

SWT – The Standard Widget Toolkit

Simon Ritchie 11/12/2002

SWT – THE STANDARD WIDGET TOOLKIT	1
INTRODUCTION.....	3
ADVANTAGES/DISADVANTAGES	4
ARCHITECTURE COMPARISON WITH AWT/SWING	6
FEATURE COMPARISON OF SWT WITH SWING	7
LEARNING ABOUT THE WIDGETS.....	7
LIST OF SWT CONTROLS	7
OTHER SWT COMPONENTS	9
USING WIDGETS.....	9
GARBAGE COLLECTION.....	9
THREADING ISSUES	9
LAYOUTS	10
<i>FillLayout</i>	10
<i>RowLayout</i>	11
<i>GridLayout</i>	12
<i>FormLayout</i>	14
<i>Attaching to a position</i>	15
<i>Attaching to the Parent</i>	15
<i>Attaching to another Widget</i>	16
RUNNING SWT AS A STANDALONE PROGRAM.....	17
FINDING THE SWT COMPONENTS	17
<i>Windows</i>	17
<i>Linux (Motif)</i>	17
<i>Linux (GTK2)</i>	17
<i>Mac OS X</i>	17
USING JFACE.....	17
SETTING UP SWT	18
DEPLOYING AN SWT APPLICATION WITH JAVA WEB START	18
EXAMPLE	19
<i>The TestSWT Class</i>	19
<i>The LongRunningThread class</i>	25
WRITING AN ECLIPSE PLUG-IN.....	27
EXAMPLE	27
<i>Using the Plug-in Project Wizard</i>	27
SUMMARY	35
RESOURCES	36

Introduction

SWT is the software component that delivers native widget functionality for the Eclipse platform in an operating system independent manner.

In order to build an Eclipse plug-in that is completely integrated into Eclipse developers need to use SWT. However, it is also possible to use SWT outside of Eclipse in a standalone program.

SWT has generated a lot of interest lately, mostly due to its speed compared to Swing. It's introduction has been quite controversial not only because it competes with Swing, but because it requires native components for each operating system it runs on, seemingly to break the Write Once Run Anywhere goal of Java.

This presentation is based largely on summarizing the various documents that are available from eclipse.org.

Advantages/Disadvantages

SWT does have the disadvantage that it requires native code to be installed on each platform in order for it to run. Each platform/windowing system has its own version of swt.jar as well as at least one shared native object. On Windows this is a .dll, on Linux it is a .so. If this native code is not available for an operating system, then SWT cannot be used on that OS. The good news is that SWT native components have been written for most operating systems.

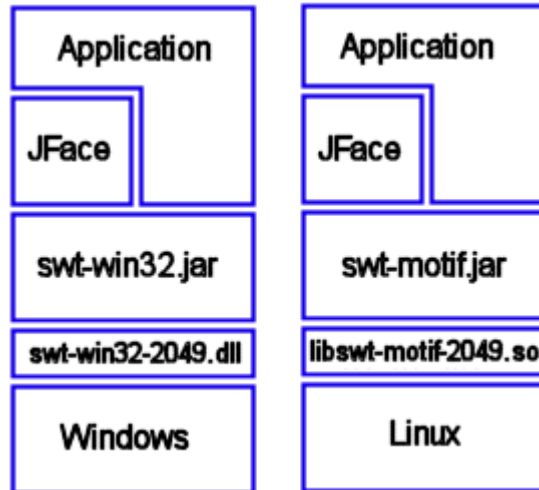
SWT's main advantage is that it is fast. In a simple comparison between IDEs, it is clear that graphical operations in Eclipse are faster than IDEs written in Swing. Swing takes the approach of getting a primitive resource (like a window) from the underlying operating system and then drawing all the graphical elements itself by either using pre-existing images or draw operations. In contrast, SWT directly uses the operating system's UI components.

Using the operating system UI components results in several advantages. Not only is it fast, but it does a better job of presenting the native GUI. Because SWT uses the platform's components, it is an exact match of the OS. In skinable OS's (like Windows XP), Swing can't represent the selected skin, whereas SWT can.

Swing has also been criticized for its memory usage. SWT uses considerably less memory and is reported to use about 50% of the memory that a comparable Swing application would use.

It should be said that the look and feel of Eclipse could be matched by Swing pluggable look and feels. There are some excellent examples at <http://www.jgoodies.com> of how it is possible to produce exactly the same look as Eclipse using Swing. The problem is that the default Sun LAFs don't provide exact representations of the platform the user is used to and speed is still a problem.

Swing, however, has comprehensive support for the Model-View-Controller pattern in JTables, JTrees and JLists. This is not built into SWT but instead built into the JFace layer of components that sit above the SWT layer. Including the JFace components in a standalone application is a bit more involved.



Architecture Comparison with AWT/Swing

The original Java AWT was implemented by having widgets written in Java which delegated to peer classes that were written in C. AWT used a 'lowest common denominator' solution where only widgets that were available on all supported platforms were provided. This meant that AWT included low level components such as lists, text and buttons, but did not include trees or rich text.

Swing solved this problem by providing emulated versions of all the high level components (like trees, tables and text). The Swing toolkit uses pre-created images and draw operations to build it's own components. This solved the problem of functionality, but applications developed in Swing stand out as being different. It also meant that Swing applications ran slower than native counterparts. This makes it difficult to build applications that compete with shrink-wrapped applications developed for a particular platform.

SWT attempts to do the best of both Swing and AWT. It defines a common portable API that is provided on all the platforms. Where possible, the API on each platform is implemented with native widgets. This allows the toolkit to immediately reflect any changes in the underlying OS GUI look and feel, while maintaining a consistent programming model on all platforms.

The "least common denominator" problem is solved by SWT in several ways.

- Features that are not available on all platforms but generally useful for the workbench and tooling plug-ins can be emulated on platforms that provide no native support. For example, the OSF/Motif 1.2 widget toolkit does not contain a tree widget. SWT provides an emulated tree widget on Motif 1.2 that is API compatible with the Windows native implementation.
- Features that are not available on all platforms but not widely used can be omitted from SWT. For example, Windows provides a widget that implements a calendar, but this is not provided in SWT.
- Features that are specific to a platform, such as ActiveX integration, are only provided on the relevant platform. Platform specific features are separated into separate packages that clearly denote the platform name in the package.

Feature Comparison of SWT with Swing

Overall, there are fewer widget classes than in Swing. This is because more function has been included in fewer classes, resulting in an API with less hierarchy than Swing. For instance, instead of a JButton, JCheckBox and JRadioButton there is just a Button class with an int parameter on the constructor to determine what the Button will look like.

Learning about the Widgets

To get familiar with the SWT UI components and layout managers, install the SWT examples and run the SWT Controls and SWT Layouts examples. This is the equivalent of the SwingSet demo. To install, use the following instructions from the Platform Plug-in Developer's Guide:

1. Open the main update manager by clicking **Help > Software Updates > Update Manager**. This opens the Install/Update perspective.
2. In the Feature Updates view, select **New > Site Bookmark** from the context menu. In the New Site Bookmark dialog that opens, give the site the name "Eclipse.org" and URL "http://update.eclipse.org/updates".
3. Expand the "Eclipse.org" item and select the feature named "Eclipse SDK Examples" (or "Eclipse SDK Examples (Windows)" if you're on Windows). Click the **Install** button within the Preview view.
4. The Feature Install wizard confirms the details of the feature you are about to install. Click **Next**.
5. Select the directory into which the example feature is to be installed and hit **Next**.
6. Click **Install** to allow the downloading and installing to proceed.
7. Click **Yes** when asked to exit and restart the workbench for the changes to take effect. The examples are now installed in the workbench.

List of SWT Controls

A Control is a widget that typically has a counterpart representation (denoted by an OS window handle) in the underlying platform. The **org.eclipse.swt.widgets** package defines the core set of widgets in SWT. They include the following control types:

- **Button** - Selectable control that issues notification when pressed and/or released.
- **Canvas** - Composite control that provides a surface for drawing arbitrary graphics. Often used to implement custom controls.
- **Caret** - An "I-beam" that is typically used as the insertion point for text.

- **Combo** - Selectable control that allows the user to choose a string from a list of strings, or optionally type a new value into an editable text field.
- **Composite** - Control that is capable of containing other widgets.
- **CoolBar** - Composite control that allows users to dynamically reposition the cool items contained in the bar.
- **CoolItem** - Selectable user interface object that represents a dynamically positioned area of a cool bar.
- **Group** - Composite control that groups other widgets and surrounds them with an etched border and/or label.
- **Label** - Non-selectable control that displays a string or an image.
- **List** - Selectable control that allows the user to choose a string or strings from a list of strings.
- **Menu** - User interface object that contains menu items.
- **MenuItem** - Selectable user interface object that represents an item in a menu.
- **ProgressBar** - Non-selectable control that displays progress to the user, typically in the form of a bar graph.
- **Sash** - Selectable control that allows the user to drag a rubber banded outline of the sash within the parent window. Used to allow users to resize child widgets by repositioning their dividing line.
- **Scale** - Selectable control that represents a range of numeric values.
- **ScrollBar** - Selectable control that represents a range of positive numeric values. Used in a Composite that has V_SCROLL and/or H_SCROLL styles.
- **Shell** - Window that is managed by the OS window manager. Shells can be parented by a Display (top-level shells) or by another shell (secondary shells).
- **Slider** - Selectable control that represents a range of numeric values. A slider is distinguished from a scale by providing a draggable thumb that can adjust the current value along the range.
- **TabFolder** - Composite control that groups pages that can be selected by the user using labeled tabs.
- **TabItem** - Selectable user interface object corresponding to a tab for a page in a tab folder
- **Table** - A Selectable control that displays a list of table items that can be selected by the user. Items are presented in rows that display multiple columns representing different aspects of the items.
- **TableColumn** - Selectable user interface object that represents a column in a table.
- **TableItem** - Selectable user interface object that represents an item in a table.
- **Text** - Editable control that allows the user to type text into it.
- **ToolBar** - Composite control that supports the layout of selectable tool bar items.
- **ToolItem** - Selectable user interface object that represents an item in a tool bar.
- **Tracker** - User interface object that implements rubber-banding rectangles.

- **Tree** – A selectable control that displays a hierarchical list of tree items that can be selected by the user.
- **TreeItem** - Selectable user interface object that represents a hierarchy of tree items in a tree.

Other SWT Components

- **ColorDialog** – A dialog that allows a user to choose a color.
- **FontDialog** – A dialog used for choosing a text font.
- **FileDialog** – Dialog for choosing a file to open or save.
- **DirectoryDialog** – Dialog for choosing a directory.
- **MessageBox** – Dialog used to contain messages for the user.

Using Widgets

In Swing, components are first instantiated and then added to a JPanel. In SWT, one of the parameters on the constructor is the Composite (equivalent of a JPanel). This seems a little odd at first, but it does enforce the rule of only placing a component within a single panel. Swing doesn't enforce this at compile time. If you do it by accident in Swing, the application errors.

Garbage Collection

Another of the more controversial features of SWT is that there is no garbage collection built in for widgets. The programmer is responsible for disposing of any widget instances they create. The rule is if you call the constructor, you must also call the dispose method. To make disposal easier, calling the dispose method on a widget will automatically call the dispose method for all of the widget's child widgets.

The following article discusses the reasons why SWT doesn't include garbage collection:
<http://www.eclipse.org/articles/swt-design-2/swt-design-2.html>

Threading issues

In any GUI program, Swing or SWT, threading is a problem. New programmers often attempt to perform a long running operation in the event handler method after a GUI event is fired. Perhaps a button is clicked and a series of database updates begin. It soon becomes obvious that something is wrong when the application locks up until the operation is completed. So the solution is to start a new thread in the event handler method. The event is fired, a thread is started and the UI-thread goes back to handling user responses. Meanwhile the thread performs the update task.

Often the long running thread must periodically update a UI component. Perhaps a progress bar needs to be incremented. The best way of handling these kinds of GUI

component updates from within a long running thread is to somehow communicate with the UI thread that a change needs to take place.

In Swing, the long running thread is supposed to call `SwingUtilities.invokeLater()` or `SwingUtilities.invokeAndWait()` to execute these operations. But Swing doesn't prevent you from directly updating a UI component from a non-UI thread. This can lead to problems with thread race conditions.

This is where SWT differs. SWT provides a similar mechanism to Swing with `display.syncExec()` and `display.asyncExec()`, but more importantly, it prevents a non-UI thread from updating any UI component. If an update to a UI component is attempted then an `SWTException` will be thrown. This can come as quite a shock to a Swing developer not used to doing every update on the UI-Thread.

Although this can be a difficult adjustment, it does two things: It seems to result in a clear separation of view classes from thread classes, which is likely to improve an application's class structure. Also it will probably force developers to put long running code in threads and do UI updates in the correct place. A large proportion of Swing's apparent lack of speed is due to developers misunderstanding threading in Swing.

Layouts

Like Swing, SWT defines several layout managers for placing widgets in a *Composite* (or *panel*). Layouts are found in the package **org.eclipse.swt.layout**.

The following article describes the layouts in detail:

<http://www.eclipse.org/articles/Understanding%20Layouts/Understanding%20Layouts.htm>





The following describes some of the features of layouts but is not exhaustive.

FillLayout

FillLayout is the simplest layout class. It lays out widgets in a single row or column, forcing them to be the same size. Initially, the widgets will all be as tall as the tallest widget, and as wide as the widest. *FillLayout* does not wrap, and you cannot specify margins or spacing.

Here is the relevant portion of the example code. First we create a *FillLayout*, then (if we want vertical) we set its **type** field, and then we set it into the *Composite* (a *Shell*). The *Shell* has three pushbutton children, `B1`, `B2`, and `Button 3`. Note that in a *FillLayout*, children are always the same size, and they fill all available space.

```
FillLayout fillLayout = new FillLayout();
fillLayout.type = SWT.VERTICAL;
shell.setLayout(fillLayout);
```

	Initial	After Resize
<code>fillLayout.type = SWT.HORIZONTAL</code> (default)		
<code>fillLayout.type = SWT.VERTICAL</code>		

RowLayout

RowLayout also lays out widgets in rows, but is more flexible than FillLayout. It can wrap the widgets, creating as many rows as needed to display them. It also provides configurable margins on each edge of the layout, and configurable spacing between widgets in the layout. You can pack a RowLayout, which will force all widgets to be the same size. If you justify a RowLayout, extra space remaining in the Composite will be allocated as margins between the widgets.

The height and width of each widget in a RowLayout can be specified in a RowData object, which should be set in the widget using `setLayoutData`.










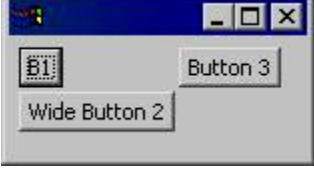
The following example code creates a *RowLayout*, sets all of its fields to non-default values, and then sets it into a *Shell*.

```
RowLayout rowLayout = new RowLayout();
rowLayout.wrap = false;
rowLayout.pack = false;
rowLayout.justify = true;
rowLayout.type = SWT.VERTICAL;
rowLayout.marginLeft = 5;
rowLayout.marginTop = 5;
rowLayout.marginRight = 5;
rowLayout.marginBottom = 5;
rowLayout.spacing = 0;
shell.setLayout(rowLayout);
```

If you are using the default field values, you only need one line of code:

```
shell.setLayout(new RowLayout());
```

In the table below, the result of setting specific fields is shown.

	Initial	After Resize
wrap = true pack = true justify = false type = SWT.HORIZONTAL (defaults)		
wrap = false (clips if not enough space)		
pack = false (all widgets are the same size)		
justify = true (widgets are spread across the available space)		
type = SWT.VERTICAL (widgets are arranged vertically in columns)		

GridLayout

GridLayout is the most powerful (and most complex) layout. With a *GridLayout*, the widget children of a *Composite* are laid out in a grid. *GridLayout* has a number of configuration fields. The widgets it lays out can have an associated layout data object, called *GridData*. The power of *GridLayout* lies in the ability to configure *GridData* for each widget controlled by the *GridLayout*.

This layout behaves similarly to the Swing *GridBagLayout*.

The following code creates a *Shell* with five *Button* children of various widths, managed by a *GridLayout*.

```

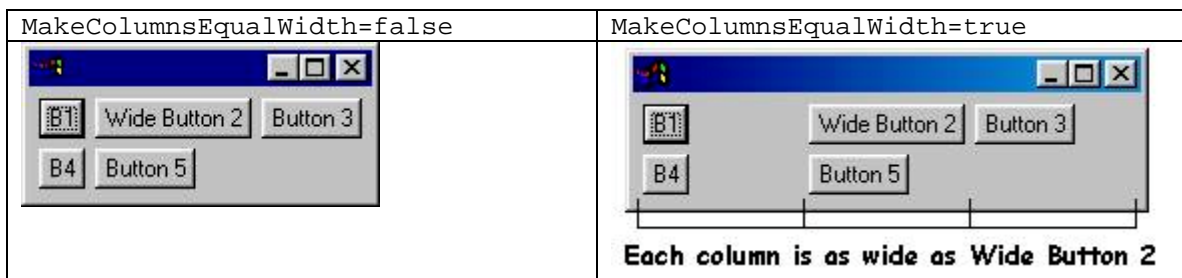
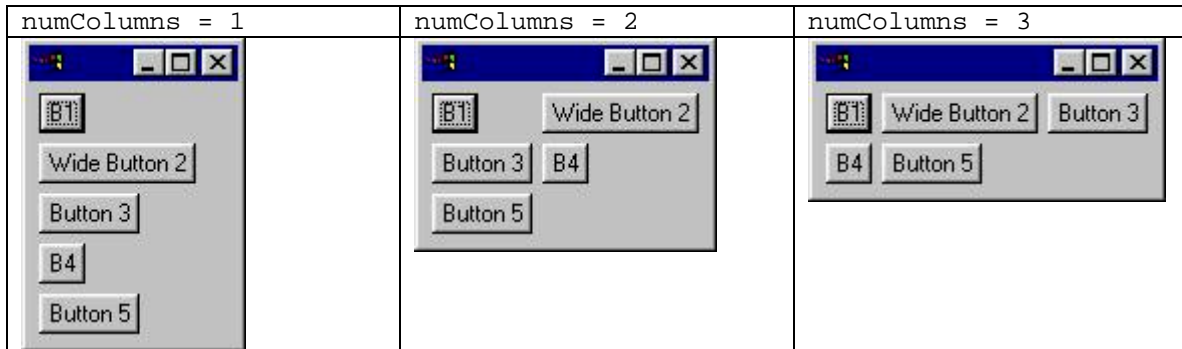
Display display = new Display();
Shell shell = new Shell(display);
GridLayout gridLayout = new GridLayout();
gridLayout.numColumns = 3;
shell.setLayout(gridLayout);
new Button(shell, SWT.PUSH).setText("B1");
new Button(shell, SWT.PUSH).setText("Wide Button 2");
new Button(shell, SWT.PUSH).setText("Button 3");

```

```

new Button(shell, SWT.PUSH).setText("B4");
new Button(shell, SWT.PUSH).setText("Button 5");
shell.pack();
shell.open();
while (!shell.isDisposed()) {
    if (!display.readAndDispatch()) display.sleep();
}

```

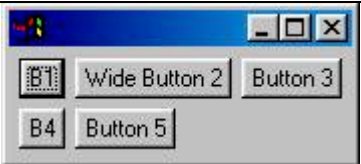


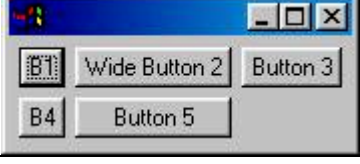


GridData is the layout data object associated with *GridLayout*. To set a widget's *GridData* object, you use the *setLayoutData* method. For example, to set the *GridData* for a *Button*, we could do the following:

```

Button button1 = new Button(shell, SWT.PUSH);
button1.setText("B1");
GridData gridData = new GridData();
gridData.horizontalAlignment = GridData.FILL;
gridData.grabExcessHorizontalSpace = true;
button1.setLayoutData(gridData);

```

horizontalAlignment = GridData.BEGINNING (default)		
horizontalAlignment = GridData.CENTER		
horizontalAlignment = GridData.END		
horizontalAlignment = GridData.FILL		

FormLayout

FormLayout works by creating *FormAttachments* for each side of the widget, and storing them in the layout data. An attachment ‘attaches’ a specific side of the widget either to a position in the parent *Composite* or to another widget within the layout. This provides tremendous flexibility when laying out, as it allows you to specify the placement of individual widgets within the layout.

FormData objects specify how each widget in a *FormLayout* will be laid out. Each *FormData* object defines the attachments for all four sides of the widget. These attachments tell where to position each side of the widget. To set a widget’s *FormData* object, you use the **setLayoutData** method, for example:

```
Display display = new Display ();
Shell shell = new Shell (display);
FormLayout layout= new FormLayout ();
layout.marginHeight = 5;
layout.marginWidth = 5;
shell.setLayout (layout);

Button button1 = new Button(shell, SWT.PUSH);
button1.setText("B1");

FormData formData = new FormData();
formData.top = new FormAttachment(0,60);
formData.bottom = new FormAttachment(100,-5);
formData.left = new FormAttachment(20,0);
formData.right = new FormAttachment(100,-3);
button1.setLayoutData(formData);
```

Attaching to a position

There are many types of attachment. The first is to attach the widget to a position in the parent *Composite*. This can be done by defining a percentage value out of 100, for example:



```
FormData formData = new FormData();  
formData.top = new FormAttachment(50,0);  
button1.setLayoutData(formData);
```

This sets the top of the *Button* to a position that represents 50% of the height of the parent *Composite* (a *Shell*), with an offset of 0. When the shell is resized, the top side of the *Button* will still be at 50%, like so:



If we chose to set an offset value, the top side of the *Button* would have been set to 50% of the *Composite* plus or minus the number of pixels set for the offset.

Attaching to the Parent

The second type of attachment is to attach a side to the edge of the parent *Composite*. This is done in much the same way as attaching to a position, but the position is either 0% or 100%. The 0 position is defined as the top of the *Composite* when going vertically, and the left when going horizontally. The right and bottom edges of the *Composite* are defined as the 100 position. Therefore, if we want to attach a widget to the right edge of the *Composite*, we simply have to create an attachment that sets the position to 100:



```
FormData formData = new FormData();  
formData.right = new FormAttachment(100,-5);  
button1.setLayoutData(formData);
```

This attaches the right side of the *Button* to the right edge of the parent (a *Shell*), with an offset of 5 pixels. Note that the offsets go in one direction only. If you want a widget offset down or to the right, the offset should be positive. For offsets that shift the widget

up or to the left, the offset should be negative. Now when the *Shell* is resized, the *Button* will always be 5 pixels away from the right edge:



Attaching to another Widget

The third type of attachment is to attach the side of the widget to another control within the parent *Composite*. The side can be attached to the adjacent side of the other control (the default), to the opposite side of the other control, or the widget can be centered on the other control, all with or without an offset.

The most common way to attach to another control is to attach to its adjacent side. For example, the following code:

```
FormData formData = new FormData();
formData.top = new FormAttachment(20,0);
button1.setLayoutData(formData);

FormData formData2 = new FormData();
formData2.top = new FormAttachment(button1,10);
button2.setLayoutData(formData2);
```

attaches the top of `button2` to the bottom of `button1`, since the bottom of `button1` is adjacent to the top of `button2`.



Note that when the window is resized, `button1` will move so that its top side is always positioned at 20% of the *Shell*, and `button2` will move so that its top side is always 10 pixels below the adjacent (bottom) side of `button1`.



Running SWT as a Standalone Program

Although SWT was primarily designed for Eclipse, there is nothing (including the license agreement) to stop a developer writing a standalone program using SWT. The license agreement is quite liberal regarding using components of Eclipse in commercial applications. The only difficulty is in distributing the SWT components.

Finding the SWT Components

In order to get the SWT components, you should copy them from the Eclipse install directory. The easiest way is to download Eclipse for your platform, then find swt.jar and the shared objects. You may be able to find current versions of the shared objects in the eclipse.org CVS tree, but you will probably have to build the swt.jar yourself. The components you will need are:

Windows

ECLIPSE\plugins\org.eclipse.swt.win32_2.0.1\os\win32\swt-win32-2049.dll
ECLIPSE\plugins\org.eclipse.swt.win32_2.0.1\ws\win32\swt.jar

Linux (Motif)

ECLIPSE/plugins/org.eclipse.swt.motif_2.0.1/os/linux/x86/libswt-gnome-motif-2049.so
ECLIPSE/plugins/org.eclipse.swt.motif_2.0.1/os/linux/x86/libswt-kde-motif-2049.so
ECLIPSE/plugins/org.eclipse.swt.motif_2.0.1/os/linux/x86/libswt-motif-2049.so
ECLIPSE/plugins/org.eclipse.swt.motif_2.0.1/ws/motif/swt.jar

Linux (GTK2)

ECLIPSE/plugins/org.eclipse.swt.gtk_2.0.1/os/linux/x86/libswt-gtk-2049.so
ECLIPSE/plugins/org.eclipse.swt.gtk_2.0.1/os/linux/x86/libswt-gtk-pi-2049.so
ECLIPSE/plugins/org.eclipse.swt.gtk_2.0.1/ws/gtk/swt.jar
ECLIPSE/plugins/org.eclipse.swt.gtk_2.0.1/ws/gtk/swt-pi.jar

Mac OS X

ECLIPSE/plugins/org.eclipse.swt.carbon_?.?.?.os/macosex/ppc/libswt-carbon-2114.jnilib
ECLIPSE/plugins/org.eclipse.swt.carbon_?.?.?.ws/carbon/swt.jar

Using JFace

If your stand-alone application requires JFace classes, then you will also need the following jar files from Eclipse. Unfortunately, these jar files have not been separated as cleanly as SWT has, and they also include a number of Eclipse workbench classes.

ECLIPSE/plugins/org.eclipse.core.runtime/runtime.jar
ECLIPSE/plugins/org.eclipse.ui/workbench.jar

Setting up SWT

There are different ways of setting up the SWT environment. Basically, it comes down to either putting the shared objects in the application path and the jar file in the Java extensions directory, or setting them up on the call to the Java executable.

Alternatives for configuring SWT on Windows (\$jre\$ is the location of the JRE):

1. Configuring the shared object so that System.loadLibrary() will find it.
 - a) Copy the swt-win32-2034.dll to \$jre\$/bin
 - b) Or, copy the swt-win32-2034.dll to c:\windows\system32 to obtain the same effect
 - c) Or, set -Djava.library.path when starting a SWT program

```
-Djava.library.path=ECLIPSE\plugins\org.eclipse.swt.win32_2.0.1\os\win32\x86
```

2. Configuring the swt.jar file so that it is in the classpath.
 - a) Add swt.jar to the Java classpath with the -cp option.
 - b) Add swt.jar to \$jre\$/lib/ext, so it will be automatically detected.

Deploying an SWT application with Java Web Start

Java applications using SWT can be deployed using Java Web Start. This is possible because Java Web Start can deploy native libraries and different sets of jar files for different operating systems. The only difficulties are firstly the shared objects (Windows .dll) must be placed in a jar file, and secondly all the jar files must be signed. This is because in order to use SWT, the program must perform JNI calls, requiring the JNLP document to request 'all-permissions' authority.

```
<?xml version="1.0" encoding="UTF-8"?>
<jnlp spec="1.0+">
  <information>
    <title>Test SWT</title>
    <description kind="short">Test SWT</description>
    <vendor>Arizona Mail Order</vendor>
  </information>
  <resources>
    <j2se version="1.3+" />
    <jar href="http://hog.amo.com/it/education/swt/testswt.jar" main="true"
        download="eager" />
  </resources>

  <resources os="Windows">
    <jar href="http://hog.amo.com/it/education/swt/swt-win32.jar" />
    <nativelib href="http://hog.amo.com/it/education/swt/swt-win32dll-2049.jar" />
  </resources>

  <resources os="Mac OS X">
    <jar href="http://hog.amo.com/it/education/swt/swt-carbon.jar" />
    <nativelib href="http://hog.amo.com/it/education/swt/swt-carbonjnilib-2111.jar"/>
  </resources>

  <application-desc main-class="com.amo.testswt.TestSWT">
  </application-desc>
  <security>
    <all-permissions>allPermissions</all-permissions>
  </security>
```

</jnlp>

Example

This example is part of an application used to build Java classes for AMO's object-to-relational-DB mapper. The GUI class has been written in SWT, and the thread class has been changed to simulate the build by just incrementing the progress bar.

SWT applications differ from Swing in that the developer must code the event loop. In this example, the event loop processing is done in the main() method. SWT applications start by creating a `Display` representing an SWT session. A `Shell` is created to serve as the main window for the application. The function and display widgets are created in the `Shell` and the `Shell` is then opened. The event loop is processed until an exit condition is detected (the user closing the window) and the display is disposed and the program ended.

```
public static void main(String[] args) {
    // Create the display
    Display display = new Display();

    // Create the main interface class
    TestSWT application = new TestSWT();

    // Open the application using the main display class
    Shell shell = application.open(display);

    // Loop until the shell is closed.
    while(!shell.isDisposed()){
        if(!display.readAndDispatch()) {
            display.sleep();
        }
    }

    // Garbage collect the main display
    display.dispose();

    System.exit(0);
}
```

The `Display` class represents the connection between SWT and the underlying windowing system. Normally only one `Display` is created in an application. It is used primarily to manage the platform event loop and control the communication between the UI thread and other threads. The `Display` must be created before any windows can be displayed, and it must be disposed when the shell is closed.

The `Shell` class represents a window managed by the OS platform window manager.

The `open()` method is responsible for creating a new `Shell`, laying out the widgets on the, packing and centering the shell and finally opening the shell for display.

The TestSWT Class

```
package com.amo.testswt;

import org.eclipse.swt.SWT;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.graphics.*;
import org.eclipse.swt.layout.*;
```

```

/**
 * This class prompts the user to build data object classes from
 * a deployment descriptor.
 */
public class TestSWT {
    private String baseDirectory = "";

    private Display display = null;

    private Text descriptorText = null;

    private Button recordCheck = null;
    private Button tableCheck = null;
    private Button doCheck = null;
    private Button finderCheck = null;
    private Button doimplCheck = null;
    private Button validatorCheck = null;

    private ProgressBar progressBar = null;
    private Label statusLabel = null;

    private Text outputDirectoryText = null;

    private final static String [] filterExtensions = {"*.xml", ""};
    private final static String [] filterNames = {"XML Files", "All Files"};

    public static void main(String[] args) {
        // Create the display
        Display display = new Display();

        // Create the main interface class
        TestSWT application = new TestSWT();

        // Open the application using the main display class
        Shell shell = application.open(display);

        // Loop until the main shell is closed.
        while(!shell.isDisposed()){
            if(!display.readAndDispatch()) {
                display.sleep();
            }
        }

        // Garbage collect the main display
        display.dispose();

        System.exit(0);
    }

    /**
     * Construct a XMLGeneratorFrame
     */
    public TestSWT() {
        super();
    }

    /**
     * Open the shell using the display passed in.
     * @param display The display to use.
     * @return The shell created to display the widgets.
     */
    public Shell open(Display display) {
        this.display = display;

        // Create the shell
        Shell shell = new Shell(display);

        // set the text at the top of the window
        shell.setText("Data Object Builder");
    }

```

```

    // create and set a layout for the shell
    GridLayout layout = new GridLayout(1, true);
    layout.marginHeight = 0;
    layout.marginWidth = 0;
    shell.setLayout(layout);

    // add the widgets
    createLabel(shell, "Descriptor file:");
    createDescriptorPanel(shell, "Test");
    createSpacer(shell);
    createClassesPanel(shell);
    createSpacer(shell);
    createLabel(shell, "Output directory:");
    createOutputDirectoryPanel(shell, baseDirectory);
    createSpacer(shell);
    createSeparator(shell);
    createButtonPanel(shell);
    createStatusPanel(shell);

    // pack the widgets in the shell
    shell.pack();

    // Center the shell on the display
    centerOnScreen(display, shell);

    // open the panel
    shell.open();

    return shell;
}

/**
 * Center the shell on the display
 */
private void centerOnScreen(Display display, Shell shell) {
    Rectangle rect = display.getClientArea();
    Point size = shell.getSize();
    int x = (rect.width - size.x) / 2;
    int y = (rect.height - size.y) / 2;
    shell.setLocation(new Point(x,y));
}

/**
 * Add a separator line to the shell
 * @param shell
 */
private void createSeparator(Shell shell) {
    Label separator = new Label(shell, SWT.SEPARATOR | SWT.HORIZONTAL);
    GridData gd = new GridData();
    gd.horizontalAlignment = GridData.FILL;
    separator.setLayoutData(gd);
}

/**
 * Add a vertical space to the shell
 * @param shell
 */
private void createSpacer(Shell shell) {
    new Label(shell, SWT.NONE);
}

/**
 * Add a label to the shell.
 * @param shell
 */
private void createLabel(Shell shell, String text) {
    Label label = new Label(shell, SWT.NONE);
    label.setText(text);
    label.setLayoutData(new GridData(GridData.FILL_HORIZONTAL));
}

```

```

/**
 * Add the descriptor panel to the shell.
 * @param shell
 */
private void createDescriptorPanel(final Shell shell, String deploymentDescriptor) {
    Composite descriptorGroup = new Composite(shell, SWT.NONE);
    descriptorGroup.setLayoutData(new GridData(GridData.FILL_HORIZONTAL));

    GridLayout gridLayout = new GridLayout();
    gridLayout.marginHeight = 0;
    gridLayout.marginWidth = 0;
    gridLayout.numColumns = 2;

    descriptorGroup.setLayout(gridLayout);

    descriptorText = new Text(descriptorGroup, SWT.SINGLE | SWT.BORDER);
    descriptorText.setText(deploymentDescriptor);
    descriptorText.setTextLimit(150);
    descriptorText.setLayoutData(new GridData(GridData.FILL_HORIZONTAL));

    Button selectButton = new Button(descriptorGroup, SWT.PUSH);
    selectButton.setText("Select...");
    selectButton.addSelectionListener(new SelectionAdapter() {
        public void widgetSelected(SelectionEvent e) {
            selectDescriptor(shell);
        }
    });
}

/**
 * Display a FileDialog to get a selected deployment descriptor.
 * @param shell
 */
private void selectDescriptor(Shell shell) {
    FileDialog fileDialog = new FileDialog(shell, SWT.OPEN);
    fileDialog.setFileName(descriptorText.getText());
    fileDialog.setText("Open a Deployment Descriptor");
    fileDialog.setFilterExtensions(filterExtensions);
    fileDialog.setFilterNames(filterNames);
    String result = fileDialog.open();
    if (result != null) {
        descriptorText.setText(result);
    }
}

/**
 * Add the classes check panel to the shell
 * @param shell
 */
private void createClassesPanel(Shell shell) {
    // Create a grouping panel
    Group classesGroup = new Group(shell, SWT.NONE);
    classesGroup.setLayoutData(new GridData(GridData.FILL_HORIZONTAL));

    RowLayout rowLayout = new RowLayout();
    classesGroup.setLayout(rowLayout);
    classesGroup.setText("Classes to generate");

    recordCheck = new Button(classesGroup, SWT.CHECK);
    recordCheck.setText("Record");
    recordCheck.setSelection(true);

    tableCheck = new Button(classesGroup, SWT.CHECK);
    tableCheck.setText("Table");
    tableCheck.setSelection(true);

    doimplCheck = new Button(classesGroup, SWT.CHECK);
    doimplCheck.setText("DOImpl");
    doimplCheck.setSelection(true);
}

```

```

doCheck = new Button(classesGroup, SWT.CHECK);
doCheck.setText("DO");
doCheck.setSelection(false);

finderCheck = new Button(classesGroup, SWT.CHECK);
finderCheck.setText("Finder");
finderCheck.setSelection(false);

validatorCheck = new Button(classesGroup, SWT.CHECK);
validatorCheck.setText("Validator");
validatorCheck.setSelection(false);
}

/**
 * Add the output directory panel to the shell
 * @param shell
 */
private void createOutputDirectoryPanel(final Shell shell, String baseDirectory) {
    Composite outDirectoryGroup = new Composite(shell, SWT.NONE);
    outDirectoryGroup.setLayoutData(new GridData(GridData.FILL_HORIZONTAL));

    GridLayout gridLayout = new GridLayout();
    gridLayout.marginHeight = 0;
    gridLayout.marginWidth = 0;
    gridLayout.numColumns = 2;

    outDirectoryGroup.setLayout(gridLayout);

    outputDirectoryText = new Text(outDirectoryGroup, SWT.SINGLE | SWT.BORDER);
    outputDirectoryText.setText(baseDirectory);
    outputDirectoryText.setTextLimit(150);
    outputDirectoryText.setLayoutData(new GridData(GridData.FILL_HORIZONTAL));

    Button selectButton = new Button(outDirectoryGroup, SWT.PUSH);
    selectButton.setText("Select...");
    selectButton.addSelectionListener(new SelectionAdapter() {
        public void widgetSelected(SelectionEvent e) {
            selectOutputDirectory(shell);
        }
    });
}

/**
 * Display a DirectoryDialog to get a selected directory.
 * @param shell
 */
private void selectOutputDirectory(Shell shell) {
    DirectoryDialog directoryDialog = new DirectoryDialog(shell, SWT.OPEN);
    directoryDialog.setFilterPath(outputDirectoryText.getText());
    directoryDialog.setMessage("Choose the base directory for creating data objects.");
    directoryDialog.setText("Select an Output Directory");
    String result = directoryDialog.open();
    if (result != null) {
        outputDirectoryText.setText(result);
    }
}

/**
 * Add a button panel to the shell
 */
private void createButtonPanel(final Shell shell) {
    Composite buttonGroup = new Composite(shell, SWT.NONE);
    buttonGroup.setLayoutData(new GridData(GridData.FILL_HORIZONTAL));

    GridLayout gridLayout = new GridLayout();
    gridLayout.numColumns = 3;

    buttonGroup.setLayout(gridLayout);

    GridData okGridData = new GridData();

```

```

GridData cancelGridData = new GridData();

Label label = new Label(buttonGroup, SWT.NONE);
GridData labelGridData = new GridData();
labelGridData.horizontalAlignment = GridData.FILL;
labelGridData.grabExcessHorizontalSpace = true;
label.setLayoutData(labelGridData);

Button okButton = new Button(buttonGroup, SWT.PUSH);
okButton.setLayoutData(okGridData);
okButton.setText("OK");
okButton.addSelectionListener(new SelectionAdapter() {
    public void widgetSelected(SelectionEvent e) {
        startBuild();
    }
});

Button cancelButton = new Button(buttonGroup, SWT.PUSH);
cancelButton.setLayoutData(cancelGridData);
cancelButton.setText("Cancel");
cancelButton.addSelectionListener(new SelectionAdapter() {
    public void widgetSelected(SelectionEvent e) {
        shell.close();
        shell.dispose();
    }
});
}

/**
 * Add a status bar to the panel.
 * @param shell
 */
private void createStatusPanel(Shell shell) {
    Composite statusGroup = new Composite(shell, SWT.NONE);
    statusGroup.setLayoutData(new GridData(GridData.FILL_HORIZONTAL));

    GridLayout gridLayout = new GridLayout();
    gridLayout.marginHeight = 0;
    gridLayout.marginWidth = 0;
    gridLayout.numColumns = 3;

    statusGroup.setLayout(gridLayout);

    progressBar = new ProgressBar(statusGroup, SWT.HORIZONTAL);
    GridData progressGridData = new GridData();
    progressGridData.heightHint = 15;
    progressGridData.widthHint = 100;
    progressGridData.horizontalAlignment = GridData.BEGINNING;
    progressBar.setLayoutData(progressGridData);

    statusLabel = new Label(statusGroup, SWT.BORDER);
    GridData labelGridData = new GridData();
    labelGridData.heightHint = 15;
    labelGridData.horizontalAlignment = GridData.FILL;
    labelGridData.grabExcessHorizontalSpace = true;
    labelGridData.horizontalSpan = 2;
    statusLabel.setLayoutData(labelGridData);
}

/**
 * Start the build
 */
public void startBuild() {
    LongRunningThread thread = new LongRunningThread(this);
    thread.start();
}

/**
 * This method is called by the thread to initialize the progress bar.
 */
public void startProgress(final int total) {

```



```

        display.asyncExec(new Runnable() {
            public void run() {
                progressBar.setMinimum(0);
                progressBar.setSelection(0);
                progressBar.setMaximum(total);
            }
        });
    }

    /**
     * This method is called by the thread to set the status text.
     */
    public void setStatusText(final String text) {
        display.asyncExec(new Runnable() {
            public void run() {
                statusLabel.setText(text);
            }
        });
    }

    /**
     * This method is called by the thread to increment the progress bar.
     */
    public void incrementProgress() {
        display.asyncExec(new Runnable() {
            public void run() {
                progressBar.setSelection(progressBar.getSelection()+1);
            }
        });
    }

    /**
     * This method is called by the thread when it has finished.
     */
    public void stopProgress() {
        display.asyncExec(new Runnable() {
            public void run() {
                progressBar.setSelection(progressBar.getMaximum());
            }
        });
    }
}

```

The LongRunningThread class

```

package com.amo.testswt;

/**
 * A simple Thread class that sends progress updates every half second
 * for MAX_SECS.
 */
public class LongRunningThread extends Thread {
    private final static int MAX_SECS = 30;
    private TestSWT view = null;

    /**
     * Constructor for LongRunningThread.
     */
    public LongRunningThread(TestSWT view) {
        super();
        this.view = view;
    }

    /**
     * Increment the progress for MAX_SECS
     */
    public void run() {
        view.setStatusText("");
        view.startProgress(MAX_SECS * 2);
        for (int i = 0; i < MAX_SECS * 2; i++) {
            waitHalfSecond();
        }
    }
}

```

```

        view.incrementProgress();
        view.setStatusText("Count = "+i);
    }
    view.stopProgress();
    view.setStatusText("Complete");
}

/**
 * Wait half a second
 */
private void waitHalfSecond() {
    try {
        Thread.sleep(500);
    } catch (Exception e) {}
}
}

```

Writing an Eclipse Plug-in

The Eclipse platform is composed of a core runtime engine with a number of connected plug-ins. Plug-ins are connected to the platform through pre-defined extension points. The workbench UI itself is a plug-in. When you start up the workbench, you are not starting up a single Java program, you are activating a platform runtime which can dynamically discover registered plug-ins and start them as needed.

The platform has a well-defined set of extension points – places where you can hook into the system and contribute system behavior. From the platform's perspective, your plug-in is no different from a basic plug-in like the resource management system or the workbench.

Example

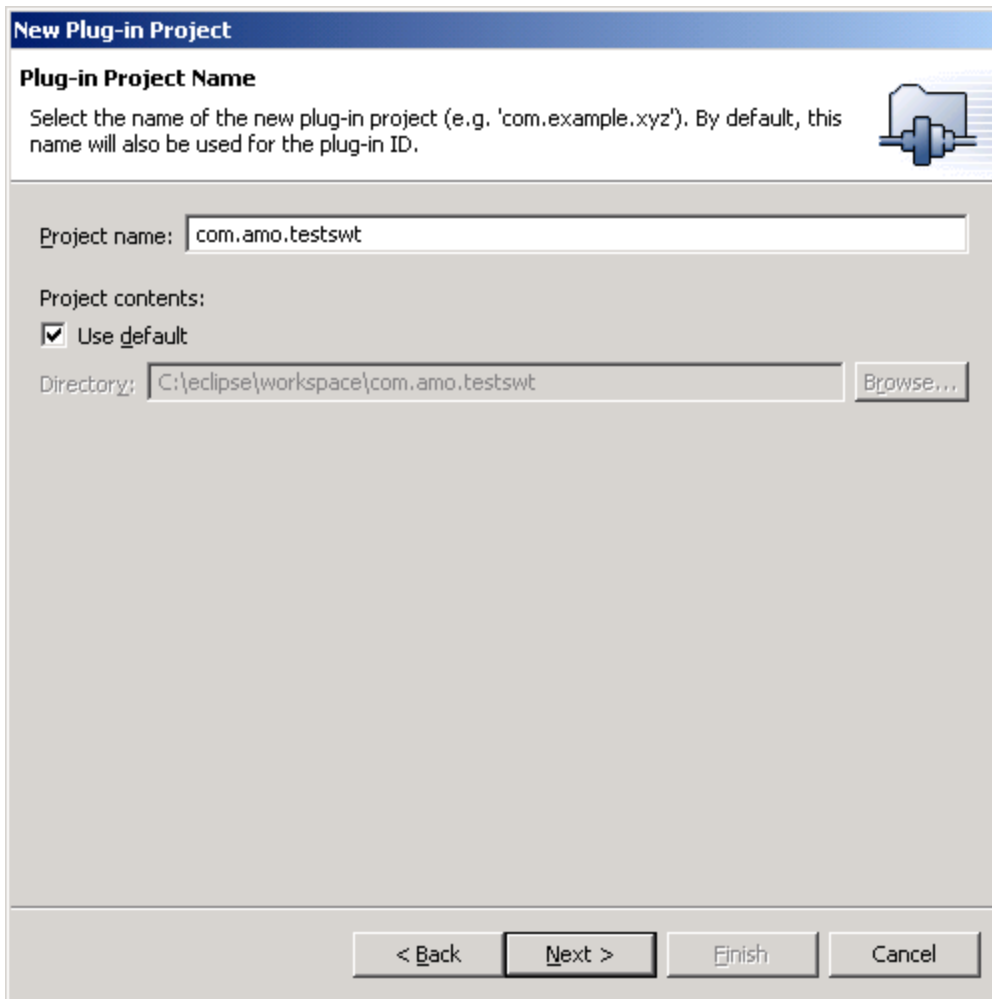
The following example shows how to create a plug-in out of the previous example. This plug-in will be connected to Eclipse via a popup menu.

Using the Plug-in Project Wizard

An easy way to get started with a plug-in is to use the wizard. This will prompt us through all the steps.

To create a plug-in, we need to first create a plug-in project. File | New... | Project starts the project wizard. Select Plug-in Development in the left pane and then choose Plug-in Project in the right pane. Click next. The following pane will be displayed:

Choose a name for the project. Although you can use any name for the project, later parts of the wizard assume the name will be a Java package name. Click next when the name has been entered.



New Plug-in Project

Plug-in Project Name

Select the name of the new plug-in project (e.g. 'com.example.xyz'). By default, this name will also be used for the plug-in ID.

Project name:

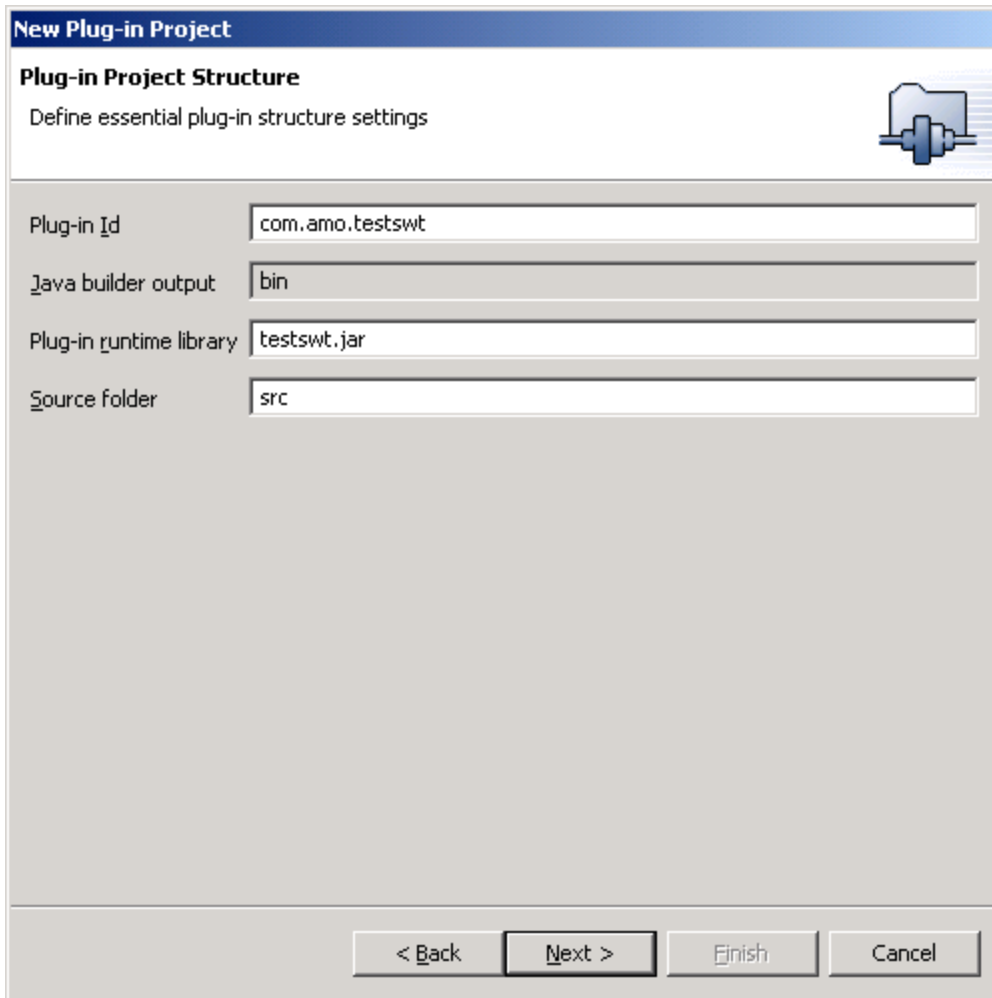
Project contents:

☒ Use default

Directory:

< Back Next > Finish Cancel

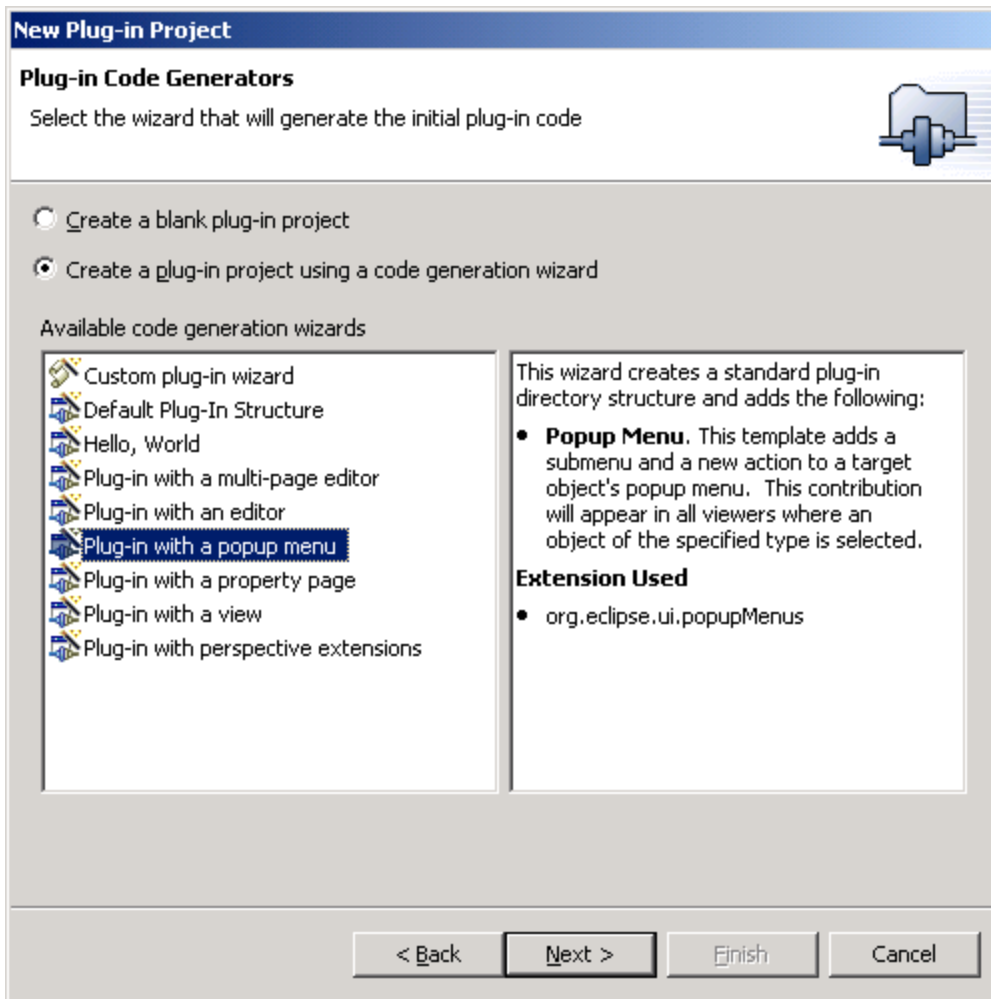
The Project Structure pane is displayed. If the right format project name was chosen then you shouldn't need to alter this pane. Click next when complete.



The image shows a 'New Plug-in Project' dialog box with a title bar. Below the title bar, the text 'Plug-in Project Structure' is followed by the instruction 'Define essential plug-in structure settings'. A folder icon with a plus sign is in the top right corner. The main area contains four text input fields: 'Plug-in Id' with 'com.amo.testswt', 'Java builder output' with 'bin', 'Plug-in runtime library' with 'testswt.jar', and 'Source folder' with 'src'. At the bottom, there are four buttons: '< Back', 'Next >', 'Finish', and 'Cancel'.

Field	Value
Plug-in Id	com.amo.testswt
Java builder output	bin
Plug-in runtime library	testswt.jar
Source folder	src

The Plug-in Code Generator pane is displayed. This is where we choose to create a plug-in with a popup menu. If you are familiar with plug-ins you could create a blank plug-in project. Click next when ready.



The Plug-in Content pane will be displayed. The Class Name is the name of the plugin class that will be created. Click next when complete.

The screenshot shows a dialog box titled "New plug-in project with popup menus". Inside, the "Simple Plug-in Content" tab is active, with the instruction "Enter the required data to generate initial plug-in files". A small icon of a computer monitor with a plug is in the top right. The form contains the following fields and options:

- Plug-in name:** TestSWT Plug-in
- Version:** 1.0.0
- Provider Name:** AMO
- Class Name:** com.amo.testswt.TestSWTPlugin
- ☒ **Generate code for the class**
- Plug-in code generation options:**
 - ☒ Add default instance access
 - ☐ Add support for resource bundles
 - ☒ Add access to the workspace

At the bottom, there are four buttons: "< Back", "Next >", "Finish", and "Cancel".

The Sample Popup Menu will be displayed. This pane prompts the user for items specific to the popup menu.

The **Target Object's Class** is the class name of the file that will be selected when the popup is shown.

The **Name Filter** is used to determine if the menu item will appear on the popup menu when a document is selected and the mouse right clicked. In this case, we only want the menu item to appear for files that end in the extension .xml

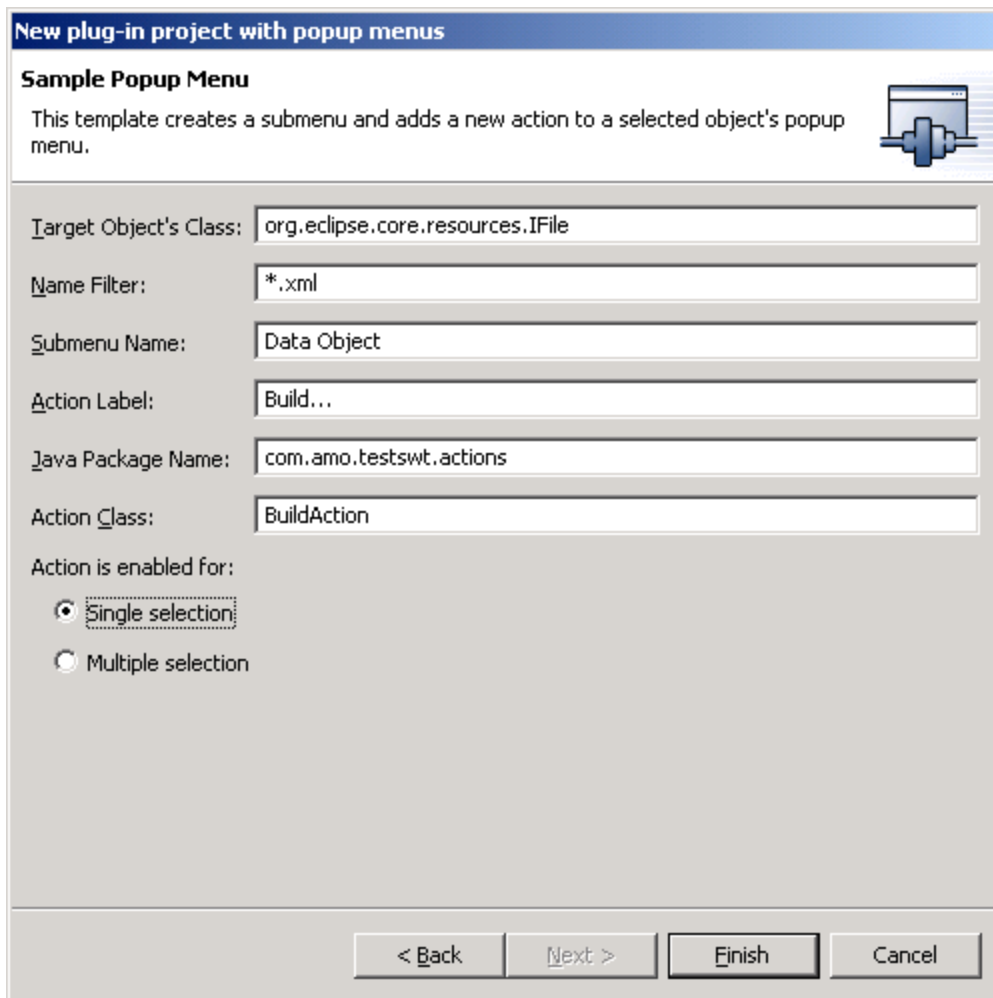
The **Submenu Name** is the name of the sub menu that will be added to the popup menu.

The **Action Label** is the text that will appear in the menu item inside the sub menu.

The **Java Package Name** is the package for the action class that will be created.

The **Action Class** is the class that will be called when the menu item is selected.

Choosing **Single Selection** restricts the action to only using a single selected node in the tree.



New plug-in project with popup menus

Sample Popup Menu

This template creates a submenu and adds a new action to a selected object's popup menu.

Target Object's Class:

Name Filter:

Submenu Name:

Action Label:

Java Package Name:

Action Class:

Action is enabled for:

☒ Single selection

☐ Multiple selection

< Back Next > Finish Cancel

Click Finish to create the project.

The project will be shown in the package explorer.

Two classes will have been created: BuildAction and TestSWTPlugin.

The files plugin.xml and build.properties will also be created. The file plugin.xml describes how the plugin extends the platform.

Now in order to make our original example into a plugin, we copy all the methods from out original TestSWT class into the TestSWTPlugin class.

Then we need to modify the BuildAction class to call the plugin.

```
package com.amo.testswt.actions;

import org.eclipse.core.resources.*;
import org.eclipse.jface.action.IAction;
import org.eclipse.jface.viewers.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.ui.IObjectActionDelegate;
import org.eclipse.ui.IWorkbenchPart;

import com.amo.testswt.*;

public class BuildAction implements IObjectActionDelegate {
    private IStructuredSelection fSelection = null;

    public BuildAction() {
        super();
    }

    public void setActivePart(IAction action, IWorkbenchPart targetPart) {
    }

    /**
     * Run the plugin with the current selection
     */
    public void run(IAction action) {
        Object obj = fSelection.getFirstElement();
        if (obj != null && obj instanceof IFile) {
            IFile file = (IFile)obj;
            String path = file.getLocation().toString();
            TestSWTPlugin plugin = TestSWTPlugin.getDefault();
            plugin.setDeploymentDescriptor(path);

            // Open the plugin
            plugin.open(Display.getCurrent());
        }
    }

    /**
     * Save the selection
     */
    public void selectionChanged(IAction action, ISelection selection) {
        if (selection instanceof IStructuredSelection)
            fSelection= (IStructuredSelection)selection;
        else
            fSelection= StructuredSelection.EMPTY;
    }
}
```

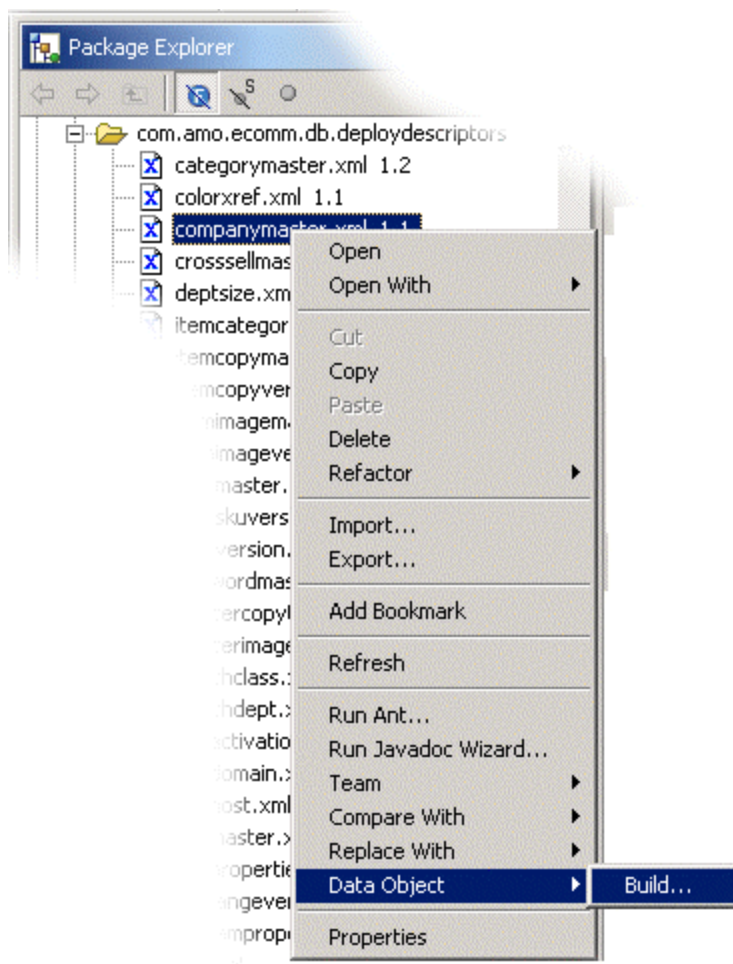
Once the plugin class is complete, and the action has been modified, the jar file can be created. Right click on the plugin.xml and choose Create Plug-in Jars. This creates an Ant build.xml script that will create the jar file.

You may have to refresh the project to see the build.xml and the jar file.

When the jar file has been created, the plugin directory needs to be created. An easy way to do this is to run the build.xml script and choose the **zip.plugin** target. This will build a zip file that contains the correct directory structure.

Unzipping the zip file into the eclipse/plugin-ins directory will setup the jar file. The plugin.xml and build.properties will also need to be copied into the same directory.

When Eclipse is restarted, the new plug-in will be registered and available for use. To try the plug-in, select a file that ends with .xml and right-click. One of the menu options will be Data Object. This option will have an item named 'Build...'. Clicking this will start the plugin.



Summary

SWT certainly runs faster than Swing, and appears to consume a lot less memory. More analysis needs to be done to show exactly what kinds of performance increases the average application can expect.

SWT is different enough from Swing that an application written in Swing cannot easily be converted to SWT. Not only are some of the widgets different, but so are the layout managers, and there are no direct equivalents to some layouts. However, writing new SWT layouts would probably not be difficult. For developers that already have frameworks and components written for Swing, there is no easy conversion. These would have to be rewritten for SWT.

The simple widget classes show an implementation that is simpler than Swing and therefore perhaps more intuitive. However, the more complex classes such as Table and Tree and their MVC counterparts in JFace have not yet been tried.

Distribution of the required native components is the primary hinderance to wide deployment. This should be possible with Java Web Start, but it would certainly be easier if Sun included these files with the JRE. This is unlikely to happen any time soon, however.

Overall, though, SWT presents a great alternative to Swing for Java GUI developers - an option that will particularly appeal to developers writing shrink-wrapped applications. It has the potential to finally allow Java applications to compete successfully on the desktop.

Resources

The Platform Plug-in Developer's Guide:

<http://dev.eclipse.org:8080/help/help.jsp>

SWT: The Standard Widget Toolkit - Part 1:

<http://www.eclipse.org/articles/Article-SWT-Design-1/SWT-Design-1.html>

SWT: The Standard Widget Toolkit - Part 2:

<http://www.eclipse.org/articles/swt-design-2/swt-design-2.html>

SWT Layouts

<http://www.eclipse.org/articles/Understanding%20Layouts/Understanding%20Layouts.htm>

SWT Examples:

<http://dev.eclipse.org/viewcvs/index.cgi/~checkout~/platform-swt-home/dev.html#snippets>

The Eclipse Wiki Site:

<http://eclipsewiki.swiki.net>

This document was created with Win2PDF available at <http://www.daneprairie.com>.
The unregistered version of Win2PDF is for evaluation or non-commercial use only.