IBM® Lotus® Symphony™ Developer's Tutorial: Building A Simple Document Workflow Plug-in

IBM® Lotus® Symphony™ Developer's Tutorial: Building A Simple Document Workflow Plug-in

Contents

Preface 1

Overview 3

Lesson 1 Add a Simple Side Shelf . . . 5
Verify the Eclipse Development Environment . . . 5
Create an Eclipse Plug-in for the Document Workflow 6

Lesson 2 Create UI on the Side Shelf 13
Define Variables 13
Complete the CreatePartControl Method 13
Add Helper Methods and Inner Classes 15
Create the Library 17
Run the Application 18

Lesson 3 Access and Modify the Documents 19
Open a Document from the Library 19
Using SelectionService and Accessing the Content of the Document 19
Modifying the Current Document 21
Run the Application 23

Conclusion 25

Appendix . Notices 27

Preface

Note: Before using this information and the product it supports, read the information in "Notices".

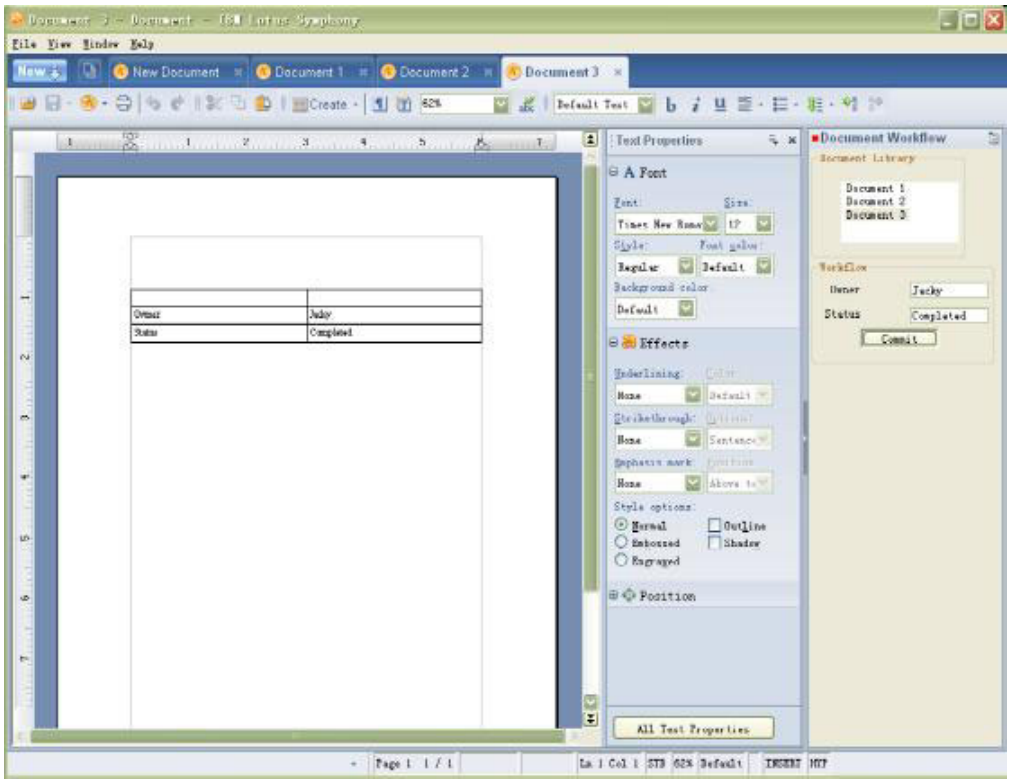
This edition applies to release IBM Lotus Symphony Toolkit 1.0 (license number L-AENR-7DSDUB) and to all subsequent releases and modifications until otherwise indicated in new editions.

Overview

When many people think of IBM® Lotus® Symphony, they rightly think of it as a group of office productivity editors, but it fact it is much more. Because Symphony is based on OpenOffice.org technology and the Eclipse and Lotus Expeditor rich client platform, it's by definition an extensible product. Lotus business partners take advantage of this flexibility to add new features that enhance the base product's capabilities, delivered as a natural extension of Symphony's user interface. As you will learn in this tutorial, it is also possible to add business-specific capabilities beyond editing. Whether you are enhancing the base editing features or adding business-specific capabilities, the approach is the same: Develop a plug-in that hooks into Symphony. The purpose of this tutorial is to demonstrate how plug-ins are used to extend the functionality of the office suite in a variety of ways.

In this tutorial you will create a simple document workflow plug-in. Using the document workflow plug-in. When a document is opened, in the Lotus Symphony window, the workflow information is displayed. Workflow information can be changed and stored in the document. The document is built on a special template and stored locally.

The following image shows the user interface with the document workflow plug-in.



With the document workflow plug-in, you will learn how to perform the following actions:

- Declare an extension to add an application to the side shelf area (right pane above).

- Open a document with Lotus Symphony API.
- Access and modify content of a document.

You will create the plug-in from start to finish and use pre-created snippets of code to speed up the manual data entry process. This tutorial is intended for developers who are familiar with Java™ programming and would like to try the plug-in development environment to extend Symphony. Eclipse and OpenOffice.org UNO programming knowledge are preferred, but not necessary.

To create the Document Workflow plug-in, you just need to learn the following three small lessons step by step:

Lesson 1: Show you how to build a simple Eclipse plug-in which adds a side shelf to Lotus Symphony.

Lesson 2: Show you how to create the User Interface (UI) on the side shelf created in lesson 1 using Eclipse's Standard Widget Toolkit (SWT).

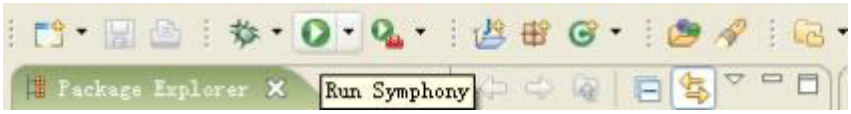
Lesson 3: Show you how to open documents from library, read workflow information when the document is loaded and modify the workflow information based on the contributions of lesson 2.

OK, now, let's begin.

Lesson 1 Add a Simple Side Shelf

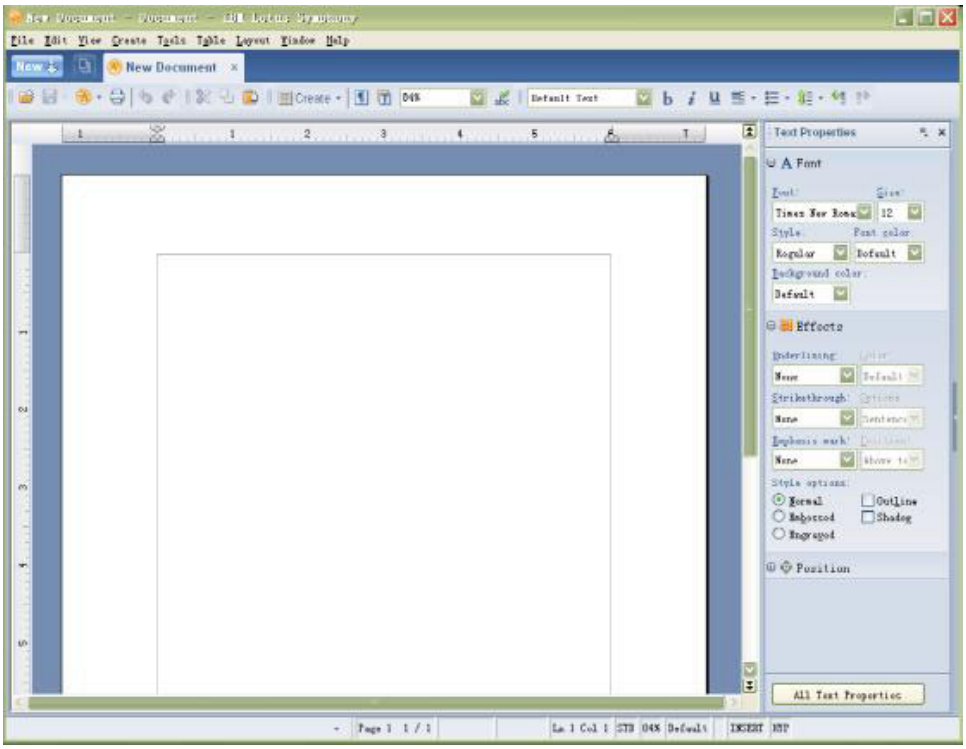
Verify the Eclipse Development Environment

1. Set up the Lotus Symphony development environment according *Symphony Developer's Guide* provided by the toolkit. Then, switch to the Plug-in development environment perspective by selecting **Window > Open Perspective > Other**, then select **Plug-in Development**, and then click **OK**.
2. Click **Run** from the toolbar, launches Lotus Symphony. (If the Run option is disabled, select **Run > Run...** to open the runtime configuration dialog. Select Eclipse **Client Services > Symphony** and then the **Run** button.) If asked if you want to clear the runtime workspace, select **YES**.



Tips: After Lotus Symphony opens, close the Welcome page.

3. When Lotus Symphony window open, click **File->New->Document**, you will see the following window:



This window is the standard Lotus Symphony document editor. In the next section you will add an Eclipse plug-in to the development environment and test that it works. Select **File > Exit** to close the runtime instance of Lotus Symphony before continuing.

Create an Eclipse Plug-in for the Document Workflow

Create a new plug-in

To create a new plug-in, follow these steps:

1. Launch the Eclipse development environment
2. Click **File > New > Project**
3. Select **Plug-in Project**, and click **Next**
4. Type`com.ibm.productivity.tools.samples.DocumentWorkflow` in the Project name field. Leave the rest of the default values.
5. Click **Next**.
6. Type a descriptive name in the **Plug-in Name** field, for example `Document Workflow Sample`. Keep the rest of the default values.
7. Click **Finish**.

Add the plug-in dependency

The following table lists plug-in dependencies used by the document library:

Dependency Plug-in	Description
org.eclipse.core.runtime, org.eclipse.ui	Eclipse core plug-ins
com.ibm.productivity.tools.ui.views com.ibm.productivity.tools.core	Lotus Symphony API plug-ins

Note: Adding these plug-in dependencies adds the following to the MANIFEST.MF file, which defines the plug-in. You can see the contents of this file by selecting the Plug-in Manifest Editor's `MANIFEST.MF` tab:

Require-Bundle: org.eclipse.ui,

org.eclipse.core.runtime,

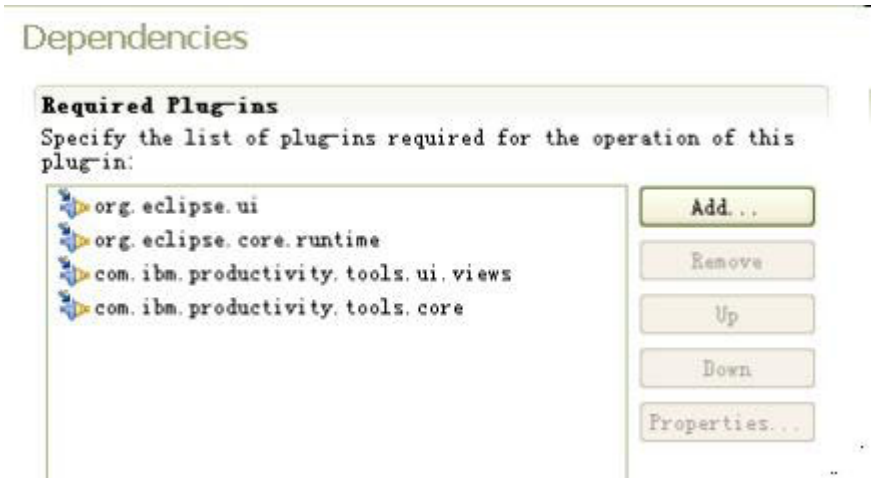
com.ibm.productivity.tools.ui.views,

com.ibm.productivity.tools.core

Perform the following steps to add the plug-in dependency.

1. Click the **Dependencies** tab of the Document Workflow plug-in manifest.
2. Click **Add**.
3. Add the following plug-ins:
   - com.ibm.productivity.tools.ui.views
   - com.ibm.productivity.tools.core

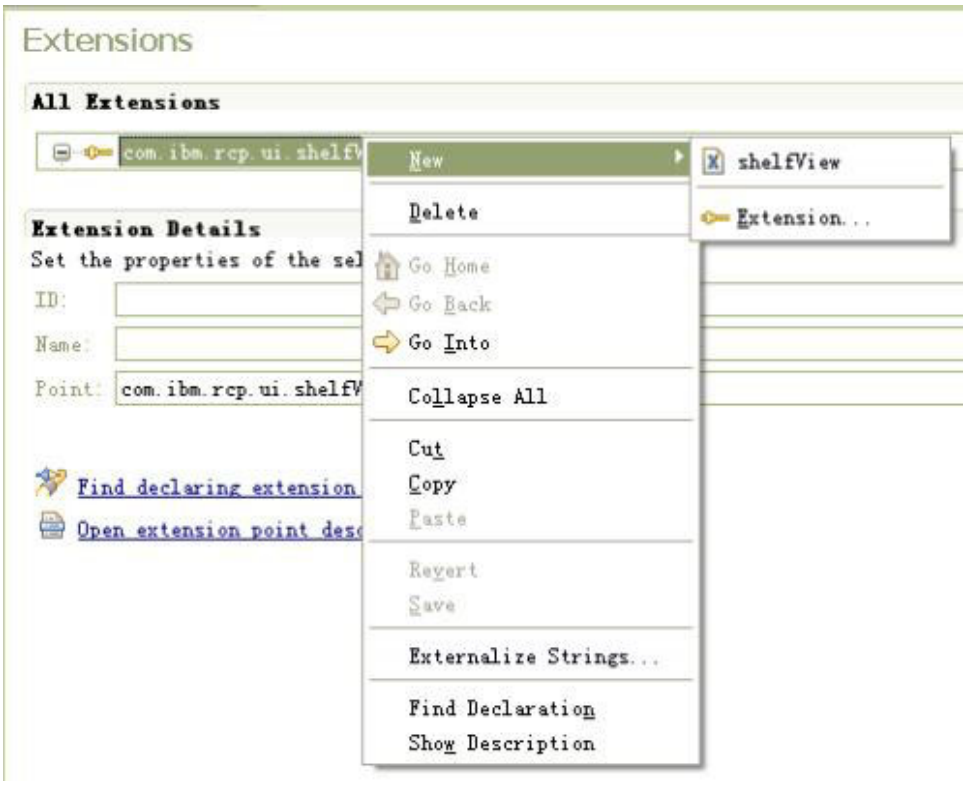
   The screen should look like the following image (the plug-in order is not important)



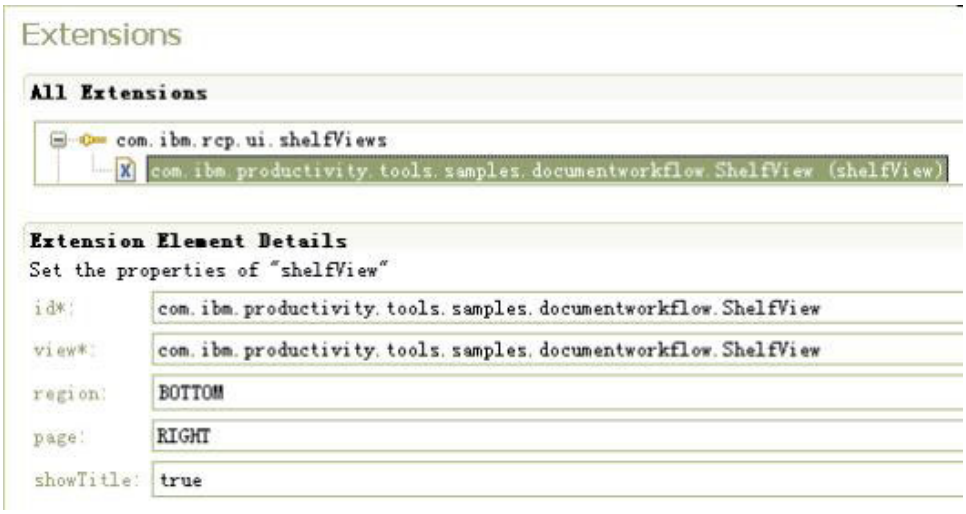
Add a side shelf

To add a side shelf, follow these steps:

1. Click the **Extensions** tab.
2. Click **Add**.
3. Add the following extension: `com.ibm.rcp.ui.shelfViews`.
4. Click **Finish**.
5. Right click the added extension and select **New > shelfView**.



Selecting this menu choice adds a shelfview element to the extension declaration. Select the newly added element and note that the Extension Element Details are updated to show the possible attributes. Fill in the fields as shown below.



The asterisk (*) indicates a required field. One of particular importance is the **class** field which indicates the Java class that implements the shelfview's behavior (for example, this class defines what the side shelf area contains, how it responds to user events.).

The id used in this tutorial is: com.ibm.productivity.tools.samples.documentworkflow.ShelfView. You can copy and paste it to the id field as figure above.

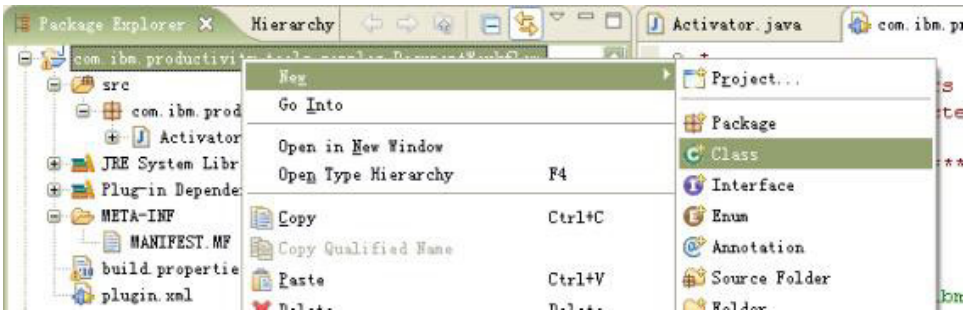
6. Click the plugin.xml tab
7. Copy and paste the following into the `plugin.xml` file.

```
<extension
      point="org.eclipse.ui.views">
   <category
         name="Sample Category"
         id="com.ibm.productivity.tools.samples">
   </category>
   <view
         name="Document Workflow"
         icon=" "
         category="com.ibm.productivity.tools.samples"
         class="com.ibm.productivity.tools.samples.documentworkflow.ShelfView"
         id="com.ibm.productivity.tools.samples.documentworkflow.ShelfView">
   </view>
</extension>
```

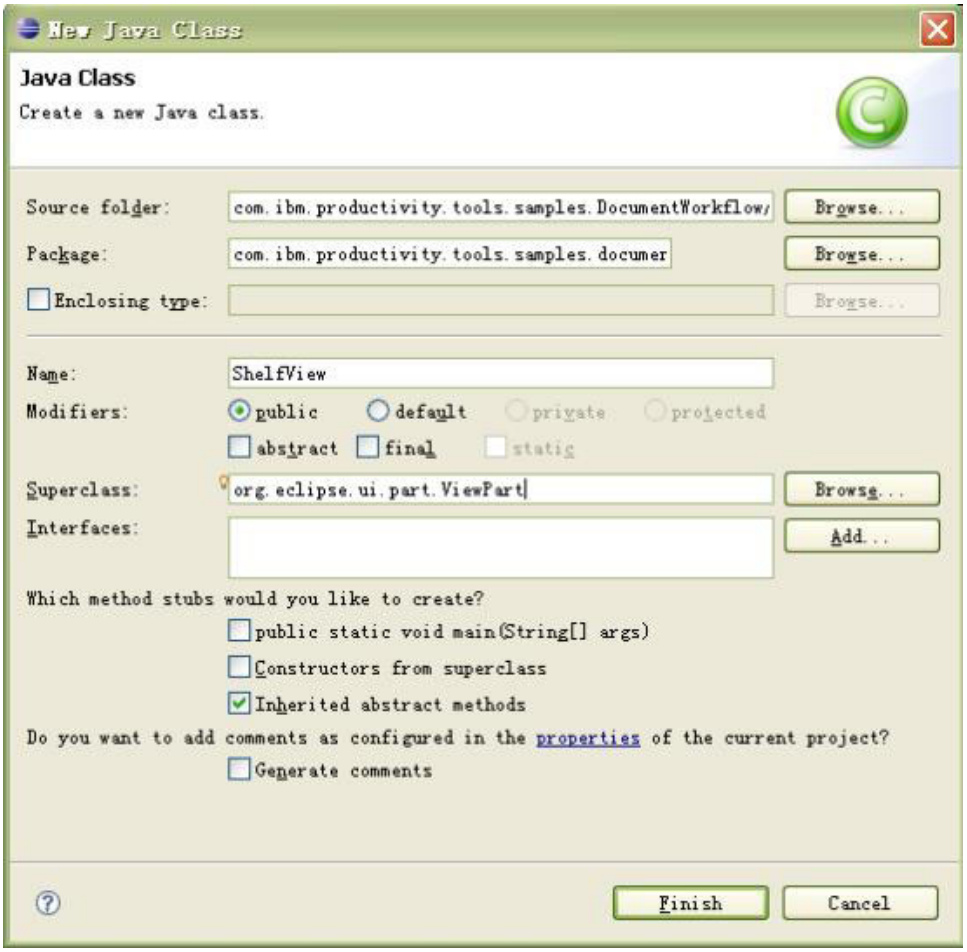
The view attribute of the <shelfView> tag in the com.ibm.rcp.ui.shelfViews extension must match exactly the id field of the <view> tag in the org.eclipse.ui.views extension above. That is, the side-shelf content is defined by the extensions' <view>/<shelfView> pairs.

The steps above adds a new Eclipse ViewPart to the platform. You can create your plug-in extensions with the Manifest Editor or enter the specifications directly in the `plugin.xml` file.

8. Right-click on the package com.ibm.productivity.tools.samples.DocumentWorkflow in **Package Explorer**, and then select **New > Class**.



9. Input the class information as showing in the following screen capture, Click the **Browse** to search the superclass of org.eclipse.ui.part.ViewPart, and then clicked **Finish**.



A new Eclipse ViewPart named ShelfView is created in the com.ibm.productivity.tools.samples.documentworkflow package.

Run the application

To run the application, follow these steps:

1. Check your plug-in

   Before running the application, look at the `plugin.xml` file and the newly created class:

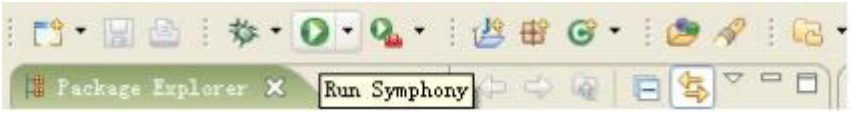
   The `plugin.xml` file should look like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>

<plugin>
   <extension
         point="com.ibm.rcp.ui.shelfViews">
      <shelfView
            id="com.ibm.productivity.tools.samples.documentworkflow.ShelfView"
            page="RIGHT"
            region="BOTTOM"
            showTitle="true"
            view="com.ibm.productivity.tools.samples.documentworkflow.ShelfView"/>
   </extension>

   <extension
         point="org.eclipse.ui.views">
      <category
            name="Sample Category"
            id="com.ibm.productivity.tools.samples">
      </category>
      <view
            name="Document Workflow"
            icon=" "
            category="com.ibm.productivity.tools.samples"
            class="com.ibm.productivity.tools.samples.documentworkflow.ShelfView"
            id="com.ibm.productivity.tools.samples.documentworkflow.ShelfView">
      </view>
   </extension>

</plugin>
```

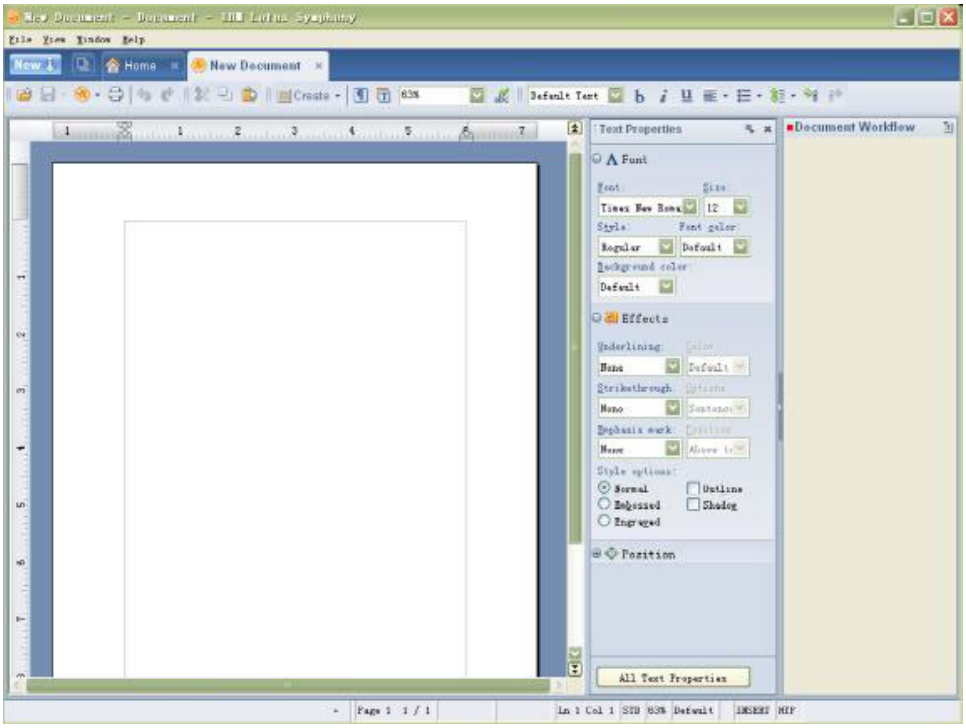
Double Click the `ShelfView.java` file in **Package Explorer**, the `ShelfView.java` file looks like the following:

```
package com.ibm.productivity.tools.samples.documentworkflow;

import org.eclipse.swt.widgets.Composite;
import org.eclipse.ui.part.ViewPart;

public class ShelfView extends ViewPart {

    public void createPartControl(Composite arg0) {
           // TODO Auto-generated method stub

    }

    public void setFocus() {
           // TODO Auto-generated method stub

    }

}
```

2. Click **Run** from the toolbar:



3. Lotus Symphony is launched. Click **File > New > Document**. Your screen should look similar to the following image:



Tip: If the newly added view does not display, check the console for a message like `org.eclipse. ui.PartInitException: Could not create view: XXX` and confirm that `XXX` = the view id. The view attribute of the <shelfView> tag in the com.ibm.rcp.ui.shelfViews extension must match the id attribute of the <view> tag in the org.eclipse.ui.views extension.

Congratulations! You have reserved space in the Lotus Symphony side shelf for your application. Next, you will open a document from the document library, and modify the document.

Lesson 2 Create UI on the Side Shelf

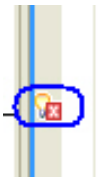
Before starting the Lotus Symphony APIs, let's add all the variables and methods that will be required for the document workflow plug-in, and some helper methods that are not specific to Lotus Symphony. We can then turn to the code that is specific to extending Lotus Symphony. All the following lines of code are within the `ShelfView.java` file.

Define Variables

The variables(field) are shown below. Copy and paste this block into the beginning of the `ShelfView. java` file after the first open brace ({) of this class.

```
private Text ownerTxt;
private Text statusTxt;
private RichDocumentView selectedView = null;
private TableViewer viewer;

final static public boolean isWindowsOS = System.getProperty("os.name")
              .startsWith("Windows");
```

Note: Use the menu choice **Source > Organize Imports** to add the needed import statements after pasting a code snippet. For example, in the following snippet, you will need to import the com.ibm.productivity.tools.ui.views.RichDocumentView package.



```
private Text ownerTxt;
private Text statusTxt;
private RichDocumentView selectedView = null;
private TableViewer viewer;
```

TableViewer and Text are defined by several packages. Be certain to select org.eclipse.jface.viewers. TableViewer and org.eclipse.swt.widgets.Text.

Complete the CreatePartControl Method

The Document Workflow adds a viewable area to the side shelf of Lotus Symphony. This area is defined in the createPartControl method using Eclipse's standard widgets, called SWT (Standard Widget Toolkit).

The SWT API is used to create user interface elements in Eclipse platform.

The following code snippet creates a document library group and a workflow group in side shelf, and other SWT controls. Copy the following code and replace the createPartControl method in the `ShelfView.java` files.

```
public void createPartControl(Composite parent) {
        parent.setLayout(new RowLayout());
        // Group of Document Library
        int y = 10 ;
        Group doclibGrp = new Group(parent, SWT.NULL);
        doclibGrp.setText("Document Library");
        doclibGrp.setSize(390, 72);
        doclibGrp.setLocation(10, y);

        viewer = new TableViewer(doclibGrp, SWT.MULTI
                        | SWT.H_SCROLL | SWT.V_SCROLL);
        viewer.getTable().setSize(120, 60);
        viewer.getTable().setLocation(30, y + 20);
        viewer.setContentProvider(new ViewContentProvider());
        viewer.setLabelProvider(new ViewLabelProvider());
        viewer.setInput(this );
        hookDoubleClickAction();

        // Group of user information
        y = 10 + doclibGrp.getBounds().height + 7;
        Group workflowGrp = new Group(parent, SWT.NULL);
        workflowGrp.setText("Workflow");
        workflowGrp.setSize(390, 72);
        workflowGrp.setLocation(10, y);

        Label ownerLbl = new Label(workflowGrp, SWT.SHADOW_NONE
                        | SWT.RIGHT);
        ownerLbl.setText("Owner");
        ownerLbl.setSize(40, 22);
        ownerLbl.setLocation(10, 22);
```

```
        Label statusLbl = new Label(workflowGrp, SWT.SHADOW_NONE
                        | SWT.RIGHT);
        statusLbl.setText("Status");
        statusLbl.setSize(40, 22);
        statusLbl.setLocation(10, 46);

        ownerTxt = new Text(workflowGrp, SWT.BORDER | SWT.LEFT);
        ownerTxt.setBackground(new Color(this.getSite().
                        getShell().getDisplay(), 255, 255, 255));
        ownerTxt.setSize(80, 16);
        ownerTxt.setLocation(100, 21);

        statusTxt = new Text(workflowGrp, SWT.BORDER | SWT.LEFT);
        statusTxt.setBackground(new Color(this.getSite().
                        getShell().getDisplay(), 255, 255, 255));
        statusTxt.setSize(80, 16);
        statusTxt.setLocation(100, 46);

        Button commitBtn = new Button( workflowGrp, SWT.BORDER
                        | SWT.PUSH );
        commitBtn.setText("Commit");
        commitBtn.setSize(80, 20);
        commitBtn.setLocation(50, 67);
        commitBtn.addSelectionListener(new SelectionListener(){

                public void widgetDefaultSelected(SelectionEvent arg0) {

                }

                public void widgetSelected(SelectionEvent arg0) {

                        commitWorkflowInfo();
                }

        });

  }
```

Reminder: Use the menu choice **Source > Organize Imports** to add the needed import statements after pasting a code snippet. If there are methods and classes missing, continue the copy and paste operation in the following document.

Label and SelectionEvent are defined by several packages. Be certain to select the ones defined in the SWT packages.

You will need to import the following packages; you can check the list or just copy and paste the following lines into the import area to replace the older.

```
import java.io.IOException;

import org.eclipse.core.runtime.FileLocator;
import org.eclipse.core.runtime.Platform;
import org.eclipse.jface.viewers.DoubleClickEvent;
import org.eclipse.jface.viewers.IDoubleClickListener;
import org.eclipse.jface.viewers.ISelection;
import org.eclipse.jface.viewers.IStructuredContentProvider;
import org.eclipse.jface.viewers.IStructuredSelection;
import org.eclipse.jface.viewers.ITableLabelProvider;
import org.eclipse.jface.viewers.LabelProvider;
import org.eclipse.jface.viewers.TableViewer;
import org.eclipse.jface.viewers.Viewer;
import org.eclipse.swt.SWT;
import org.eclipse.swt.events.SelectionEvent;
import org.eclipse.swt.events.SelectionListener;
import org.eclipse.swt.graphics.Color;
import org.eclipse.swt.graphics.Image;
import org.eclipse.swt.layout.RowLayout;
import org.eclipse.swt.widgets.Button;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Group;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.widgets.Text;
import org.eclipse.ui.part.ViewPart;
import org.osgi.framework.Bundle;
import com.ibm.productivity.tools.ui.views.RichDocumentView;
```

Add Helper Methods and Inner Classes

SWT includes the low-level widgets that are common across different platforms. For example, a label, a text entry field, a button, and so on. Eclipse developers can use another framework called JFace to simplify the widget code by mapping widget friendly data types like strings into application friendly objects like clients, documents, and similar high-level classes. For example, the TableViewer works with helper objects that handle the mapping of higher-level classes like documents into lower-level data types expected by widgets, like strings. The next two classes define these helper classes.

The following methods and inner classes are required; copy each of them to the end of the `ShelfView.java` file one by one, before the last close brace (}) of this class.

1. Copy ViewContentProvider inner class.

The ContentProvider provides the file name of sample documents listed in the document library:

```
class ViewContentProvider implements IStructuredContentProvider {
        public void inputChanged(Viewer v, Object oldInput,
                        Object newInput) {
        }

        public void dispose() {
        }

        public Object[] getElements(Object parent) {
                return new String[] { "Document 1", "Document 2",
                                "Document 3" };
        }
}
```

2. Copy ViewLabelProvider inner class.

The LabelProvider provides the viewable text of file name of each sample document in document library:

```
class ViewLabelProvider extends LabelProvider implements
        ITableLabelProvider {
        public String getColumnText(Object obj, int index) {
                return getText(obj);
        }
        public Image getColumnImage(Object obj, int index) {
                return getImage(obj);
        }
        public Image getImage(Object obj) {
                return null;
        }
}
```

Class Image is defined by more than one package. Be certain to select the ones defined in the SWT package.

3. Copy hookDoubleClickAction method.

The method handles double-click event in the document library:

```
private void hookDoubleClickAction() {
        viewer.addDoubleClickListener(new IDoubleClickListener() {
                public void doubleClick(DoubleClickEvent event) {
                        ISelection selection = viewer.getSelection();
                        Object obj = ((IStructuredSelection)selection).
                                getFirstElement();

                        String displayName = obj.toString();
                        String url = getDocumentURL( displayName );



                }
        });
}
```

4. Copy getDocumentURL method.

The method translates the display name of each sample document in the document library into an absolute URL:

```
private String getDocumentURL( String displayName ) {
          String url = "";

          String res = "docs/" + displayName + ".odt";

          url = getResolvedPath( res );

          return url;
  }

  private String getResolvedPath(String file) {
          String resolvedPath = null;
       Bundle bundle = Platform.getBundle(Activator.PLUGIN_ID );
       if (bundle != null) {
           java.net.URL bundleURL = bundle.getEntry( "/" );
           if (bundleURL != null) {
               try {
                   resolvedPath = FileLocator.resolve(
                                  bundleURL ).getFile();
               } catch (IOException e) {
                 e.printStackTrace();
               }
           }
       }

       String ret = resolvedPath+file;
       if(isWindowsOS) {
          ret = ret.substring(1);
          ret = ret.replace('/', '\\');
       }
       return ret;
  }
```

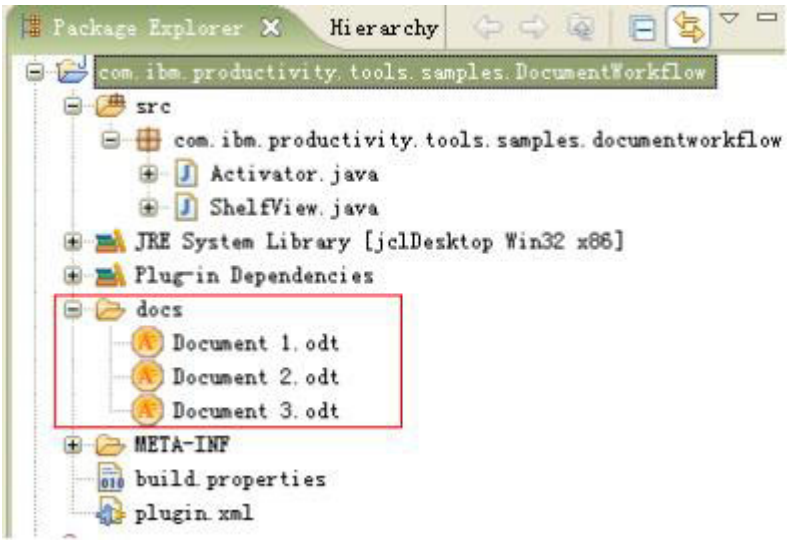
IOException and Platform are defined by more than one package. Be certain to select the ones defined in the Java and Eclipse packages.

Create the Library

We use a dummy library in this tutorial, meaning that all files are local files. Complete the following steps to create the library:

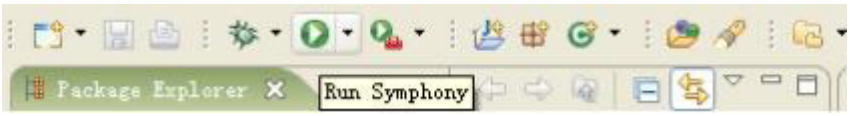
1. Open Windows Explorer, locate the tutorial directory, right-click the `docs` directory, and then select **Copy**.
2. Switch to the Eclipse environment. Right click the com.ibm.productivity.tools.sample.DocumentWorkflow plug-in, and select **Paste**.

The docs directory is created as shown in the following screen capture:

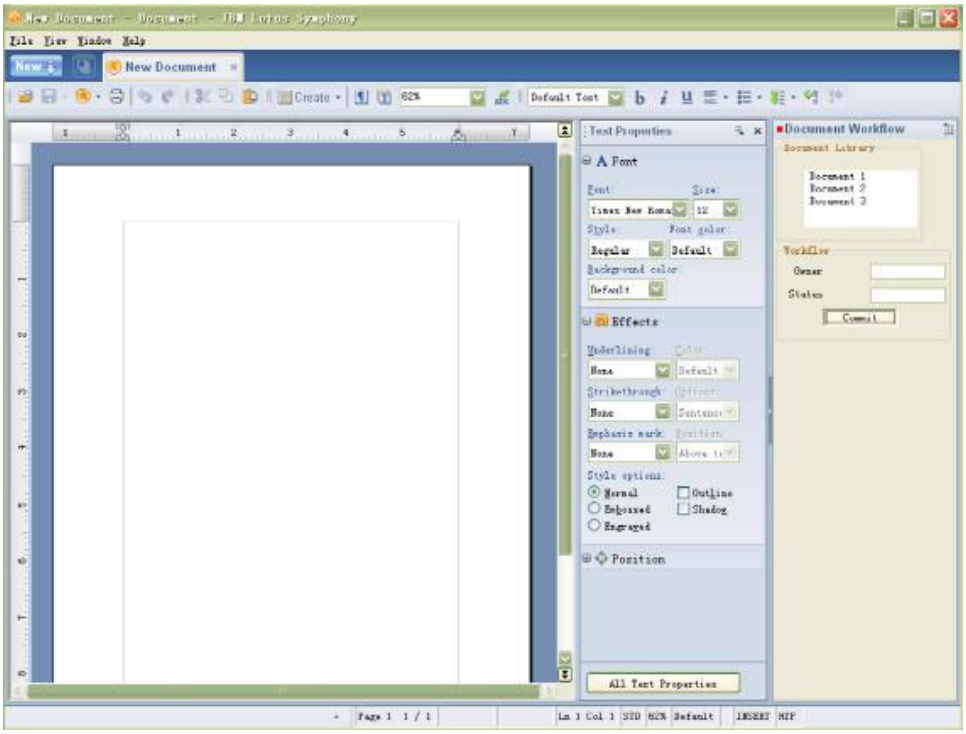


Run the Application

1. Confirm the codes `ShelfView.java` file is correct, and click **File > Save All**.
2. Click **Run** from the toolbar.



The application is launched. Click **File > New > Document**, and your view should be similar to the following screen capture.



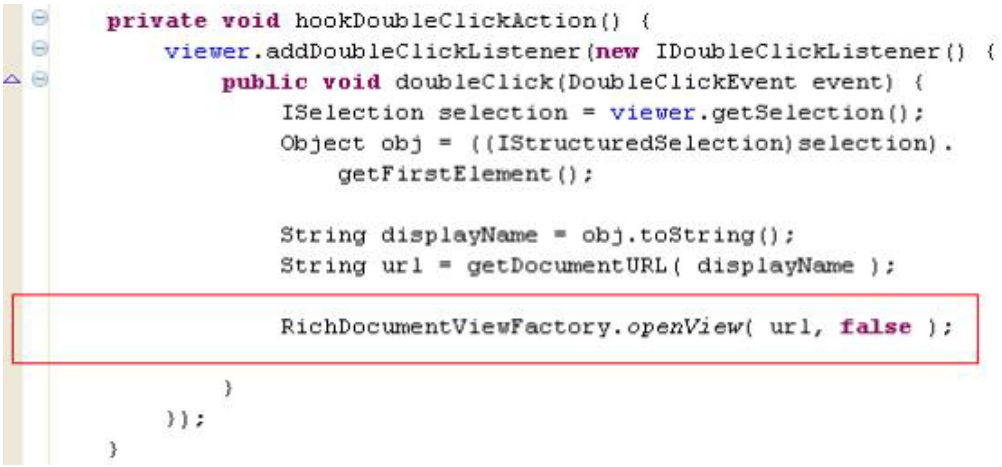
Lesson 3 Access and Modify the Documents

Open a Document from the Library

Locate the hookDoubleClickAction method in the `ShelfView.java` file, and add the following line to the end of this function. When you double-click a file entry in the document library, the document is opened with the following code automatically:

```
RichDocumentViewFactory.openView( url, false );
```

Note: Use the menu choice **Source > Organize Imports** to add the needed import statements after pasting a code snippet. You will need to import com.ibm.productivity.tools.ui.views.RichDocumentViewFactory package. Your screen should look similar to the following image:



Using SelectionService and Accessing the Content of the Document

The selection service provided by the Eclipse workbench allows efficient linking of different parts within the workbench window. Each workbench window has its own *selection service* instance which service keeps track of the selection in the currently active part and propagates selection changes to all registered listeners. Such selection events occur when the selection in the current part is changed or when a different part is activated. Selection events can be triggered by user interaction or programmatically.

Each Lotus Symphony view registers the selection provider, so it is possible to monitor the selection change event in Lotus Symphony.

The following code demonstrates how to use the selection provider in Lotus Symphony. Perform the following steps:

- **Create an instance of ISelectionListener**

  The selection listener is invoked when users switch between opened views. It will perform the following tasks at runtime:

  1. Gets the selected RichDocumentView.
  2. Gets the UNO model of the currently selected view.
  3. Queries the workflow information from the document with the UNO API.
  4. Updates the user interface of the document workflow plug-in:

Copy and paste the code into the `ShelfView.java` file. Copy it after the field *isWindowsOS* defined.

```
private ISelectionListener selectionListener = new ISelectionListener(){

          public void selectionChanged(IWorkbenchPart arg0, final ISelection arg1) {

                 //It is suggested to create a new Job for UNO call which will be
                 //invoked by Symphony C++ process
                 //the selctionChanged() is called when selection happened within
                 //symphony document
              Job job = new Job("Update workflow info") {
                       public IStatus run(IProgressMonitor progress) {

                         IAdaptable adaptable = ( IAdaptable ) arg1;

                         RichDocumentViewSelection selection =
   ( RichDocumentViewSelection )
                              adaptable.getAdapter(RichDocumentViewSelection.class);

                              selectedView = selection.getView();
                              Object model = selectedView.getUNOModel();

                              XTextTablesSupplier tablesSupplier =
   ( XTextTablesSupplier )
                              UnoRuntime.queryInterface( XTextTablesSupplier.class, model
   );

                              if( tablesSupplier == null)
                                  return Status.OK_STATUS;

                              XNameAccess nameAccess = tablesSupplier.getTextTables();

                              try {
                                  XTextTable table = ( XTextTable )
   UnoRuntime.queryInterface
                                          ( XTextTable.class,nameAccess.getByName
   ("Workflow"));
                                  XCell cell = table.getCellByName( "B2" );
                                  XText text = ( XText ) UnoRuntime.queryInterface
                                  (XText.class, cell);
                                  final String owner = text.getString();
                                  cell = table.getCellByName("B3");
                                  text = (XText)UnoRuntime.queryInterface(XText.class,
   cell);
                                  final String reviewer = text.getString();
```

```
                        //Execute the SWT UI related functions in display thread
                               Display.getDefault().asyncExec(new Runnable(){

                                       public void run() {
                                              updateWorkflowInfo( owner, reviewer );

                                              }

                                  });

                              } catch (NoSuchElementException e) {
                                       e.printStackTrace();
                              } catch (WrappedTargetException e) {
                                       e.printStackTrace();
                              }
                              return Status.OK_STATUS;
                       }
              };
              job.schedule();

          }
};
```

Notes 1. NoSuchElementException and XText are defined by more than one package. Be certain to select the ones defined in the Sun packages not the Java packages.

2. Please notice the following line in this step; it is used to get the UNO document model of current document. The UNO document model is the entry point to access and modify document content. You can get almost every object within the document. In this tutorial, you will learn how to access a pre-defined

table named as workflow in the document.

```
Object model = selectedView.getUNOModel();
```

3. Use the menu choice **Source > Organize Imports** (Ctrl+Shift+O) to add the needed import statements after pasting a code snippet. You will need to import more packages. The following classes are listed here:

```
import org.eclipse.core.runtime.IAdaptable;
import com.sun.star.container.NoSuchElementException;
import com.sun.star.container.XNameAccess;
import com.sun.star.lang.WrappedTargetException;
import com.sun.star.table.XCell;
import com.sun.star.text.XText;
import com.sun.star.text.XTextTable;
import com.sun.star.text.XTextTablesSupplier;
import com.sun.star.uno.UnoRuntime;
import com.ibm.productivity.tools.ui.views.RichDocumentViewSelection;
import org.eclipse.ui.ISelectionListener;
import org.eclipse.ui.IWorkbenchPart;
```

- **Use the updateWorkflowInfo method**

  The following code updates the user interface of document workflow plug-in. Copy the following code after the end of the field selectionListener defined after this method's final close brace (} ;).

```
private void updateWorkflowInfo(String owner, String reviewer) {

        ownerTxt.setText(owner);
        statusTxt.setText(reviewer);

}
```

- **Register the selection listener**

  To make the selection listener work, register the listener into the selection service. Copy the following code into the end of the createPartControl() method before this method's final close brace (}).

```
        IWorkbenchWindow window = PlatformUI.getWorkbench().
            getActiveWorkbenchWindow();
        ISelectionService service = window.getSelectionService();
        service.addSelectionListener(selectionListener);
```

  Use the menu choice **Source > Organize Imports** to add the needed import statements after pasting a code snippet. You will need to import more packages. The following import statements are listed here.

```
import org.eclipse.ui.ISelectionService;
import org.eclipse.ui.IWorkbenchWindow;
import org.eclipse.ui.PlatformUI;
```

Modifying the Current Document

In this section, you will learn how to modify the current Lotus Symphony document with content from the document workflow plug-in. Perform the following steps to add workflow information into the Lotus Symphony document.

- **Add function to a commit button**

  Locate the commitBtn in createPartControl, and change it as shown in the following sample. The commitWorkflowInfo method is invoked when the widget is selected.

```
commitBtn.addSelectionListener(new SelectionListener(){

        public void widgetDefaultSelected(SelectionEvent arg0) {
                // TODO Auto-generated method stub

        }

        public void widgetSelected(SelectionEvent arg0) {

                commitWorkflowInfo();

        }

});
```

- **Add the commitWorkflowInfo method**

  This method reads content from the document workflow plug-in and invokes a function to write data into the current document. Copy and paste it into the `ShelfView.java` file after the last close brace (}) of the method createPartControl:

```
private void commitWorkflowInfo( ) {
        String owner = ownerTxt.getText();
        String reviewer = statusTxt.getText();

        writeWorkflowInfo( owner, reviewer );
}
```

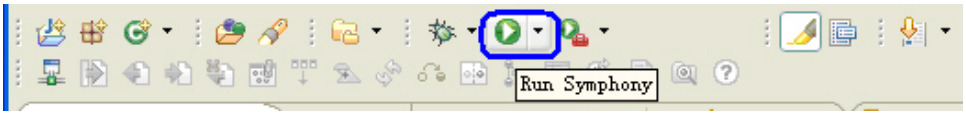
- **Add writeWorkflowInfo method**

  This method commits workflow data into the current document. Copy and paste the following function into the `ShelfView.java` file after last close brace (}) of the method commitWorkflowInfo() .

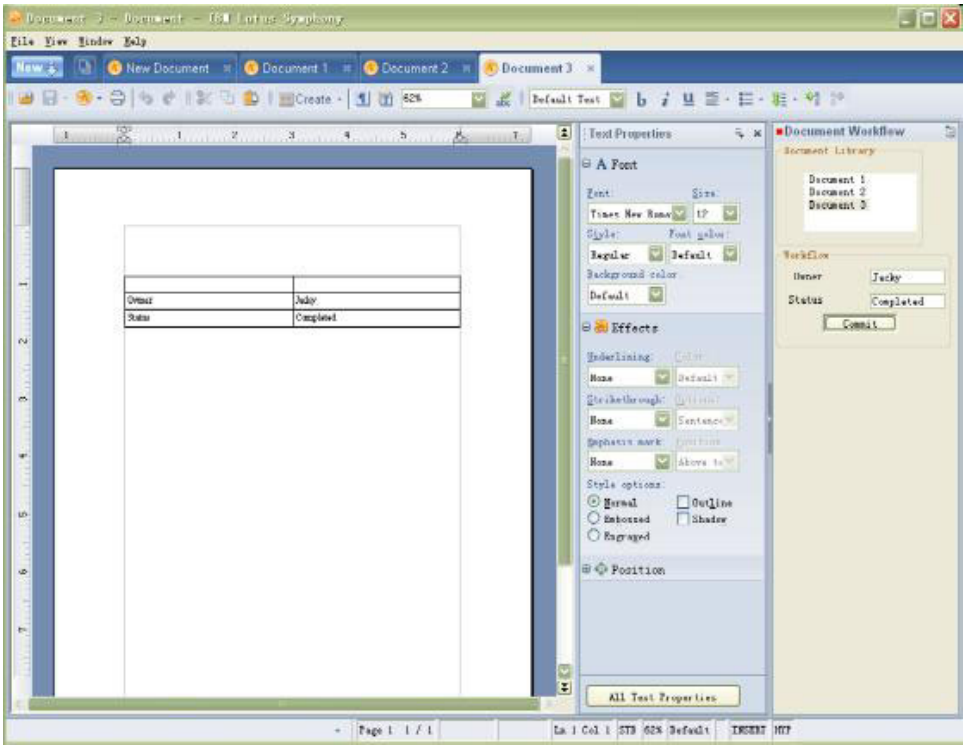
```
private void writeWorkflowInfo(String owner, String reviewer) {
        Object model = selectedView.getUNOModel();

        XTextTablesSupplier tableSupplier = ( XTextTablesSupplier )
          UnoRuntime.queryInterface( XTextTablesSupplier.class, model );

        XNameAccess nameAccess = tableSupplier.getTextTables();

        try {
                XTextTable table = ( XTextTable )UnoRuntime.queryInterface
                  ( XTextTable.class,nameAccess.getByName("Workflow"));
                XCell cell = table.getCellByName( "B2" );
                XText text = ( XText ) UnoRuntime.queryInterface(
                                XText.class, cell);
                text.setString(owner);
                cell = table.getCellByName("B3");
                text = ( XText ) UnoRuntime.queryInterface(XText.class,
                                cell);
                text.setString(reviewer);

        } catch (NoSuchElementException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
        } catch (WrappedTargetException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
        }

}
```

Run the Application

1. Confirm the codes in the `ShelfView.java` file is correct, and click **File > Save ALL**.
2. Click **Run** from the toolbar.



The application is launched; click **File > New > Document**, your screen show should be similar to the following screen capture:



3. Double click each document in the Document Library

   The document is opened in a new tab window.
4. Switch between “Document 1” “Document 2” and “Document 3”

   The owner and status in the workflow group is updated automatically.
5. Change the text of the owner and status, click **Commit**.

   The modified content is updated into the Lotus Symphony document automatically.

Conclusion

This tutorial leads you through the creation of a Lotus Symphony side shelf extension. You learned how to create a plug-in, and plug-in extensions, and how to use the Java APIs supported by Lotus Symphony to read and update content of Lotus Symphony documents.

Appendix . Notices

Notices

The information contained in this publication is provided for informational purposes only. While efforts were made to verify the completeness and accuracy of the information contained in this publication, it is provided AS IS without warranty of any kind, express or implied. In addition, this information is based on IBM's current product plans and strategy, which are subject to change by IBM without notice. IBM shall not be responsible for any damages arising out of the use of, or otherwise related to, this publication or any other materials. Nothing contained in this publication is intended to, nor shall have the effect of, creating any warranties or representations from IBM or its suppliers or licensors, or altering the terms and conditions of the applicable license agreement governing the use of IBM software.

Copyright

Licensed Materials - Property of IBM

Lotus Software

IBM Software Group

One Rogers Street

Cambridge, MA 02142

Revision History: Original material produced for IBM Lotus Symphony Release Beta 4.

List of Trademarks

IBM, the IBM logo, AIX, DB2, Domino, iSeries, i5/OS, Lotus, Lotus Notes, LotusScript, Notes, Quickplace, Sametime, WebSphere, Workplace, z/OS, and zSeries are trademarks or registered trademarks of IBM Corporation in the United States, other countries, or both.

Additional IBM copyright information can be found at:

http://www.ibm.com/legal/copytrade.shtml

This information also refers to products built on Eclipse™ (http://www.eclipse.org)

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Intel and Pentium are trademarks of Intel Corporation in the United States, other countries, or both.

The Graphics Interchange Format© is the Copyright property of CompuServe Incorporated. GIF(sm) is a Service Mark property of CompuServe Incorporated.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product and service names may be trademarks or service marks of others.

Printed in USA