSlider1, null);
    jPanel1.add(jCameraButtonGrid1, null);
    jPanel1.add(jLayoutComboBox1, null);
    contentPane.add(jViewer1, BorderLayout.CENTER);
}

```

- Click the Design tab at the bottom of the window. Select **jViewer1** in the Structure pane. Find **defaultLayoutPlugin** in the Properties inspector. Click on its value and a drop down menu should appear. Select **myLayoutPlugin** from the menu.
- Rebuild and execute your viewer and click "**Live**" to show some video.

You should find that the application is using your new **LayoutPlugin**.

The `LayoutComboBox` will have only 2 items to choose between, Single and Four Way. Try toggling between them.

# A MULTI-SITE VIEWER TUTORIAL

## PURPOSE AND SCOPE

The purpose of this tutorial is guide you through the creation of a Multi-Site client application to view and control video from a number of Image Servers.

The client application will connection to multiple video servers and allow you to :

- Select between the cameras on the different servers
- Control playback of the video stream
- Select a number of servers to connect to simultaneously

## Pre-requisites

For this tutorial it would be desirable, but certainly not essential, to have a basic understanding of the Java programming language, this will help you understand the processes you will be asked to follow and why.

Before you start the tutorial you will need the following:

- Java SDK JAR file ("netvu\_awt.jar" or "netvu\_swing.jar" file) - provided on the CD within the Java SDK
- A suitably configured JBuilder installation
- At least two networked image servers with at least one live camera (you will be connecting to this unit through the LAN connection) on each server
- You have completed the Simple SDK Tutorial

## Creating a Viewing Application

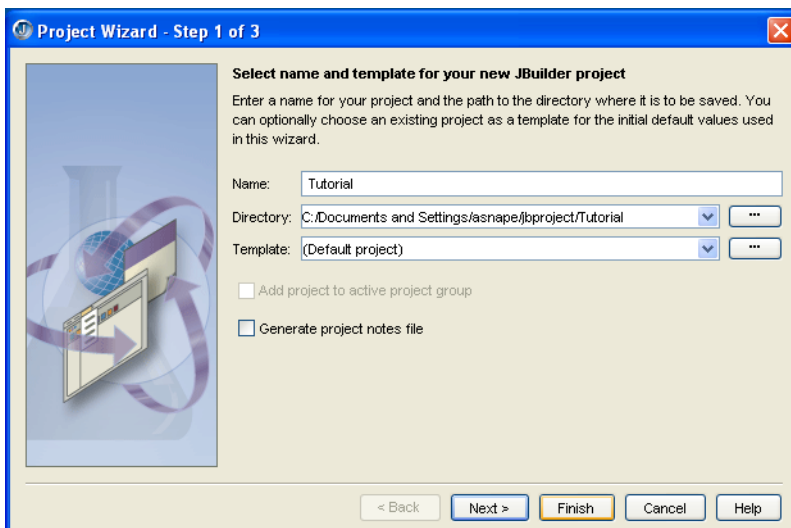
The next few sections will detail the steps you follow to create a PC Viewing application. If you follow the Steps carefully at the end of the tutorial you will have an application that has the functionality described above.

### Step 1 Starting a new JBuilder project

The first step is to create a JBuilder project.

Select "**File -> New Project**" from the drop down menu.

Name the new project "**Tutorial**" and click "**Finish**" to close the dialog box.



## Step 2 Creating the application window

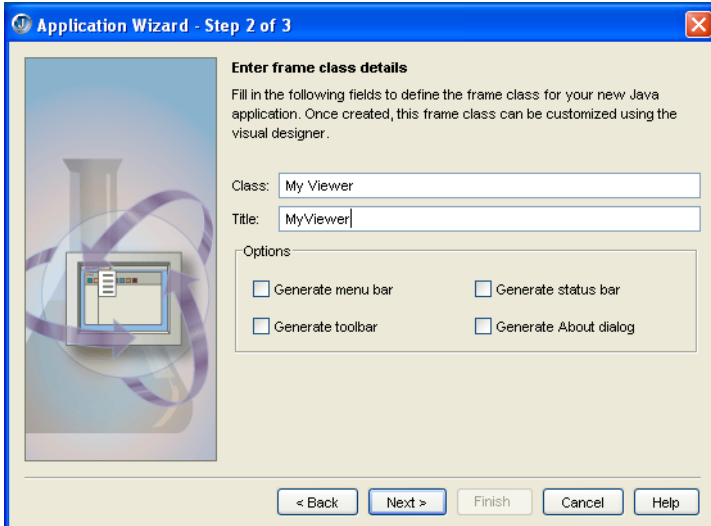
Now you have basis to build your application. The first step in creating the viewer is to produce an application window here you will build your network viewer. JBuilder has a handy wizard that can help us do this.

Select "**File -> New**" from the drop down menu

Select "**Application**" from under the "**General**" tab in the Object Gallery that appears and click "**OK**".

When the Application Wizard appears, change the Class Name to "**Tutorial**"

Click the "**Next**" button

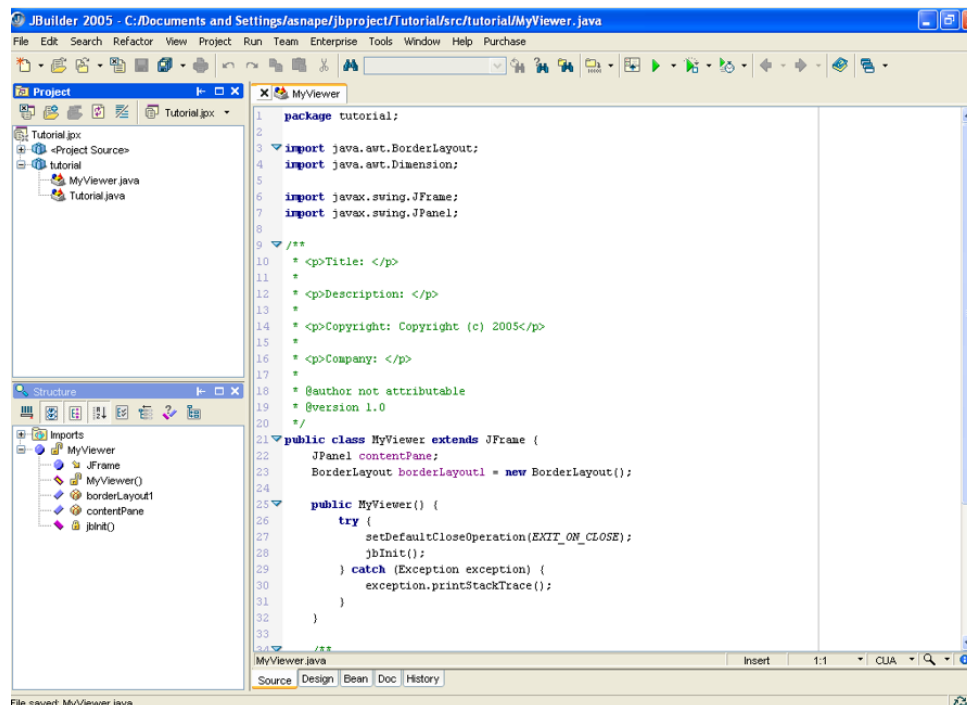


Set the Class to "**MyViewer**"

Set the Title to "**My Viewer**".

Click on the "**Finish**" button.

This process will add two new files to your project, which you will be able to see in the project pane.



Tutorial.java is the program code that encapsulates your application.

MyViewer.java is the program code that creates the application window, to which you will add the Java SDK components when building your video client application.

**Note** You may notice that some code has appeared in the content pane don't panic, this is not relevant to you and you will not work with this at all!

### Step 3 Preparing the application frame for the SDK controls

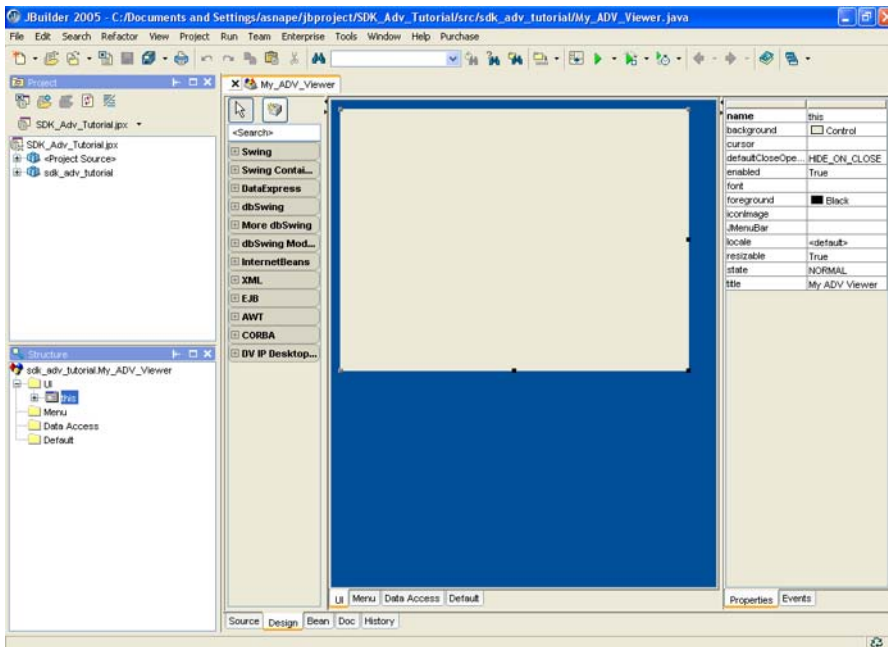
Before you can add any of the SDK controls you need to add a JPanel to the bottom of the frame that the wizard has already created for us.

In this panel later we will be adding the JVideoControls and JLayoutCombo components from the SDK.

A frame is just an application window.

Double click on **MyViewer.java** in the project panel to make the program code appear in the content pane (if it isn't there already).

Click on the **"Design"** tab under the content pane to change to a visual representation of your application window.



To add the JPanel, click on the Swing Containers tab in the component palette.

The first icon on this tab sheet should be the JPanel (which you can confirm by holding the mouse over the icon until it's name appears, click once on the JPanel icon

Click once near the bottom of the frame - this should add the JPanel to the bottom of the frame for you.

**Note** *If the JPanel does not appear at the bottom of the frame, click once on the JPanel either in the content pane or the component tree.*

Change the constraint property to **"South"**.

Next we need to add another JPanel to the frame and set the constraints property to **"East"**.

## Step 4 Adding the Viewer component

The JViewer is the interface between the application and the Image Server.



### JViewer

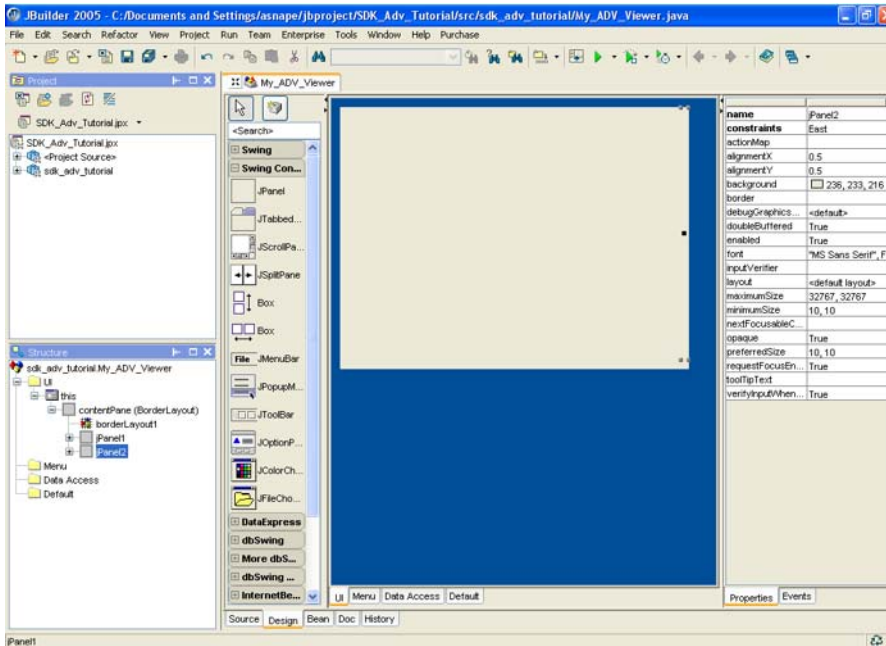
Select the "**DV IP Swing**" tab on the component palette; you should see the components that were added when you installed the toolkit into JBuilder.

**Note** *If you do not see this component palette, go back and follow the instructions for*

Locate the **JViewer** component and click it once

Click once in the middle of the frame to add the JViewer to the frame

Make sure the constraint property of the JViewer is set to "**Center**".



## Step 5 Adding the ControlSet component

The ControlSet is the component that links the controls and viewer together.

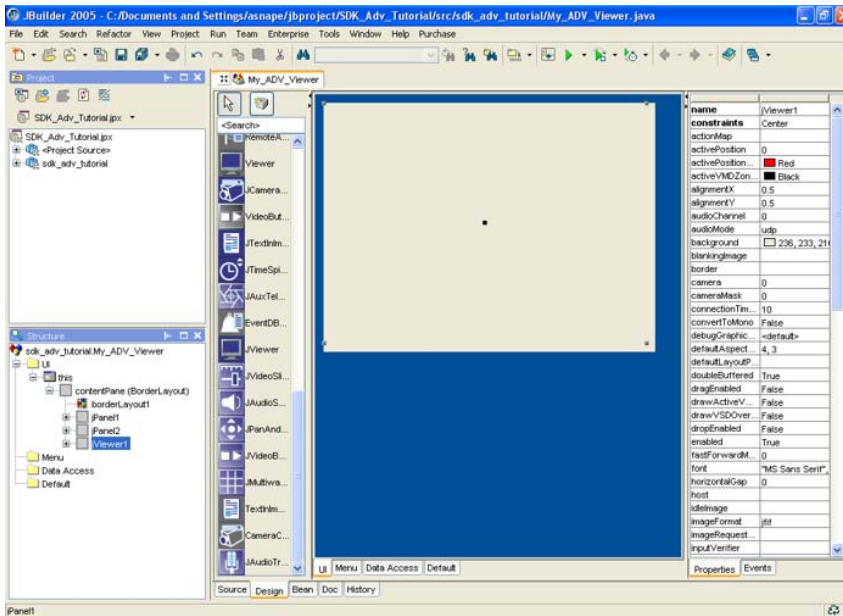


### ControlSet

Locate the **ControlSet** component in the component palette and click it once

Click once anywhere in the frame to add the ControlSet

Once you have added the ControlSet change the ViewerControl property in the inspector to the Viewer object we created earlier, which should be called "**jviewer1**".



## Step 6 Adding the JVideoButtons component

The JVideoButtons allow you to select Live mode, playback video (normal speed, fast forward, etc) or pause the video.

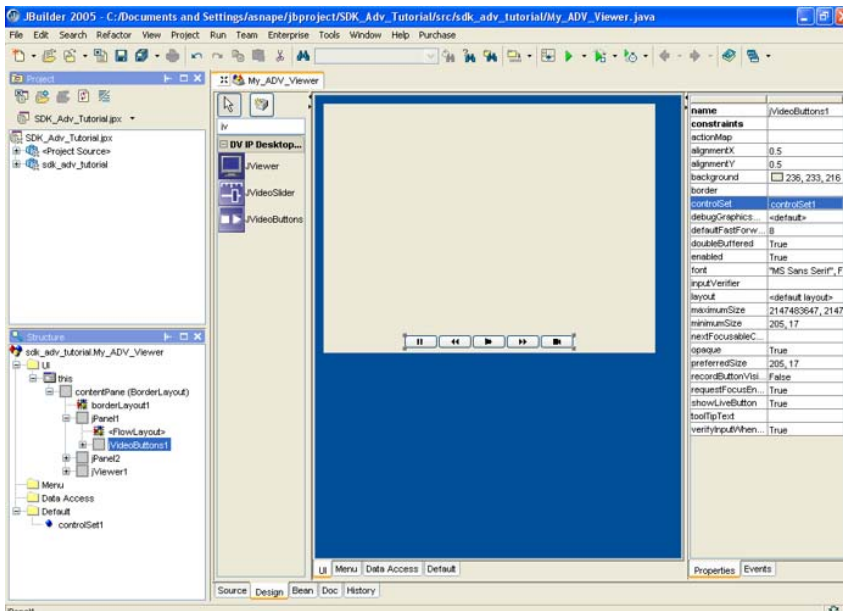


### JVideoButtons

Locate the **JVideoButtons** component in the component palette and click it once

Click once on JPanel1 in the component tree - The reason for clicking in the component tree this time rather than the frame is to make it easier to ensure that the JVideoButtons component is added in the right place.

Once you have added it change the ControlSet property in the inspector to the ControlSet object we created earlier, which should be called "**controlSet1**".



## Step 7 Adding the JLayoutCombo

The JLayoutCombo component selects between the screen layouts available for the video images being displayed in the Viewer.



### JLayoutCombo

Locate the **JLayoutCombo** component in the component palette and click it once.

Click once just in the "jPanel1" component. It should appear along side the video controls we placed earlier.

Switch to source mode to add the following lines of code. This will synchronize the changes in the layout selection with the JViewer.

After the text line

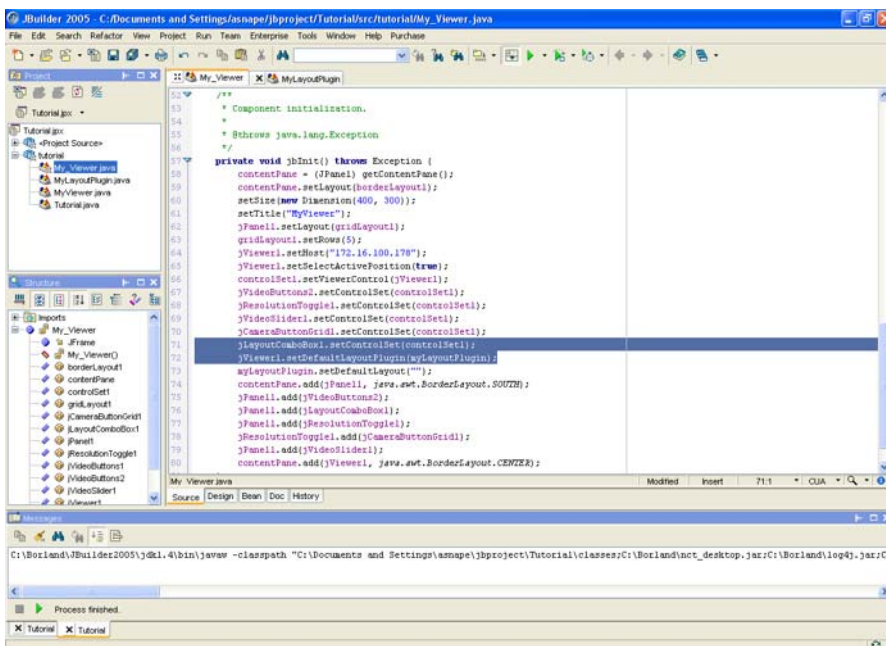
```
private void jblnit() throws Exception  
{
```

Add

```
jLayoutComboBox1.setControlSet(controlSet1);  
jViewer1.setDefaultLayoutPlugin(myLayoutPlugin);
```

before the closing bracket.

```
}
```



## Step 8 Adding the JCameraButtonGrid component

The JCameraButtonGrid component will allow you to select between the cameras on the Image Server for display in the Viewer.

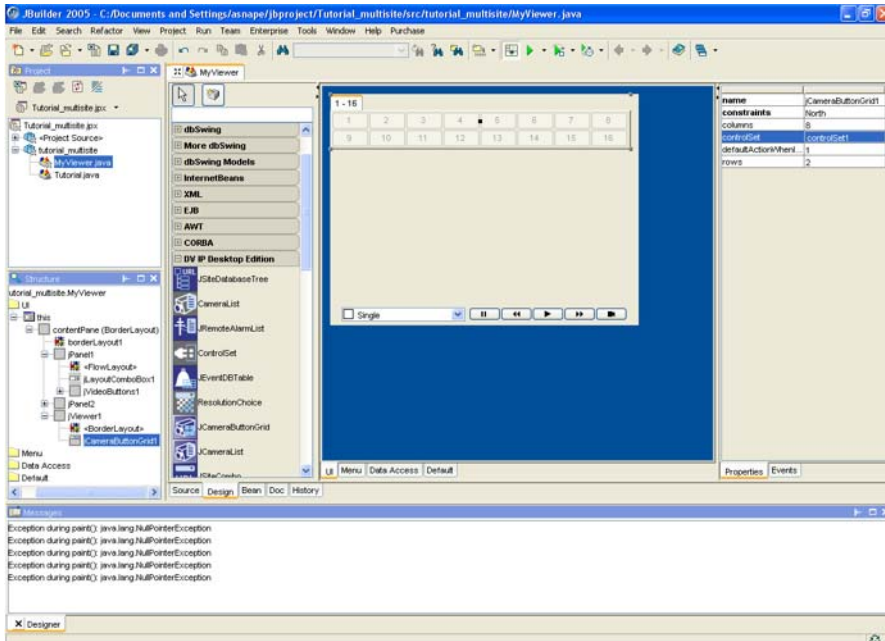


### JCameraButtonGrid

Locate the **JCameraButtonGrid** component in the component palette and click it once.

Click once at the top of the frame displayed in the content panel and make sure the constraint property is set to **"North"**.

Once you have added it change the ControlSet property in the inspector to the ControlSet object we created earlier, which should be called **"controlSet1"**.



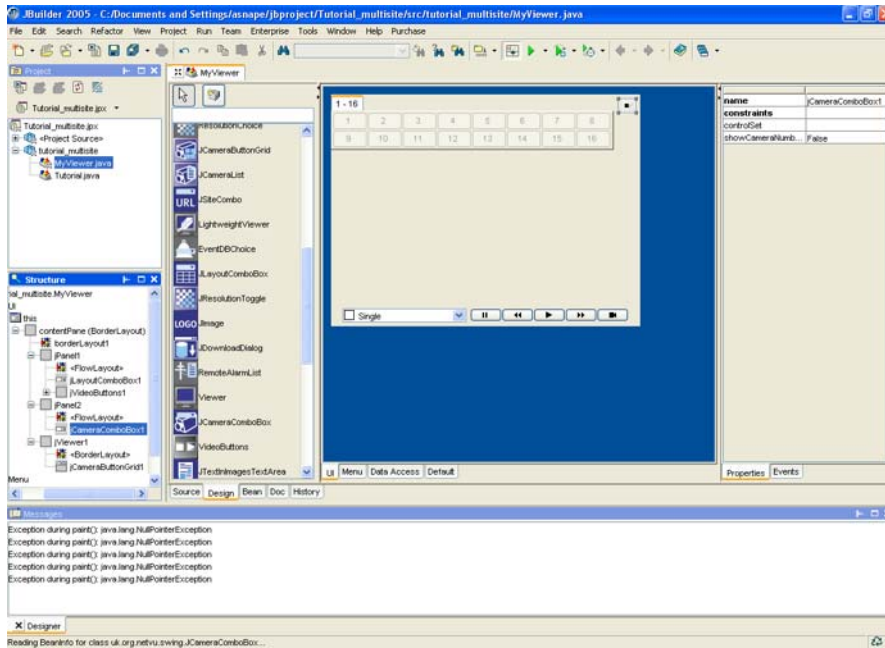
## Step 9 Adding a JComboBox

The JComboBox component will allow you to select different video servers to display in the JViewer.

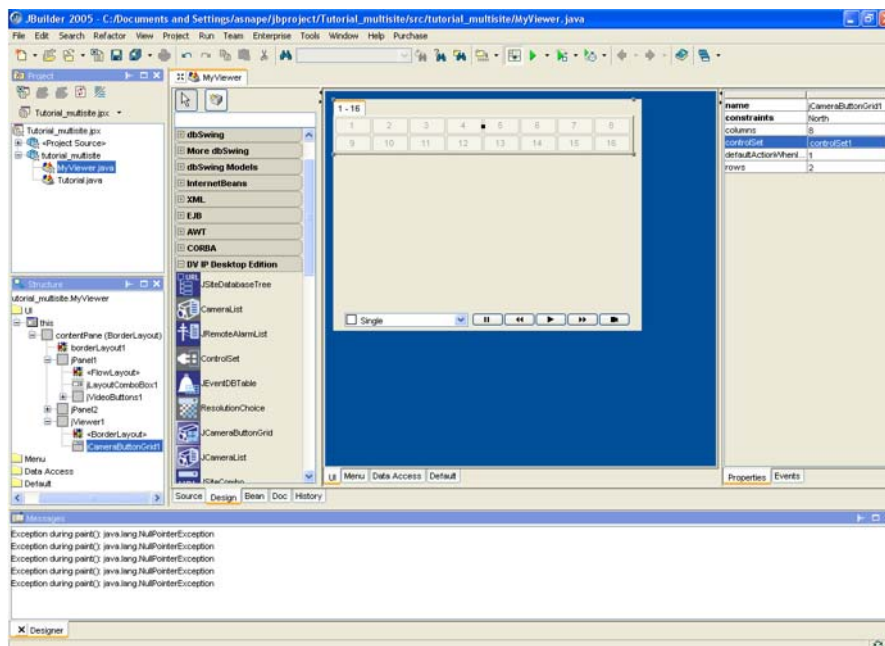


### JComboBox

Locate the JComboBox component in the Swing component palette and click it once. Click once in the JPanel2 we added earlier.



Switch to view the source of Myviewer.java and add the following code to connect to multiple servers.



After the text line

```
public MyViewer() {  
    try {  
        setDefaultCloseOperation(EXIT_ON_CLOSE);  
        jblnit();  
    } catch (Exception exception) {  
        exception.printStackTrace();  
    }  
}
```

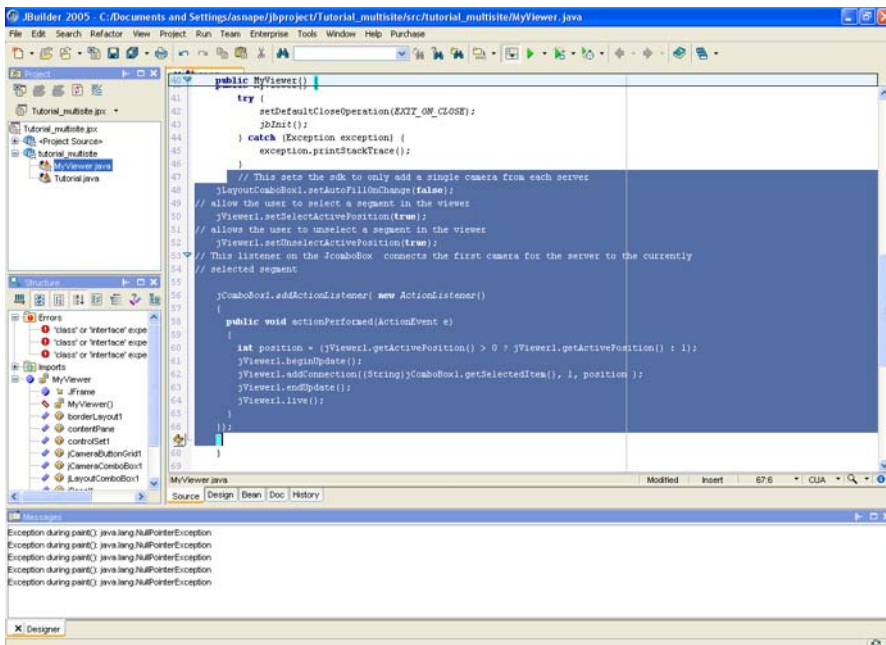
Add

```
// This sets the sdk to only add a single camera from each server
jLayoutComboBox1.setAutoFillOnChange(false);
// allow the user to select a segment in the viewer
jViewer1.setSelectedActivePosition(true);
// allows the user to unselect a segment in the viewer
jViewer1.setSelectedActivePosition(false);
// This listener on the JComboBox connects the first camera for the server to the currently
// selected segment
```

```
jComboBox1.addActionListener( new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        int position = (jViewer1.getActivePosition() > 0 ? jViewer1.getActivePosition() : 1);
        jViewer1.beginUpdate();
        jViewer1.addConnection(((String)jComboBox1.getSelectedItem(), 1, position );
        jViewer1.endUpdate();
        jViewer1.live();
    }
});
}
```

Before

```
/**
 * Component initialization.
```



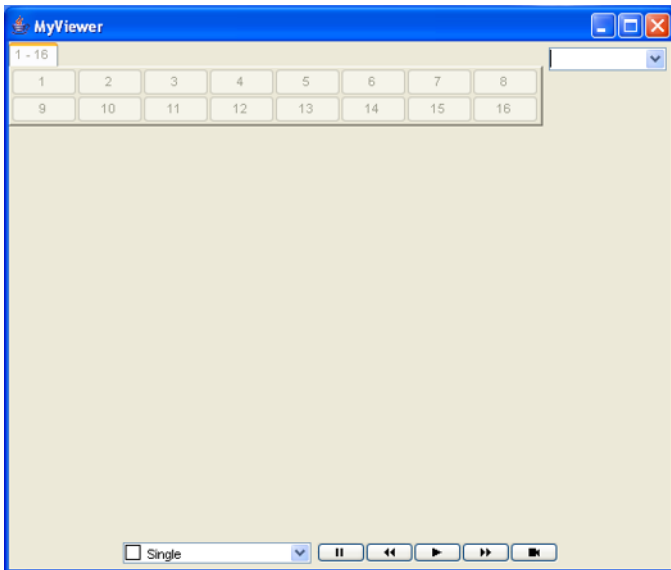
Switch back to design view and change the editable property to 'True'. This will allow you to enter an IP address in the Combo Box.

## Step 10 Compiling and running the application

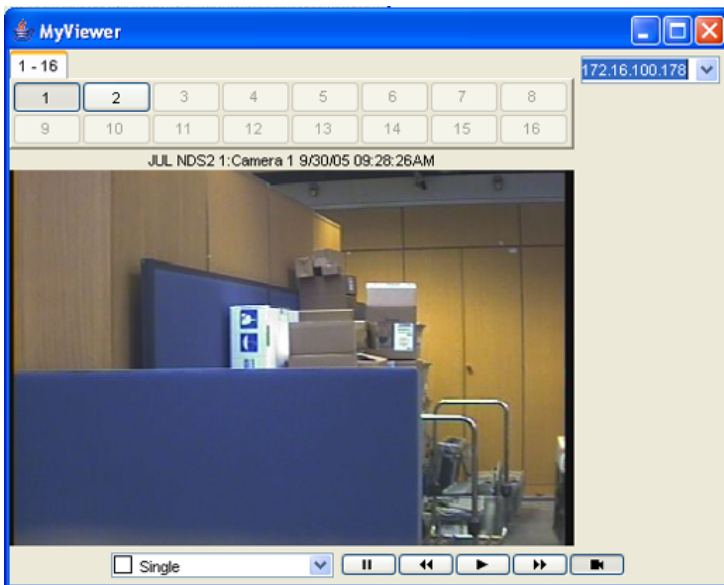
That completes the construction needed to build a multi site video client when you use the Java SDK!

All that remains now is to actually test out the program you have created, which you can do by clicking on the run button (green arrow) in the toolbar, or by pressing the F9 key.

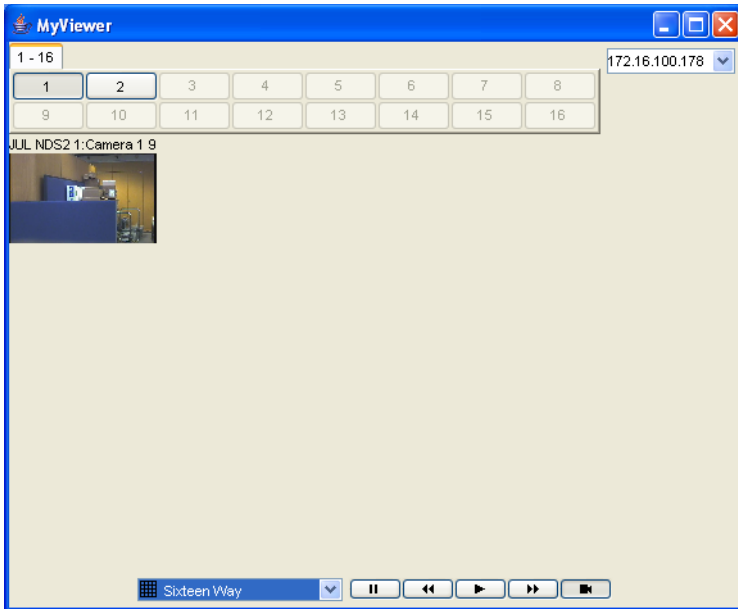
When the program runs you should be presented with the following window.



Type in the ip-address of a unit into the **JcomboBox** and video will be display in the center of the application.



Next switch the layout from single to sixteen way, so we can add other servers to the viewer.



Next select the bottom right segment in the viewer using the mouse, you should have a red square highlighting the segment and then type in a new ip address for another video server and hit return.

How you can add as many video servers to the viewer by repeatedly selecting a new segment and type in a new ip address for each server.

### Java Source Code for Multisite application.

```

package sdk_advanced;

import java.awt.BorderLayout;
import java.awt.Dimension;

import javax.swing.JFrame;
import javax.swing.JPanel;
import java.awt.*;
import uk.org.netvu.swing.JViewer;
import uk.org.netvu.control.ControlSet;
import uk.org.netvu.swing.JVideoButtons;
import uk.org.netvu.swing.JLayoutComboBox;
import uk.org.netvu.swing.JCameraButtonGrid;
import javax.swing.JComboBox;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

/**
 * <p>Title: </p>
 *
 * <p>Description: </p>
 *
 * <p>Copyright: Copyright (c) 2005</p>
 *
 * <p>Company: </p>
 *
 * @author not attributable
 * @version 1.0
 */
public class MyViewer extends JFrame {
    JPanel contentPane;

```

```

BorderLayout borderLayout1 = new BorderLayout();
JPanel jPanel1 = new JPanel();
JPanel jPanel2 = new JPanel();
JViewer jViewer1 = new JViewer();
ControlSet controlSet1 = new ControlSet();
JVideoButtons jVideoButtons1 = new JVideoButtons();
JLayoutComboBox jLayoutComboBox1 = new JLayoutComboBox();
JCameraButtonGrid jCameraButtonGrid1 = new JCameraButtonGrid();
JComboBox jComboBox1 = new JComboBox();

public MyViewer() {
    try {
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        jblnit();
    } catch (Exception exception) {
        exception.printStackTrace();
    }
    // This sets the sdk to only add a single camera from each server
    jLayoutComboBox1.setAutoFillOnChange(false);
    // allow the user to select a segment in the viewer
    jViewer1.setSelectedActivePosition(true);
    // allows the user to unselect a segment in the viewer
    jViewer1.setUnselectActivePosition(true);
    jComboBox1.addActionListener( new ActionListener()
    {
        public void actionPerformed(ActionEvent e)
        {
            int position = (jViewer1.getActivePosition() > 0 ? jViewer1.getActivePosition() : 1);
            jViewer1.beginUpdate();
            jViewer1.addConnection((String)jComboBox1.getSelectedItem(), 1, position );
            jViewer1.endUpdate();
            jViewer1.live();
        }
    });
}

/**
 * Component initialization.
 *
 * @throws java.lang.Exception
 */
private void jblnit() throws Exception {
    contentPane = (JPanel) getContentPane();
    contentPane.setLayout(borderLayout1);
    setSize(new Dimension(400, 300));
    setTitle("MyViewer");
    controlSet1.setViewerControl(jViewer1);
    jVideoButtons1.setControlSet(controlSet1);
    jLayoutComboBox1.setControlSet(controlSet1);
    jCameraButtonGrid1.setControlSet(controlSet1);
    jComboBox1.setMinimumSize(new Dimension(100, 20));
    jComboBox1.setPreferredSize(new Dimension(100, 20));
    jComboBox1.setEditable(true);
    contentPane.add(jPanel1, java.awt.BorderLayout.SOUTH);
    jPanel1.add(jLayoutComboBox1);
    jPanel1.add(jVideoButtons1);
    contentPane.add(jPanel2, java.awt.BorderLayout.EAST);
}

```

```
jPanel2.add(jComboBox1);
contentPane.add(jViewer1, java.awt.BorderLayout.CENTER);
jViewer1.add(jCameraButtonGrid1, java.awt.BorderLayout.NORTH);
}
}
```