Lab: Introduction to rich-client development

.

Elena Lowery
ISV Business Strategy and Enablement
June 2006

Table of contents

Abstract and introduction **1**

Tools **1**

Skill prerequisites **1**

Lab setup **1**

Part 1: Reviewing SWT controls **2**

Checkpoint 1: 5

Part 2: Creating composites and shells with Visual Editor **6**

Checkpoint 2: 20

Part 3: Import pre-implemented classes and JAR files **21**

Checkpoint 3: 26

Part 4: Implementing basic navigation **27**

Checkpoint 4: 36

Part 5: Integrating business logic **37**

Checkpoint 5: 45

Trademarks and special notices **46**

Abstract and introduction

In this lab, you will create a stand-alone Standard Widget Toolkit (SWT) application for Eclipse 3.1 for System i programmers. You will learn how to create SWT controls, invoke business logic, and display results in a rich-client application.

Tools

This lab will use Eclipse 3.1, and not IBM WebSphere® Development Studio Client for IBM iSeries™, because Eclipse 3.1 contains significant enhancements for SWT and the Eclipse Rich Client Platform (RCP) application development. WebSphere Development Studio Client and IBM Rational® tooling (Version 6) are currently based on Eclipse 3.0.

Eclipse 3.1 is an open source (free-of-charge) integrated development environment (IDE). You can download Eclipse 3.1 from the Eclipse Web site, http://www.eclipse.org. You will use Visual Editor (VE), an open source plug-in for SWT, JFace and Eclipse RCP development, to build your SWT application. VE has to be installed into Eclipse 3.1 projects using the Eclipse Update feature. VE is packaged with IBM Rational and WebSphere Studio tooling (with no additional installation required).

Skill prerequisites

Prerequisite skills for this lab consist of an understanding of Java™ programming language concepts and beginner Java programming skills.

Lab setup

1. Extract the RichClientLab.zip file to your C:\ drive.
2. Create the flght400C library on your IBM System i™ model (notice no “i” in the word *flght*).
3. Implement a File Transfer Protocol (FTP) and restore the custdb.savf save file on your System i model by entering the following string:

```
RSTOBJ OBJ(*ALL) SAVLIB(FLGHT400C) DEV(*SAVF) SAVF(QGPL/CUSTDB)
MBROPT(*ALL) ALWOBJDIF(*ALL)
```

Part 1: Reviewing SWT controls

In this section, you will review SWT widgets. SWT widgets are building blocks of a rich-client application:

1. Open the Eclipse 3.1 IDE. Double-click the **Eclipse 3.1** icon on your desktop (see Figure 1).

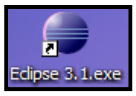

Figure 1. Eclipse 3.1 icon

2. In the **Workspace** field, type `C:/ITSO Lab` (see Figure 2).

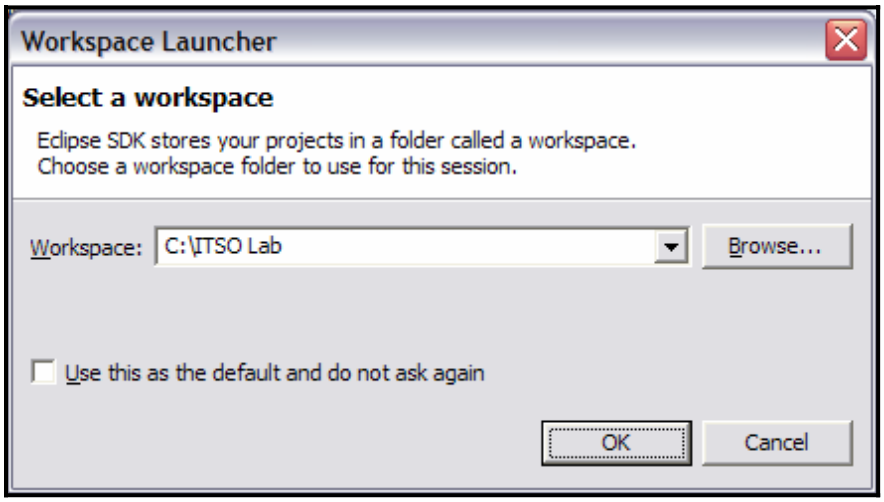

Figure 2. Select a workspace: C:\ITSO Lab

3. On the Welcome panel, click **Samples**. If the Welcome panel does not display, you can open it by clicking **Help -> Welcome** (see Figure 3).

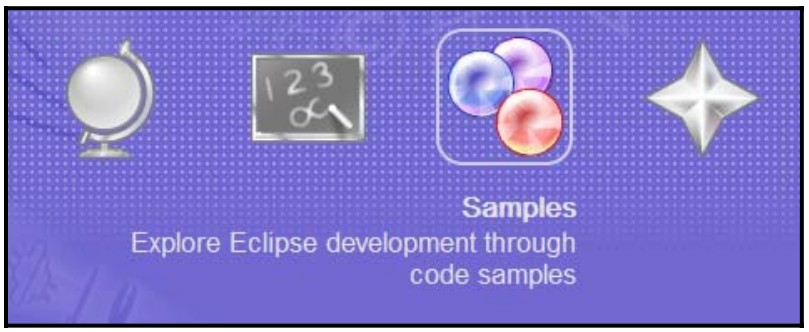

Figure 3. Samples selection

4. Under SWT, click he red round button (see Figure 4).

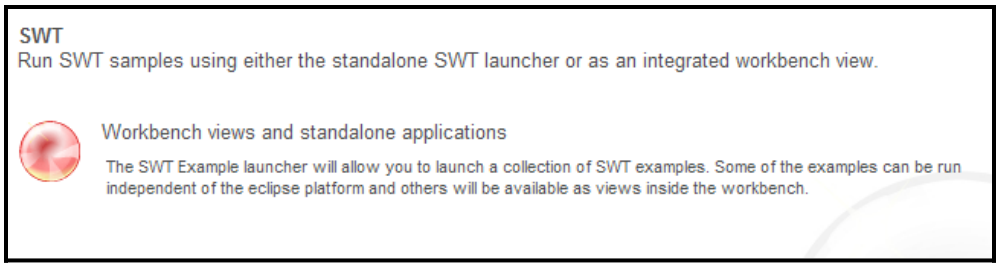

Figure 4. Workbench views and stand-alone applications

5. On the Eclipse Samples window, click **Finish** (see Figure 5).

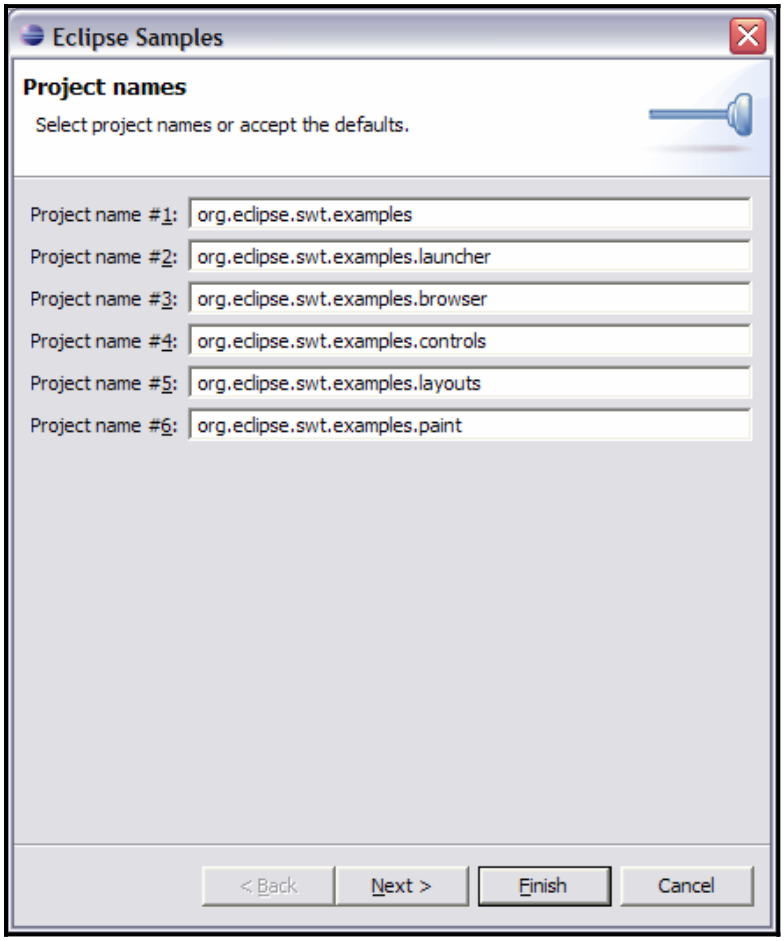

*Figure 5. Click **Finish***

6. Verify that you are in the **Java** development perspective. If not, switch to the **Java** development perspective by selecting **Window -> Open Perspective -> Java** (see Figure 6).

Figure 6. The Java development perspective

7. In the Package Explorer tab, right-click **org.eclipse.swt.examples**. Click **Run as -> SWT Application** (see Figure 7).

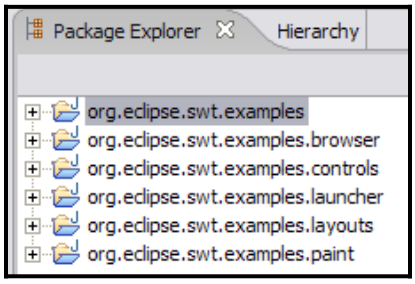

Figure 7: Package Explorer selections

8. On the Run Type dialog window, click **ControlExample** for the matching type and click **OK** (see Figure 8)**.**

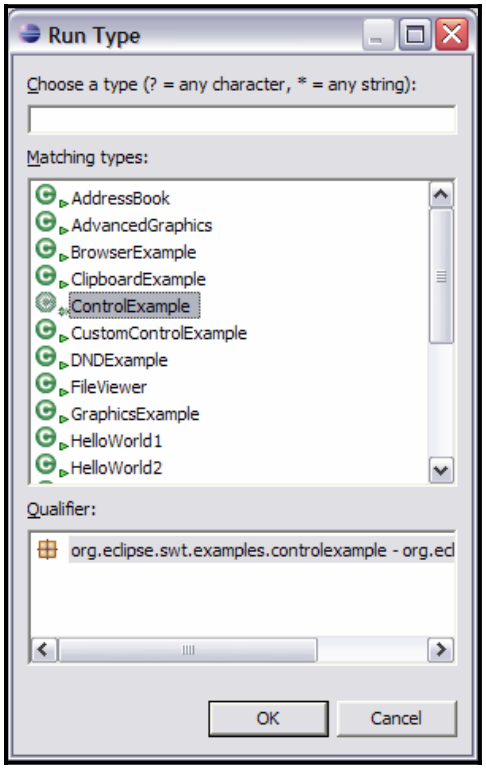

*Figure 8: **ControlExample** is the Run Type*

9. A sample SWT application is displayed in Figure 9.

   This application shows different types of SWT controls, their properties, and events. Try setting different properties for SWT controls and observe the change in their visual appearance. On the tabs that have **Select Listeners** enabled, observe when certain events are invoked on controls.

   If you are new to SWT programming, this application provides an excellent reference on how you can use SWT controls to construct a rich-client application.

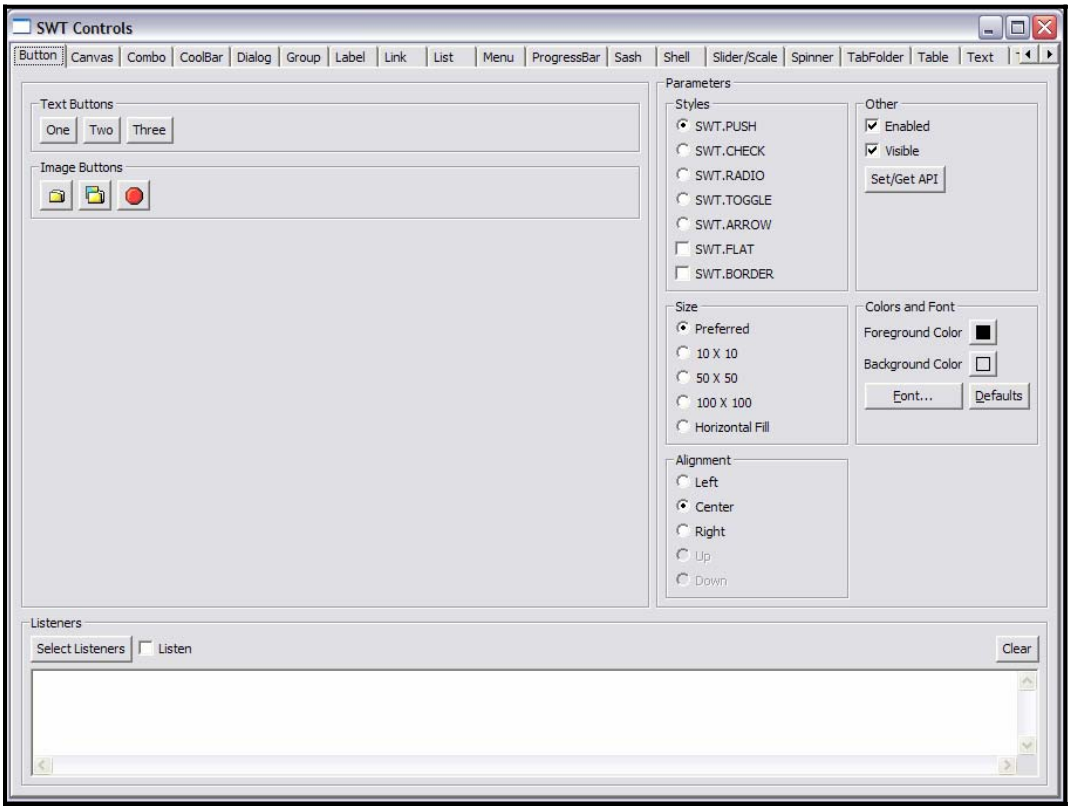

Figure 9. A sample SWT application

10. Close the sample application by clicking the x in the red box.

Checkpoint 1:

Take a few minutes to reflect on the things that you have learned in this section:

- List some of the most-commonly used SWT widgets.
- What is an SWT widget property? List some of the SWT widget properties.
- What is an SWT widget event? List some of SWT widget events.

Part 2: Creating composites and shells with Visual Editor

In this section, you will create an SWT application with Visual Editor. You will complete the following tasks (represented in Figure 10):

1. *Create composites*. A composite is a type of an SWT widget that contains other SWT widgets.
2. *Create SWT shells* and place composites on shells. Shells are application windows.
3. *Implement events for SWT widgets*. Events are user-driven actions that invoke some business logic. Examples of events are click, menu selection, and double-click.

The completed SWT application allows you to search for a customer name on the Customer Summary shell and perform view, add, update, and delete operations on the Customer Details shell.

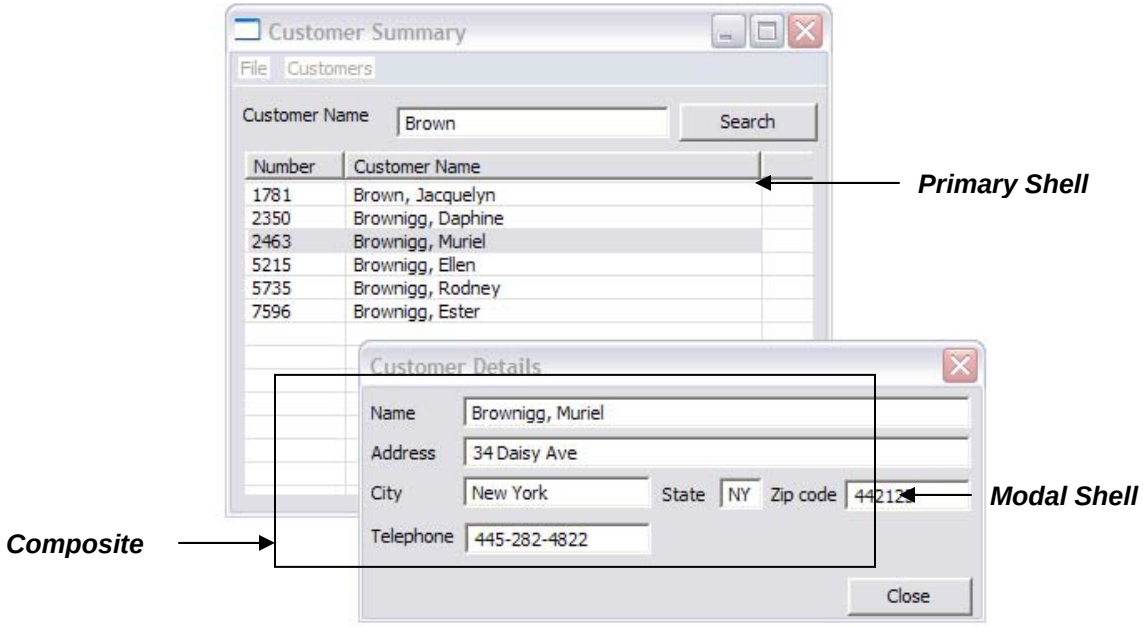

Figure 10. Customer Detail shell input

1. Select **File -> New -> Project**, then, on the **New Project** window, click **Java Project**. Click **Next** (see Figure 11).

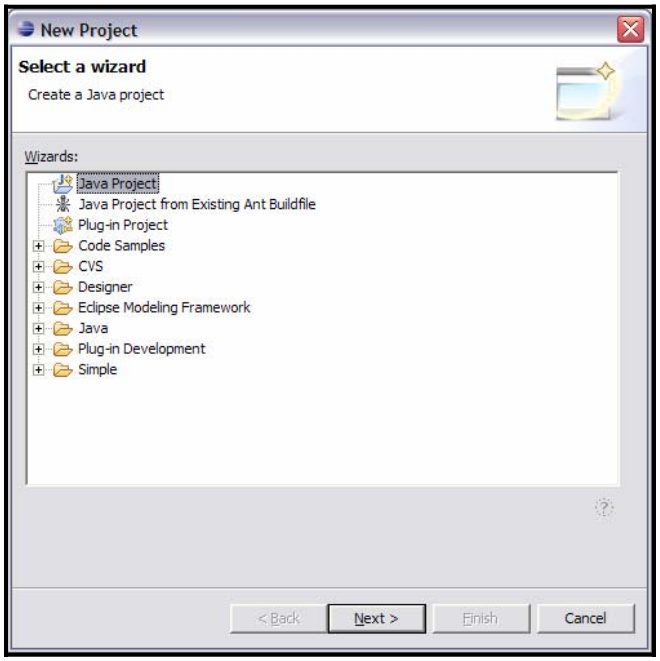

Figure 11. Java Project wizard

2. Type `Team`*`XX`*`SWTProject` as the project name, where *XX* is your team number. Click **Finish** (see Figure 12).

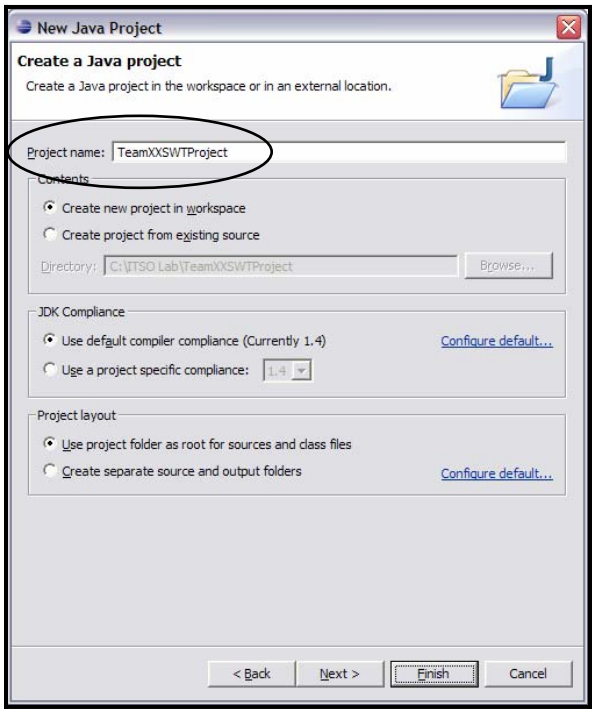

*Figure 12. Project name. **TeamXXSWTProject***

3. Create the Customer Summary composite:
   a. In Package Explorer, right-click the newly created Team*XX*SWTSample project and select **New - > Visual Class**.
   b. On the New Java Visual Class dialog, type or verify the following values (see Figure 13):
      - Source folder: **Team*XX*SWTProject**
      - Package: **com.ibm.swt.comp**
      - Name: **CustomerSummaryComp**
      - Style: **SWT -> Composite**
   c. Click **Finish**.

      Note: *SWT shell* is a stand-alone window in a rich-client application. *SWT composite* is a widget that contains other controls. Although you can place controls directly on a shell, you can achieve better reuse of code if you create composites and place them on different shells.

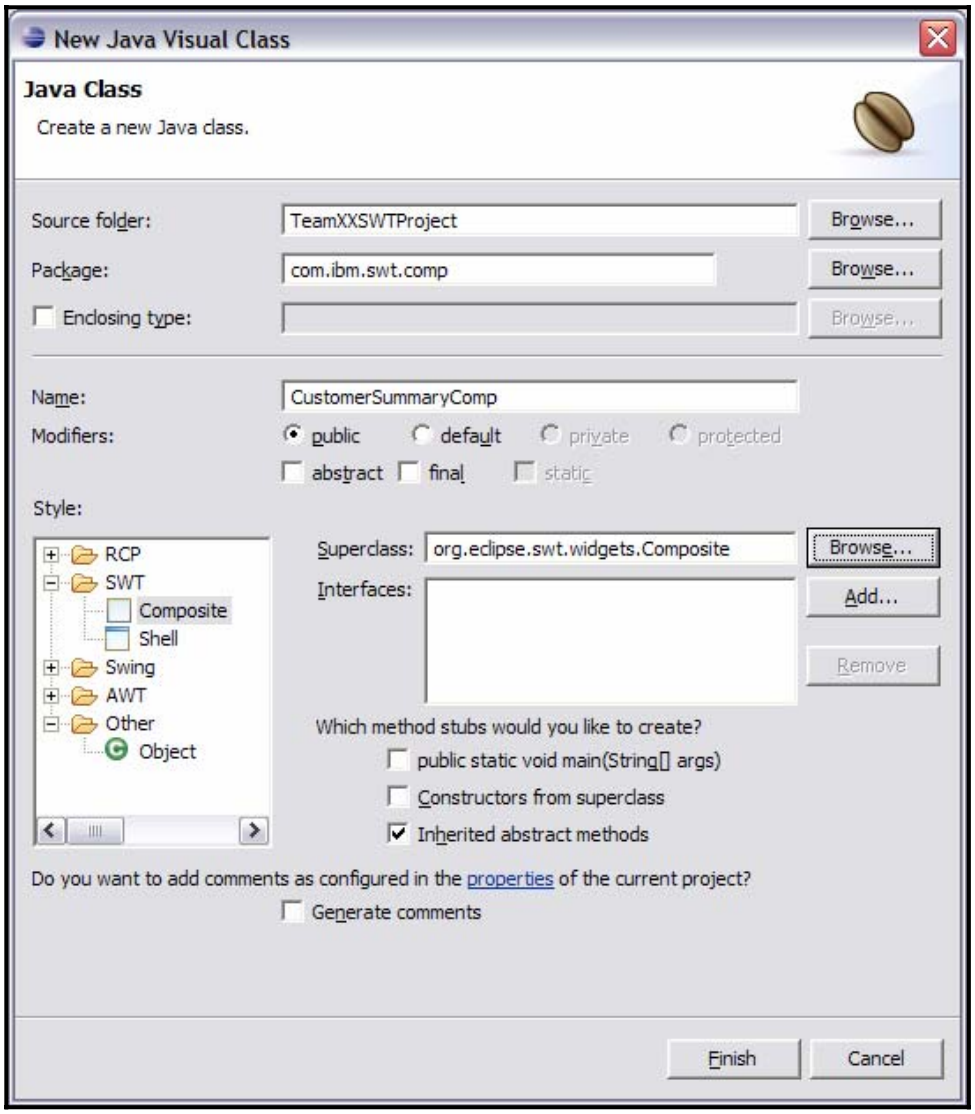

*Figure 13. Click **Finish** to finish creating a new Java class*

d. The CustomerSummaryComp.java class is displayed in the Visual Editor. Notice that there is a hidden Palette view on the right side of the editor area. Click the arrow in the top-right corner to display the Palette view. In the editor area, you can work in a *design view* or a *source view* (see Figure 14).

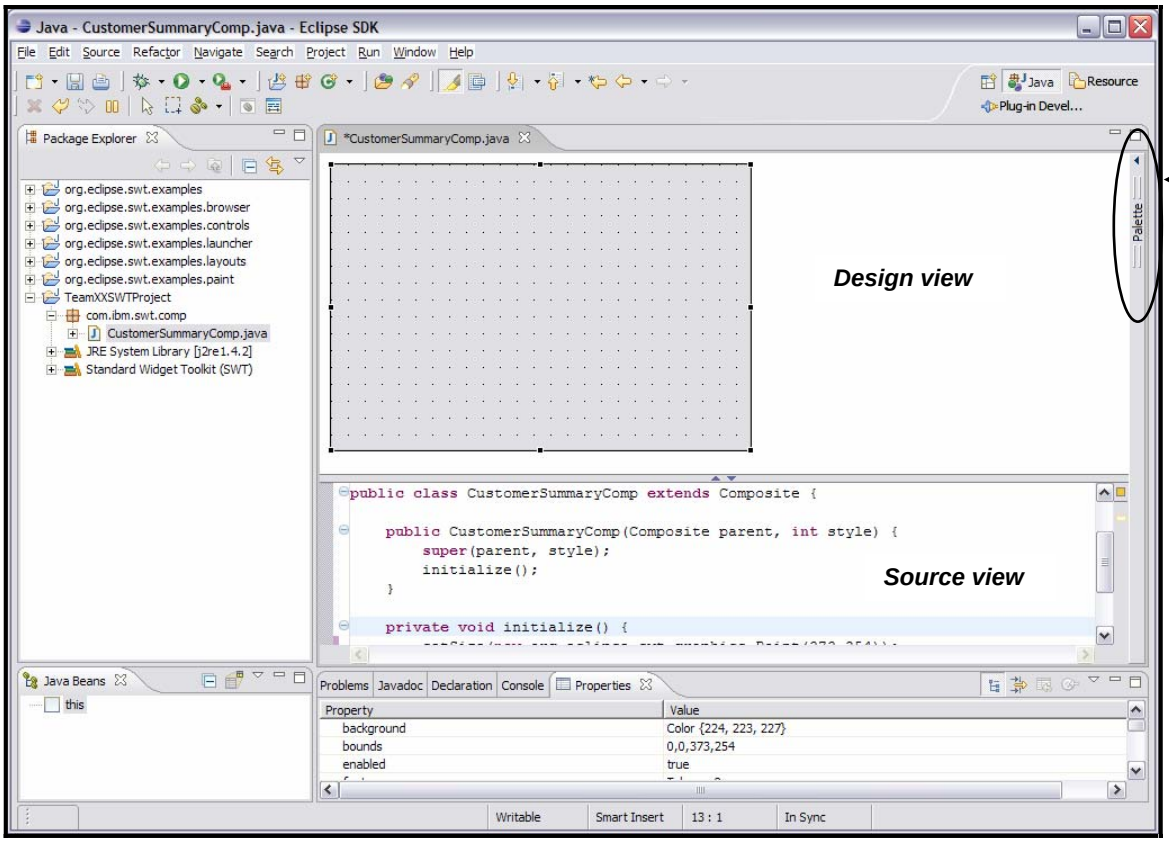

Figure 14. Palette, design, and source views

e. Move the following components (see Table 1) from the Palette view to the CustomerSummaryComp design view.

Type	Name	Text
Label	lblCustomerName	Customer Name
Text	txtCustomerName	
Button	btnSearch	Search
Table	tblCustomers	
TableColumn	colCustomerNumber	Number
TableColumn	colCustomerName	Customer Name

Table 1. Type, Name, and Text components from the Palette view

When placing a widget on the composite, you can use the Name window to type the name component (see Figure 15). In the Properties view, you can modify the Name, Text, and other properties. If the Properties view is not open, you can open it by selecting **Window -> Show View -> Properties** (see Figure 16).

Figure 15. Provide names for beans

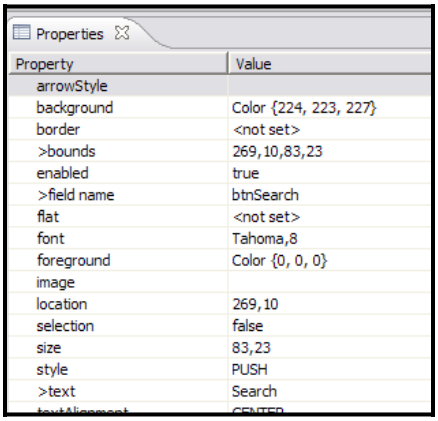

Figure 16. Property identification

When finished, your composite will look like Figure 17.

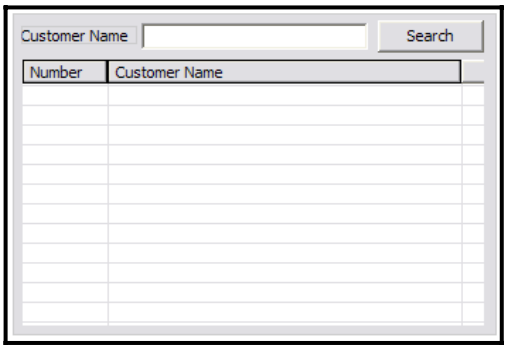

Figure 17. Panel of finished composite

f. Set the **fullSelection** property for the tblCustomers table to **FULL_SELECTION**. *Full selection* for a table means that an entire row (as compared to a cell) is selected in a table when a user clicks a table row (see Figure 18Figure 18).

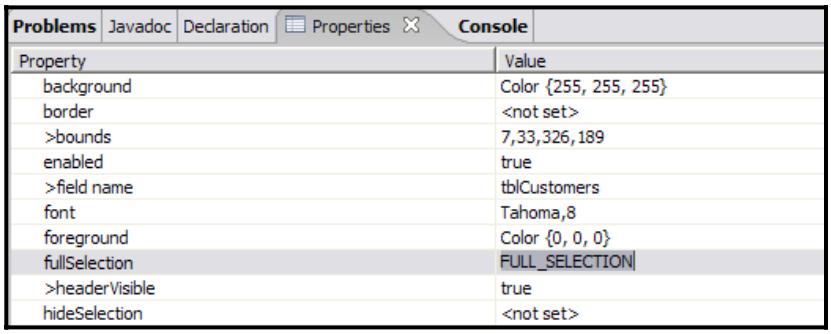

*Figure 18. Setting the **fullSelection** property*

g. Add a `getter` method to allow access to the Customers table from other SWT controls (for example, the SWT shell):

i. In the Source view, right-click any **tblCustomers** label and click **Source -> Generate Getters and Setters** (see **Error! Reference source not found.**).

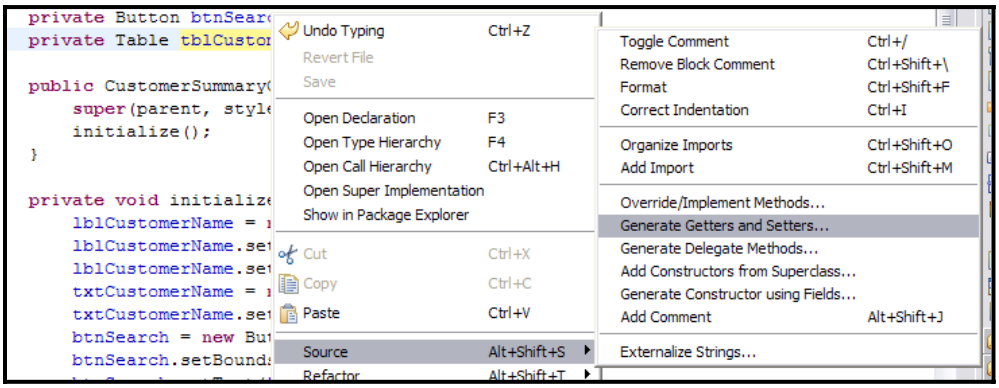

Figure 19. Source view requirement

ii. Click Deselect All, then on the Generate Getters and Setters window, select the getTblCustomers() check box. Click OK (see Figure 20).

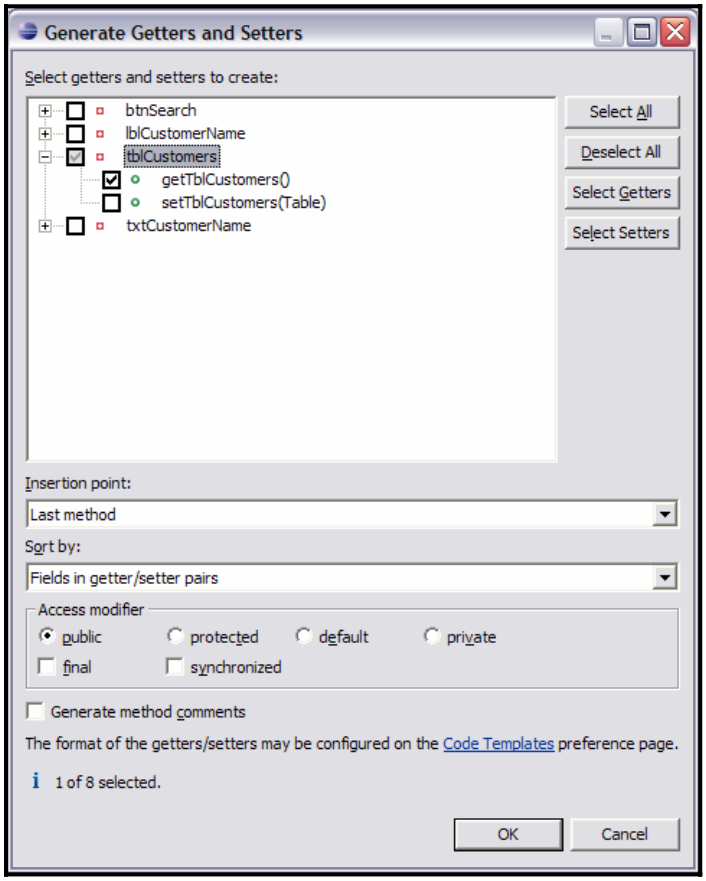

*Figure 20. Select **Generate method comments***

h. Save the changes to the CustomerSummaryComp.java class.

4. Import and review the Customer Details composite:
   a. In Package Explorer, right-click **Team*XX*SWTProject** and click **Import**.
   b. On the Import window, click **File system** (see Figure 21).

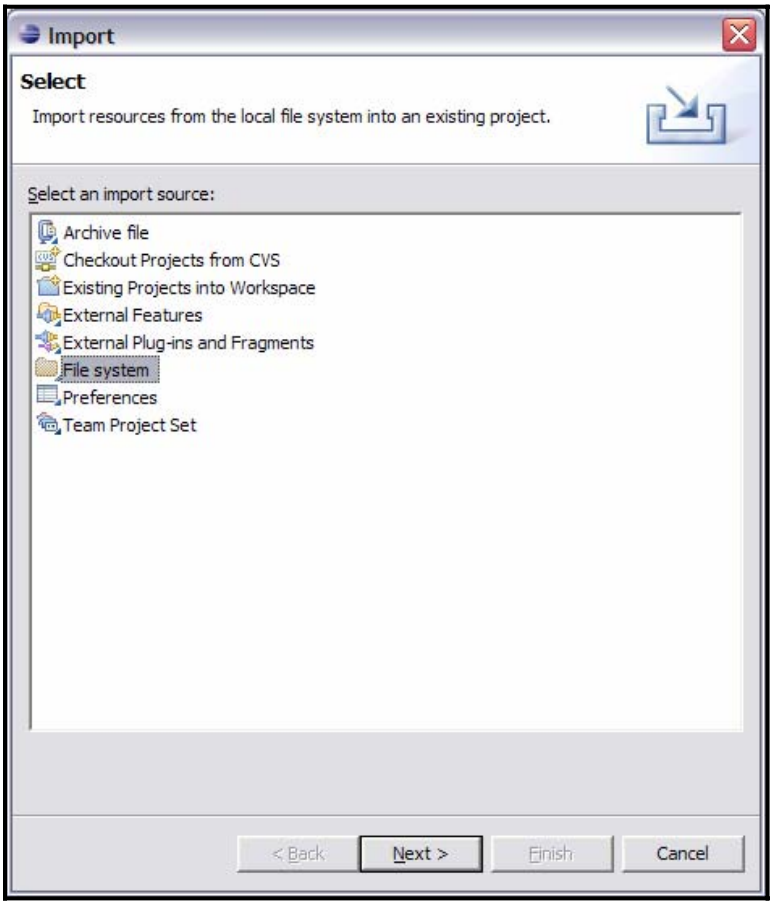

*Figure 21. Select **File system***

c. Navigate to the C:\Rich Client Lab\com\ibm\flight400\comp directory and click **CustomerDetailsComposite.java**. Click **Finish** (see Figure 22).

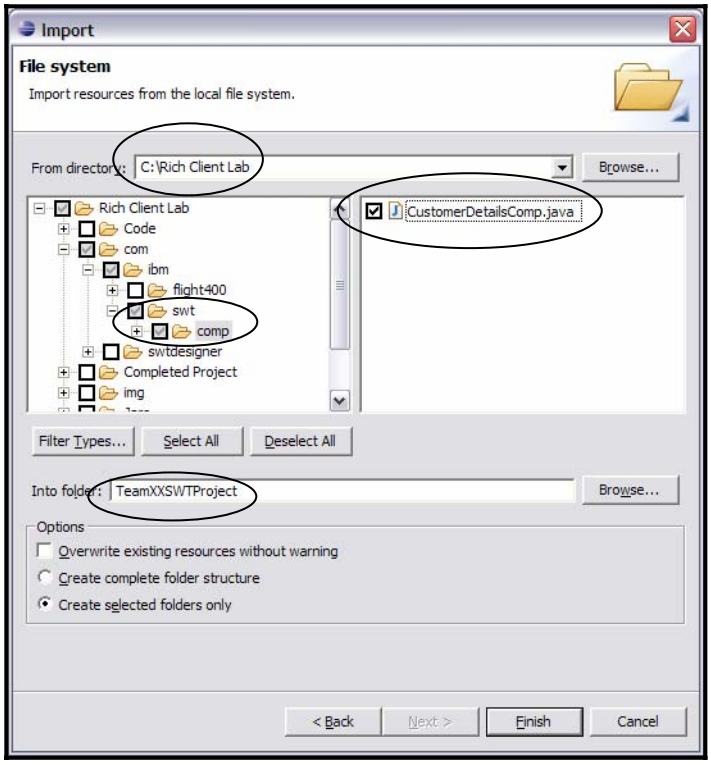

*Figure 22. Create selected folders and click **Finish***

d. Verify that the CustomerDetailsComp.java class was imported into the com.ibm.swt.comp package. Inform the instructor if you see any errors after the import (see Figure 23).

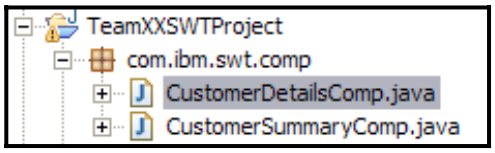

Figure 23. Ensure that CustomerDetailsComp.java has been imported into the com.ibm.swt.comp package

e. Review the Customer Details composite.

Notice that double-clicking **CustomerDetailsComp.java**, opens this class in a default Java editor. Right-click **CustomerDetailsComp.java** and select **Open With -> Visual Editor** (see Figure 24).

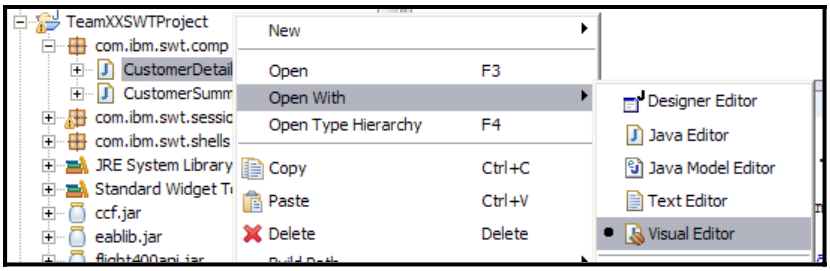

Figure 24. Opening the Visual Editor

Development of this composite is similar to the development of the Customer Summary composite; it consists of placing controls on the composite and modifying control properties (see Figure 25).

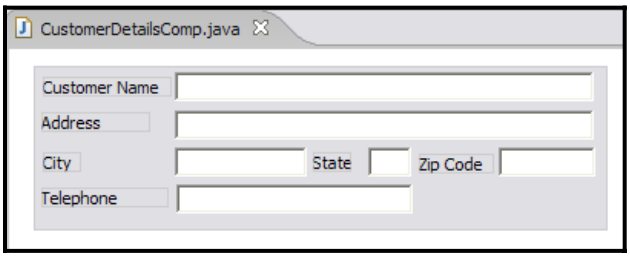

Figure 25. Development of the CustomerDetailsComp.java composite

5. Create a Customer Summary shell. A *Customer Summary shell* is the top-level application window that you use to display customer summary information:
   a. In Package Explorer, right-click **Team*XX*SWTProject** and click **New -> Visual Class**.
   b. On the New Java Visual Class dialog, type or verify the following values (see Figure 26):
      - Source folder: **Team*XX*SWTProject**
      - Package: **com.ibm.swt.shells**
      - Name: **CustomerSummaryShell**
      - Style: **SWT** -> **Shell**
      - For the **Which method stubs would you like to create?** value, select the **public static void main(…)** check box.

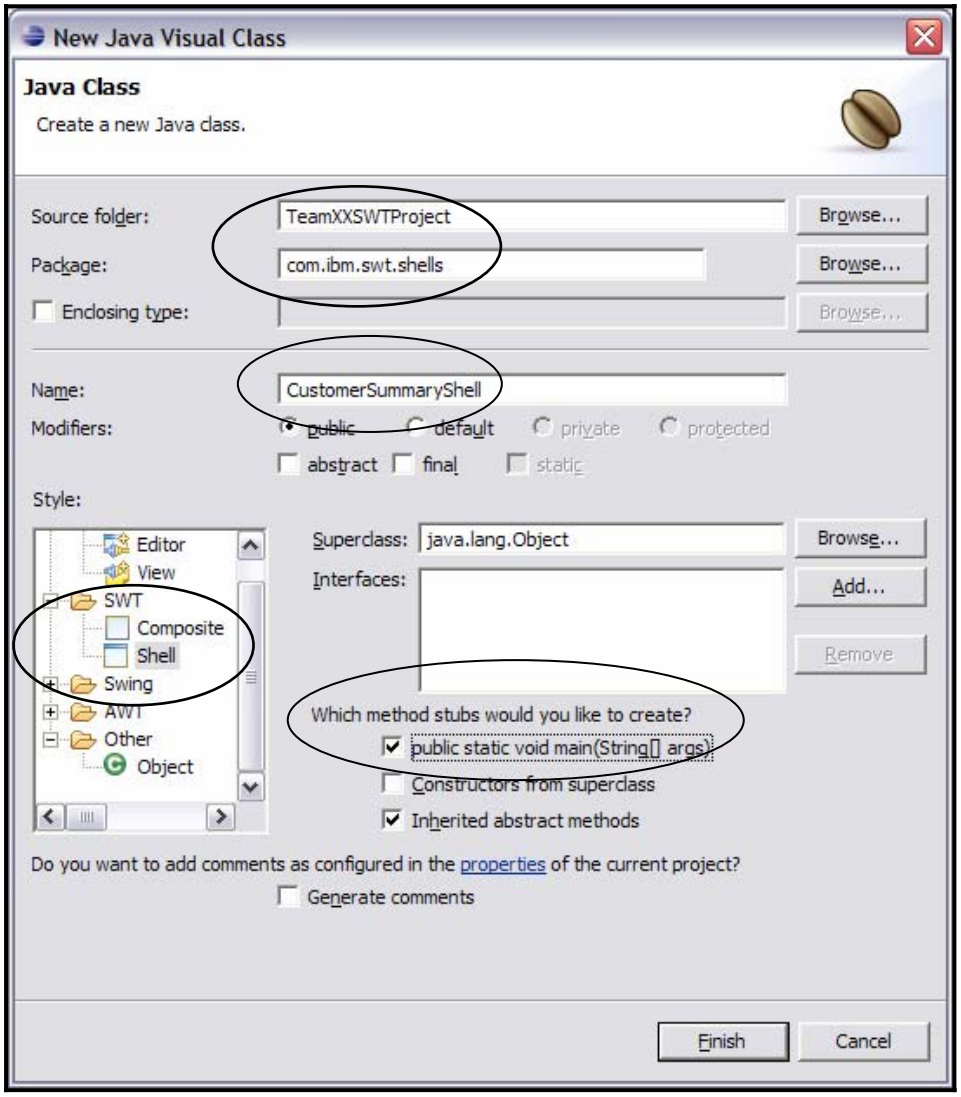

Figure 26. Create a new Java class

   c. Click **Finish**.

d. Add the Customer Summary composite to the Customer Summary shell:

i. Using the following code, declare **CustomerSummaryComp** as a private field immediately after the shell declaration:

```
private CustomerSummaryComp customerSummaryComp;
```

See Figure 27 for a display of the completed code.

```
public class CustomerSummaryShell {

    private Shell sShell = null;
    private CustomerSummaryComp customerSummaryComp;
```

Figure 27. Competed code - CustomerSummaryComp

ii. Add the following method in the CustomerSummaryShell.java class.

```
public void createContents(){
            customerSummaryComp = new CustomerSummaryComp
                  (sShell.getShell(), SWT.NONE);
      }
```

You can copy and paste the code in Figure 27 from the following text file:
C:\Rich Client Lab\Code\CustomerSummaryShell_CreateContents.txt

Note: Throughout this lab, you will reference Java classes that are not imported by default. You can easily add imports by clicking the **error** icon (Figure 28) on the left border of the Java source editor and selecting an option to add the required import. Alternatively, you can add an import manually.

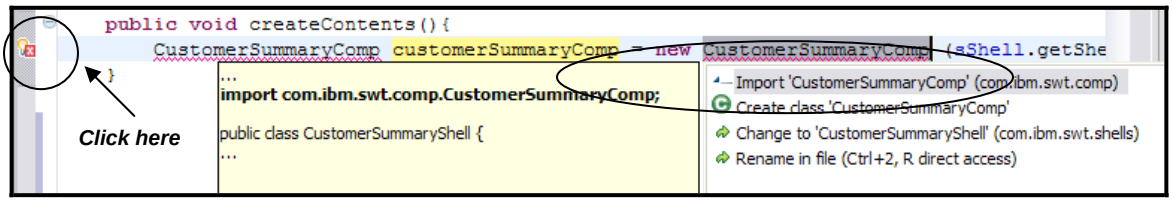

Figure 28. Adding imports

iii. After the last line of code in the createSShell() method, add a call to the createContents() method as follows:

```
private void createSShell() {
      sShell = new Shell();
      sShell.setText("Customer Summary");
      sShell.setSize(new org.eclipse.swt.graphics.Point(375,294));
      createContents();
}
```

e. Save the changes to the CustomerSummaryShell.java class. At this time, your Customer Summary shell will look like Figure 29.

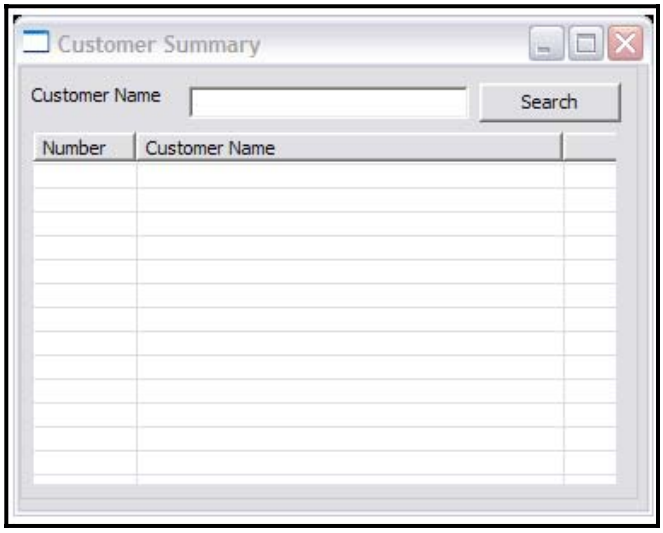

Figure 29. Example of the Customer Summary shell

6. Create a Customer Details shell. A Customer Details shell is displayed by a user-driven event (menu selection) from the Customer Summary shell.

   Note: In a typical stand-alone SWT application, you have one shell with a main() method (which is a starting point for any Java application). Actions on the primary application shell invoke all other shells.

   a. In Package Explorer, right–click **Team*XX*SWTProject** and select **New -> Visual Class**.
   b. On the New Java Visual Class dialog, type or verify the following values:
      - Source folder: **TeamXXSWTProject**
      - Package: **com.ibm.swt.shells**
      - Name: **CustomerDetailsShell**
      - Style: **SWT** => **Shell**
      - For the **Which method stubs would you like to create?** value, make sure the **public static void main(…)** check box is cleared.
   c. Click **Finish**.
   d. Add the Customer Details composite to the Customer Details shell:
      i. Using the following statement, declare **CustomerDetailsComp** as a private field immediately after the shell declaration:

```
private CustomerDetailsComp customerDetailsComp;
```

See Figure 30 for a display of the completed code.

```
public class CustomerDetailsShell {

    private Shell sShell = null;
    private CustomerDetailsComp customerDetailsComp;
```

Figure 30. Completed code - CustomerDetailsComp

ii. Add the following method in the CustomerDetailsShell.java class:

```
public void createContents(){
                customerDetailsComp = new
                CustomerDetailsComp(sShell,SWT.NONE);
        }
```

You can copy and paste the code listed in Figure 30 from the following text file:

C:\Rich Client Lab\Code\CustomerDetailsShell_CreateContents_method.txt

iii. After the last line of code in the createSShell() method that is contained in the CustomerDetailsShell.java class, add a call to the createContents() method as follows:

```
private void createSShell() {
      sShell = new Shell();
      sShell.setText("Customer Details");
      sShell.setSize(new org.eclipse.swt.graphics.Point(450,170));
      createContents();
}
```

e. Add a constructor to the CustomerDetailsShell.java class.

A *constructor* is a method in a Java class that initializes the class. You can add it immediately after the class variable declaration as follows:

```
public CustomerDetailsShell(){
      createSShell();
}
```

Figure 31 shows the completed code.

```
public class CustomerDetailsShell {

    private Shell sShell = null;
    private CustomerDetailsComp customerDetailsComp;

    public CustomerDetailsShell(){
        createSShell();
    }
```

Figure 31. Completed code – CustomerDetailsComp with createSShell() method

f. Make the following changes to change the type of shell that is being created in the createSShell() method to create a modal shell:

Change: `sShell = new Shell();`

To:

```
sShell = new Shell(SWT.DIALOG_TRIM + SWT.APPLICATION_MODAL);
```

g. Add the open() method to the CustomerDetailsShell.java class. This method allows other shells to open the Customer Details shell. You can add this method after the last method of the class as follows:

```
public void open(){
      sShell.open();
}
```

h. Switch to the design view of the CustomerDetailsShell.java class and under the **Customer Details** composite, add the two buttons listed in Table 2.

Type	Name	Text
Button	btnAction	
Button	btnCancel	Cancel

Table 2. Design view: Adding two buttons to the Customer Details composite

i. Save changes to the CustomerDetailsShell.java file. Figure 32 shows the Customer Details shell.

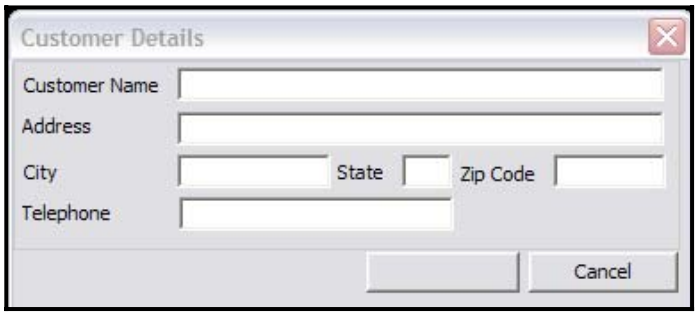

Figure 32. Example of Customer Details shell

Checkpoint 2:

Up to this point, you have created composites and shells. Take a few minutes to reflect on the things that you have learned in this section:

- What is the difference between a composite and a shell?
- Why is it a good idea to place widgets in a composite compared to directly onto a shell?
- How do you place a composite on a shell?
- How many shells with a main() method do you have in a SWT application?
- How can you specify the different types of an SWT shell?

Part 3: Import pre-implemented classes and JAR files

In this section, you will import some Java archive (JAR) files and classes that contain business logic. (IBM instructors created these files and classes in order to help you complete this lab.)

1. Import the JAR files that are required for business-logic implementation. You will review the content of JAR files in "Part 5: Integrate business logic."
   a. In Package Explorer, right–click **Team*XX*SWTProject** and click **Import**. On the Import panel, click **File system** (see Figure 33).

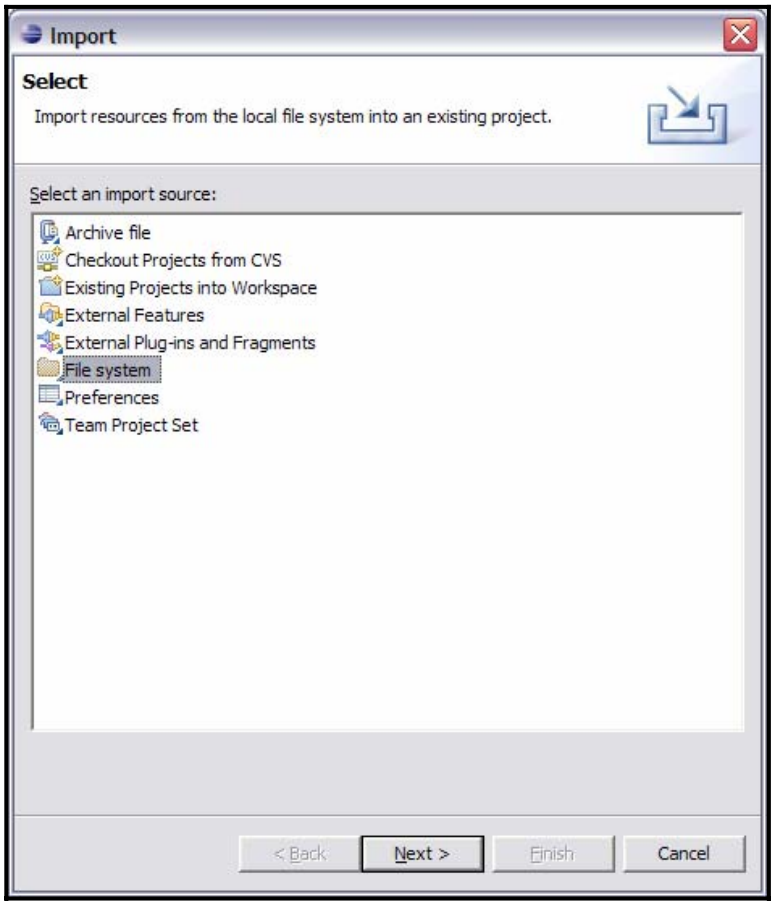

*Figure 33. On the Import panel, select **File system***

b. Navigate to the C:\Rich Client Lab\Jars directory and select all JAR files in this directory. Click **Finish** (see Figure 34).

Note: The application business logic is implemented in the flight400api.jar file. The rest of the JAR files contain Java classes that the business logic uses.

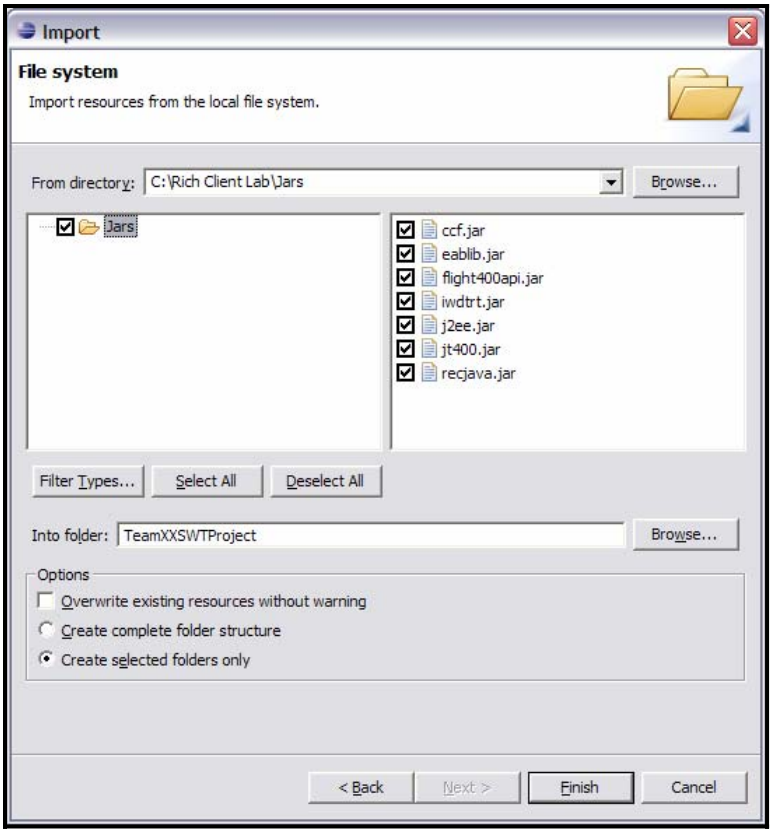

Figure 34. Selecting all JAR files in the directory

c. Add JAR files to Java Build Path (see Figure 35):

i. In Package Explorer, right-click **Team*XX*SWTProject** and click **Properties**.

ii. In the left pane, click **Java Build Path**.

iii. Click the **Libraries** tab and click **Add JARs**.

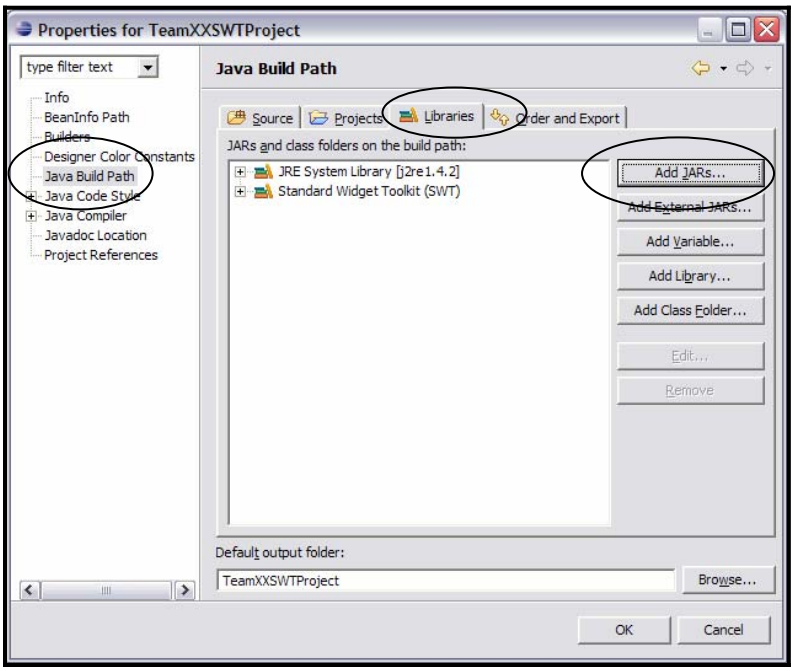

*Figure 35. Selecting properties for TeamXX*SWTProject*

iv. On the JAR Selection dialog, select all JAR files under **Team*XX*SWTProject** (hold the **Shift** key when selecting the JAR files). Click **OK** (see Figure 36).

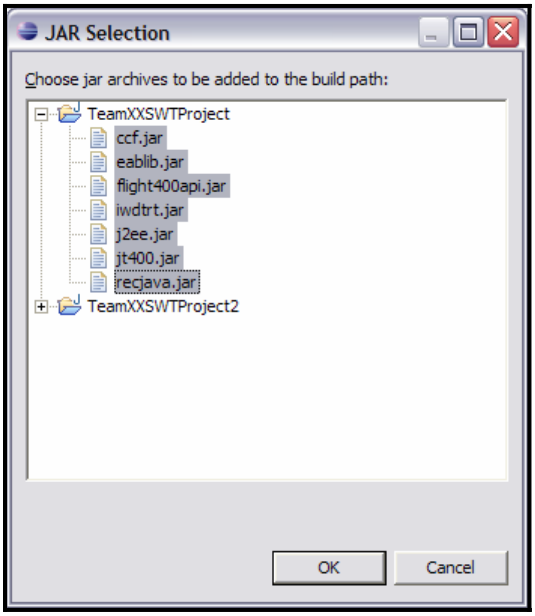

Figure 36. Choosing the JAR archives for the build path

v. Click **OK** to close the Properties window.

d. Repeat steps 1a and 1b to import the dbconn.properties file from the C:\Rich Client Lab\jdbc directory into Team*XX*SWTProject. This properties file contains database connection information.

e. Edit the `hostname`, `userid`, and `password` values in the dbconn.properties file.

When finished, your project will look similar to Figure 37.

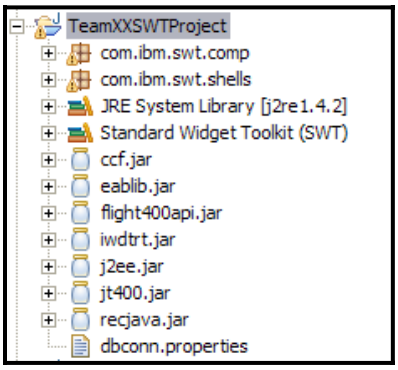

Figure 37. Project appearance

2. Import the Session Manager Java (SessionMgr.java) class. This class has been implemented to provide some basic session management functionality for this lab. Unlike Web applications, rich-client applications do not provide a default implementation for session management. A session in an application is a period of time during which a user interacts with an application. Conceptually, the use of a session object in a rich-client application is similar to the use of a temporary data area or a user space in an RPG application.
3. One of the primary uses of a session object is to pass information between SWT shells. Although it is possible to pass information between shells directly, using an intermediate object creates better architecture and allows for code reuse:
   a. In Package Explorer, right-click **Team*XX*SWTProject** and click **Import**.
   b. On the **Import** panel, click **File System** And click **Next**.

c. Navigate to the C:\Rich Client Lab\Session directory and select **Session**. Click **Finish** (see Figure 38).

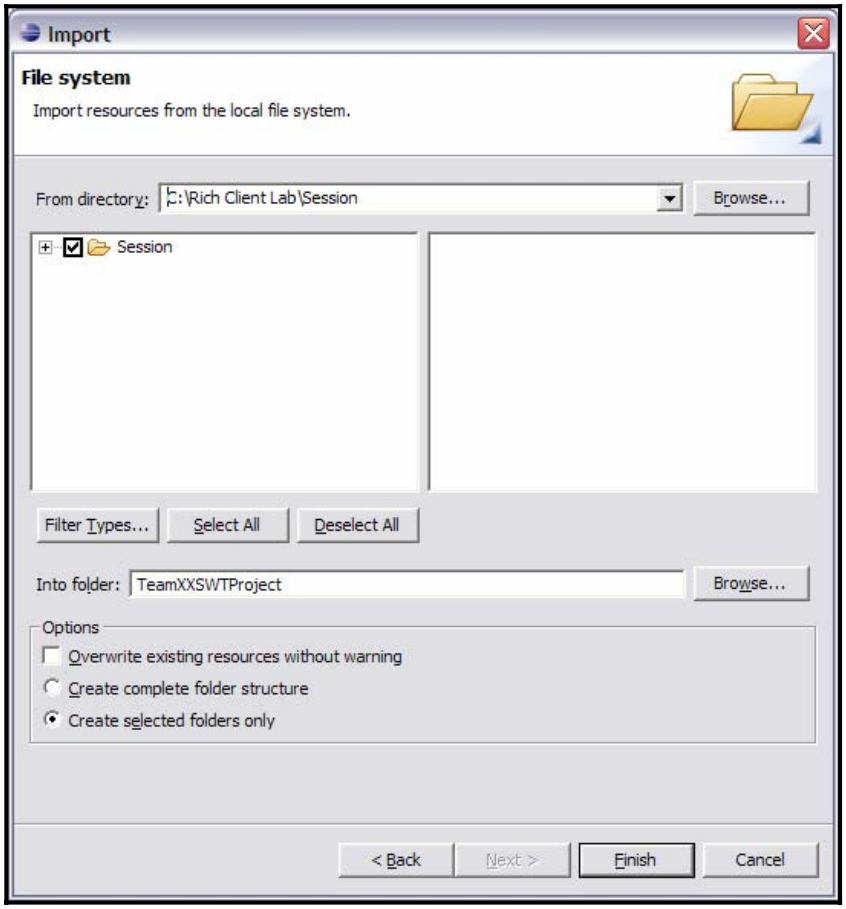

*Figure 38. Create selected folders and click **Finish***

4. Review the implementation of the SessionMgr.java class (located in Team*XX*SWTProject\com.ibm.swt.session). In a Java editor, double-click **SessionMgr.java** to open this class (see Figure 39).

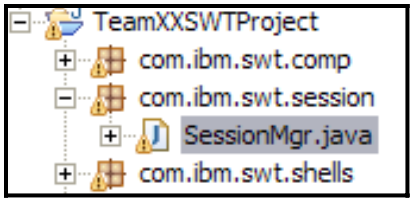

Figure 39. Opening the SessionMgr.java class

Notice that this class is a singleton. *Singleton implementation* means that there will be only one instance of this class in an application. This implementation is necessary to support the session functionality that was described earlier:

```
public static SessionMgr getInstance() {
   if(sessionMgr == null){
         sessionMgr = new SessionMgr();
   }
   return sessionMgr;
}
```

Here is a review some of the most interesting Java methods in this class:

- **getCurrentMode() and setCurrentMode():** This application uses the concept of a mode (view, add, update, or delete) to reuse SWT shells. At the same time, these methods support different behavior, depending on the selected mode. This getter/setter set allows the Customer Summary shell to set the mode, based on the selected menu. The Customer Details shell has a slightly different look and behavior, based on the selected mode.
- **getSelectedCustomer():** This method allows the application to pass information about the selected customer from the Customer Summary shell to the Customer Details shell.
- **setRefresh and isRefresh():** These methods tell the Customer Summary shell when to refresh search results.

Checkpoint 3:

Take a few minutes to reflect on the things that you have learned in this section:

- What is the role of a session object in an application?
- Do all rich-client applications need a session object?

Part 4: Implementing basic navigation

In this section, you will implement basic navigation between the Customer Summary and Customer Details shells.

1. Add a main menu bar to the Customer Summary shell:
   a. You can build a main menu by using Menu components in the Palette. Make sure to select components from the SWT Menus palette (see Figure 40).

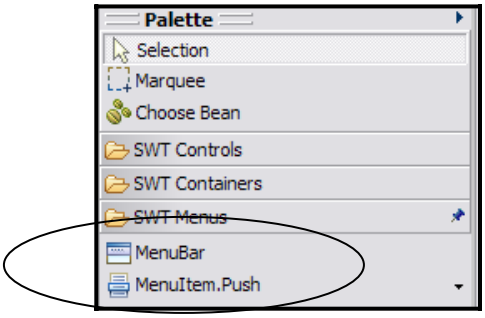

Figure 40. Adding a main menu bar to the Customer Summary shell

   b. Start by placing a **MenuBar** object onto the Customer Summary shell. Accept the default values in the JavaBeans name dialogs (see Figure 41).

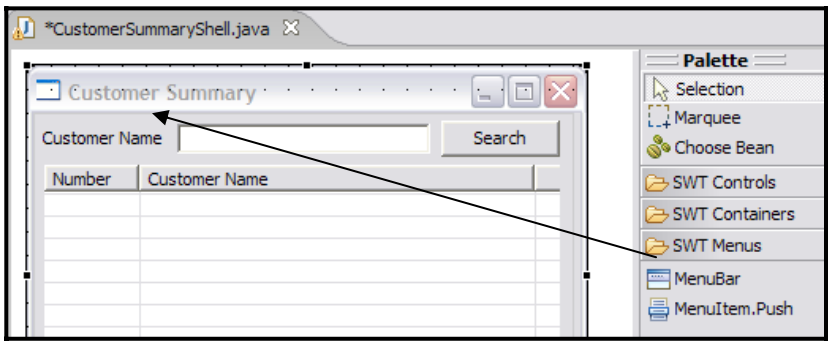

*Figure 41. Placing a **MenuBar** object onto the Customer Summary shell*

   c. Notice that the menu bar is displayed in the JavaBeans view (located in the left-bottom corner of the Workbench). To build the rest of the menu, drop menu components in the Java Beans view (see Figure 42).

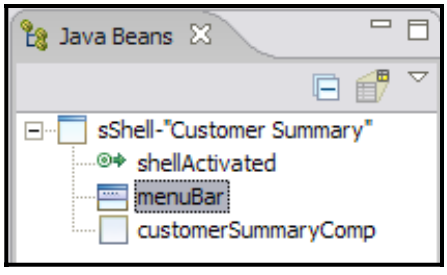

Figure 42. menuBar is displayed in the JavaBeans view

d. Drop two **SubMenu** objects under the **menuBar** object. You can change the type of menu by clicking the arrow in the Palette view (see Figure 43).

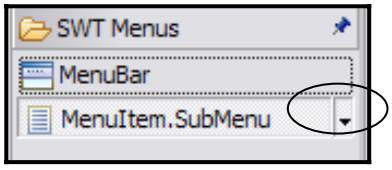

*Figure 43: Dropping **SubMenu** objects under the MenuBar*

At this time, your menu bar will look similar to Figure 44.

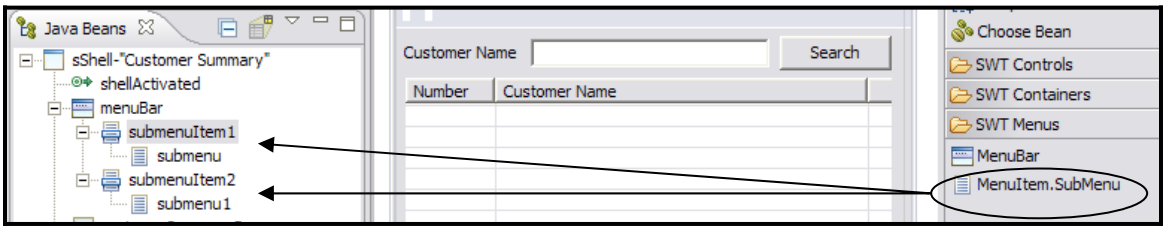

*Figure 44: **MenuBar** screen display*

e. Add one **push** menu object under the first **submenu** object and four **push** menu objects under the second **submenu** object (see Figure 45).

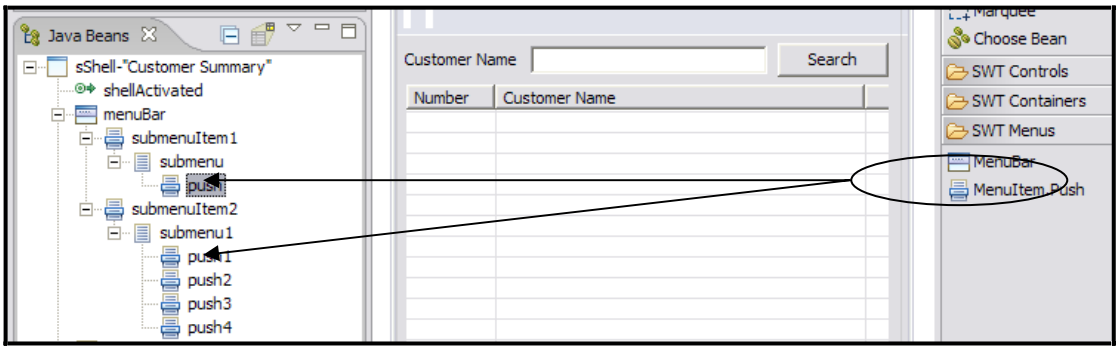

*Figure 45: Adding **push** menu objects*

f. In the Properties view, rename the **menu** objects by changing their field names and text properties (see Table 3).

Generated name	Field name	Text
submenuItem	mnuFile	File
push	mnuExit	Exit
submenuItem1	mnuCustomer	Customer
push1	mnuViewCustomer	View Customer
push2	mnuAddCustomer	Add Customer
push3	mnuUpdateCustomer	Update Customer
push4	mnuDeleteCustomer	Delete Customer

Table 3. menu object field names and text properties

g. When finished, your menu tree will look similar to Figure 46.

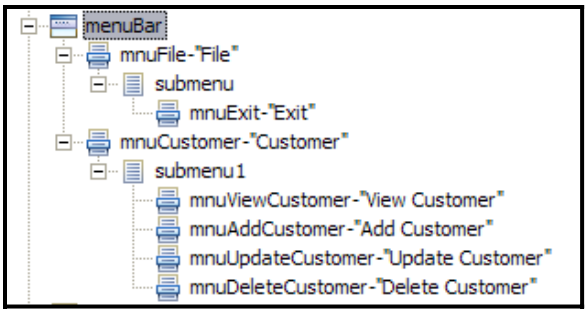

Figure 46: Menu tree display

2. Test the SWT application:
   a. In Package Explorer, right–click **Team*XX*SWTProject** and click **Run As -> SWT Application** (see Figure 47).

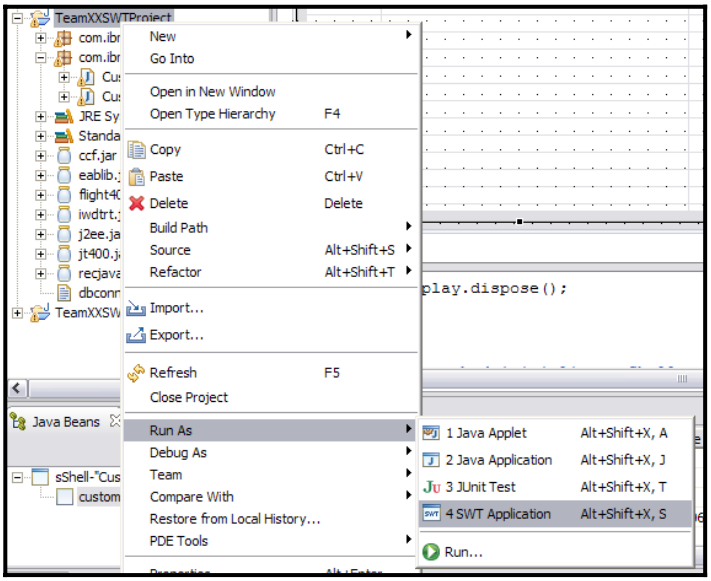

Figure 47. Testing the SWT application

b. In the Run Type dialog, type `cu` to display the CustomerSummaryShell matching type. Click **OK** (see Figure 48).

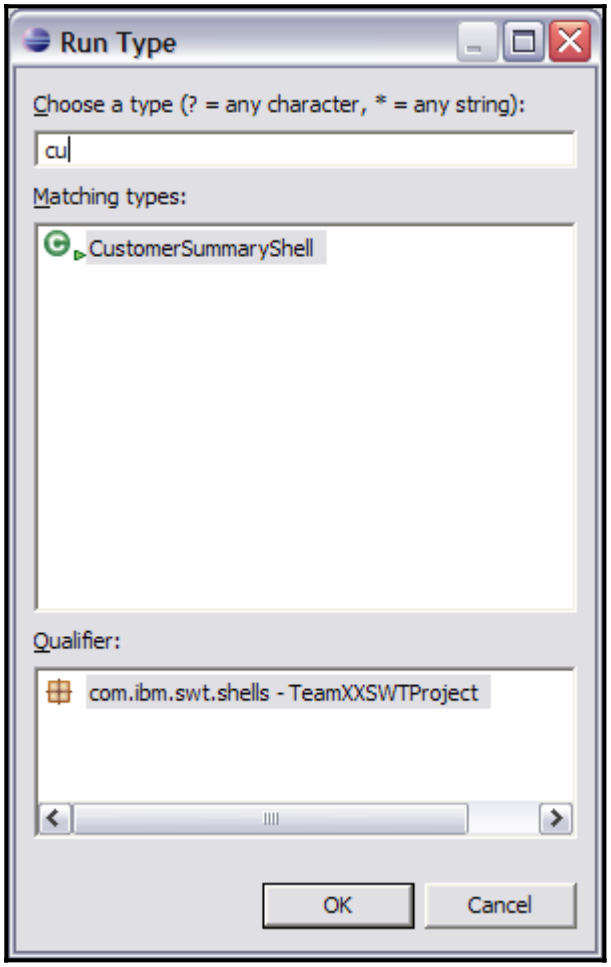

Figure 48. Displaying the CustomerSummaryShell matching type

c. Your application will look similar to Figure 49.

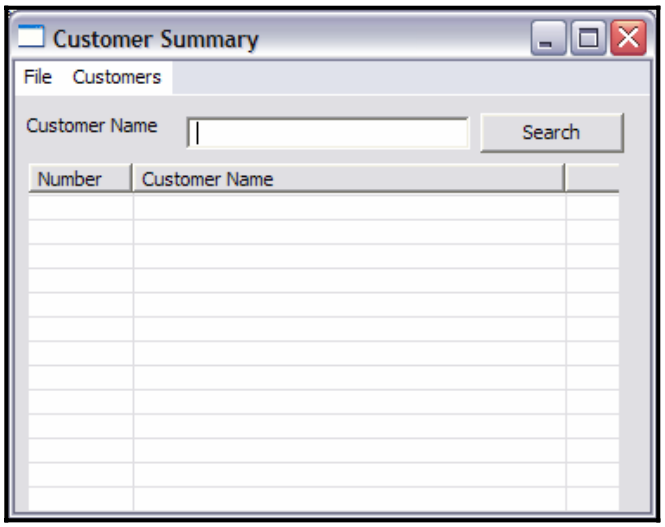

Figure 49. Customer Summary window

Notice that you can display the main and pop-up menus but nothing happens when you select the menus. Later in the lab, you will implement event handling for menu items and other SWT widgets. The only default actions that an SWT shell implements are the `minimize`, `maximize`, and `close` actions.

d. Close the Customer Summary shell.

3. Next, implement a basic operation of closing the application with the Exit menu:
   a. Open the Customer Summary shell in the Visual Editor.
   b. In the JavaBeans view, right-click **mnuExit** and click **Add Events** (see Figure 50).

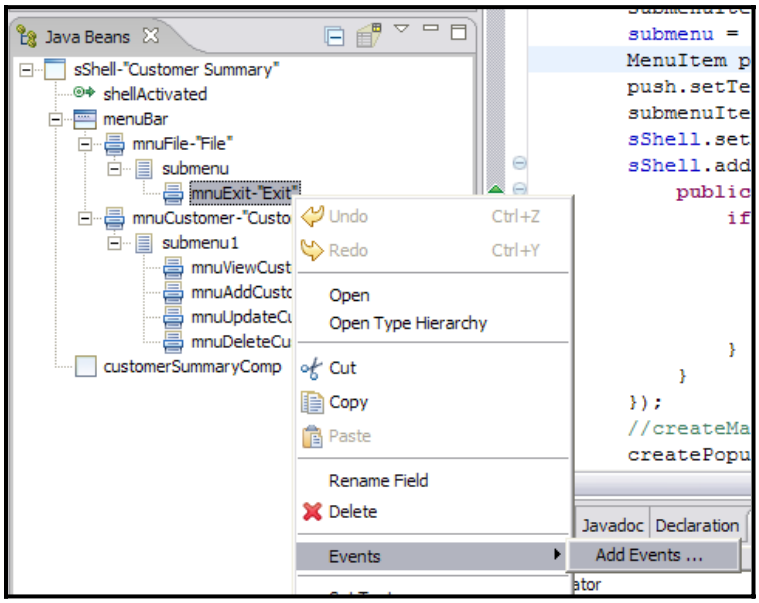

*Figure 50. **Add Events** in the JavaBeans view*

c. On the Add Event dialog, expand **selection**, click **widgetSelected,** and click **Finish** (see Figure 51).

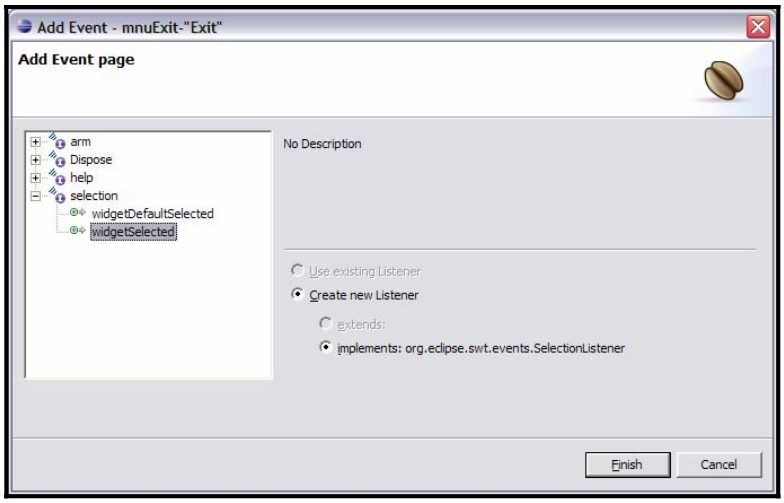

Figure 51. Selecting widgets

d. Review the generated code (see Figure 52). Notice that a selection listener has been added to the Exit menu. The role of a listener is to respond to a specific event, in this case, a selection event.

```
MenuItem push = new MenuItem(submenu, SWT.PUSH);
push.setText("Exit");
push.addSelectionListener(new org.eclipse.swt.events.SelectionListener() {
    public void widgetSelected(org.eclipse.swt.events.SelectionEvent e) {
        System.out.println("widgetSelected()"); // TODO Auto-generated Ever
    }
    public void widgetDefaultSelected(org.eclipse.swt.events.SelectionEvent
    }
});
```

Figure 52. Generated code for the Exit menu

Replace the generated code with the code that closes the shell.

Replace: `System.out.println("widgetSelected()");`

With: `sShell.close();`

Figure 53 shows completed code for the **Exit** menu.

```
MenuItem push = new MenuItem(submenu, SWT.PUSH);
push.setText("Exit");
push.addSelectionListener(new org.eclipse.swt.events.SelectionListener() {
    public void widgetSelected(org.eclipse.swt.events.SelectionEvent e) {
        sShell.close();
    }
    public void widgetDefaultSelected(org.eclipse.swt.events.SelectionEvent e) {
    }
});
```

*Figure 53. Completed code for the **Exit** menu*

e. Save the changes and test the application (try clicking the **Exit** menu).

4. Implement event handling for the View Customer, Add Customer, Update Customer, and Delete Customer menus:
   a. Add the displayCustomerDetails(int mode) method to the CustomerSummaryShell.java shell. You can copy and paste this code from the following text file:

      `C:\Rich Client Lab\Code\DisplayCustomerDetails.txt`

      This method creates an instance of the CustomerDetailsShell panel and opens it. The `mode` parameter determined if the details panel is used for display, add, update, and delete operations.

```
public void displayCustomerDetails(int mode){

   SessionMgr sessionMgr = SessionMgr.getInstance();
   sessionMgr.setCurrentMode(mode);

   if(mode == SessionMgr.MODE_VIEW ||
      mode == SessionMgr.MODE_UPDATE ||
      mode == SessionMgr.MODE_DELETE) {
   // Get selected row
   sessionMgr.setSelectedCustomerIndex(customerSummaryComp.
               getTblCustomers().getSelectionIndex());
}

CustomerDetailsShell custDetailShell = new CustomerDetailsShell();

custDetailShell.open();

}
```

   b. Add selection listeners and a call to the displayCustomerDetails() method to each customer-related menu. Use the Add Events menu that you used in the previous step. For each menu, replace the default code with:

      View menu: `displayCustomerDetails(SessionMgr.MODE_VIEW);`

      Add menu: `displayCustomerDetails(SessionMgr.MODE_ADD);`

      Update menu: `displayCustomerDetails(SessionMgr.MODE_UPDATE);`

      Delete menu: `displayCustomerDetails(SessionMgr.MODE_DELETE);`

Figure 54 shows a sample set of code for the customer menus.

```
MenuItem push1 = new MenuItem(submenu1, SWT.PUSH);
push1.setText("View Customer");
push1.addSelectionListener(new org.eclipse.swt.events.SelectionListener() {
    public void widgetSelected(org.eclipse.swt.events.SelectionEvent e) {
        displayCustomerDetails(SessionMgr.MODE_VIEW);
    }
    public void widgetDefaultSelected(org.eclipse.swt.events.SelectionEvent e) {
    }
});
MenuItem push2 = new MenuItem(submenu1, SWT.PUSH);
push2.setText("Add Customer");
push2.addSelectionListener(new org.eclipse.swt.events.SelectionListener() {
    public void widgetSelected(org.eclipse.swt.events.SelectionEvent e) {
        displayCustomerDetails(SessionMgr.MODE_ADD);
    }
    public void widgetDefaultSelected(org.eclipse.swt.events.SelectionEvent e) {
    }
});
```

Figure 54. Sample code for the customer menus

Although the JFace API is out of the scope of this lab, it is important to note that it is possible to minimize the amount of repeating code for action processing with the JFace Action Processing API. The JFace API allows you to set up one action-processing method for multiple events (main menu, pop-up menu, and toolbar buttons). In this case, if you create a JFace action-processing class, you do not have to add event listeners to each menu.

c. Test the application. Each customer-related menu item brings up a Customer Details panel. Notice that the Customer Details shell is modal.

5. Add a pop-up menu to the Customers table.

   Note: At this time, Visual Editor does not provide a visual wizard for creating a pop-up menu. You will create the pop-up menu manually.

   a. Add the createPopupMenu() method to CustomerSummaryShell.java class. You can copy and paste this code from following text file:

      C:\Rich Client Lab\Code\CustomerSummaryPopupMenu.txt

      In this method, you will create a Customer pop-up menu with submenus: View Details, Add Customer, Update Customer, and Delete Customer. This pop-up menu will be displayed when the user right-clicks the Customers table. Examine the following code:

```
private void createPopupMenu(){

   final Menu menu_3 = new Menu(customerSummaryComp);
   customerSummaryComp.getTblCustomers().setMenu(menu_3);

   final MenuItem mntmViewDetails_1 = new MenuItem(menu_3, SWT.NONE);

   mntmViewDetails_1.setText("View Details");
   mntmViewDetails_1.addSelectionListener(new SelectionAdapter() {
      public void widgetSelected(final SelectionEvent e) {
         SessionMgr.getInstance().setCurrentMode(SessionMgr.MODE_VIEW);
         displayCustomerDetails(SessionMgr.MODE_VIEW);
      }
   });

   new MenuItem(menu_3, SWT.SEPARATOR);
   final MenuItem mntmAddCustomer_1 = new MenuItem(menu_3, SWT.NONE);
   mntmAddCustomer_1.setText("Add Customer");
   mntmAddCustomer_1.addSelectionListener(new SelectionAdapter() {
      public void widgetSelected(final SelectionEvent e) {
         SessionMgr.getInstance().setCurrentMode(SessionMgr.MODE_ADD);
         displayCustomerDetails(SessionMgr.MODE_ADD);
      }
   });

   final MenuItem mntmUpdateCustomer_1 = new MenuItem(menu_3, SWT.NONE);
   mntmUpdateCustomer_1.setText("Update Customer");
   mntmUpdateCustomer_1.addSelectionListener(new SelectionAdapter() {
      public void widgetSelected(final SelectionEvent e) {
         SessionMgr.getInstance().setCurrentMode(SessionMgr.MODE_UPDATE);
         displayCustomerDetails(SessionMgr.MODE_UPDATE);
      }
   });

   final MenuItem mntmDeleteCustomer_1 = new MenuItem(menu_3, SWT.NONE);
   mntmDeleteCustomer_1.setText("Delete Customer");
   mntmDeleteCustomer_1.addSelectionListener(new SelectionAdapter() {
      public void widgetSelected(final SelectionEvent e) {
         SessionMgr.getInstance().setCurrentMode(SessionMgr.MODE_DELETE);
         displayCustomerDetails(SessionMgr.MODE_DELETE);
      }
   });
```

b. After the last line of code in the createSShell() method, add a call to the createPopupMenu() method. Figure 55 is a display of the completed code.

```
submenuItem.setMenu(submenu);
sShell.setMenuBar(menuBar);
createPopupMenu();
```

Figure 55. Sample of completed code for a pop-up menu

Checkpoint 4:

In this section, you added some basic navigation in a SWT application. Take a few minutes to reflect on the things that you have learned:

- What is an event?
- What is the role of a listener?
- How did you implement event processing for SWT widgets?
- Does SWT implement any default event processing?

Part 5: Integrating business logic

Business logic is usually invoked from GUI events that the end user initiates. Creating business logic is outside the scope of this lab. You will use the business logic that was previously created for you.

Business logic in this application performs the following functions:

1. Search for customers based on the customer name provided by the user
2. Display detailed customer information
3. Add a customer
4. Update a customer
5. Delete a customer

You implemented the business logic using ProgramCall JavaBeans (JavaBeans that invoke an RPG program) and JDBC JavaBeans (JavaBeans that run Structured Query Language [SQL] statements against an IBM DB2® for i5/OS® database).

As a GUI layer programmer, you do not need to know the details of the business-logic layer of the implementation. Your goal is to provide input parameters that the business logic methods require and display results that the business logic returns.

Separating the presentation and business logic is something that all programmers know they are required to do, but during implementation you might have a tendency to do what is faster and more convenient. You can use some techniques to move programmers into a good programming model. For example, you can develop business logic in a separate project from the presentation logic (in IBM WebSphere Development Studio Client), package it into a JAR file, and import it into the presentation layer project. By using this process, programmers can call business logic from the presentation layer, but they will be less likely to add business logic to the presentation layer.

1. Add business logic to search for customers:
   a. Add the getCustomers() and refresh() methods to the CustomerSummaryComp.java class:
      Note: To add these two methods, you can copy and paste the code listed on the next page from the following text file:
      C:\Rich Client Lab\Code\SummaryCompositeActions.txt
      - The getCustomers() method calls another method that is packaged in one of the Java classes that are included in the flight400api.jar JAR file and populates the table on the composite with the returned results.
      - The refresh() method refreshes the search results after returning from add, update, and delete operations on the Customer Details shell.

```
private void getCustomers(){

      if (txtCustomerName.getText().trim() != "") {

            Customer customer = null;
            int i = 0;

            try {
                  // Call business logic method to retrieve customers
                  Vector customers =
                        iSeriesDataManager.getInstance().
                        getCustomerByName(txtCustomerName.
                        getText());
                  if (customers != null) {
                        Iterator iter = customers.iterator();
                        // Remove previous records
                        tblCustomers.removeAll();
                        // Create an array of table rows
                        final TableItem[] items = new
                                          TableItem[customers.size()];

                        while (iter.hasNext()) {
                              customer = (Customer) iter.next();
                              items[i] = new TableItem(tblCustomers,
                                                SWT.NONE);
                              items[i].setText(new String[] {
                              customer.getCustomerNumber().toString(),
                              customer.getCustomerName() });
                              i++;
                        }

                        // Save results in the "session"
                  SessionMgr.getInstance().setSearchResults(customers);
                  }

            } catch (Exception ex) {
                  ex.printStackTrace();
            }
      }else{
            // Display a message box if the customer name is not provided
            MessageBox validMsg = new MessageBox(this.getShell(), SWT.OK);
            validMsg.setMessage("Please enter a customer name.");
            validMsg.open();
      }
}

public void refresh(){
      getCustomers();
}}
```

b. Add an event handling process for the Search button.

i. You need to call the getCustomers() method when the user clicks the Search button. The click action on the button corresponds to the SWT widgetSelected event. On the **Design** view, right-click **Search** and click **Events -> widgetSelected** (see Figure 56).

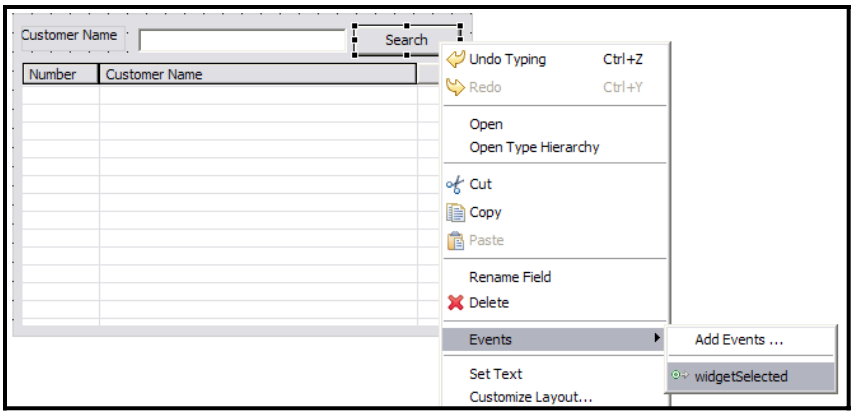

*Figure 56:Select **widgetSelected***

ii. The View Editor brings you to the widgetSelected event code in the Source Code view.

Replace the following code: `System.out.println("widgetSelected()")`

with: `getCustomers()`

(See Figure 57. The widgetSelected event code.)

```
btnSearch.addSelectionListener(new org.eclipse.swt.events.SelectionAdapter() {
    public void widgetSelected(org.eclipse.swt.events.SelectionEvent e) {
        getCustomers();
    }
});
```

Figure 57. The widgetSelected event code

c. Add the **s**hellActivated() event to the CustomerSummaryShell.java class. You need to refresh the search results after returning from add, update, and delete operations on the Customer Details shell.

i. Switch to design mode in the Customer Summary shell editor. Right-click the shell and click **Events -> shellActivated** (see Figure 58).

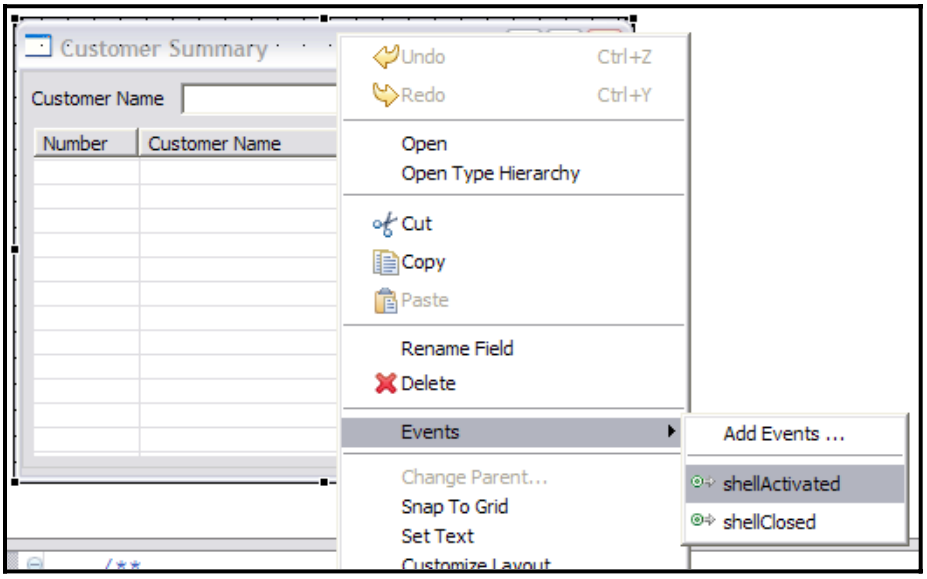

*Figure 58. Select **shellActivated***

ii. Replace the generated code with the following code:

```
if(SessionMgr.getInstance().isRefresh()){
      // If there was a previous search result
      if(SessionMgr.getInstance().getSearchResults() != null){
            customerSummaryComp.refresh();
      }
}
```

Note: You can copy and paste this code from the following text file:

C:\Rich Client Lab\Code\CustomerSummaryShell_ShellActivated.txt

You can see an example of the completed code in Figure 59:

```
sShell.addShellListener(new org.eclipse.swt.events.ShellAdapter() {
    public void shellActivated(org.eclipse.swt.events.ShellEvent e) {
        if(SessionMgr.getInstance().isRefresh()){
            // If there was a previous search result
            if(SessionMgr.getInstance().getSearchResults() != null){
                customerSummaryComp.refresh();
            }
        }
    }
```

Figure 59. The completed shellActivated code

d. Save the changes to the CustomerSummaryComp.java class and test your Team*XX*SWTProject:

iii. Type `Brown` for the customer name and click **Search**. You will see results similar to Figure 60.

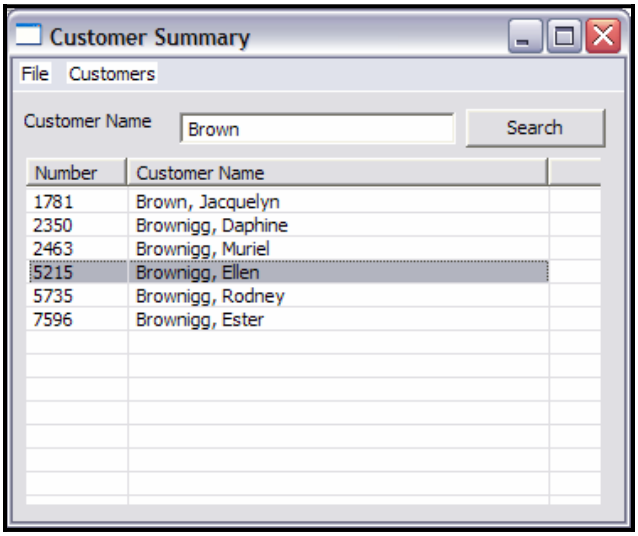

Figure 60. Customer name: Brown

iv. Leave the customer name blank and **Search**. You will see a validation message box (see Figure 61).

Figure 61. Validation message box

2. Add business logic to the Customer Detail**s** shell.

   Note: This step allows you to do some manual coding. If you do not want to complete this step, but want to see the completed application, you can copy the completed version of the Customer Details shell from the Solution project in your Eclipse Workbench (see Figure 62).

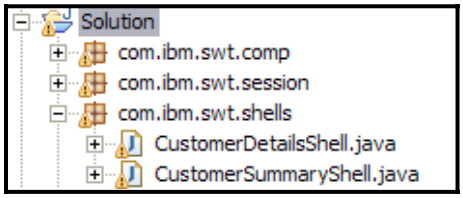

Figure 62. Adding business logic to the Customer Details shell

   a. Add the prepareForMode() method (see the following code) to the CustomerDetailsShell.java class:

```
private void prepareForMode(){

            int mode = SessionMgr.getInstance().getCurrentMode();

            if(mode == SessionMgr.MODE_ADD){
                  btnAction.setText("Add");
            }else if (mode == SessionMgr.MODE_UPDATE){
                  btnAction.setText("Update");
            }else if (mode == SessionMgr.MODE_DELETE){
                  btnAction.setText("Delete");
                  // Disable all fields
                  customerDetailsComp.getTxtCustomerName().setEnabled(false);
                  customerDetailsComp.getTxtAddress().setEnabled(false);
                  customerDetailsComp.getTxtCity().setEnabled(false);
                  customerDetailsComp.getTxtState().setEnabled(false);
                  customerDetailsComp.getTxtZipCode().setEnabled(false);
                  customerDetailsComp.getTxtTelephone().setEnabled(false);
            }else{
                  btnAction.setVisible(false);
            }


      }
```

   You can copy this method code from the following text file:

   C:\Rich Client Lab\Code\PrepareForMode.txt

   Note: The prepareForMode() method displays a different caption for the action button (based on the selected mode) and disables fields when they are not editable.

b. Add the populateCustomerDetails() method to the CustomerDetailsShell.java shell. The code for this method is as follows:

```
private void populateCustomerDetails() {

    Customer customer = SessionMgr.getInstance().getSelectedCustomer();
    if(customer != null){
          customerDetailsComp.getTxtCustomerName().setText(customer.
                      getCustomerNme());

    customerDetailsComp.getTxtAddress().setText(customer.getAddress());
          customerDetailsComp.getTxtCity().setText(customer.getCity());
          customerDetailsComp.getTxtState().setText(customer.getState());

    customerDetailsComp.getTxtZipCode().setText(customer.getZipcode());

    customerDetailsComp.getTxtTelephone().setText(customer.getTelephone());
   }
```

You can copy the populateCustomerDetails() code from the following text file:
C:\Rich Client Lab\Code\PopulateCustomerDetails.txt

Note: The populateCustomerDetails() method populates the text fields, based on the record selected in the Customer Summary shell.

c. In the CustomerDetailsShell constructor, add a call to the following code after the call to the createSShell() method:

```
          public CustomerDetailsShell(){
                createSShell();
                prepareForMode();
                if(SessionMgr.getInstance().getCurrentMode() ==

                      SessionMgr.MODE_VIEW ||
                   SessionMgr.getInstance().getCurrentMode() ==
                      SessionMgr.MODE_UPDATE ||
                SessionMgr.getInstance().getCurrentMode() ==
                SessionMgr.MODE_DELETE ) {
                      populateCustomerDetails();
                }

          }
```

You can copy and paste the complete code for the createSShell() method from the following text file:
C:\Rich Client Lab\CustomerDetailsShellCreateShell.txt

Note: If you are copying and pasting the createSShell() code, make sure you replace the existing constructor; that is, do not create a second constructor.

d. Test the application, using `Brown` as a search value. Then select all menus. Notice that the behavior of the Summary Details panel differs, depending on the mode that you select.

e. Add the perfomAction() method (see the following code) to the CustomerDetailsShell.java class. This method invokes business logic to perform an action that corresponds to the selected mode (add, update, or delete).

```
private void performAction(){

          SessionMgr sessionMgr = SessionMgr.getInstance();

          try{
             Customer customer = new Customer();
             // We don't have customer number in the Add mode
             if(sessionMgr.getCurrentMode() != SessionMgr.MODE_ADD){
                    customer.setCustomerNumber(sessionMgr.getSelectedCustomer()
                    .getCustomerNumber());
             }

          customer.setCustomerName(customerDetailsComp.getTxtCustomerName().
                         getText());

          customer.setAddress(customerDetailsComp.getTxtAddress().getText());
          customer.setCity(customerDetailsComp.getTxtCity().getText());
          customer.setState(customerDetailsComp.getTxtState().getText());
          customer.setZipcode(customerDetailsComp.getTxtZipCode().getText());
          customer.setTelephone(customerDetailsComp.getTxtTelephone().getText());

          iSeriesDataManager dataMgr = iSeriesDataManager.getInstance();
          int mode = SessionMgr.getInstance().getCurrentMode();

          if(mode == SessionMgr.MODE_ADD){
             dataMgr.addCustomer(customer);
          }else if (mode == SessionMgr.MODE_UPDATE){
             dataMgr.updateCustomer(customer);
          }else if (mode == SessionMgr.MODE_DELETE){
             // Display a confirmation message box
             MessageBox confirmMsg = new MessageBox(sShell, SWT.YES + SWT.NO);
             confirmMsg.setMessage("Delete this customer?");
             int result = confirmMsg.open();
             if(result == SWT.YES){
                    dataMgr.deleteCustomer(customer.getCustomerNumber().
                           intValue());
             }
          }

          // If we were in the Add, Update or Deelte mode,
          // refresh the summary view
          if(sessionMgr.getCurrentMode() == SessionMgr.MODE_ADD ||
             sessionMgr.getCurrentMode() == SessionMgr.MODE_UPDATE ||
             sessionMgr.getCurrentMode() == SessionMgr.MODE_DELETE) {
                    sessionMgr.setRefresh(true);
             }else{
                    sessionMgr.setRefresh(false);
             }

          }catch(Exception e){
             e.printStackTrace();
          }finally{
             sShell.close();
          }

          }
```

You can copy and paste the perfomAction() code from the following text file:
C:\Rich Client Lab\Code\CustomerDetailsPerformAction.txt

f. On the Customer Details shell, add the widgetSelected() event to the btnAction class.
Hint: Use the right-click menu in the design mode.

g. Replace the generated code with a call to the performAction() method (see Figure 63):

```
btnAction.addSelectionListener(new org.eclipse.swt.events.SelectionAdapter() {
    public void widgetSelected(org.eclipse.swt.events.SelectionEvent e) {
        performAction();
    }
});
```

Figure 63. Replacing generated code with a call to performAction()

h. Incorporate the handling process for the Cancel button by adding the widgetSelected() event to the btnCancel class on the Customer Details shell, and replace the generated code with the sShell.close() method (see Figure 64):

```
btnCancel.addSelectionListener(new org.eclipse.swt.events.SelectionAdapter()
    public void widgetSelected(org.eclipse.swt.events.SelectionEvent e) {
        sShell.close();
    }
});
```

Figure 64. Replace generated code with the sShell.close() method

i. Test the application. Try adding, updating, and deleting customer records.

Note: Create your own customer records to avoid conflict with other students in the class. (This lab application does not implement concurrency handling.)

Checkpoint 5:

Take a few minutes to reflect on what you have learned in this section:

- How can you add business logic to an SWT application?
- Is it possible to reuse SWT shells for different business functions?

This is the end of this lab. Congratulations. You have implemented your first SWT rich-client application.

Trademarks and special notices

References in this document to IBM products or services do not imply that IBM intends to make them available in every country.

The following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both:

IBM	i5/OS	System i
DB2	iSeries	WebSphere
eServer	Rational	

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

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

Information is provided "AS IS" without warranty of any kind.