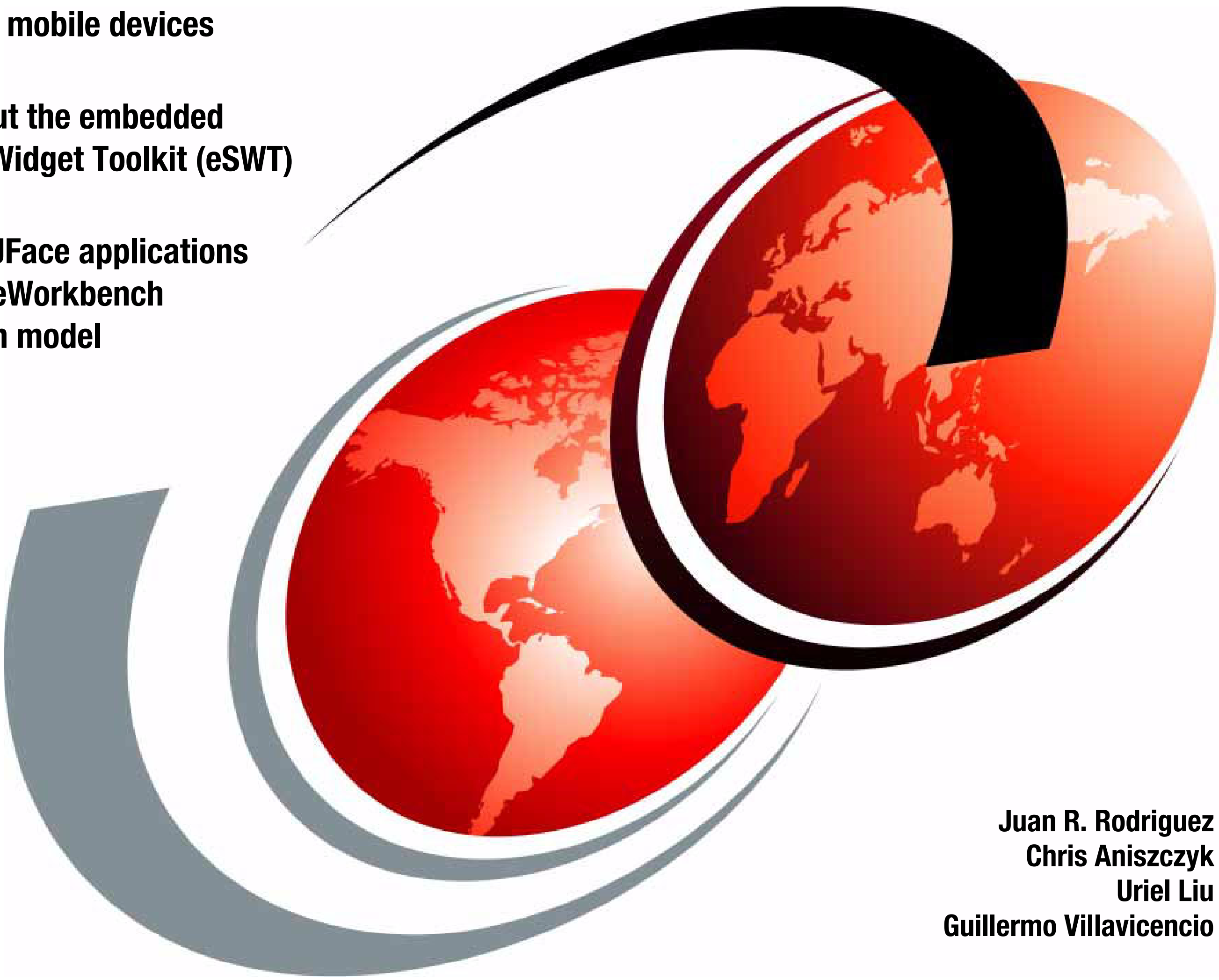
The Eclipse embedded Rich Client Platform

A Graphical User Interface for Small Devices

Understand eRCP applications for embedded mobile devices

Learn about the embedded Standard Widget Toolkit (eSWT)

Develop eJFace applications using the eWorkbench application model



Juan R. Rodriguez
Chris Aniszczyk
Uriel Liu
Guillermo Villavicencio

Redpaper

ibm.com/redbooks

International Technical Support Organization

The Eclipse embedded Rich Client Platform: A Graphical User Interface for Small Devices

June 2006

Note: Before using this information and the product it supports, read the information in “Notices” on page vii.

First Edition (June 2006)

This edition applies to Version 3.1.2 of Eclipse SDK and Version 1.4.2 of Java Runtime Environment (JRE)

Contents

Notices

Trademarks

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

Redbooks (logo) ™	Extreme Blue™	Workplace™
DB2®	IBM®	Workplace Client Technology™
DPI®	Redbooks™	
Everyplace®	WebSphere®	

The following terms are trademarks of other companies:

Java, JDBC, JVM, J2EE, J2ME, J2SE, and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

ActiveSync, Microsoft, Windows Mobile, Windows, Win32, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

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

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

Preface

The embedded Rich Client Platform (eRCP) is an open source project under the Eclipse Technology Project using OSGi standards. This IBM® Redpaper covers the eRCP Extension Point Framework as the basis for creating embedded client platforms and implementing eSWT, eJFace, and eWorkbench for embedded devices having fewer resources and smaller screen sizes than a desktop computer.

In this redpaper, you will find information about what APIs, as a subset of SWT for desktops, are available and apply to Eclipse eSWT basic components, including Core eSWT and Expanded eSWT. You will also find information to better use JFace as a means for providing model, view, and controller mobile application paradigms.

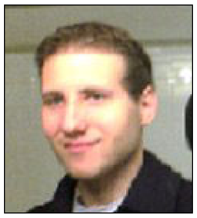
This redpaper includes also a sample scenario chosen to illustrate how eRCP applications are developed to access a local database using DB2® Everyplace®. You will find the elements of the business processes being put into place, and understand what technologies were chosen to solve the different parts of the implementation. The sample scenario includes step by step guidelines to use the proper tools, runtimes, and APIs needed to build, test and deploy an eRCP application for small devices.

A basic knowledge of Java™ programming and Java technologies is required.

The team that wrote this Redpaper

This Redpaper was produced by a team of specialists from around the world working at the International Technical Support Organization (ITSO), Raleigh Center.

Juan R. Rodriguez is a Consulting IT professional at the IBM ITSO Center, Raleigh. He has an Master of Science degree in Computer Science from Iowa State University. He writes extensively and teaches IBM classes worldwide on Web technologies, and information security. Before joining the IBM ITSO, he worked at the IBM laboratory in Research Triangle Park, North Carolina as a designer and developer of networking products.



Chris Aniszczyk is a Software Engineer in the IBM Software Group and a graduate of the Extreme Blue™ internship program with IBM. During his college days, Chris attended Worcester Polytechnic Institute at Worcester, Massachussets. He is an open source enthusiast at heart, and he works on the Gentoo Linux® distribution and is a committer on the Eclipse Communication Framework (ECF) and Eclipse Modeling Framework Tools (EMFT) projects.

Uriel Liu is a Staff Software Engineer at IBM China Development Lab in Taipei, Taiwan. He received his M.S. degree in Computer Science and Information Engineering from National Taiwan University. He joined IBM and devoted in pervasive computing software design, development and testing, especially on embedded devices. His area of expertise covers embedded Linux, Microsoft® Windows® Mobile, Qt/e, J2ME™, WebSphere® Everyplace Access, and WebSphere Everyplace Deployment. He is also a certified Project Management Professional.

Guillermo Villavicencio holds a degree in Informatics Engineering from the Pontifical Catholic University of Peru. He is the Chief Information Officer for Carhados Peru, an IBM Business Partner. He has been the architect for several e-business projects including wireless and Portal solutions. His current area of expertise is centered around Web technologies and pervasive computing. He writes and teaches classes worldwide on Web technologies, and Java Application Development using Eclipse platforms.

Thanks to the following people for their contributions to this project:

Debbie Willmschen, Editor
ITSO, Raleigh Center

Become a published author

Comments welcome

1

Introduction to eRCP

This chapter introduces you to the embedded Rich Client Platform (eRCP) fundamental topics, terminology, and architecture.

1.1 Rich Client Platform

Eclipse 3.0 SDK allows developers to enhance easily the functionality of the Integrated Development Environment (IDE) portion of Eclipse. In addition, these pieces of functionality (plug-ins) can be installed dynamically or updated without restarting Eclipse. Eclipse developers ask questions such as can I dump all the plug-ins that come with Eclipse and use the platform to host only business specific plug-ins that I built?

Users of Eclipse demanded an application model with open architecture, easy configurablity, install and update support, and user configuration and preferences. Some non-IDE application developers want to remove IDE-specific features such as built-in editors, views, and perspectives to have rich functionality with a low footprint. These requirements have resulted in the development of the Rich Client Platform (RCP). RCP requires a minimum set of plug-ins with the following major characteristics:

- Specified interfaces: A component must declare its public API and extensions
- Lazy loading: Components are loaded on demand not on startup
- Versioning: Prerequisite components are referenced by name and version
- Dynamic detection: Components are detected dynamically (no need to restart)
- Management: Install, update, remove, and discover components

With these features, RCP application developers can focus effectively on their business logic without worrying about the core infrastructure. Multiple developers can also create functionality simultaneously and then access the RCP plug-in infrastructure easily.

1.2 The embedded Rich Client Platform

The RCP application model can be used widely among different device types, including embedded devices. The RCP platform architecture is well adapted to low-end devices because multiple applications can run in a single Java Virtual Machine (JVM™) with the introduction of Open Service Gateway interface (OSGi) technology. The only issue with the RCP platform is that performance and footprint becomes sensitive and significant in an embedded environment. The embedded version of RCP, embedded Rich Client Platform (eRCP) was created to solve these issues.

In essence, eRCP is a smaller application model which is slightly different than RCP. eRCP includes subset of various components that make up RCP to conserve memory usage and to produce a smaller footprint. The eRCP team describes eRCP as follows:

> *eRCP is an Eclipse technology project primarily slated to investigate the suitability of using various components of the Eclipse platform in a variety of embedded devices such as mobile phones and PDAs.*

The eRCP application model still preserves what's majorly included in RCP, especially most of the OSGi framework capabilities. However, eRCP provides only one single system workbench, which is branded and optimized for a particular device. All applications in eRCP are plug-ins to this system workbench. Better yet, eRCP applications have no need to provide their own workbench. That usually makes for a smaller application footprint than what would normally be found in RCP.

1.3 Terminology

This section introduces you to the terminologies that are used in the eRCP platform.

1.3.1 Eclipse

Eclipse is an open-source development framework, or workbench, and set of widgets that are designed for tools developers to leverage code reuse, a consistent user interface, and a plug-in architecture for developing new packages. While Eclipse was designed originally for tools developers, it has evolved to become a more general purpose application platform with its RCP. You can find more information about Eclipse at:

http://www.eclipse.org

1.3.2 Java 2 Micro Edition

Java 2 Micro Edition (J2ME) is a Java platform for consumer and embedded devices. Similar to the enterprise (J2EE™) and desktop (J2SE™) platforms, J2ME is a set of standard Java APIs that delivers the power and benefits of Java technology tailored for embedded devices. It includes a flexible user interface, robust security model, broad range of built-in network protocols, and support for networked and disconnected applications.

J2ME configurations

J2ME configurations are compromised of a virtual machine (VM) and a minimal set of libraries that provide the base functionality for a particular range of devices that share similar characteristics, such as network connectivity and memory footprint. Currently, there are two J2ME configurations: Connected Limited Device Configuration (CLDC) and Connected Device Configuration (CDC).

J2ME profiles

Profiles are higher-level APIs that, combined with device configurations, provide a complete runtime environment that is targeted at specific device categories. Profiles further define the application life cycle model, the user interface, and access to device-specific properties.

Foundation Profile

CDC profiles are layered so that you can add further profiles as needed to provide application functionality for different types of devices. The Foundation Profile (FP) is the lowest level profile for the CDC. FP provides a network-capable implementation of CDC that you can use for deeply embedded implementations without a user interface. In addition, you can combine FP with Personal Basis Profile (PBP) and Personal Profile (PP or PPro) for devices that require a graphical user interface (GUI). It is also possible to write eSWT applications on top of Foundation Profile to create a rich user interface on top of the foundation.

1.3.3 Open Service Gateway initiative

The Open Service Gateway initiative (OSGi) was founded in March 1999. Its creates open specifications for the network delivery of managed services to local networks and devices. The OSGi organization is the leading standard for next-generation internet services for homes, cars, small offices, and other environments.

The OSGi service platform specification delivers an open, common architecture for service providers, developers, software vendors, gateway operators, and equipment vendors to develop, to deploy, and to manage services in a coordinated fashion. Because of its flexible and managed deployment of services, it enables an entirely new category of smart devices. The primary targets for the OSGi specifications are set-top boxes, service gateways, cable modems, consumer electronics, PCs, industrial computers, cars, and more. Devices that implement the OSGi specifications enable service providers such as telcos, cable operators, utilities, and others, to deliver valuable services over their networks.

The OSGi specification enables the separation of the service interface from the service implementation, allowing scalability and extensibility. The SMF framework uses the information in the bundle manifests to populate its service registry and to manage and to resolve bundle dependencies. Each bundle has its own class loader and name space as well as the references between bundles to access each others' services and is managed by the framework.

The OSGi platform enables devices of any shape and size to execute a consistent, modular program model on a well-architected set of framework and core services. This framework provides services and bundle life cycle management to enable dynamic loading, starting, and stopping, and more importantly, bundle and services sharing in the VM instance that is running the platform.

1.3.4 Service Management Framework

Service Management Framework (SMF) is an implementation of the OSGi) Service Platform specification. The SMF acts as a layer that enables operators to deploy multiple applications on a single JVM. In addition, it provides a framework for application life cycle, including delivery to the device as well as dynamic starting and stopping of the applications. Application developers partition applications into *services* and other resources. Services and resources are packaged into *bundles*. Bundles are files that serve as the delivery *unit* for applications and have manifests with special headers that enable the sharing of classes and services at the package level. Bundles can be started and stopped dynamically, allowing you to update systems without extended services or downtime. In many cases, you can perform these updates over the air without any user intervention, interaction, or even awareness.

Additionally, the Eclipse framework has incorporated SMF into its underpinnings. So, ultimately, the plug-ins that run in Eclipse run as bundles, and you can load, unload, start, and stop them dynamically. This adds an additional element of flexibility and choice to the application as well as a more robust platform underneath.

1.4 Technology overview

Similar to the RCP application model, eRCP provides minimum sets of RCP components to enable plug-in architecture on an embedded environment. Figure 1-1 shows the architecture of eRCP. It consists of blocks that are embraced inside a bold orange line. All components are subsets of the corresponding components in the RCP application model, except for mobile extension in eSWT, which is a set of controls, widgets, or listeners that are designed specifically for embedded offerings.

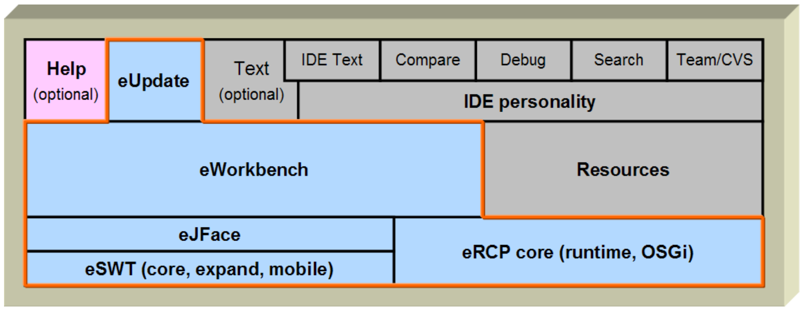


Figure 1-1 The eRCP architecture

1.4.1 The eRCP core

The eRCP core includes a set of runtimes and utilities that support the plug-in application model. It contains a set of non-graphical tools that facilitate the operations that underlie eRCP applications. The eXML component that is packaged within eRCP helps applications handle XML-related issues. The *org.eclipse.core.runtimes* plug-in works as it does in RCP, as a thin layer over the OSGi APIs to support plug-ins architecture. The org.eclipse.osgi plug-in is an implementation that complies with the OSGi service framework that enables life cycle management of bundle services.

1.4.2 The embedded Standard Widget Toolkit

The embedded Standard Widget Toolkit (eSWT) is a subset of Standard Widget Toolkit (SWT) that provides a set of controls, panels, and other widgets that are building blocks of user interfaces. eSWT provides a completely platform-independent API that is integrated tightly with an operating system's graphics system. All Java widgets map to a platform's native widgets. eSWT offers a seamless environment because the look and feel of Java applications are visually indistinguishable from native applications.

As a subset of SWT, eSWT provides a set of APIs that are related closely to those of SWT and encourages reuse of existing SWT applications.

Figure 1-2 shows how eSWT and eSWT: Mobile Extensions fit into the overall programming architecture. eSWT is a subset of SWT and is made up of two components:

- The Core portion, which is required on devices.
- The Expanded portion, which is optional.

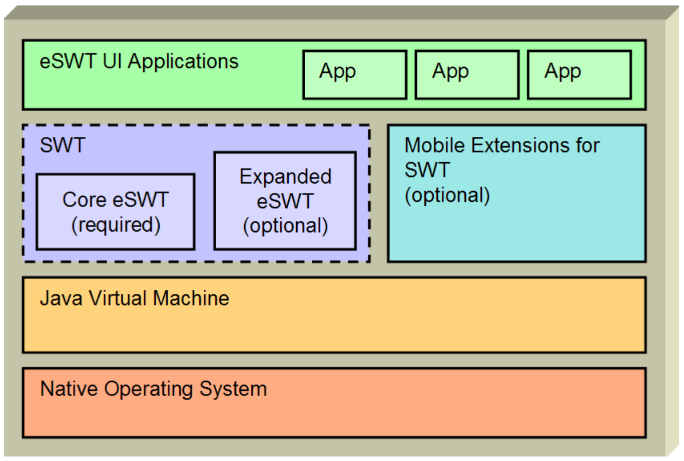


Figure 1-2 The eSWT UI architecture

The eSWT Mobile Extensions is an optional set that contains the following embedded specific features:

- eSWT Core

  Contains fundamental user interface elements, including low-level graphics, events, and basic widget infrastructure. This portion of eSWT is device-agnostic and applicable to a wider range of embedded devices. This component is a strict subset of SWT.

- eSWT Expanded

  An optional package that provides more sophisticated user interface elements that are found commonly on high-end mobile devices and PDAs. This component is a subset of SWT.

- eSWT Mobile Extension

  An optional component that provides user interface elements that are found commonly on mobile devices. Although many of the mobile device requirements are fulfilled already by eSWT, this package focuses on critical features that are not defined currently by eSWT.

1.4.3 The embedded JFace

JFace is a user interface toolkit that handles common user interface controls and tasks. It is a pure Java implementation that works with SWT directly, regardless of the Windows operating system. JFace includes *Action* to support shared user interface resources, such as menus, tool bars, and status lines. *Operation* provides support for long-running operations, *Preferences* provide a preference framework, and *Resources* support fonts and images resource management. *Util* provides useful utilities that are used throughout JFace, including property change events, a listener list implementation, and runtime checked assertions. *Viewers* contains model-based content adapters for SWT widgets, providing common behavior and higher level operations than available in SWT.

The embedded JFace (eJFace) is a pure subset of the JFace library. It provides advanced user interface capabilities for embedded devices, and it excludes functions that are either desk-top oriented or too large to provide reasonable benefits for the footprint that is consumed.

eJFace depends on the eSWT: Core and eSWT: Expanded portions of eSWT and J2ME CDC profile.

1.4.4 The embedded Workbench

The workbench provides user interface functionality and supplies an infrastructure in which tools can interact with users. In the workbench plug-in model, applications can run simultaneously inside a single workbench window that controls where and when applications are displayed. The workbench application model is aligned closely with Eclipse IDE workbench model. Workbench plug-ins generally provide one or more views.

One difference between the IDE workbench plug-ins and eRCP plug-ins is that eRCP plug-ins do not provide perspectives. Furthermore, the generic eWorkbench is a stand-alone application that owns the JVMs GUI thread and manages the launch and display of all eRCP workbench applications. It can be used as the basis for more advanced workbenches which take advantage of particular hardware features. For example, a mobile phone with multiple displays can display some limited information about the front of the phone and a full or normal view on the larger display. The generic eWorkbench initially displays a list of available eWorkbench applications. When a user selects one of these available applications, the application launches and displays. There is also a command for the user to switch back to application list.

1.4.5 eUpdate

eUpdate provides the update logic and user interface plug-ins for plug-ins that want to contain update capabilities. Application developers can either write their own update functionality with the help of these plug-ins or an eUpdate workbench application can provide a complete GUI for users to configure update sites. In addition, update status information is contained within this workbench application for user awareness.

2

eSWT fundamentals

This chapter provides an overview of the fundamentals that are contained within the embedded Standard Widget Toolkit (eSWT). It explains how eSWT is structured and analyzes its components. It then walks through each of these fundamental components, which behave as building blocks for other eSWT components. This chapter introduces you to the eSWT programming model and explains how you can start writing a simple eSWT application with some mandatory elements.

2.1 eSWT packaging

eSWT consists of three *physical* package sets:

- eSWT core
- eSWT expanded
- eSWT mobile extensions

The core and expanded packages are a strict subset of Standard Widget Toolkit (SWT). The mobile extensions component is a newly created package in embedded Rich Client Platform (eRCP). It enables the creation of common applications and is modeled after the interfaces of typical mobile devices, such as phones and PDAs.

Note: In terms of what we mean by *physical*, these three packages can be separated and put onto devices depending on functionality desire.

2.1.1 eSWT core

eSWT core is the mandatory component that must reside on a device. It is strictly a subset of SWT and provides the fundamental user interface elements such as display, shell, widget, controls, and events or listeners. Table 2-1 shows the packages and classes or interfaces that are contained in eSWT core.

Table 2-1 Packages and classes/interfaces in eSWT core

Package	Class/Interfaces	Description
org.eclipse.swt.widgets	Button	A selectable user interface object that issues notifications when pressed and released
	Canvas	A surface for drawing arbitrary graphics
	Combo	A control that allows the user to choose an item from a list of items, or optionally enter a new value by typing it into an editable text field
	Composite	A control that can contain other controls
	Control	Abstract superclass of all windowed user interface classes
	Decorations	It provides the appearance and behavior of shells, but are not top level shells or dialogs
	Dialog	Abstract superclass of the classes that represent the built in platform dialogs
	Display	It is responsible for managing the connection between SWT and the underlying operating system, also implements the SWT event loop in terms of the platform event model
	Event	It provides a description of a particular event which occurred within SWT
	FileDialog	It allows the user to navigate the file system and select or enter a file name
	Item	Abstract superclass of all non-windowed user interface objects that occur within specific controls

Package	Class/Interfaces	Description
org.eclipse.swt.widgets	Label	It represents a non-selectable user interface object that displays a string or image
	Layout	A layout controls the position and size of the children of a composite widget, it is the abstract base class for layouts
	List	It represents a selectable user interface object that displays a list of strings and issues notification when a string selected
	Menu	User interface objects that contain menu items
	MenuItem	It represents a selectable user interface object that issues notification when pressed and released
	MessageBox	It is used to inform or warn the user
	ProgressBar	It represents an unselectable user interface object that is used to display progress, typically in the form of a bar
	Scrollable	Abstract superclass of all classes which represent controls that have standard scroll bars
	Scrollbar	Selectable user interface objects that represent a range of positive, numeric value
	Shell	It represents the windows which the desktop or window manager is managing
	Slider	Selectable user interface objects that represent a range of positive, numeric values
	Synchronizer	It provides synchronization support for displays. A default instance is created automatically for each display, and this instance is sufficient for almost all applications
	Text	Selectable user interface objects that allow the user to enter and modify text
	TypedListener	Instances of this class are *internal SWT implementation* objects that provide a mapping between the typed and untyped listener mechanisms that SWT supports

Package	Class/Interfaces	Description
org.eclipse.swt.graphics	Color	Instances of this class manage the operating system resources that implement SWTs RGB color model
	Device	Abstract superclass of all device objects, such as the Display device
	Font	Instances of this class manage operating system resources that define how text looks when it is displayed
	FontData	Instances of this class describe operating system fonts
	FontMetrics	Instances of this class provide measurement information about fonts including ascent, descent, height, leading space between rows, and average character width
	GC	It is where all of the drawing capabilities that are supported by SWT are located
	Image	Instances of this class are graphics which have been prepared for display on a specific device
	ImageData	Instances of this class are device-independent descriptions of images
	PaletteData	Instances of this class describe the color data used by an image
	Point	Instances of this class represent places on the (x, y) coordinate plane
	Rectangle	Instances of this class represent rectangular areas in an (x, y) coordinate system
	Resource	Abstract superclass of all graphics resource objects
	RGB	Instances of this class are descriptions of colors in terms of the primary additive color model (red, green, and blue)

Package	Class/Interfaces	Description
org.eclipse.swt.events	ControlEvent/ ControlListener	Deal with the events that are generated by moving and resizing controls
	DisposeEvent/ DisposeListener	Deal with the event that is generated when a widget is disposed
	FocusEvent/ FocusListener	Deal with the events that are generated as controls gain and lose focus.
	KeyEvent/ KeyListener	Deal with the events that are generated as keys are pressed on the system keyboard
	MenuEvent/ MenuListener	Deal with the hiding and showing of menus
	ModifyEvent/ ModifyListener	Deal with the events that are generated when text is modified
	MouseEvent/ MouseListener/ MouseMoveListener	Deal with the events that are generated as mouse buttons are pressed or mouse move
	PaintEvent/ PaintListener	Deal with the events that are generated when the control needs to be painted
	SelectionEvent/ SelectionListener	Deal with the events that are generated when selection occurs in a control
	ShellEvent/ ShellListener	Deal with changes in state of Shells
	TraverseEvent/ TraverseListener	Deal with the events that are generated when a traverse event occurs in a control
	TypedEvent	Super class for all typed event classes provided by SWT
	VerifyEvent/ VerifyListener	Deal with the events that are generated when text is about to be modified
org.eclipse.swt.layout	FormLayout/ FormData/ FormAttachment	Instances of this class control the position and size of the children of a composite control by using `FormAttachments` to optionally configure the left, top, right, and bottom edges of each child
org.eclipse.swt	SWT	It provides access to a small number of SWT system-wide methods, and in addition defines the public constants provided by SWT
	SWTException	This runtime exception is thrown whenever a recoverable error occurs internally in SWT
	SWTError	This error is thrown whenever an unrecoverable error occurs internally in SWT

2.1.2 eSWT expanded

eSWT expanded is an optional component that contains more sophisticated user interface elements and layouts. This functionality is found commonly on high-end mobile devices and PDAs. Table 2-2 lists the packages and classes or interfaces that are contained in eSWT expanded.

Table 2-2 Packages and classes/interfaces in eSWT expanded

Package	Class/Interfaces	Description
org.eclipse.swt.widgets	ColorDialog	Allows the user to select a color from a predefined set of available colors
	DirectoryDialog	Allows the user to navigate the file system and select a directory
	FontDialog	Allows the user to select a font from all available fonts in the system
	Table/ TableColumn TableItem	A selectable user interface object that displays a list of images and strings and issue notification when selected
	Tree/ TreeItem	A selectable user interface object that displays a hierarchy of items and issue notification when an item in the hierarchy is selected
org.eclipse.swt.browser	Browser	Implements the browser user interface metaphor. Allows the user to visualize and navigate through HTML documents
	LocationEvent/ LocationListener	Deal with the events that are generated when a Browser navigates to a different URL
	ProgressEvent/ ProgressListener	Deal with the events that are generated during the loading of the current URL or when the loading of the current URL has been completed
	StatusTextEvent/Sta tusTextListener	Deal with the events that are generated when the status text for a Browser needs to be updated
	TitleEvent/ TitleListener	Deal with the events that are generated when the title of the current document is available or when it is modified
org.eclipse.swt.dnd	ByteArrayTransfer	Provides a platform specific mechanism for converting a Java byte[] to a platform specific representation of the byte array and vice versa
	Clipboard	Provides a mechanism for transferring data from one application to another or within an application
	TextTransfer	Provides a platform specific mechanism for converting plain text represented as a java String to a platform specific representation of the data and vice versa
	Transfer	Provides a mechanism for converting between a Java representation of data and a platform specific representation of data and vice versa
	TransferData	A platform specific data structure for describing the type and the contents of data being converted by a transfer agent
org.eclipse.swt.graphics	ImageLoader	Used to load images from and save images to a file or stream

2.1.3 eSWT mobile extensions

eSWT mobile extensions is an optional package that provides user interface elements that are found commonly on mobile devices. Table 2-3 lists the packages and classes or interfaces that are contained in eSWT mobile extensions.

Table 2-3 Packages and classes/interfaces in eSWT mobile extensions

Package	Class/Interfaces	Description
org.eclipse.ercp.swt.mobile	CaptionedControl	Used to display a label (caption) in front of a control. An optional trailing text can be used after the control
	Command	A metaphor that represents a general action
	ConstrainedText	A single-line Text control which constrains the user input by styles
	DateEditor	A special data entry control that allows users to enter or choose a date
	HyperLink	Represents a selectable user interface object that launches other applications when activated by the user
	Input	Instances of this class represent key based input features
	ListBox/ ListBoxItem	Represents a selectable user interface object that displays a list of items consisting of text and icons from a data model
	ListView	A widget that allows the user to select one or more items from a collection of items that can be displayed in a multi-column way with different styles
	MobileDevice/ MovileDeviceEvent/ MobileDeviceListener	Instances of this class represent the device that is being used. It provides methods that enable applications to learn more about the device specific characteristics and capabilities
	MobileShell	A shell particularly suitable for devices that require dynamic change of trims at runtime
	MultiPageDialog	Instances of this class represents a tabbed dialog
	QueryDialog	A modal window used to prompt the user for data input
	Screen/ ScreenEvent/ ScreenListener	Instances of this class represent display screens available for application use
	SortedList	Represents a selectable user interface object that displays a sorted list of text items
	TaskTip	Provides feedback to the user about the state of a long-running task
	TextExtension	Contains methods for extending the functionality of the Text control
	TimedMessageBox	A modal window that is used to inform the user of limited information using a standard style

2.2 Creating a complete eSWT program

Example 2-1 produces a complete eSWT program that creates an empty window that displays the message `HelloWorld from eSWT` on the application title bar.

Example 2-1 HelloWorld application in eSWT

```
import org.eclipse.swt.widgets.*;

public class HelloWorldeSWT {

   public static void main(String[] args) {
      Display display = new Display();
      Shell shell = new Shell(display);
      shell.setSize(300,300);
      shell.setText("HelloWorld from eSWT");
      shell.open();

      while (!shell.isDisposed()) {
         if (!display.readAndDispatch())
            display.sleep();
      }
      display.dispose();
   }
}
```

The org.eclipse.swt.widgets package contains the most commonly used widgets in eSWT applications. In `main(String[] args])`, every eSWT application needs to create a display to build up a connection between the eSWT application and the underlying graphics system. Next, we create a shell and set the shell's size and title.

Shells that are created based on the display are referred to as *top-level windows*. The `shell.open()` method makes the shell visible to the user. Inside the `while ()` loop, we establish an explicit loop that reads and dispatches user events from the operating system. If no more events are available, `display.sleep()` is called and goes to sleep waiting for the next event. The `display.dispose()` call at the last line of code explicitly disposes the display and releases any associated resources from the eSWT application.

Note: Even though resources are disposed automatically when an application exits, it is considered good programming style to release all widgets that you create manually.

Figure 2-1 shows the result of the HelloWorld application running on Windows XP. The size of the shell that we set is 300x300.

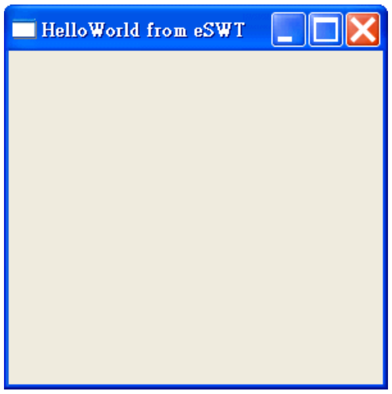


Figure 2-1 Result of the HelloWorld application running on Windows XP

Figure 2-2 shows the same application running on Pocket PC. eSWT uses underlying graphics systems to draw its user interface elements. The user cannot tell the native applications apart because they look and feel very similar. In this case, the only visual difference is that on Pocket PC, eSWT maximizes the top-level shell for a consistent user interface behavior with native applications. So, `shell.setSize()` does not take effect for top-level windows on Pocket PC.

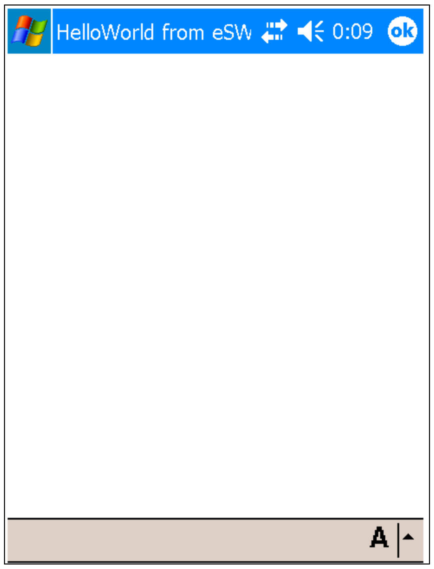


Figure 2-2 Result of the HelloWorld application running on Pocket PC

2.3 The Display class

The `Display` class is responsible for managing the connection between SWT and the underlying window system. It implements the SWT event loop in terms of the system event loop. The display provides other important methods for inter-thread communications, timers, and access to resources of operating system.

The display in SWT is not `Widget`, although many concepts are shared between these classes. On the whole, `Display` is a very powerful class that abstracts common operating system APIs to provide a way to access low-level resources such as fonts, colors, or other system attributes.

2.3.1 Events and listeners

The `Display` sends events using *untyped* listeners. There are two methods that are related to listeners defined in `Display`:

- `addListener(int event, Listener listener)`

  Adds the listener to the collection of listeners who are notified when an event of the given type occurs. When the event does occur in the display, the listener is notified by the `handleEvent()` message.

- `removeListener(int event, Listener listener)`

  Removes the listener from the collection of listeners who are notified when an event of the given type occurs.

2.3.2 Event filters

Event filters acts similar to a global listener. To filter an event and then do something works similar to an event listener. Filters run before all the other event listeners, which gives an application an opportunity to modify an event's attribute or just stop issuing the event. There are two methods that are related to filters that are defined in `Display`:

- `addFilter(int eventType, Listener listener)`

  Adds the listener to the collection of listeners who are notified when an event of the given type occurs anywhere in this display. When the event does occur, the listener is notified by the `handleEvent()` message.

- `removeFilter(int eventType, Listener listener)`

  Removes the listener from the collection of listeners who are notified when an event of the given type occurs.

Example 2-2 shows how to add a filter via the `addFitler()` method. It adds a filter that listens to the `SWT.FocusIn` event of all controls. In `handleEvent()`, we output the widget that triggers the event. On the whole, the `addFilter()` method is a global listener of a specified event.

Example 2-2 addFilter snippet

```
display.addFilter(SWT.FocusIn, new Listener(){

            public void handleEvent(Event event) {
               System.out.println("Focused widget: "+event.widget);
            }});
```

Note: Pay attention to using filters. Filters are powerful because they grab all events, but on the other hand, they block all controls' listeners as well. For example, if you have a time-consuming operation for SWT.FocusIn inside `handleEvent()`, all controls get focused slowly, which definitely impacts usability. Note, however, that even filters should be used sparingly.

2.3.3 Shells and focus control

The `Display` keeps a list of shells that are created upon it, including the active shell that is ready for user manipulation. Furthermore, it keeps focus control information. There are three methods that are related to shells and to focus control:

- `getActiveShell()`

  Returns the currently active shell, or `null` if no shell is active that belongs to the currently running application.

  Example 2-3 demonstrates how we get active shell and then set its title.

Example 2-3 getActiveShell() snippet

```
Shell shell = display.getActiveShell();
shell.setText(“Active Shell“);
```

- `getShells()`

  Returns a (possibly empty) array that contains all shells that have not been disposed and that have the receiver as their display.

  Example 2-4 demonstrates how we get all shells and then output their titles via walking through the array of shells.

Example 2-4 getShells() snippet

```
Shell[] shells = display.getShells();
for (int i=0; i< shells.length; i++) {
    System.out.println(“title of shell “+”i”+” is “+shells[i].getText());
}
```

- `getFocusControl()`

  Returns the control that has keyboard focus currently, or `null` if keyboard events are not going to any of the controls that are built by the currently running application.

2.3.4 System information

The `Display` can be used to retrieve system information (resources such as display depth, DPI®, fonts, and colors). These objects are available for use but owned by the operating system.

Note: Do not try to release these objects via a method similar to the `dispose()` method because this might injure the operating system with unpredictable results.

There are several methods that are related to system information:

- `getDepth()`

  Returns the bit depth of the screen that is the number of bits that it takes to represent the number of unique colors that the screen is capable of displaying currently. This number is typically one of 1, 8, 15, 16, 24, or 32. `getDepth()` is not defined in `Display` but is inherited from `Device`.

- `getDPI()`

  Returns a point whose x coordinate is the horizontal dots per inch of the display and whose y coordinate is the vertical dots per inch of the display. `getDpi()` is not defined in `Display` but is inherited from `Device`.

- `getIconDepth()`

  Returns the maximum allowed depth of icons on this display in bits per pixel. On some platforms, this value might be different than the actual depth of the display.
- `getSystemColor()`

  Returns the matching standard color for the given constant, which should be one of the color constants that is specified in class SWT. System colors are pre-allocated, and because you do not allocate these colors, they should not be discarded or disposed. Example 2-5 shows how to get a `Color` object of white in system.

Example 2-5 getSystemColor() snippet

```
Color white = display.getSystemColor(SWT.COLOR_WHITE);
```

2.4 Composite and shell

This section presents an overview of composites and shells.

2.4.1 Composite

`Composite` is a control that can be used to contain other controls. It acts as a container with layout to position its children automatically.

Composite style

Table 2-4 shows what styles can be used to construct a `Composite` instance.

Table 2-4 Composite styles

Style	Description
SWT.H_SCROLL	Create a horizontal scroll bar
SWT.V_SCROLL	Create a vertical scroll bar
SWT.NO_BACKGROUND	Do not fill the background when painting
SWT.NO_REDRAW_RESIZE	Do not redraw when resize event happens
SWT.NO_MERGE_PAINTS	Do not merge invalidated rectangles when painting
SWT.NO_FOCUS	Do not take focus
SWT.NO_RADIO_GROUP	Disable radio button behavior

Getting the children

`Composite` keeps a list of all its children. Applications can use `getChildren()` to query this information using the following:

- `getChildren()`

  Returns a (possibly empty) array that contains the receiver's children. Children are returned in the order that they are drawn. `getChildren()` is a copy of the array of children and takes no effect on real children that are contained within `Composite` if an application tries to modify the array.

Example 2-6 shows how to print out each control that is contained in an instance of `Composite`.

Example 2-6 snippet of getChildren()

```
Control[] children = composite.getChildren();
for (int i=0; i<children.length; i++) {
   System.out.println("Child "+i+" is "+children[i]);
}
```

Getting the tab traversal order

Applications can explicitly get or set the tab traversal order within a `Composite`. There are two methods that are related to tab traversal:

- `getTabList()`

  Gets the (possibly empty) tabbing order for the control.

- `setTabList(Control[] tablist)`

  Sets the tabbing order for the specified controls to match the order that they occur in the argument list.

Layout

`Composite` can set layout related data to help in positioning and sizing its children according to the characteristic of `Layout`.

2.4.2 Shell

Instances of `Shell` represent a window that a desktop or window manager is controlling. Instances without a parent, where only one instance of `Display` is passed in as argument, are called *top-level* shells. Shells that are parented under another shell are called *dialog* shells.

Note: Top-level shells are maximized at windows mobile device, because this behavior is similar with other native applications. `MobileShell` contained within the mobile extensions component of eSWT should be used on mobile devices because it works similar to `Shell` but provides addition embedded specific features such as full-screen support.

Shell styles

Table 2-5 shows styles used in `Shell`.

Table 2-5 Shell styles

Style	Description
SWT.BORDER	Provide a border around the shell
SWT.CLOSE	Provide a close button
SWT.MIN	Provide a minimize box
SWT.MAX	Provide a minimize box
SWT.RESIZE	Provide a resizable border
SWT.TITLE	Provide a title bar
SWT.NO_TRIM	Force the shell to have no trim
SWT.APPLICATION_MODAL	Make the shell application

Style	Description
SWT.MODELESS	Make the shell modeless
SWT.PRIMARY_MODAL	Make the shell primary modal
SWT.SYSTEM_MODAL	Make the shell system modal

Shell events

Applications can use the following types of `Shell` events via `ShellListener`:

- `SWT.Activate`

  Sent whenever the shell is activated.
- `SWT.Deactivate`

  Sent whenever the shell is inactive.
- `SWT.Iconfy`

  Sent whenever the shell is iconized.
- `SWT.Deiconify`

  Sent whenever the shell is no longer iconized.
- `SWT.Close`

  Sent whenever the shell is closed.

Opening and closing the shell

Unlike other controls in eSWT, shells are invisible when created. Thus, applications have to make shells visible so that the shell is usable. You can use the `open()` method to bring the shell in front of user. The `open()` method moves the shell to the top of the drawing order for the display on which it was created. All other shells on that display, which are not this shell's children, are drawn behind it. The `open()` method also marks the shell as visible, sets the focus, and asks the window manager to make the shell active.

Users can close the shell by tapping the close box of the shell. Alternatively, the application can close the shell programatically using the `close()` method. The `close()` method requests that the window manager close the shell in the same way that the window would close if the user clicks the *close box* or performs some other platform specific key or mouse combination that indicates the window should be removed.

Using MenuBar

`MenuBar` provides a means for a user to manipulate shortcut functionalities while traversing in menus. `Menu` is discouraged for use in mobile device because `Command` in the mobile extensions component provides a more sophisticated way to handle menus. For example, `Command` can map its action to configurable hardware button on device automatically, which provides the user a handy means to trigger a default action.

2.5 Events and listeners

An *event* is an indication that something has happened. It could be issued as *key down* or *mouse release* when the user presses a key or releases the mouse. The `Event` class contains detailed information about the event, such as `widget` to represent which widget triggered the event, `doit` to represent if the event should be performed or not, `type` to represent the event type, and so forth. The triggered event is delivered to application via a *listener*.

An instance of listener provides an interface to implement the methods that are invoked when appropriate. Listener methods take an instance of an event as an argument.

eSWT has two types of listeners: *untyped* and *typed*.

2.5.1 Untyped listeners

Untyped listeners provide a generic way to monitor any type of events that are defined in eSWT. In terms of Java classes, only two Java classes are used in untyped listeners: org.eclipse.swt.widgets.Listener and org.eclipse.swt.widgets.Event.

The following methods are related to untyped listeners:

- `addListener(int event, Listener listener)`

  Adds the listener to the collection of listeners who are notified when an event of the given type occurs. When the event does occur in the widget, the listener is notified by the `handleEvent()` message.

- `notifyListener(int type, Event event)`

  Notifies all of the receiver's listeners for events of the given type that one such event has occurred by invoking their `handleEvent()` method. Applications can use this method to send a specific type of event; however, calling `notifyListener()` does not cause real operating system event happens.

- `removeListener(int type, Listener listener)`

  Removes the listener from the collection of listeners who are notified when an event of the given type occurs.

Note: In order to remove a listener, you should remove the *exact* listener that you added. If you added an instance of listeners multiple times, you should also remove that instance if you do not want the listener to take effect any more.

Table 2-6 shows event types along with description of all untyped events that are defined in eSWT.

Table 2-6 Untyped events

Event Type	Description
`SWT.KeyDown`	A key was pressed
`SWT.KeyUp`	A key was released
`SWT.MouseDown`	A mouse button was pressed
`SWT.MouseUp`	A mouse button was released
`SWT.MouseMove`	The mouse was moved
`SWT.MouseEnter`	The mouse entered the client area of control
`SWT.MouseDoubleClick`	A mouse button was double clicked
`SWT.Paint`	A control was asked to paint
`SWT.Move`	Control’s position changed
`SWT.Resize`	The size of control’s client area changed
`SWT.Dispose`	The control was disposed

Event Type	Description
`SWT.Selection`	The widget was selected
`SWT.DefaultSelection`	The default selection event happened in the widget
`SWT.FocusIn`	The widget gets focus
`SWT.FocusOut`	The widget lost focus
`SWT.Expand`	A tree item was expanded
`SWT.Collapse`	A tree item was collapsed
`SWT.Iconfy`	The shell was minimized
`SWT.Deiconfy`	The shell goes back to its original size before iconized
`SWT.Close`	The shell is closed
`SWT.Show`	The widget becomes visible
`SWT.Hide`	The widget is hidden
`SWT.Modify`	Content has been changed in the control
`SWT.Verify`	Text is to be verified in the control
`SWT.Active`	The control is activated
`SWT.Deactive`	The control is deactivated
`SWT.MenuDetect`	The user requests a context menu
`SWT.Traverse`	A keyboard navigation happened
`SWT.HardKeyDown`	A hardware button is pressed
`SWT.HardKeyUp`	A hardware button is released

2.5.2 Typed listeners

Typed listeners provide a more meaningful listener class with corresponding events. Some agreed upon methods are defined for each listener interface for applications to implement. For example, to listen for selection on a widget, you can use the following two methods:

- `addSelectionListener(SelectionListener listener)`

  Adds the listener to the collection of listeners who are notified when the receiver's selection changes by sending it one of the messages that are defined in the `SelectionListener` interface.

  The `widgetSelected()` method is called when the combo's list selection changes. The `widgetDefaultSelected()` method is called typically when Enter is pressed the combo's text area.

- `removeSelectionListener(SelectionListener listener)`

  Removes the listener from the collection of listeners who are notified when the receiver's selection changes. The listener to be removed should be the *exact* one you that added with `addSelectionListener()`.

Table 2-7 shows all typed events that are supported in eSWT. Some are not defined in SWT because they are newly created in mobile extension.

Table 2-7 Typed events

Event	Listener	Methods

`ControlEvent`	`ControlListener`	`controlMoved(ControlEvent)`
		`controlResized(ControlEvent)`

`DisposeEvent`	`DisposeListener`	`widgetDisposed(DisposeEvent)`

`FocusEvent`	`FocusListener`	`focusGained(FocusEvent)`
		`focusLost(FocusEvent)`

`KeyEvent`	`KeyListener`	`keyPressed(KeyEvent)`
		`keyRelease(KeyEvent)`

`MenuEvent`	`MenuListener`	`menuHidden(MenuEvent)`
		`menuShown(MenuEvent)`

`Modifyevent`	`ModifyListener`	`modifyText(ModifyEvent)`

`MouseEvent`	`MouseListener`	`mouseDoubleClick(MouseEvent)`
		`mouseDown(MouseEvent)`
		`mouseUp(MouseEvent)`

`MouseEvent`	`MouseMoveListener`	`mouseMove(MouseEvent)`
`PaintEvent`	`PaintListener`	`paintControl(PaintEvent)`

`SelectionEvent`	`SelectionListener`	`widgetDefaultSelected(SelectionEvent)`
		`widgetSelected(SelectionEvent)`

`ShellEvent`	`ShellListener`	`shellActivitated(ShellEvent)`
		`shellClosed(ShellEvent)`
		`shellDeactivated(ShellEvent)`
		`shellDeiconified(ShellEvent)`
		`shellIconfied(ShellEvent)`

`TraverseEvent`	`TraverseListener`	`keyTraversed(TraverseEvent)`

`TreeEvent`	`TreeListener`	`treeCollapsed(TreeEvent)`
		`treeExpanded(TreeEvent)`

`VerifyEvent`	`VerifyListener`	`verifyText(VerifyEvent)`

`MobileDeviceEvent`	`MobileDeviceListene r`	`deviceChanged(MobileDeviceEvent)`
		`inputChanged(MobileDeviceEvent)`
		`screenChanged(MobileDeviceEvent)`

`ScreenEvent`	`ScreenListener`	`screenActivated(ScreenEvent)`
		`screenDeactivated(ScreenEvent)`
		`screenOrientationChanged(ScreenEvent)`

2.6 Mouse

Mouse in eSWT is a piece of existing code that represents code in SWT. You should not use Mouse because applications will not get into your Mouse handling code in devices without a pointer mechanism such as a stylus.

Mouse events

There are typed events and untyped events that relate to Mouse. Table 2-8 shows untyped events, and Table 2-9 shows typed events that are provided by eSWT.

Table 2-8 Untyped events in Mouse

Event Type	Description
`SWT.MouseDown`	The mouse was pressed
`SWT.MouseUp`	The mouse was pressed
`SWT.MouseMove`	The mouse was moved
`SWT.MouseEnter`	The mouse entered a control
`SWT.MouseExit`	The mouse exited a control
`SWT.MouseDoubleClick`	The mouse was double clicked

Table 2-9 Typed events in Mouse

Typed Event	Listener	Methods

`MouseEvent`	`MouseListener`	`mouseDouvleClick(MouseEvent)`
		`mouseDown(MouseEvent)`
		`mouseUp(MouseEvent)`

`MouseEvent`	`MouseMoveListener`	`mouseMove(MouseEvent)`

Selection

Selection is a major action when users tap on controls. `SelectionEvent` is sent when some control is selected. `SelectionEvent` can happen with other events simultaneously. For example, when the user selects text, `SelectionEvent` and `FocusIn` events are triggered within a single click. Table 2-10 shows two selection events for untyped events, and Table 2-11 shows for typed events.

Table 2-10 Untyped Selection events

Untyped Event	Description
`SWT.Selection`	The user selected a control
`SWT.DefaultSelection`	The user did a default selection to a control, for example, press Enter in text.

Table 2-11 Typed Selection events

Typed Event	Listener	Methods

`SelectionEvent`	`SelectionListener`	`widgetDefaultSelected(SelectionEvent)`
		`widgetSelected(SelectionEvent)`

2.7 Keyboard

In this section, we discuss characters and keyboard input, including keyboard events, focus events, key events, and accelerators.

2.7.1 Keyboard events

When a control gets focus, all keyboard events go into the control until the focus is caught by another control. In addition to user operation to make a control obtain focus, the application can assign the focus to a control by using the `setFocus()` method. The `setFocus()` method causes the receiver to have the keyboard focus, such that all keyboard events are delivered to it. Not all controls can get focus. `Label` is one of them. For `Composite`, it tries to reassign focus to its first child, if one exists. Focus is not acquired in disabled or hidden control.

Applications can also use `isFocusControl()` to check if a control gets focus.

2.7.2 Focus events

eSWT provides some focus events. Table 2-12 lists the untyped events, and Table 2-13 lists the typed events.

Table 2-12 Untyped events for focus

Untyped Event	Description
`SWT.FocusIn`	The control gets focus
`SWT.FocusOut`	The control looses focus

Table 2-13 Typed event for focus

Typed event	Listener	Method

`FocusEvent`	`FocusListener`	`focusGained(FocusEvent)`
		`focusLost(FocusEvent)`

Example 2-7 demonstrates how to use `FocusListener`.

Example 2-7 FocusListener sample

```
import org.eclipse.swt.SWT;
import org.eclipse.swt.events.FocusEvent;
import org.eclipse.swt.events.FocusListener;
import org.eclipse.swt.layout.RowLayout;
import org.eclipse.swt.widgets.*;

public class FocusSample {

   public static void main(String[] args) {
      final Display display = new Display();
      final Shell shell = new Shell(display);
      shell.setLayout(new RowLayout());
      final Text text1 = new Text(shell, SWT.SINGLE | SWT.BORDER);
      text1.setText("text1");
      final Text text2 = new Text(shell, SWT.SINGLE | SWT.BORDER);
      text2.setText("text2");

      final Label label = new Label(shell, SWT.BORDER);
      FocusListener focusListener = new FocusListener(){
         public void focusGained(FocusEvent e) {
            if (e.widget == text1) {
               label.setText("text 1 gains focus");
            }
            if (e.widget == text2) {
               label.setText("text 2 gains focus");
            }
            shell.layout();
         }

         public void focusLost(FocusEvent e) {
         }};
      text1.addFocusListener(focusListener);
      text2.addFocusListener(focusListener);
      shell.setSize(300,300);
      shell.setText("Focus");
      shell.open();

      while (!shell.isDisposed()) {
         if (!display.readAndDispatch())
            display.sleep();
      }
      display.dispose();

   }
}
```

Example 2-7 creates two `Text` widgets and a `Label` widget. The content of `Label` changes along with the focus between the two `Text` widgets. Figure 2-3 shows the result of this example when `text1` gets focus.

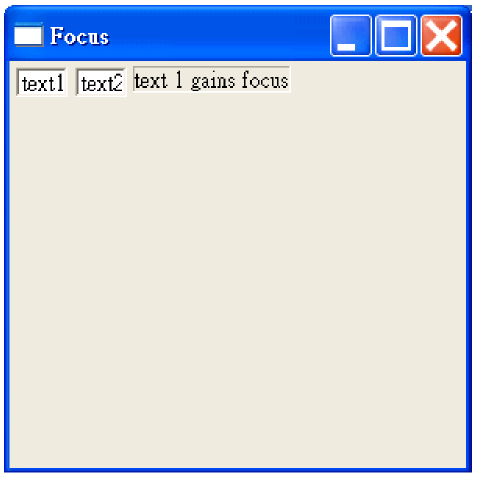


Figure 2-3 Focus effect when text1 gets focus

Figure 2-4 shows the result when `text2` gets focus.

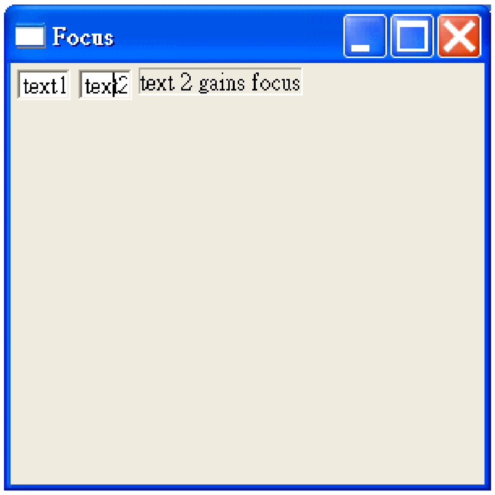


Figure 2-4 Focus effect when text2 gets focus

2.7.3 Key events

When a user presses or releases a key, key event is delivered to application. eSWT provides some key events. Table 2-14 shows the untyped key events and Table 2-15 shows typed key events.

Table 2-14 Untyped key events

Untyped Event	Description
`SWT.KeyDown`	A key was pressed
`SWT.KeyUp`	A key was released

Table 2-15 Typed key events

Typed Event	Listener	Methods
`KeyEvent`	`KeyListener`	`keyPressed(KeyEvent)` `keyReleased(KeyEvent)`

2.7.4 Accelerators

eSWT has `MenuItem` and `Command` to support accelerators. Accelerators simply represent keyboard shortcuts. Accelerators produce the same result when users tap on a menu item. When applications have accelerators in `MenuItem` or `Command`, they are global to the window. When users input an accelerator key, no matter which control gets focus, this combination of keys is consumed, and an accelerator action is triggered.

In eSWT, accelerators are defined as integer variables. Table 2-16 shows some accelerator examples.

Table 2-16 Accelerator example

Accelerator	Description
`SWT.CONTROL | 'C'`	Combination of Ctrl + C keys
`SWT.CONTROL | SWT.SHIFT | 'T"`	Combination of Ctrl + Shift + T keys

There are two methods that you can set to retrieve an accelerator of `MenuItem` and `Command`:

- `setAccelerator(int accelerator)`

  Sets the widget accelerator. An accelerator is the bit-wise OR of zero or more modifier masks and a key.

- `getAccelerator()`

  Returns the widget accelerator. An accelerator is the bit-wise OR of zero or more modifier masks and a key.

Note: However, the use of accelerators does not generate exceptions or unexpected failures for your application. It simply has no effect in the Pocket PC.

3

eSWT core

This chapter provides a detailed, API-level description of the components that are found within embedded Standard Widget Toolkit (eSWT) core that were not discussed in Chapter 2, “eSWT fundamentals” on page 9.

eSWT core is discussed first because it contains the most commonly used widgets in embedded Rich Client Platform (eRCP). Also, it provides a great introduction to learning about eSWT. If you are already familiar with SWT, then this chapter provides an easy-to-follow guide, due to similarities between the two toolkits.

This chapter discusses the following types of eSWT core components:

- Controls
  - Label
  - Button
  - Text
  - List
  - Combo box
  - Dialog window
  - MessageBox
  - FileDialog
  - Menu
  - ScrollBar and Slider widgets
  - ProgressBar
- Layouts
  - FormLayout

3.1 Controls

Controls are the basic building blocks in eSWT application programs. This section describes eSWT core controls.

3.1.1 Label

Labels are the simplest widgets in eSWT. They represent a non-selectable object that acts as a string, image, or separator. Labels used to adorn other eSWT widgets such as a dialog.

Example

Figure 3-1 demonstrates several different types of labels, including text, image, and separator based labels.

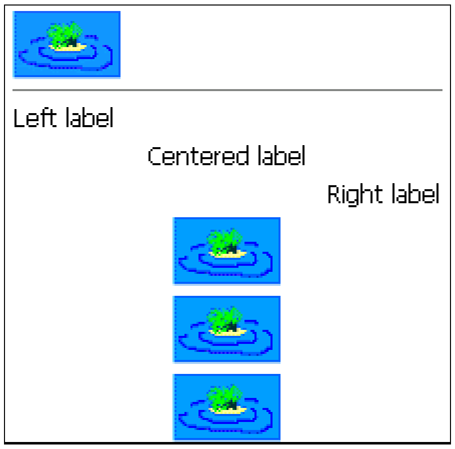


Figure 3-1 An example of a few label types

Styles

Table 3-1 describes label styles.

Table 3-1 Label styles

Style	Description
`SWT.SEPARATOR`	Draw a separator instead of text or image
`SWT.HORIZONTAL`	Draw a separator horizontally
`SWT.VERTICAL`	Draw a separator vertically
`SWT.SHADOW_IN`	Draw a separator with an inward shadow effect
`SWT.SHADOW_OUT`	Draw a separator with an outward shadow effect
`SWT.SHADOW_NONE`	Draw a separator with no shadow effects
`SWT.CENTER`	Center align a label
`SWT.LEFT`	Left align a label
`SWT.RIGHT`	Right align a label
`SWT.WRAP`	Wrap text to fit within the visible area

Events

Labels are non-selectable widgets (commonly known as *static components*) that take no action when focused or selected. Therefore, no events are available to use within a label.

Text and Images

Labels allow you to set a string (text) or image (but not both). You can use the following methods with labels:

- `setText(String string)`

  Sets the text of the label.
- `getText()`

  Returns the text of the label. Note that this is an empty string if the label is a separator, is an image, or has not been set at all.
- `setImage(Image image)`

  Sets the image of the label.
- `getImage()`

  Returns the image of the label or `null` if not set.

Alignment

Labels can be aligned with SWT.LEFT, SWT.CENTER, or SWT.RIGHT styles.

- `setAlignment(int alignment)`

  Sets the alignment of how the text or image is displayed. The valid styles are SWT.LEFT, SWT.CENTER, or SWT.RIGHT.
- `getAlignment()`

  Returns the alignment.

Snippets

A *snippet* is a minimal program that demonstrates a specific piece of functionality. Snippets are used throughout this book as method to understand widgets in a simple way. Example 3-1 is a snippet that creates a Label with a separator. Example 3-2 is a snippet that creates a label with text that is left aligned. Example 3-3 is a snippet that creates a label with an image that is center aligned.

Example 3-1 Label: Creating a Label with a separator

```
Label separatorLabel = new Label(composite, SWT.SEPARATOR);
```

Example 3-2 Label: Creating a Label with text, left aligned

```
Label textLabel = new Label(composite, SWT.LEFT);
textLabel.setText("A left-aligned label");
```

Example 3-3 Label: Creating a Label with an image, center aligned

```
Label imageLabel = new Label(composite, SWT.NONE);
Image image = new Image(composite.getDisplay(), getClass().getResourceAsStream("img.gif");
imageLabel.setImage(image);
```

3.1.2 Button

Buttons are commonly found in desktop applications because desktop computers have pointer devices (such as a mouse). Pointer devices allow users to select buttons and handle selection events.

Tip: The use of buttons is discouraged because most mobile devices are not paired with a pointer device. Therefore, buttons on these devices must be selected using controls such as arrow keys. To get around this issue, you can check via the `MobileDevice` class to see if the screen is a touch screen and create buttons when appropriate.

Example

Figure 3-2 demonstrates the appearance of a button.

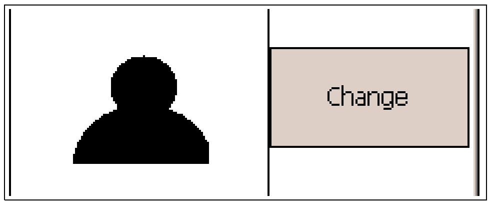


Figure 3-2 A sample button and label

Styles

Table 3-2 describes button styles.

Table 3-2 Button styles

Style	Description
`SWT.CHECK`	Create a check button
`SWT.PUSH`	Create a push button
`SWT.RADIO`	Create a radio button
`SWT.TOGGLE`	Create a toggle button
`SWT.FLAT`	Draw a button with a flat look
`SWT.LEFT`	Left align a button
`SWT.RIGHT`	Right align a button
`SWT.CENTER`	Center align a button

Note: Only one of the button styles: `SWT.CHECK`, `SWT.PUSH`, `SWT.RADIO` and `SWT.TOGGLE` can be specified. This also applies to the alignment styles: `SWT.LEFT`, `SWT.CENTER` and `SWT.RIGHT`.

Event

Table 3-3 describes the button event.

Table 3-3 Button event

Event	Description
SWT.Selection	The button was selected

The `SWT.Selection` event is sent whenever a user interacts with a button (via a click or a keyboard selection). There are two ways to listen for selections on a button:

- Using the convenience method found on the `Button` class (Example 3-4)
- Using the `addListener(int event, Listener listener)` method from the inherited `Widget` class.

Example 3-4 Button: Adding a listener to a button via the convenience method

```
button.addSelectionListener(new SelectionListener() {
   public void widgetDefaultSelected(SelectionEvent e) {}
   public void widgetSelected(SelectionEvent e) {
      System.out.println("The button was pressed!");
   }
});
```

Note: To listen for radio button events, you have to get the selection before performing your desired action.

Text and images

Buttons allow you to set a string (text) or image (but not both). You can use the following methods with buttons:

- `setText(String string)`

  Sets the text of the button.
- `getText()`

  Returns the text of the button. Note that this is an empty string if the button is an image or has not been set at all.
- `setImage(Image image)`

  Sets the image of the button.
- `getImage()`

  Returns the image of the label or `null` if not set.

Alignment

Buttons can be aligned with SWT.LEFT, SWT.CENTER, or SWT.RIGHT styles. The use of button alignment is discouraged, because it can cause the buttons to appear out of place. You can use the following methods with button alignment:

- `setAlignment(int alignment)`

  Sets the alignment of how the text or image is displayed. The valid styles are SWT.LEFT, SWT.CENTER, or SWT.RIGHT.
- `getAlignment()`

  Returns the alignment.

Snippets

Example 3-5 illustrates how a simple push button is created.

Example 3-5 Creating a simple push button

```
Button pushButton = new Button(composite, SWT.PUSH);
pushButton.setText("Push me!");
```

3.1.3 Text

Text widgets allow users to edit strings. These strings can range from multi-line text to even password fields. Text widgets are selectable which means a user has control over what characters are selected. If no characters are selected, it is indicated by a caret (a thin vertical stripe).

Example

Figure 3-3 displays a couple of captioned controls along with corresponding text widgets. Text widgets are perfect for user input.

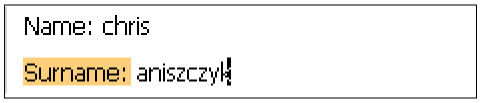


Figure 3-3 A couple of captioned controls with corresponding text widgets

Styles

Table 3-4 describes text styles.

Table 3-4 Text styles

Style	Description
`SWT.SINGLE`	Text spans a single line
`SWT.MULTI`	Text spans multiple lines
`SWT.PASSWORD`	Hide text input via an echo character
`SWT.READ_ONLY`	Text cannot be edited by the user
`SWT.WRAP`	Text is wrapped if it does not fit in the control
`SWT.LEFT`	Text is left aligned
`SWT.CENTER`	Text is centered
`SWT.RIGHT`	Text is right aligned

Note: Only one of the styles, `SWT.SINGLE` or `SWT.MULTI`, can be specified.

Events

The following text events are available:

- `SWT.DefaultSelection`

  This event is only relevant to single-line text controls. It commonly happens when a user presses Enter (or the equivalent) as input.

- `SWT.Modify`

  This event is only reached **after** text has been entered by a user or programmatically modified. A common use of `SWT.Modify` is to warn a user of invalid input. An example within Eclipse would be when you try to create a package with a lowercase starting character, Eclipse warns you that it is bad practice to have a lowercase starting character.

- `SWT.Verify`

  This event is only reached **before** text has been entered by a user or programmatically modified. A common use of `SWT.Verify` is to restrict a user's input to a set of valid characters. An example of this would be a text control which restricts the user to only input numbers valid to a ZIP code.

Table 3-5 describes the text events.

Table 3-5 Text events

Event	Description
`SWT.DefaultSelection`	The default selection event type
`SWT.Modify`	Text has been modified
`SWT.Verify`	Text needs to be validated

Snippets

Example 3-6 shows how to create a simple single line text widget for password input.

Example 3-6 Text: Prompt a user for their password

```
Text passwordText = new Text(composite, SWT.SINGLE | SWT.PASSWORD);
```

3.1.4 List

A list is used to represent a set of strings that can be used in a single or multi-select manner.

Example

Figure 3-4 shows a single-selection list with a few sample values.

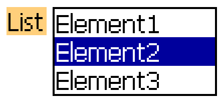


Figure 3-4 A sample list

Styles

Table 3-6 describes list styles.

Table 3-6 List styles

Style	Description
`SWT.SINGLE`	A list where only one item can be selected
`SWT.MULTI`	A list where multiple items can be selected

Note: Only one feature, `SWT.SINGLE` or `SWT.MULTI`, can be specified at a time.

Events

Table 3-7 describes list events.

Table 3-7 List events

Event	Description
SWT.DefaultSelection	The user double clicked on an item
SWT.Selection	The user selected an item

There are only two events found within `List` and both deal with selection. Example 3-7 prints the user's selection based on a double-click.

Example 3-7 Printing a user selection on a double click

```
List list = new List(composite, SWT.SINGLE);
list.add("A");
list.add("B");
list.addListener(SWT.DefaultSelection, new Listener() {
   public void handleEvent(Event e) {
      System.out.println(list.getSelection());
   }
});
```

Snippets

Example 3-8 creates a list with a few inserted items and allows for vertical scrolling.

Example 3-8 List: Creating a new single-select list with a few items (vertical scrolling)

```
List list = new List(composite, SWT.SINGLE | SWT.V_SCROLL);
list.add("Apples");
list.add("Oranges");
list.add("Bananas");
```

3.1.5 Combo box

Figure 3-5 demonstrates a captioned control with a combo box. In this particular example, the combo box is filled with values that represent various world languages.

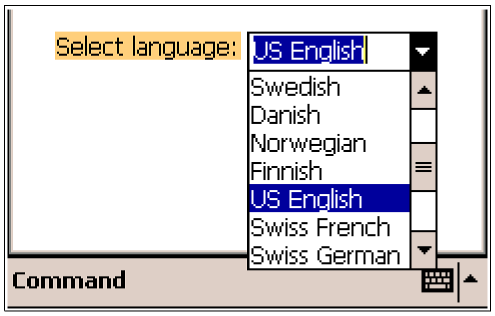


Figure 3-5 A language selection utility using a combo box

Styles

Table 3-8 describes combo styles.

Table 3-8 Combo styles

Style	Description
`SWT.DROP_DOWN`	The list becomes a drop down
`SWT.READ_ONLY`	The user cannot modify the list

Events

Table 3-9 describes combo events.

Table 3-9 Combo events

Event	Description
`SWT.DefaultSelection`	The user pressed Enter
`SWT.Modify`	The user selected an item
`SWT.Selection`	The text has been modified

Snippets

Example 3-9 contains a small snippet of code that creates a read-only combo box and populates the combo box with values.

Example 3-9 Combo: Create and populate a read-only combo

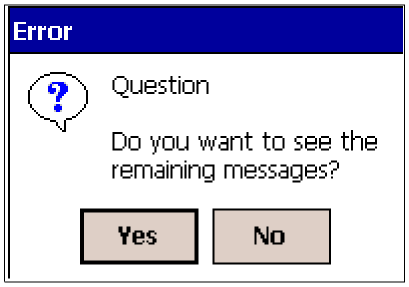
```
Combo combo = new Combo(composite, SWT.READ_ONLY);
combo.add(“Oranges”);
combo.add(“Apples”);
combo.add(“Bananas”);
```

3.1.6 Dialog window

A dialog window provides a way to prompt the user for some input with a degree of modality. The dialog window is commonly used to report error messages or to ask simple questions regarding actions to the user. It is also important to note that a dialog is not a widget and therefore shares no common widget functionality. Dialog is an abstract superclass that is shared by all dialog types.

Example

Figure 3-6 shows a special type of dialog: a message box with styles: `SWT.YES | SWT.NO | SWT.ICON_QUESTION`.


Error

Question

Do you want to see the remaining messages?

Figure 3-6 A sample message box

Styles

Table 3-10 describes dialog styles.

Table 3-10 Dialog styles

Style	Description
`SWT.APPLICATION_MODAL`	Does not allow interaction with any window except its direct secondary descendants
`SWT.PRIMARY_MODAL`	Does not allow interaction with any ancestor window

Note: Not every subclass of `Dialog` supports a modality style. If a modality style is not supported, it is adjusted automatically to a more restrictive modality style.

Dialog types

There are several dialog implementations available to users within eSWT. They are each discussed in detail within their respective sections. This chapter discusses eSWT core, so we cover `MessageBox` and `FileDialog`. Table 3-11 lists all of the dialog implementations that are available and where they reside within eSWT.

Table 3-11 Dialog types

Component	Type	Description
eSWT core	`MessageBox`	Informs a user of a message, sometimes accompanied with a system beep.
eSWT core	`FileDialog`	Allows a user to navigate the file system and select a file.
eSWT expanded	`ColorDialog`	Allows a user to select a color for usage.
eSWT expanded	`DirectoryDialog`	Allows a user to navigate the file system and select a directory.
eSWT expanded	`FontDialog`	Allows a user to select a font for usage.
eSWT mobile	`QueryDialog`	A modal window that prompts the user for different types of input: standard, numeric, password, time and date.
eSWT mobile	`MultiPageDialog`	Allows an application to show multiple pages (tabbed dialog) which are displayed one at a time. It provides function similar to `TabFolder` in SWT.
eSWT mobile	`TimedMessageBox`	A dialog that closes automatically after a assigned period of time.

Titles

All dialogs have the ability to contain a title string.

- `setText(String title)`

  Sets the title of the dialog. If the title is not set, it is simply blank.

- `getText()`

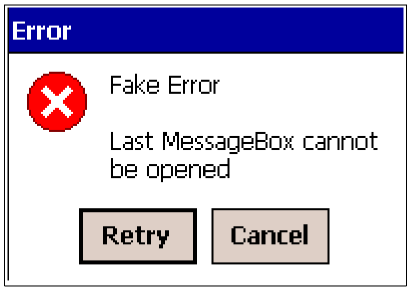
  Returns the title of the dialog.

3.1.7 MessageBox

Message boxes, to put it simply, provide a convenient way to display information or prompt a user with a question.

Example

Figure 3-7 displays a error message box with styles: `SWT.RETRY | SWT.CANCEL | SWT.ICON_ERROR`.



Fake Error

Last MessageBox cannot be opened

Figure 3-7 A sample error message box

Styles

Message boxes can include buttons and icons. These options are configured via style bits. Table 3-12 describes MessageBox button styles.

Table 3-12 MessageBox buttons

Style	**Description**
`SWT.OK`	A single OK button is displayed
`SWT.OK | SWT.CANCEL`	An OK and cancel button are displayed
`SWT.YES | SWT.NO`	A yes and no button are displayed
`SWT.YES | SWT.NO | SWT.CANCEL`	A yes, no and cancel button are displayed
`SWT.RETRY | SWT.CANCEL`	A retry and cancel button are displayed

Table 3-13 describes MessageBox icon styles.

Table 3-13 MessageBox icons

Style	**Description**
`SWT.ICON_ERROR`	Inform the user of an error
`SWT.ICON_INFORMATION`	Display a message to a user
`SWT.ICON_QUESTION`	Display a question to a user
`SWT.ICON_WARNING`	Inform the user of a warning
`SWT.ICON_WORKING`	Inform the user that work is in progress

Snippets

Example 3-10 illustrates how to display a warning message using a MessageBox.

Example 3-10 MessageBox: Display a warning

```
MessageBox warningBox = new MessageBox(shell, SWT.ICON_WARNING | SWT.OK);
warningBox.setText("Warning");
warningBox.setMessage("I'm warning you...!");
warningBox.open();
```

3.1.8 FileDialog

A file dialog allows a user to navigate the file system. Selecting a file or files performs an action (save or open) on the selection.

Example

Figure 3-8 shows a sample file dialog with style `SWT.OPEN`.

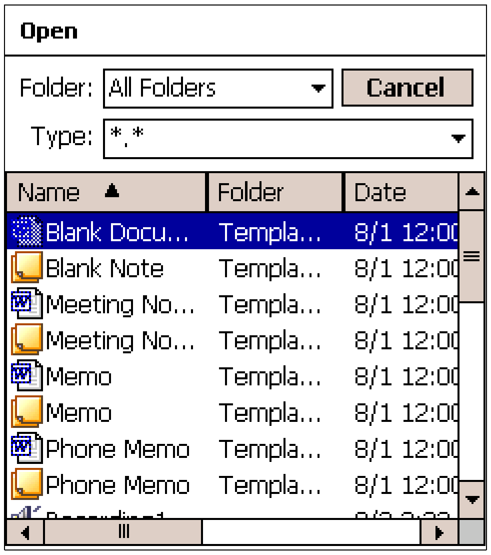

Open
Folder: All Folders
Cancel
Type: *.*
Name
Folder
Date
Blank Docu... Templa... 8/1 12:00
Blank Note Templa... 8/1 12:00
Meeting No... Templa... 8/1 12:00
Meeting No... Templa... 8/1 12:00
Memo Templa... 8/1 12:00
Memo Templa... 8/1 12:00
Phone Memo Templa... 8/1 12:00
Phone Memo Templa... 8/1 12:00

Figure 3-8 An example file dialog

Styles

Table 3-14 describes FileDialog styles.

Table 3-14 FileDialog styles

Style	Description
`SWT.SAVE`	Used when a user needs to save a file
`SWT.OPEN`	Used when a user needs to open a file
`SWT.SINGLE`	Allows the selection of a single file only
`SWT.MULTI`	Allows multiple file selections

Filters

A special feature of FileDialog allows a user to filter paths and file extensions. For path filtering, the following methods are available:

- `setFilterPath(String path)`

  Sets the path that is displayed first to a user when opening a file dialog. If no path is set, the operating system default is used.

- `getFilterPath()`

  Returns the filter path.

A common use of path filtering allows some convenience to the user in deciding the best location for the file to save or open. For example, a word processor on the Windows Mobile® operating system would most likely set the filter path to the My Documents folder to save the user time (Example 3-11).

Example 3-11 FileDialog: Filter the path so saves start in the My Documents folder

```
FileDialog dialog = new FileDialog(shell, SWT.SAVE);
dialog.setText("Save");
dialog.setFilterPath("C:\My Documents");
dialog.open();
```

For extension filtering, the following methods are available:

- `setFilterExtensions(String[] extensions)`

  Sets the extensions to which filter items are displayed in the file dialog. If no extensions are set, all files are matched.

- `getFilterExtensions()`

  Returns the filter extensions.

An example of a filter extension would be to limit a user to only opening music files because they are using a music player program (Example 3-12).

Example 3-12 FileDialog: Create a dialog and filter for music files only

```
FileDialog dialog = new FileDialog(shell, SWT.OPEN);
dialog.setText("What do you want to hear?");
dialog.setFilterExtensions(new String[] {"*.mp3,"*.wav","*.ogg"});
dialog.open();
```

Snippets

Example 3-13 illustrates how to create a dialog and filter for text files to open.

Example 3-13 FileDialog: Create a dialog and filter for text files to open

```
FileDialog dialog = new FileDialog(shell, SWT.OPEN);
dialog.setText("Select the text file...");
dialog.setFilterExtensions(new String[] {"Text Files (*.txt)", "Rich Text Files (*.rtf)"});
dialog.open();
```

3.1.9 Menu

Menus are one of the most common widgets found on desktop user interfaces. They allow for a list of items to be displayed, giving a user the option to choose an item. It is recommended that menus are used sparingly on mobile devices.

Important: The use of menus is discouraged because many mobile devices do not have pointer devices. Furthermore, it is difficult for an application to determine exactly how many top-level menu items can be displayed due to variable device displays. It is recommend that you use `Command` for interactions.

Example

Figure 3-9 shows a menu with a couple of sub-menu entries.

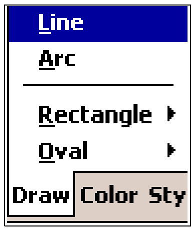


Figure 3-9 A sample menu with a couple of sub-menus.

Menu styles

Table 3-15 describes menu styles.

Table 3-15 Menu styles

Style	Description
`SWT.BAR`	Create a menu bar
`SWT.DROP_DOWN`	Create a drop-down menu
`SWT.POP_UP`	Create a pop-up menu
`SWT.NO_RADIO_GROUP`	Removes radio button behavior
`SWT.LEFT_TO_RIGHT`	Left-to-right menu orientation
`SWT.RIGHT_TO_LEFT`	Right-to-left menu orientation

Menu events

Table 3-16 describes menu events.

Table 3-16 Menu events

Event	Description
`SWT.Hide`	The menu is being hidden
`SWT.Show`	The menu is being shown

MenuItem styles

Table 3-17 describes MenuItem styles.

Table 3-17 MenuItem styles

Style	Description
`SWT.CHECK`	Creates a check button menu item
`SWT.CASCADE`	Creates a cascade styled menu item
`SWT.PUSH`	Creates a push button menu item
`SWT.RADIO`	Creates a radio button menu item
`SWT.SEPARATOR`	Creates a separator styled menu item

MenuItem events

Table 3-18 describes MenuItem events.

Table 3-18 MenuItem Events

Event	Description
`SWT.Selection`	The menu item was selected

Snippets

Example 3-14 illustrates how to create a pop-up menu.

Example 3-14 Menu: Create a pop-up menu

```
Menu menu = new Menu(shell, SWT.POP_UP);
MenuItem radioItem = new MenuItem(menu, SWT.RADIO);
radioItem.addText("Radio");
MenuItem checkItem = new MenuItem(menu, SWT.CHECK);
checkItem.addText("Check");
```

3.1.10 ScrollBar and Slider widgets

`ScrollBar` and `Slider` are widgets that allow a user to select a value between a minimum and maximum. They are manipulated via keyboard keys or using the thumb. The big difference between the two is that `Slider` widgets are controls, where as `ScrollBar` widgets are not. A scroll bar usually appears to the right or bottom of a control. Both of the widgets have a single selection that is considered to be its value, which is constrained based on a range of values specified by a user.

Example

Figure 3-10 shows a sample slider.

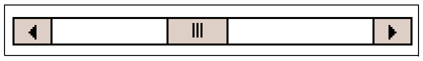


Figure 3-10 A sample slider

Styles

Table 3-19 describes `ScrollBar` and `Slider` styles.

Table 3-19 ScrollBar and Slider styles

Style	Description
`SWT.HORIZONTAL`	Creates a horizontal scroll bar or slider
`SWT.VERTICAL`	Creates a vertical scroll bar or slider

Events

Table 3-20 describes `ScrollBar` and `Slider` events.

Table 3-20 ScrollBar and Slider events

Event	Description
`SWT.Selection`	The widget was selected

ScrollBar creation

Scroll bars are created by specifying the `SWT.H_SCROLL` and `SWT.V_SCROLL` styles to the parent that is interested in receiving scroll bars. After the scroll bar is created, you can access it via its parent using the following methods:

- `getHorizontalBar()`

  Returns the horizontal scroll bar.

- `getVerticalBar()`

  Returns the vertical scroll bar.

Ranges

`ScrollBar` and `Slider` have the ability to set range operations, particularly, minimum and maximum integer values. The following methods are used to set the ranges.

- `setMinimum(int minimum)`

  Sets the minimum value of the `ScrollBar` or `Slider`.

- `getMinimum()`

  Returns the minimum value.

- `setMaximum()`

  Sets the maximum value of the `ScrollBar` or `Slider`.

- `getMaximum()`

  Returns the maximum value.

Increments

`ScrollBar` and `Slider` have the ability to set increment and page increment values. These are the values that pertain to when a user moves the thumb. Typically, arrow keys increment the scroll bar or slider and Page Up or Page Down keys page increment the scroll bar or slider. The following methods are used to set the different types of increment values.

- `setIncrement(int increment)`

  Sets the increment value. This value defaults to one (1) .

- `getIncrement()`

  Returns the increment value.
- `setPageIncrement()`

  Sets the page increment value. This value defaults to one (1) .
- `getPageIncrement()`

  Returns the page increment value.

3.1.11 ProgressBar

Progress bars are responsible for indicating the progress of operations to users. Progress bars are considered non-selectable controls within eSWT. Therefore, no event is fired with progress bars.

Example

Figure 3-11 shows a partially completed progress bar on the Pocket PC platform.

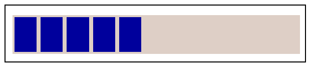


Figure 3-11 A sample progress bar

Styles

Table 3-21 describes ProgressBar styles.

Table 3-21 ProgressBar styles

Style	Description
`SWT.SMOOTH`	Fills the progress bar whole, instead of segmenting the progress.
`SWT.HORIZONTAL`	Create a horizontal progress bar.
`SWT.VERTICAL`	Create a vertical progress bar.
`SWT.INDETERMINATE`	Create a progress bar that has runs on indeterminate operations and is animated accordingly.

Ranges

Progress bars provide the following range operations:

- `setMinimum(int minimum)`

  Sets the minimum value.
- `getMinimum()`

  Returns the minimum value. This value defaults to one (1) if nothing is set.
- `setMaximum(int maximum)`

  Sets the maximum value.
- `getMaximum()`

  Returns the maximum value. This value defaults to 100 if nothing is set.

Snippets

Example 3-15 creates an indeterminate progress bar.

Example 3-15 ProgressBar: Create an indeterminate progress bar

```
final ProgressBar bar = new ProgressBar(shell, SWT.INDETERMINATE);
new Thread() {
   public void run() {
      try {Thread.sleep(60000);} catch (Throwable t) {} // sleep for 60 seconds
      bar.setSelection(bar.getMaximum());
   }
}.start();
```

3.2 Layouts

Layouts are used to arrange controls. This section provides details to design an interface for data input using a form.

3.2.1 FormLayout

`FormLayout` is a flexible layout that is highly recommended for usage in mobile devices because the position and size of the components only depend on a single control. `FormLayout` is highly resilient to the varied nature of mobile device screen sizes.

Table 3-22 lists the properties to aid in layout for FormLayout.

Table 3-22 FormLayout properties

Property	Description
`marginBottom`	The space in pixels to be used as a bottom margin
`marginHeight`	The space in pixels to be used as a margin at the top and bottom of controls
`marginLeft`	The space in pixels to be used as a left margin
`marginRight`	The space in pixels to be used as a right margin
`marginTop`	The space in pixels to be used as a top margin
`marginWidth`	The space in pixels to be used as a margin between the left and right of controls
`spacing`	The amount of spacing in pixels between controls

Example 3-16 illustrates how to create a simple FormLayout.

Example 3-16 FormLayout: Creating a simple FormLayout

```
FormLayout layout = new FormLayout();
layout.marginWidth = 5;
layout.marginHeight = 5;
layout.spacing = 2;
shell.setLayout(layout);
```

FormData

`FormLayout` uses `FormData` to set the different layout properties on a widget (Table 3-23). Inside of each `FormData` resides a `FormAttachment` that describes the edges of a widget.

Table 3-23 FormData Layout properties

Property	Description
`width`	Specifies the width of the control
`height`	Specifies the height of the control
`top`	Specifies the form attachment that resides on the top edge control
`bottom`	Specifies the form attachment that resides on the bottom edge control
`left`	Specifies the form attachment that resides on the left edge control
`right`	Specifies the form attachment that resides on the right edge control

FormAttachment

`FormAttachment` describes the layout of the control. Table 3-24 lists the properties that are associated with `FormAttachment`.

Table 3-24 FormAttachment properties

Property	Description
alignment	Specifies the alignment of the control to which this edge is attached
control	Specifies the control to which this edge is attached
denominator	Specifies the denominator that describes where the edge will be placed
numerator	Specifies the numerator that describes where the edge will be placed
offset	Specifies the offset in pixels that is added to the edge

To use attachments, we need to create a FormData instance and set the properties directly. Example 3-17 creates two buttons and attaches them to each other with some spacing.

Example 3-17 Creating buttons with attachment

```
FormLayout layout = new FormLayout();
layout.spacing = 3;
shell.setLayout(layout);

Button firstButton = new Button(shell, SWT.PUSH);
firstButton.setText(“First”);
Button secondButton = new Button(shell, SWT.PUSH);
secondButton.setText(“Second”);

FormData data = new FormData();
data.left = new FormAttatchment(firstButton);
secondButton.setLayoutData(data);
```

4

eSWT mobile extensions

This chapter provides a detailed description of the following components found within embedded Standard Widget Toolkit (eSWT) mobile extensions:

- **Controls**
  - CaptionedControl
  - ConstrainedText
  - HyperLink
  - DateEditor
  - ListBox
  - ListView
  - MobileShell
  - SortedList
  - TextExtension
- **Widgets**
  - TaskTip
- **Dialogs**
  - MultiPageDialog
  - TimedMessageBox
- **Items**
  - Command
- **Device-related**
  - MobileDevice
  - Screen
  - Input

4.1 Controls

Controls are the basic building blocks in eSWT application programs. This section describes controls that are provided by the eSWT mobile extensions.

4.1.1 CaptionedControl

A `CaptionedControl` is used to display a label (caption) in front of a control. You can use optional trailing text after the control, for example to indicate units of measurement.

Tip: Determining which control has focus can be difficult on mobile devices where lighting conditions are often less than optimal. With `CaptionedControl`, a label shows focus highlighting whenever the control has focus. With an entire label highlighted, it is easy to locate the focus.

Example

Figure 4-1 shows an example of `CaptionedControl` text. In this case, `Name:` is the text of `CaptionedControl`, `John Marshall` is the embedded `Text` widget that is contained in `CaptionedControl`, and `Software Engineer` is the training text. When `CaptionedControl` gets focus, text and training text is highlighted to express focus effect.

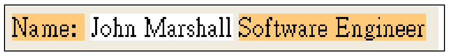


Figure 4-1 CaptionedControl example

Styles

Table 4-1 shows the styles for `CaptionedControl`.

Table 4-1 CaptionedControl styles

Styles	Description
SWT.LEFT_TO_RIGHT	Default style for `CaptionedControl`, elements inside `CaptionedControl` are aligned from left to right
SWT.RIGHT_TO_LEFT	Elements inside `CaptionedControl` are aligned from right to left

Text, trailing text, and image

`CaptionedControl` provides the following methods to set text, trailing text, and image:

- `setText(java.lang.String string)`

  Sets the caption label.
- `getText()`

  Gets the caption text, which is an empty string if it has never been set.
- `setTrailingText(java.lang.String string)`

  Sets the trailing label.
- `getTrailingText()`

  Gets the trailing text, which is an empty string if it has never been set.

- `setImage(Image image)`

  Sets the image as an icon to the `CaptionedControl`.
- `getImage()`

  Returns the `CaptionedControl` icon image or `null` if it has never been set.

Snippets

Example 4-1 is the code snippet for the `CaptionedControl` sample that is shown in Figure 4-1.

Example 4-1 CaptionedControl snippet

```
CaptionedControl captionedControl = new CaptionedControl(shell, SWT.LEFT_TO_RIGHT);
captionedControl.setText("Name: ");
Text text = new Text(captionedControl, SWT.SINGLE);
text.setText("John Marshall");
captionedControl.setTrailingText("Software Engineer");
```

4.1.2 ConstrainedText

`ConstrainedText` is a single-line `Text` control that constrains the user input by styles. This control limits the characters that users can enter by styles to make data entry more efficient. This capability is usually implemented by the underlying platform. The application is not allowed to change or override the constraints. An `IllegalArgumentException` is thrown if illegal content is set by applications programmatically.

Tip: `ConstrainedText` is a convenient widget that sets the initial input mode of a text field and also limits the characters than can be entered within the field.

Example

Figure 4-2 shows a `ConstrainedText` control with `ConstrainedText.PHONENUMBER` style for a user to input a phone number only.

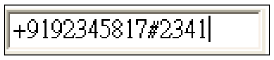

+9192345817#2341

Figure 4-2 ConstrainedText example

Styles

`ConstrainedText` is similar to `Text` in that almost all styles that are used in `Text` can also be used in `ConstrainedText`, except for `SWT.MULTI`. Because `ConstrainedText` only supports single line input, `SWT.MULTI` takes no effect if it is set in an application.

Table 4-2 lists the `ConstrainedText` styles.

Table 4-2 ConstrainedText styles

Style	Description
DECIMAL	Allows the input of numeric values with optional decimal fractions (for example, -123, 0.123, or .5 are all valid input)
NUMERIC	Allows the input of numeric values
PHONENUMBER	Allows the input of numeric values with optional phone specific characters such as a plus sign (+), an asterisk (*), and a number sign (#)

Events

Similar to `Text`, `ConstrainedText` has two events: `SelectionEvent` and `ModifyEvent`. These events can be accessed via following methods:

- `addSelectionListener(SelectionListener listener)`

  Adds the listener to the collection of listeners who are notified when the control is selected by sending it one of the messages that are defined in the `SelectionListener` interface.

- `removeSelectionListener(SelectionListener listener)`

  Removes the listener from the collection of listeners who are notified when the control is selected.

- `addModifyListener(ModifyListener listener)`

  Adds the listener to the collection of listeners who are notified when the `ConstrainedText`'s text is modified, by sending it one of the messages that are defined in the `ModifyListener` interface.

- `removeModifyListener(ModifyListener listener)`

  Removes the listener from the collection of listeners who are notified when the `ConstrainedText`'s text is modified.

Snippets

Example 4-2 is code snippet for `CaptionedControl` sample that is shown in Figure 4-2 on page 53.

Example 4-2 CaptionedControl snippet

```
ConstrainedText cT = new ConstrainedText(shell, SWT.BORDER, ConstrainedText.PHONENUMBER);
cT.setSize(150,25);
```

4.1.3 HyperLink

Instances of this class represent a selectable user interface object that launches other applications when activated by the user. This class represents several types of hyperlinks associated with certain functionalities. The user can activate the associated program determined by the style.

A `HyperLink` instance accepts general characters as other controls do, but the appearance is dependent upon implementation and locale. For example, a `HyperLink` object with the `PHONE` style might display as follows:

```
(416) 123-4567
```

However, the actual content of the object is visible to the application through the API. For example `getText()` can be the string `4161234567`.

Note: Support of the native program is platform dependent. Not all platforms can have corresponding platform function. For example, PDAs might not have a built-in phone.

Example

Figure 4-3 shows a `HyperLink` with `EMAIL` style.

Figure 4-3 HyperLink example

When a user selects the hyperlink, HyperLink control brings up the operating system default e-mail client with the e-mail address set as the receiver. Figure 4-4 shows the result on a Pocket PC. It behaves similar to a hyperlink in HTML.

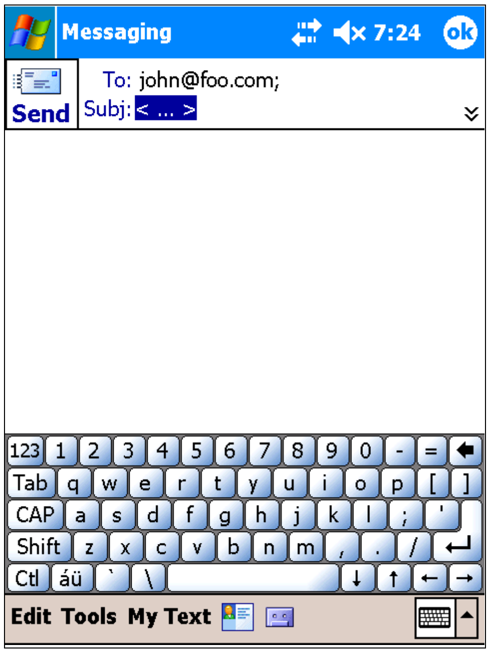


Figure 4-4 E-mail client brought up by HyperLink

Styles

Table 4-3 shows the format styles that are used in HyperLink.

Table 4-3 HyperLink format style

Style	Description
EMAIL	Opens the implementation-dependent e-mail client when activated
PHONE	Shows the implementation-dependent dialer interface when activated
URL	Launches a implementation-dependent Web browser when activated

Snippets

Example 4-3 is code snippet for CaptionedControl sample that is shown in Figure 4-5. Note that this example uses the PHONE format style and sets the text to 9191257328. The HyperLink class reformats the appearance so that it is more meaningful to the user.

Example 4-3 HyperLink snippet

```
HyperLink hyperLink = new HyperLink(shell, SWT.NONE, HyperLink.PHONE);
hyperLink.setText("9191257328");
```

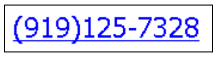


Figure 4-5 PHONE style HyperLink

4.1.4 DateEditor

`DateEditor` is a special data entry control that allows users to enter or choose a date. The return value of `getDate()` is an instance of a `Date` class.

The default locale and time zone for the date and time formatting reflects the current configuration in the device. The default date that is used if a date is not set is the current date. Applications might call `setTimeZone(timeZone)` to change the differential that is added to UTC time. This change only affects the widget instance and does not affect other applications.

Example

Figure 4-6 shows a `DateEditor` with `DATE` style.

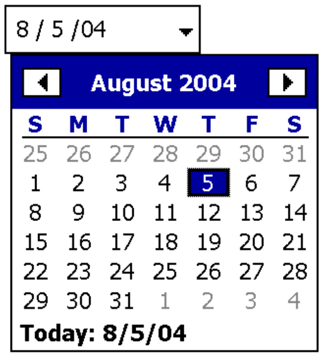


Figure 4-6 DateEditor example

Styles

Table 4-4 shows the `DateEditor` styles.

Note: `COMPACT` and `FULL` are hints that might not take effect on some embedded devices (for example, Pocket PC).

Table 4-4 DateEditor style

Style	Description
COMPACT	A hint that the widget should be displayed in a format which is smaller or less full featured
DATE	A date entry type for year, month and day
DATE_TIME	An entry type for date and time
DURATION	An entry type for a period of time in hours, minutes and seconds
FULL	A hint that the widget should be displayed in a format which is full featured and emphasizes ease of use rather than compactness
OFFSET	An entry type for a period of time in hours, minutes, and seconds which can be subtracted or added to another time value
TIME	A time entry type for hours, minutes and seconds

Events

`DateEditor` has two events: `SelectionEvent` and `ModifyEvent`. You can access these events can via the following methods:

- `addSelectionListener(SelectionListener listener)`

  Adds the listener to the collection of listeners who are notified when the control is selected by sending it one of the messages that are defined in the `SelectionListener` interface.

- `removeSelectionListener(SelectionListener listener)`

  Removes the listener from the collection of listeners who are notified when the control is selected.

- `addModifyListener(ModifyListener listener)`

  Adds the listener to the collection of listeners who are notified when `DateEditor` text is modified by sending one of the messages that are defined in the `ModifyListener` interface.

- `removeModifyListener(ModifyListener listener)`

  Removes the listener from the collection of listeners who are notified when `DateEditor` text is modified.

Date and Time

eSWT provides the following methods to perform set and retrieval operation on `DateEditor`:

- `setDate(java.util.Date date)`

  Sets the date for the editor when `TIME` or `DATE_TIME` styles are used.

- `getDate()`

  Returns the date when `DATE` or `DATE_TIME` styles are used.

- `setTime(int seconds)`

  Sets the time for the editor when `TIME`, `OFFSET`, or `DURATION` styles are used.

- `getTime()`

  Returns the number of seconds when `TIME`, `OFFSET`, or `DURATION` styles are used.

- `setTimeZone(java.util.TimeZone timeZone)`

  Defines the time zone, which is applied on the `DateEditor` (UTC time).

Snippets

Example 4-4 is code snippet for the `CaptionedControl` sample that is shown in Figure 4-7 on page 58. The first is `DATE` style, which shows the current date if the application does not set the date. The second is `TIME` style, which shows current time if the application does not set the time. The third is `DURATION` style, which takes seconds as the argument in `setTime()` and displays the format of the time in hour/minutes/seconds.

Example 4-4 DateEditor snippet

```
DateEditor dateEditor1 = new DateEditor(shell, SWT.NONE, DateEditor.DATE);
DateEditor dateEditor2 = new DateEditor(shell, SWT.NONE, DateEditor.TIME);
DateEditor dateEditor3 = new DateEditor(shell, SWT.NONE, DateEditor.DURATION);
dateEditor3.setTime(1000);
```

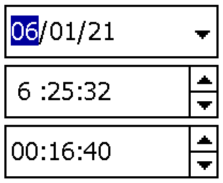


Figure 4-7 DateEditor snippet result

4.1.5 ListBox

Instances of the ListBox class represent a selectable user interface object that displays a list of items consisting of text and icons from a data model. Each list item can include combinations of heading text, heading icons, detail text, and detail icons. The layout and display of the various text and icons is variable, depending upon the style and modifier constants that are passed to the constructor.

Only one LB_STYLE_xxxx layout style constant can be specified. The LB_STYLE_NO_HEADING_TEXT style displays detail text on a single line per item. The LB_STYLE_1LINE_ITEM style displays heading text next to detail text in a single row. The LB_STYLE_2LINE_ITEM style displays heading text above detail text in a two line cell.

ListBox is a model-view-control (MVC) style widget that displays a list of items in the most efficient or useful way for a given platform.

Example

Figure 4-8 shows a ListBox example.

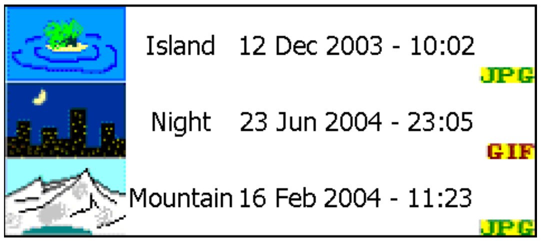

Island 12 Dec 2003 - 10:02
JPG
Night 23 Jun 2004 - 23:05
GIF
Mountain 16 Feb 2004 - 11:23
JPG

Figure 4-8 ListBox example

Styles

Table 4-5 shows several styles that are used in ListBox. You can specify only one.

Table 4-5 ListBox styles

Style	Description
LB_MOD_SHOW_DETAIL_ICONS	Shows icons that are associated with detail text
LB_MOD_SHOW_HEADING_ICONS	Shows icon that are associated with heading text
LB_MOD_SHOW_SELECTION_NUMBER	Shows a single digit number that is aligned with each item that can be used to select the item
LB_STYLE_1LINE_ITEM	A single-line item, 2 columns
LB_STYLE_2LINE_ITEM	A double-line item, 1 column with heading and detail combined
LB_STYLE_NO_HEADING_TEXT	A single-line item, 1 column (default)

Events

`DateEditor` has one event, `SelectionEvent`, which can be accessed via the following methods:

- `addSelectionListener(SelectionListener listener)`

  Adds the listener to the collection of listeners who are notified when the control is selected by sending it one of the messages that are defined in the `SelectionListener` interface.

- `removeSelectionListener(SelectionListener listener)`

  Removes the listener from the collection of listeners who are notified when the control is selected.

ListBoxItem

Unlike `List`, data that is contained in `ListBox` comes from a array of `ListBoxItem` rather than adding or inserting one by one, so as to provide an MVC model. Instances of `ListBoxItem` represent an item in a `ListBox` widget. Heading and detail icons, if provided, are displayed at the size provided or stretched to fit the `ListBox` styles. For consistency, all heading icons should be the same size and all detail icons should be the same size for any one list. Elements of a `ListBoxItem` which are null are not displayed in the `ListBox` layout.

The following methods are provided to set or to get attributes within an instance of `ListBox`:

- `setDetailIcons(Image[] icons)`

  Sets the detail icons of the item.

- `getDetailIcons()`

  Gets the detail icons of the item.

- `setDetailText(java.lang.String detailText)`

  Sets the detail text of the item.

- `getDetailText()`

  Gets the detail text of the item.

- `setHeadingIcons(Image[] icons)`

  Sets the heading icons for this item.

- `getHeadingIcons()`

  Gets the heading icons for this item.

- `setHeadingText(java.lang.String headingText)`

  Sets the heading text of the item.

- `getHeadingText()`

  Gets the heading text of the item.

Set data model and selection

This section describes the Set data model and selection methods.

Set data model

To update the content that is contained in `ListBox`, we need to create an array of `ListBoxItem`, set the content properly. We then need to call `setDataModel()` to set the content within `ListBox` using the `setDataModel(ListBoxItem[] items)` method. This method establishes the data model for this `ListBox`. The provided array is used for the life of the `ListBox` or until a new data model is set. Elements of a `ListBoxItem` which are null or disposed are not displayed, leaving a blank area within the layout.

When applications make changes to `ListBox` array, `ListBox` does not take visual change immediately. So, applications need to refresh the `ListBox` explicitly using the following methods:

- `public void refreshItem(int index)`

  Notifies this `ListBox` that the data for the item at the given index has been updated and the item display needs to be refreshed.
- `refreshList()`

  Notifies this `ListBox` that multiple items might have been updated and the entire list display needs to be refreshed.

Selection

`ListBox` provides several ways to set selection, almost all of which are index-based. The following methods are provided to set or to get selection:

- `setSelection(int index)`

  Selects the item at the given zero-relative index in the `ListBox`.
- `setSelection(int[] indices)`

  Selects the items at the given zero-relative indices in the `ListBox`.
- `setSelection(int start, int end)`

  Selects the items in the range specified by the given zero-relative indices in the `ListBox`.
- `getSelectionCount()`

  Returns the number of items currently selected.
- `getSelectionIndices()`

  Returns the zero-relative indices of the items which are currently selected in the `ListBox`.

`ListBox` also provides several methods to perform deselecting or selecting items:

- `deselect(int index)`

  Deselects the item at the given zero-relative index in the `ListBox`.
- `deselect(int[] indices)`

  Deselects the items at the given zero-relative indices in the `ListBox`.
- `deselect(int start, int end)`

  Deselects the items at the given zero-relative indices in the `ListBox`.
- `deselectAll()`

  Deselects all selected items in the `ListBox`.
- `select(int index)`

  Selects the item at the given zero-relative index in the `ListBox`'s list.
- `void select(int[] indices)`

  Selects the items at the given zero-relative indices in the ListBox.
- `void select(int start, int end)`

  Selects the items in the range specified by the given zero-relative indices in the `ListBox`.
- `void selectAll()`

  Selects all of the items in the `ListBox`.

Snippets

Example 4-5 creates a array of styles and then uses the array to create different ListBox with the same data model.

Example 4-5 ListBox snippet

```
// Different styles for use in creation of ListBox
int[] modes={
            ListBox.LB_STYLE_2LINE_ITEM | ListBox.LB_MOD_SHOW_HEADING_ICONS |
ListBox.LB_MOD_SHOW_DETAIL_ICONS,
            ListBox.LB_STYLE_1LINE_ITEM | ListBox.LB_MOD_SHOW_HEADING_ICONS |
ListBox.LB_MOD_SHOW_DETAIL_ICONS,
            ListBox.LB_STYLE_1LINE_ITEM | ListBox.LB_MOD_SHOW_HEADING_ICONS |
ListBox.LB_MOD_SHOW_DETAIL_ICONS | ListBox.LB_MOD_SHOW_SELECTION_NUMBER,
            ListBox.LB_STYLE_1LINE_ITEM | ListBox.LB_MOD_SHOW_DETAIL_ICONS |
ListBox.LB_MOD_SHOW_SELECTION_NUMBER,
            ListBox.LB_STYLE_NO_HEADING_TEXT|ListBox.LB_MOD_SHOW_DETAIL_ICONS,
            ListBox.LB_STYLE_NO_HEADING_TEXT
            };

// Create the data modal
private void createListBoxItems() {
   listboxitems = new ListBoxItem[3];

   // Get images
   lb1details = new Image(getComposite().getDisplay(),
getClass().getResourceAsStream("/res/lb1details.png"));
   lb1heading = new Image(getComposite().getDisplay(),
getClass().getResourceAsStream("/res/lb1heading.png"));
   lb2details = new Image(getComposite().getDisplay(),
getClass().getResourceAsStream("/res/lb2details.png"));
   lb2heading = new Image(getComposite().getDisplay(),
getClass().getResourceAsStream("/res/lb2heading.png"));
   lb3details = new Image(getComposite().getDisplay(),
getClass().getResourceAsStream("/res/lb3details.png"));
   lb3heading = new Image(getComposite().getDisplay(),
getClass().getResourceAsStream("/res/lb3heading.png"));

   // create the ListboxItems
   listboxitems[0] = new ListBoxItem("12 Dec 2003 - 10:02",lb1details,"Island",lb1heading);
   listboxitems[1] = new ListBoxItem("23 Jun 2004 - 23:05",lb2details,"Night",lb2heading);
   listboxitems[2] = new ListBoxItem("16 Feb 2004 -
11:23",lb3details,"Mountain",lb3heading);
}

//Then we call creation of ListBox several times with different styles to see different
visual effect

ListBox[] listbox = new ListBox[6];

for (int i=0; i<6; i++) {
   // create a new Listbox
   listbox[i] = new ListBox(getComposite(), SWT.SINGLE, modes[i] );
   // add the ListBoxItems to the ListBox
   listbox[i].setDataModel(listboxitems);
}
```

Figure 4-9 shows ListBox created with the following:

```
ListBox.LB_STYLE_2LINE_ITEM | ListBox.LB_MOD_SHOW_HEADING_ICONS |
ListBox.LB_MOD_SHOW_DETAIL_ICONS:
```

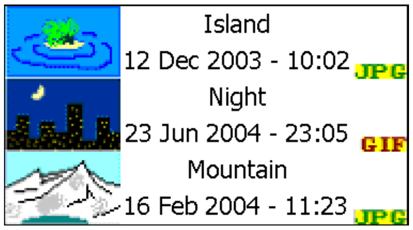


Figure 4-9 ListBox snippet 1

Figure 4-10 shows ListBox created with the following:

```
ListBox.LB_STYLE_1LINE_ITEM | ListBox.LB_MOD_SHOW_HEADING_ICONS |
ListBox.LB_MOD_SHOW_DETAIL_ICONS:
```

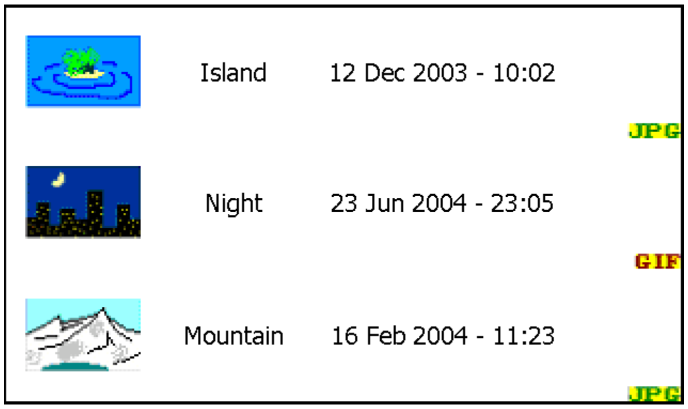


Figure 4-10 ListBox snippet 2

Figure 4-11 shows ListBox created with the following:

```
ListBox.LB_STYLE_1LINE_ITEM | ListBox.LB_MOD_SHOW_HEADING_ICONS |
ListBox.LB_MOD_SHOW_DETAIL_ICONS | ListBox.LB_MOD_SHOW_SELECTION_NUMBER:
```

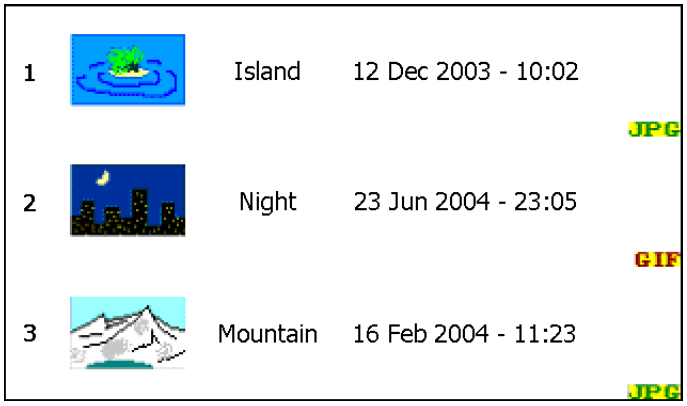
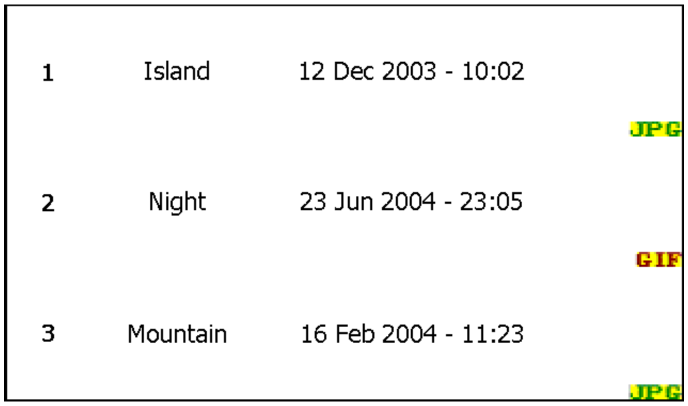


Figure 4-11 ListBox snippet 3

Figure 4-12 shows ListBox created with the following:

```
ListBox.LB_STYLE_1LINE_ITEM | ListBox.LB_MOD_SHOW_DETAIL_ICONS |
ListBox.LB_MOD_SHOW_SELECTION_NUMBER:
```

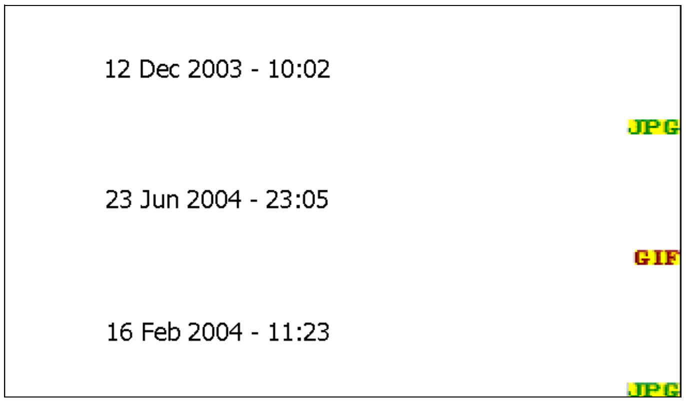


```
1    Island      12 Dec 2003 - 10:02
                                          JPG
2    Night       23 Jun 2004 - 23:05
                                          GIF
3    Mountain    16 Feb 2004 - 11:23
                                          JPG
```

Figure 4-12   ListBox snippet 4

Figure 4-13 shows ListBox created with the following:

```
ListBox.LB_STYLE_NO_HEADING_TEXT|ListBox.LB_MOD_SHOW_DETAIL_ICONS:
```

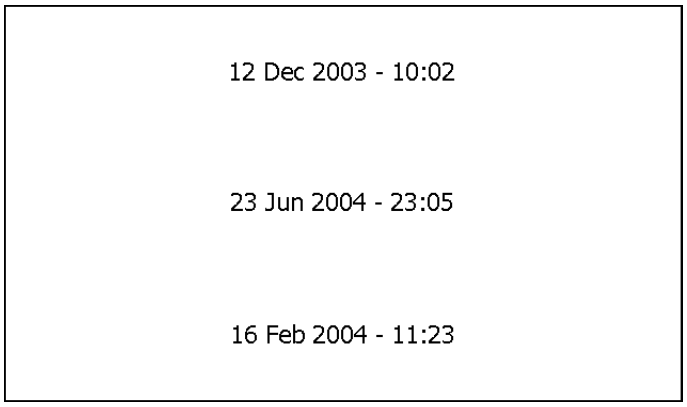


```
12 Dec 2003 - 10:02
                                  JPG
23 Jun 2004 - 23:05
                                  GIF
16 Feb 2004 - 11:23
                                  JPG
```

Figure 4-13   ListBox snippet 5

Figure 4-14 shows ListBox created with the following:

```
ListBox.LB_STYLE_NO_HEADING_TEXT:
```



```
12 Dec 2003 - 10:02

23 Jun 2004 - 23:05

16 Feb 2004 - 11:23
```

Figure 4-14   ListBox snippet 6

4.1.6 ListView

`ListView` allows users to select one or more items from a collection of items that can be displayed in multi-columns with different styles. `ListView` is similar to `List`, but it allows icon support in each item.

`ListView` lays out its children items in one or more columns from top to bottom. If a layout orientation hint is not specified, the implementation chooses the orientation. If there is only enough screen width for one column, the list scrolls vertically. If there is room to display multiple columns within the widget, then the list scrolls horizontally. The list never scrolls in more than one direction. You can set the layout orientation at runtime by calling the `setLayout(int)` method.

The item density hint determines the size and positioning of items in order to fit more or less within the widget. Applications can query the preferred sizes of the icons for each density level. Note that the sizes can be diverse in different platforms. When the given icons do not match the preferred size, the implementation might adjust icon sizes without throwing any exception. Applications can change the item density level at runtime by calling the `setLayoutDensity(int)` method. Some platforms might use context-sensitive device keys to allow the user to change the `ListView` density level at runtime (for example, by activating zoom in or zoom out device keys when the `ListView` is focused).

Example

Figure 4-15 shows a `ListView` example.

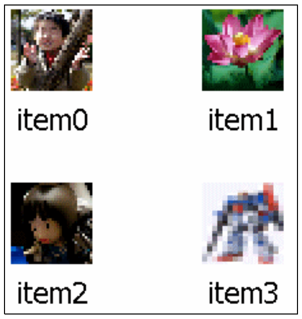


Figure 4-15 ListView example

Styles

`ListView` has two style variables in its constructor. One is a SWT style and another is a `ListView` density style.

Table 4-6 shows the SWT styles, and Table 4-7 shows the density styles.

`SWT.SINGLE` and `SWT.MULTI` are used exclusively. The same limitation applies to `SWT.VERTICAL` and `SWT.HORIZONTAL`. However, you can use `SWT.SINGLE | SWT.VERTICAL` to create a single selection, vertical layout `ListView`.

Table 4-6 SWT styles

Style	Description
SWT.SINGLE	Single selection
SWT.MULTI	Multiple selection

Table 4-7 Density styles

Style	Description
ListView.LOW	Low density
ListView.MEDIUM	Medium density
ListView.HIGH	High density

Note: Density is defined as a *hint* to be used. Not all platforms can support all three types of density. For example, Pocket PC supports only `LOW` and `MEDIUM`. Thus, with Pocket PC, `HIGH` has no visual difference with `MEDIUM`.

Events

`ListView` has one event, `SelectionEvent`, which can be accessed via the following methods:

- `addSelectionListener(SelectionListener listener)`

  Adds the listener to the collection of listeners who are notified when the control is selected by sending it one of the messages that are defined in the `SelectionListener` interface.

- `removeSelectionListener(SelectionListener listener)`

  Removes the listener from the collection of listeners who are notified when the control is selected.

Setting, adding, and removing items

The methods that are provided to set, add, and remove items in `ListView` are very similar to those in `List`. In addition, you can add icons for adding and setting `ListView` items.

Setting Items

Items inside `ListView` can be initialized with setting methods. `ListView` provides the following methods to get and to set items:

- `setItem(int index, java.lang.String string, Image icon)`

  Sets the text and icon of the item in the `ListView` list at the given zero-relative index to the string argument.

- `setItems(java.lang.String[] items, Image[] icons)`

  Sets the `ListView` items to be the given array of items and icons.

- `getItem(int index)`

  Returns the item at the given, zero-relative index in the `ListView`.

- `getItems()`

  Returns an array of String s which are the items in the `ListView`.

- `getItemCount()`

  Returns the number of items contained in the `ListView`.

Add Items

You can also add items into `ListView` by adding methods, `ListView` provides the following methods to perform the add operation:

- `add(java.lang.String item, Image icon)`

  Adds the string item and an optional icon to the end of the `ListView` list.
- `add(java.lang.String string, Image icon, int index)`

  Adds the string item and an optional icon to the `ListView` list at the given zero-relative index.

Removing Items

For removing items, `ListView` provides the following methods:

- `remove(int index)`

  Removes the item from the `ListView` at the given zero-relative index.
- `remove(int[] indices)`

  Removes the items from the `ListView` at the given zero-relative indices.
- `remove(int start, int end)`

  Removes the items from the `ListView` that are between the given zero-relative start and end indices (inclusive).
- `remove(java.lang.String string)`

  Searches the `ListView` list, starting at the first item, until an item is found that is equal to the argument and removes that item from the list.
- `removeAll()`

  Removes all of the items from the `ListView`.

Selection

`ListView` provides several ways to set selection, almost all of which are index-based. The following methods are provided to set or to get selection:

- `setSelection(int index)`

  Selects the item at the given zero-relative index in the `ListView`.
- `setSelection(int[] indices)`

  Selects the items at the given zero-relative indices in the `ListView`.
- `setSelection(int start, int end)`

  Selects the items in the range that is specified by the given zero-relative indices in the `ListView`.
- `getSelectionCount()`

  Returns the number of items currently selected.
- `getSelectionIndices()`

  Returns the zero-relative indices of the items which are currently selected in the `ListView`.

`ListView` also provides several methods to perform deselecting or selecting items:

- `deselect(int index)`

  Deselects the item at the given zero-relative index in the `ListView`.
- `deselect(int[] indices)`

  Deselects the items at the given zero-relative indices in the `ListView`.
- `deselect(int start, int end)`

  Deselects the items at the given zero-relative indices in the `ListView`.
- `deselectAll()`

  Deselects all selected items in the `ListView`.
- `select(int index)`

  Selects the item at the given zero-relative index in the `ListView`'s list.
- `void select(int[] indices)`

  Selects the items at the given zero-relative indices in the `ListView`.
- `void select(int start, int end)`

  Selects the items in the range specified by the given zero-relative indices in the `ListView`.
- `void selectAll()`

  Selects all of the items in the `ListView`.

Snippets

Example 4-6 creates a `ListView` that contains 20 items, all of which are added with icons. It also creates two `Commands` for the user to change the density of `ListView`.

Example 4-6 ListView example

```
final ListView lv = new ListView(shell, SWT.MULTI, ListView.HIGH);

//set Image array
Image[] image = new Image[4];
image[0] = new Image(Display.getDefault(), "/eSWT/childs.jpg");
image[1] = new Image(Display.getDefault(), "/eSWT/lotus.png");
image[2] = new Image(Display.getDefault(), "/eSWT/osaka_2.jpg");
image[3] = new Image(Display.getDefault(), "/eSWT/pgz_2.png");
lv.setSize(200,250);

//Create a Command for setting low density
Command lowCommand = new Command(lv, Command.SELECT, 0);
lowCommand.setText("LOW");
lowCommand.addSelectionListener(new SelectionListener(){
   public void widgetSelected(SelectionEvent e) {
      lv.setLayoutDensity(ListView.LOW);
   }
   public void widgetDefaultSelected(SelectionEvent e) {
   }});

//Create a Command for setting high density
Command midCommand = new Command(lv, Command.SELECT, 0);
midCommand.setText("MEDIUM");
midCommand.addSelectionListener(new SelectionListener(){
   public void widgetSelected(SelectionEvent e) {
      lv.setLayoutDensity(ListView.MEDIUM);
   }
   public void widgetDefaultSelected(SelectionEvent e) {
   }});

//add ListView items
for (int i=0; i<20; i++) {
   lv.add("item"+i, image[i % 4]);
}
```

Figure 4-16 shows the result of `LOW` density.

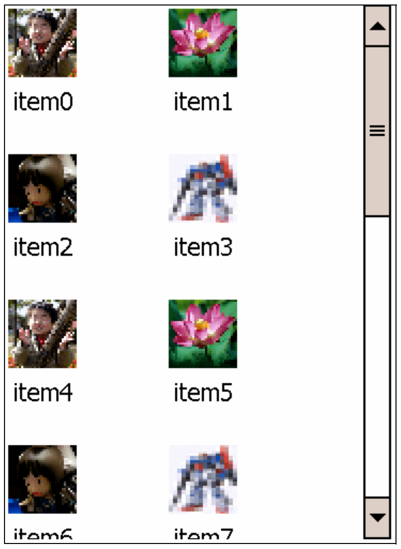


Figure 4-16 LOW density

Figure 4-17 is `HIGH` density.

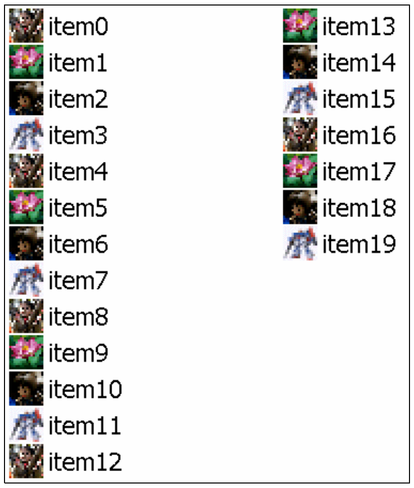


Figure 4-17 HIGH density

4.1.7 MobileShell

`MobileShell` is a shell that is particularly suitable for devices that require dynamic change of trims at runtime. It is a top-level window that is usually visible and managed by the user or a window manager. Two unique features of `MobileShell` over a normal `Shell` are a new `changeTrim(int, int)` method to change `Shell` trim styles dynamically and full-screen mode.

`MobileShell` uses the entire available application space in the normal mode. In this mode, a little screen area (title bar) is often reserved to display text with or without an icon and optional trim information. The text shows the application name text by default. (See `setText(String)` to change the text value and `setImage(Image)` to set the icon.)

Outside the application screen space, mobile devices often reserve a screen space, named *status area*, for displaying some extra information regarding the device and other useful data. Such reserved space is usually accessible and programmable. `MobileShell` provides a `setStatus(String)` method to display text on the status area. The implementation decides how to process the string (for example, if the content is too long to display).

Unlike a normal shell, `MobileShell` supports a special screen mode to use the entire device screen space instead of the application space in the normal mode. Applications can call a `setFullScreenMode(boolean)` method to switch between normal and full-screen modes at runtime. This feature is often used in applications such as media players and Web browsers where requests for full-screen mode rendering better effects.

`MobileShell` can also be used to poll for key state as commonly done within game execution loops.

Example

Figure 4-18 is a full-screen `MobileShell`, no title bar or task bar is displayed. Full-screen display is useful in mobile devices when the application needs to display more information to the user.

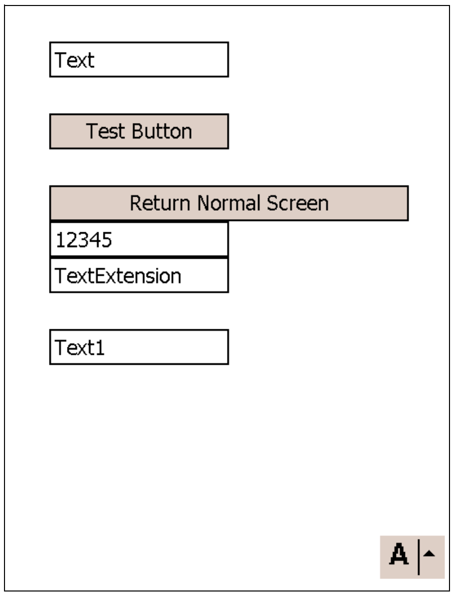


Figure 4-18 Full-screen MobileShell

Styles

`MobileShell` has two kind of styles. Table 4-8 shows the trim styles, and Table 4-9 shows status styles.

Table 4-8 Trim styles

Styles	Description
SHELL_TRIM	Shell trim
DIALOG_TRIM	Dialog trim
NO_TRIM	No trim
TITLE	Title trim
CLOSE	Shell with close box
BORDER	Shell with border

Table 4-9 Status styles

Styles	Description
NO_STATUS_PANE	Hint to show no status pane
SMALL_STATUS_PANE	Hint to show small status pane with shell
LARGE_STATUS_PANE	Hint show large status pane with shell

Trim and status

`MobileShell` provides the following methods to change the trim or status bar. These methods are device specific and not all platforms are able to support them. For example, Pocket PC cannot change the trim of `MobileShell` during runtime, and it has no status bar. The methods provided are:

- `changeTrim(int style, int statusStyle)`

  Changes the window trim mode. The implementation decides how to re-layout the content if the style changes.
- `getStatusText()`

  Returns the status text.
- `setStatusText(java.lang.String statusText)`

  Sets the status text. The change takes effect immediately.

FullScreenMode and KeyState

MobileShell has full-screen support and can keep key state for querying. The following methods are available:

- `setFullScreenMode(boolean mode)`

  Sets the screen mode. This method controls whether the shell is in normal or full-screen mode. If set to `true`, the shell switches to full-screen state. If set to `false` and the shell was in full-screen state, the shell switches back to the normal state.
- `getFullScreenMode()`

  Gets the full-screen state. Returns `true` if in full screen mode; `false` otherwise.

- `getKeyState(int keyCode)`

  Reports whether the key has been pressed. A return value of `true` indicates the key is pressed or has been pressed since the state of this key was last checked. A return value of `false` indicates the key has not been pressed since the state of this key was last checked. Only the active Shell can return `true` for this method. Calling this method on a `Shell` that is not the currently active always returns `false` and does not clear the key's has-been-pressed state.

Snippets

Example 4-7 creates an instance of `MobileShell` and sets it to be full-screen mode.

Example 4-7 MobileShell snippet

```
MobileShell mobileShell = new MobileShell(display, SWT.SHELL_TRIM, LARGE_STATUS_PANE);
mobileShell.setFullScreenMode(true);
```

4.1.8 SortedList

`SortedList` represents a selectable user interface object that displays a sorted list of text items. The items might be displayed in ascending or descending order. The sorting algorithm is platform and locale dependent.

If the `FILTER` style is specified during construction, an associated label is also displayed that shows characters that are entered. These characters are then used to filter the list to show fewer items. The selection state of items filtered out of the list is cleared.

Example

Figure 4-19 shows an instance of `SortedList` with `FILTER` style.

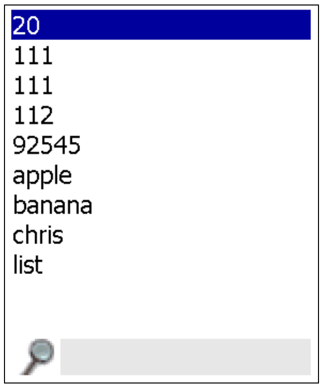


Figure 4-19 SortedList example

Styles

Table 4-10 shows the styles that `SortedList` supports.

Table 4-10 SortedList styles

Style	Description
SWT.SINGLE	Single selection
SWT.MULTI	Multiple selection
SWT.UP	Sorted so that numbers go from low to high when the list is examined from top to bottom
SWT.DOWN	Sorted so that numbers go from high to low when examined from top to bottom

Note: You can specify only one of `SINGLE` and `MULTI`. Only one of `UP` and `DOWN` can be specified.

Events

`SortedList` has one event, `SelectionEvent`, which can be accessed via following methods:

- `addSelectionListener(SelectionListener listener)`

  Adds the listener to the collection of listeners who are notified when the control is selected by sending it one of the messages that are defined in the `SelectionListener` interface.

- `removeSelectionListener(SelectionListener listener)`

  Removes the listener from the collection of listeners who are notified when the control is selected.

Setting, adding, and removing items

This section describes the methods to set, add, and remove items.

Setting items

Items inside `SortedList` can be initialized using the `setItems(java.lang.String[] items)` method. This method sets the `SortedList` items to be the given array of items.

Adding items

Items can also be added into `SortedList` using the `add(java.lang.String item)` method. This method adds the string item to the end of the `SortedList` list.

Removing items

For removing items, `SortedList` provides the following methods:

- `remove(java.lang.String string)`

  Searches the `SortedList` list, starting at the first item, until an item is found that is equal to the argument and removes that item from the list.

- `removeAll()`

  Removes all of the items from the `SortedList`.

Selection

`SortedList` provides several ways to set selection, almost all of which are index-based. The following methods are provided to set or to get selection:

- `setSelection(java.lang.String[] items)`

  Sets the `SortedList` selection to be the given array of items.

- `getSelection()`

  Returns an array of strings of items that are currently selected in the `SortedList`.

- `getSelectionCount()`

  Returns the number of items currently selected.

Snippets

Example 4-8 creates a `SortedList` instance with `SortedList.FILTER` style and adds nine items.

Example 4-8 SortedList snippet

```
SortedList sortedList = new SortedList(shell, SWT.SINGLE, SortedList.FILTER);
sortedList.add("111");
sortedList.add("apple");
sortedList.add("112");
sortedList.add("banana");
sortedList.add("92545");
sortedList.add("111");
sortedList.add("chris");
sortedList.add("list");
sortedList.add("20");
```

Figure 4-19 on page 71 shows all the items sorted in ascending order. When the user presses the *a* key, `SortedList` filters out all items that do not contain the letter *a*, as shown in Figure 4-20.

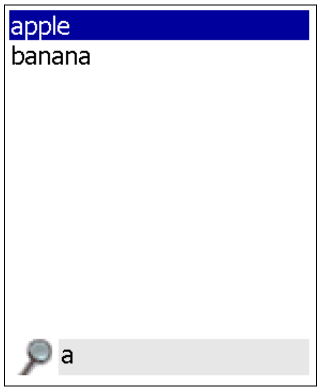


Figure 4-20 SortedList filter result

4.1.9 TextExtension

`TextExtension` contains methods for extending the functionality of the `Text` control. The functionality is specific to non-full keyboard devices. Applications can request certain text input modes when a `TextExtension` control gets focused. The effective input mode takes into account two style aspects. One aspect is based on the content semantics. The other is based on the content character set.

The editing behavior and appearance are otherwise identical to the `Text` control. The differences in `TextExtension` are to ease the possible switch of initial input modes, such as to enable or disable predictive input, changing initial casing styles and particular input modes of some languages. The initial input mode does not persist if it is changed by the user during editing. Whether the mode will be persist during the application life cycle is implementation-dependent.

Example

Figure 4-21 shows two examples of `TextExtension`, above one is `TextExtension.EMAILADDRESS` and below one is `TextExtension.URL`. Instances of `TextExtension` that are created with `TextExtension` styles display a hint to the user for input. This is just a hint that does not overwrite any input behavior of the `TextExtension`.

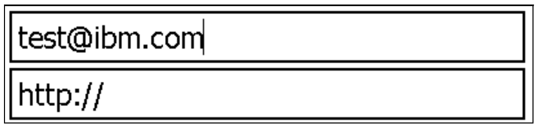


Figure 4-21 TextExtension example

Styles

`TextExtension` has two sets of styles. Table 4-11 shows the content styles, and Table 4-12 shows the input modifier styles. Content styles provide visual assistance to help user input, providing a user interface to access the device's personal information management (PIM) or browser bookmarks, if available. Input modifier styles act as a modifier to input behavior.

Table 4-11 Content styles

Content Styles	Description
EMAILADDRESS	An e-mail address
URL	A Web address

Table 4-12 Input modifier styles

Input Modifier Styles	Description
NON_PREDICTIVE	A hint for turning off possible predictive text input. By default any predictive input facilities should be turned on if available
LATIN_INPUT_ONLY	Forces that locale specific input modes should not be available. This is used in some situations when only Latin characters are allowed, for example, password text field

Note: `EMAILADDRESS` and `URL` are exclusive. You can use only one in content styles.

Set initial input mode

`TextExtension` provides a feature to set initial input mode to assist user input. The `setInitialInputMode(int casingModifier, java.lang.String mode)` method hints to the implementation as to the input mode that should be used when the user initiates editing of this control.

Table 4-13 contains the casing modifiers that you can use in `setInitialInputMode`. You can specify only one.

Table 4-13 Casing modifier

Modifier	Description
UPPERCASE	The capital letters of a typeface
LOWERCASE	The small letters of a typeface, as opposed to the capital letters, or uppercase letters
TEXTCASE	The first word of a sentence is capitalized
TITLECASE	Every word is capitalized

Snippets

Example 4-9 creates a instance of `TextExtension` and set its initial input mode with `TextExtension.TITLECASE` modifier and `UCB_BASIC_LATIN` mode.

Example 4-9 TextExtension snippet

```
TextExtension textExtension = new TextExtension(shell, SWT.SINGLE|SWT.BORDER);
textExtension.setInitialInputMode(TextExtension.TITLECASE, "UCB_BASIC_LATIN");
```

Figure 4-23 is the result when the user types the following:

```
This Is Textextension With Title Style
```

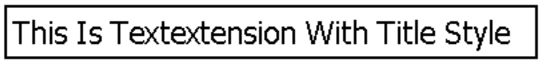

This Is Textextension With Title Style

Figure 4-22 TextExtension with TITLECASE

`TextExtension` transforms the inputted characters to follow `TITLECASE` modifier rule automatically. However, if you want to overwrite this input behavior, you can backspace to clear the characters and then type again. Modifier behavior is turned off when backspace is used. Figure 4-23 shows the result. The first characters of *style* are not forced to use a capital letter.

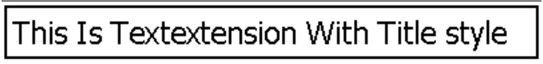

This Is Textextension With Title style

Figure 4-23 TextExtension after turning off modifier

4.2 Widgets

A widget is the interface element used to interact with the user. This section describes widgets that are provided by the eSWT mobile extensions.

4.2.1 TaskTip

`TaskTip` provides feedback to the user about the state of a long-running task.

A `TaskTip` can contain text and an optional progress bar or another object to illustrate current task state. Similar to a `MessageBox`, the look and feel of `TaskTip` is platform-dependent, which means that there is no API level access to the user interface components inside a `TaskTip`.

Unlike a `MessageBox`, the `TaskTip` is a non-focusable window and does not change the current window's activation or focus.

When constructed without a style constant, a `TaskTip` displays plain text to indicate task progress.

A `TaskTip` becomes visible by calling `setVisible(true)` and remains visible until the application calls `setVisible(false)`. When a new `TaskTip` is created before hiding or disposing of a prior `TaskTip`, the newest becomes the top-most `TaskTip` and obscures the prior ones, if any exist.

A `TaskTip` cannot be positioned programmatically or by the user. The position is implementation-dependent.

Example

Figure 4-24 shows an example of `TaskTip`.

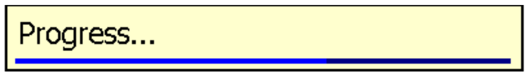


Figure 4-24 TaskTip example

Styles

Figure 4-14 shows the styles that are used in `TaskTip`.

Table 4-14 TaskTip styles

Styles	Description
SMOOTH	Displays a visual indicator of what portion of the progress is left to go
INDETERMINATE	Displays a visual indicator that a long-running process is progressing

Set text and progress indicator

`TaskTip` provides several methods to set text and progress indicator:

- `setMaximum(int value)`

  Sets the maximum value that the `TaskTip` allows.
- `setMinimum(int value)`

  Sets the minimum value that the `TaskTip` allows.
- `setSelection(int value)`

  Sets the position of the `TaskTip` indicator to the provided value.
- `setText(java.lang.String string)`

  Sets the label text.
- `setVisible(boolean visible)`

  Makes the `TaskTip` visible and brings it to the front of the display.

Note: `TaskTip` is intended to be invisible when created. The application can use `setVisible` to turn on and to turn off the display of `TaskTip`.

Snippets

Example 4-10 is a snippet for the result that is shown in Figure 4-24 on page 76.

Example 4-10 TaskTip snippet

```
TaskTip tasktip = new TaskTip(shell, SWT.SMOOTH);
tasktip.setMinimum(0);
tasktip.setMaximum(100);
tasktip.setSelection(63);
tasktip.setText("Progress...");
tasktip.setVisible(true);
```

4.3 Dialogs

This section describes dialogs that are provided by the eSWT mobile extensions.

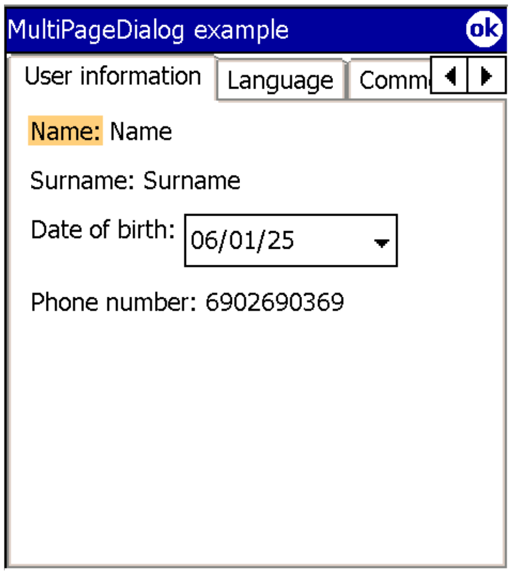
4.3.1 MultiPageDialog

Instances of `MultiPageDialog` represents a tabbed dialog. The dialog contains multiple pages. Each page contains a composite. At any given time, only one page is visible. The page visibility can be selected by users or applications programmatically. It provides function similar to `TabFolder` in desktop SWT, but in a more platform-dependent fashion.

Each page has a label. The platform might display the label as text, an icon, or both together. The size and position of page labels is implementation-dependent. There is no fixed limit on the number of pages. A runtime exception might be thrown when resources are insufficient to create a new page.

Example

Figure 4-25 shows an example of `MultiPageDialog`. For pages that cannot fit in the screen, `MultiPageDialog` provides two arrow buttons for user to browse.



Name: Name

Surname: Surname

Date of birth: 06/01/25

Phone number: 6902690369

Figure 4-25 MultiPageDialog example

Events

`MultiPageDialog` has one event, `SelectionEvent`, which can be accessed via the following methods:

- `addSelectionListener(SelectionListener listener)`

  Adds the listener to the collection of listeners who are notified when the page changes by sending it one of the messages that are defined in the `SelectionListener` interface.

  When `widgetSelected` is called, the item field of the event object is valid. `widgetDefaultSelected` is not called.

- `removeSelectionListener(SelectionListener listener)`

  Removes the listener from the collection of listeners who are notified when the `MultiPageDialog` selection changes.

Open and close dialog

Like other dialogs, MultiPageDialog has methods to open and close dialog, as follows:

- `open()`

  Makes the dialog visible, and brings it to the front of the display.

- `close()`

  Requests that the window manager close the dialog in the same way it would be closed when the user selects the close box or performs some other platform-specific key or mouse combination that indicates that the window should be removed.

Pages manipulation

`MultiPageDialog` provides several ways to perform pages operation, including the following:

- `createPage(java.lang.String title, Image icon)`

  Creates a new page with the specified title string and icon.

- `deletePage(int index)`

  Deletes the page from the `MultiPageDialog` at the given zero-relative index.

- `getPage(int index)`

  Returns the composite of the page at the given zero-relative index in the `MultiPageDialog`.

- `getPageCount()`

  Returns the number of pages contained in the `MultiPageDialog`.

- `getTitle(int index)`

  Returns the title string of the page at the given zero-relative index in the `MultiPageDialog`.

- `setImage(int index, Image icon)`

  Sets the icon image of the page at the given zero-relative index. Note that the icon size is implementation-dependent so that the icon image can be stretched or shrunk automatically.

- `setSelection(int index)`

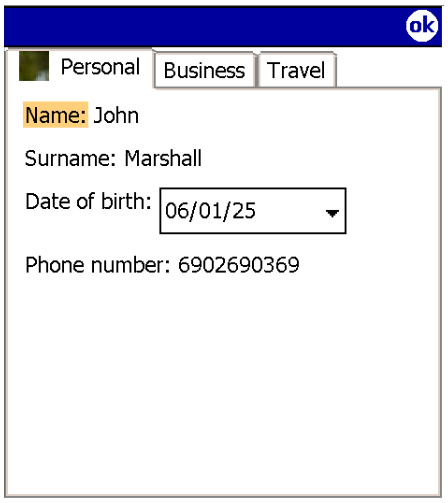
  Selects the active page by the given zero-relative index.

Snippets

Example 4-11 creates an instance of MultiPageDialog that is shown in Figure 4-26. It creates three pages. First one is created with an image set on the title. Within the first pages, it puts some fields to represent personal information.

Example 4-11 MultiPageDialog snippet

```
Composite pages[] = new Composite[3];
MultiPageDialog mpd = new MultiPageDialog(shell);
pages[0] = mpd.createPage("Personal",new Image(display,"\\eSWT\\childs.jpg"));
pages[1] = mpd.createPage("Business",null);
pages[2] = mpd.createPage("Travel",null);

pages[0].setLayout(new RowLayout(SWT.VERTICAL));

// Name
CaptionedControl name = new CaptionedControl(pages[0], SWT.LEFT_TO_RIGHT);
name.setText("Name:");
Text userName = new Text(name, SWT.SINGLE);
userName.setText("John");
userName.setEditable(true);

// Surname
CaptionedControl surname = new CaptionedControl(pages[0], SWT.LEFT_TO_RIGHT);
surname.setText("Surname:");
Text userSurname = new Text(surname, SWT.SINGLE);
userSurname.setText("Marshall");

// Date of birth
CaptionedControl birthDate = new CaptionedControl(pages[0], SWT.LEFT_TO_RIGHT);
birthDate.setSize(200,30);
birthDate.setText("Date of birth:");
DateEditor birthDateEditor = new DateEditor(birthDate, SWT.NORMAL, DateEditor.DATE |
DateEditor.FULL);

// Phone number
CaptionedControl phoneNumber = new CaptionedControl(pages[0], SWT.LEFT_TO_RIGHT);
phoneNumber.setText("Phone number:");
ConstrainedText userPhoneNumber = new ConstrainedText(phoneNumber, SWT.NORMAL,
ConstrainedText.PHONENUMBER);
userPhoneNumber.setText("6902690369");

pages[0].layout();
mpd.open();
```



Name: John

Surname: Marshall

Date of birth: 06/01/25

Phone number: 6902690369

Figure 4-26 MultiPageDialog snippet result

4.3.2 TimedMessageBox

`TimedMessageBox` is a modal window that is used to inform the user of limited information using a standard style.

A `TimedMessageBox` is capable of closing itself automatically after a certain period of time. There is no need to define button styles for `TimedMessageBox`.

Unlike `TaskTip`, the `TimedMessageBox` is a modal dialog and execution is blocked until the dialog is closed after a short period of time. The exact time-out duration is implementation dependent. Applications cannot change the period of time.

Example

Example 4-27 shows an example of `TimedMessageBox`.

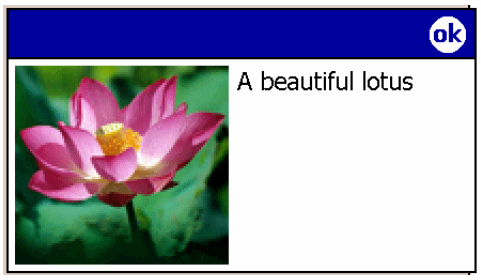


Figure 4-27 TimedMessageBox example

Styles

Table 4-15 shows the styles that are used in `TimedMessageBox`.

Table 4-15 TimedMessageBox styles

Styles	Description
ICON_WORKING	Informs that an action was successful
ICON_INFORMATION	Informs of a situation that might not require user action
ICON_WARNING	Informs of a situation that might require user intervention
ICON_ERROR	Informs that a serious situation has occurred

Set image and message

`TimedMessageBox` is similar to `MessageBox`, but it provides image support. The following methods are available with `TimedMessageBox`:

- `setImage(Image image)`

  Sets the `TimedMessageBox` icon image.
- `getImage()`

  Returns the icon object.
- `setMessage(java.lang.String string)`

  Sets the dialog's message, which is a description of the purpose for which it was opened.
- `getMessage()`

  Returns the dialog's message, which is a description of the purpose for which it was opened.

Snippets

Example 4-12 creates a `TimedMessageBox` with one icon that displays some messages as shown in Figure 4-27 on page 80.

Example 4-12 TimedMessageBox snippet

```
TimedMessageBox tmb = new TimedMessageBox(shell);
tmb.setImage(new Image(display, "\\eSWT\\lotus.png"));
tmb.setMessage("A beautiful lotus");
tmb.open();
```

4.4 Items

Items are widgets contained by other controls. This section describes items that are provided by the eSWT mobile extensions.

4.4.1 Command

`Command` is a metaphor that represents a general action. `Command` contains no information about the behavior that happens when a command is activated. The concrete action is defined in a `SelectionListener`. You can implement this command using any user interface construct that has semantics for activating a single action. Some implementations might implement commands as widgets, such as buttons or menu items, or voice tags. However, the implementation should not affect an application's layout adversely when realizing commands.

`Command` must be associated with a control and only becomes accessible when that control is in the current focus context. The current focus context includes the control that currently has focus and all of its visible ancestor controls up through the lowest level `Shell`. The term *visible* refers to widgets that are not explicitly hidden by calling `setVisible(false)`. The focus context does not include siblings of the control with focus or ancestors of the lowest level `Shell`.

Note: Use of `Command` is highly encouraged. `Command` is an abstract that the Mobile Extensions library maps to a specific mechanism, depending upon the device capabilities. This is usually pointer-driven menus or soft keys.

Example

Figure 4-28 shows an example of `TimedMessageBox` running on a Pocket PC.

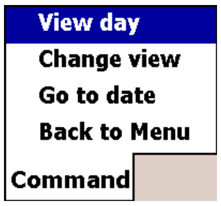


Figure 4-28 Command example

Types

The type value of `Command` describes `Command` behavior and priority for positioning hints, as shown in Table 4-16.

Table 4-16 Command type

Types	Description
BACK	Hints to the implementation to navigate backward
CANCEL	A standard negative answer to an action
COMMANDGROUP	Used for grouping commands
DELETE	Hints to the implementation to destroy data
EXIT	Indicates an exiting action
GENERAL	Can be bound to any hardware keys or softkeys
HELP	Specifies a request for online help
OK	Returns the user to the logical OK action
SELECT	Represents the context-sensitive action
STOP	Stops some currently running process, operation, and so forth

Note: When `Command` is created with another `Command` as parent, the parent `Command` type must be `COMMANDGROUP`. This type of `Command` does not fire `SelectionEvent`.

Events

`Command` has one event, `SelectionEvent`, which can be accessed via the following methods:

- `addSelectionListener(SelectionListener listener)`

  Adds the listener to the collection of listeners who are notified when the control is selected by sending it one of the messages that are defined in the `SelectionListener` interface.

- `removeSelectionListener(SelectionListener listener)`

  Removes the listener from the collection of listeners who are notified when the control is selected.

Long label and accelerator

In addition to `setText()`, which sets the preliminary information of `Command`, `Command` also provides `setLongLable()` to set longer information that is associated with `Command` if enough displaying area. The following methods are available:

- `setLongLabel(java.lang.String label)`

  Sets the command's long label text. Commands might optionally have long labels, enabling the application to specify a short form for button or softkey and a more verbose form when realized as menu item.

- `getLongLabel()`

  Returns the command's long label, which shall be `null` if it has never been set.

Default command and priority

`Command` is a mobile-specific control in eSWT mobile extension. It provides some methods to map buttons to default `Command` or high priority `Command`. This feature is platform dependent. For example, implementation on Pocket PC does not support mapping buttons due to no programmable buttons that are available on generic Pocket PC. The following methods are available:

- `setDefaultCommand()`

  Sets the default command.
- `getPriority()`

  Returns the command's priority value.
- `isEnabled()`

  Returns `true` if the `Command` is enabled; `false` otherwise.

Snippets

Example 4-13 creates one `Command` for `Shell` with `SELECT` style and one `Command` for `Text` with `SELECT`. Followed by creation of `Button`, it also creates a `CommandGROUP Command` and then associates three `Select Command` with this group `Command`.

Example 4-13 Command snippets

```
//Create a shell and set layout
Shell shell = new Shell(display);
RowLayout rowLayout = new RowLayout();
rowLayout.type = SWT.VERTICAL;
shell.setLayout(rowLayout);

// Command for Shell
Command shellCommand = new Command(shell, Command.SELECT, 0);
shellCommand.setText("Shell Command");

// Create a text and command for it
Text text = new Text(shell, SWT.SINGLE|SWT.BORDER);
text.setText("This is a text");
Command textCommand = new Command(text, Command.SELECT, 0);
textCommand.setText("Text Command");

// Create a button and a group command for it
Button button = new Button(shell, SWT.PUSH);
button.setText("Push Me");
Command groupCommand = new Command(button, Command.COMMANDGROUP, 0);
groupCommand.setText("Button Command");

// Create a set of commands associated with group command
Command buttonCommand1 = new Command(groupCommand, Command.SELECT, 0);
buttonCommand1.setText("Inside group - 1");
Command buttonCommand2 = new Command(groupCommand, Command.SELECT, 0);
buttonCommand2.setText("Inside group - 2");
Command buttonCommand3 = new Command(groupCommand, Command.SELECT, 0);
buttonCommand3.setText("Inside group - 3");
```

Figure 4-29 shows the result of this snippet. When text gets focus, its associated `Command` shows along with the parent's `Command` (`shellCommand`). When the user selects `Button`, the `Command` list changes automatically to contain `Button Command` and the parent's `Command` as well (as shown in Figure 4-30 on page 84).

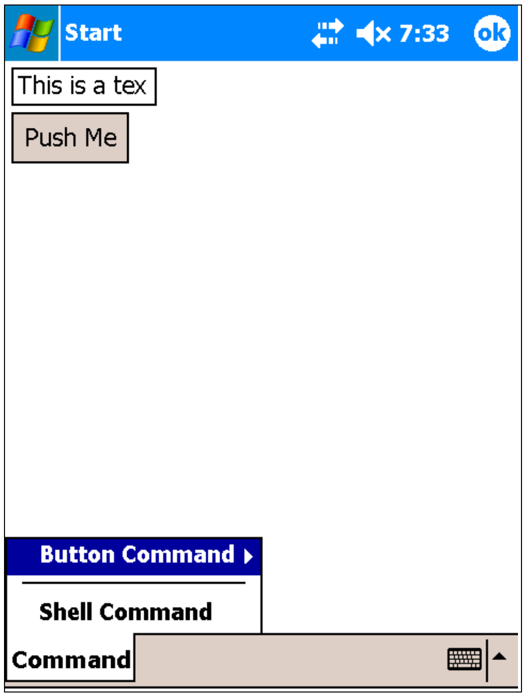


Figure 4-29 Command example when text gets focus

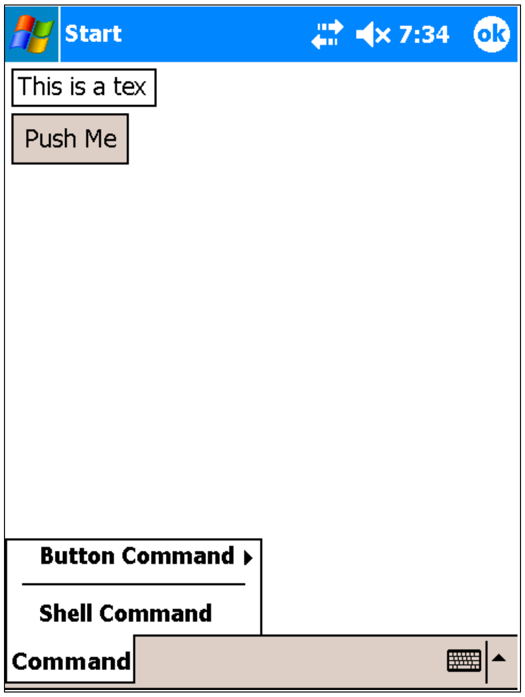


Figure 4-30 Command example when button gets focus

Group `Command` does not fire `SelectionEvent`. When the user selects it, it brings up the `Command` that is associated with it (Figure 4-31).

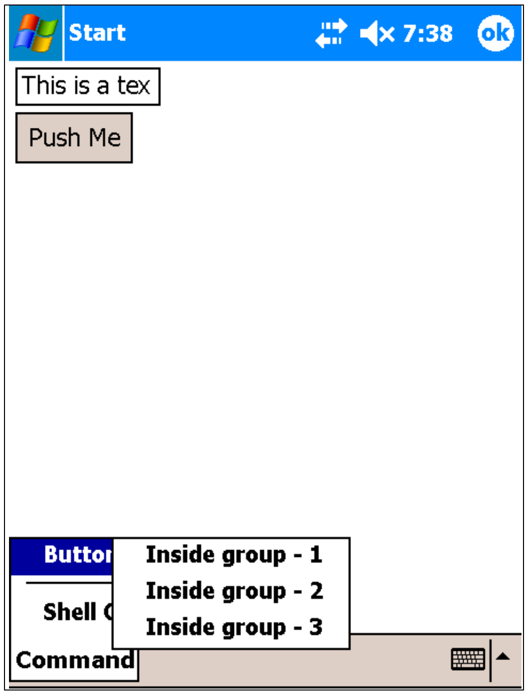


Figure 4-31 Group command

4.5 Device-related

This section describes device-related methods.

4.5.1 MobileDevice

`MobileDevice` represents the device that is being used. It provides methods that enable applications to learn more about the device-specific characteristics and capabilities.

Applications can query what input features and display screens are a part of the device permanently. These features are considered *local* features. Some devices also allow input mechanisms or screens to be attached to the device at runtime. These features are considered *remote* features. Because local features do not come and go, it is sufficient to query for them once. On the other hand, because remote devices can be added or removed at any time, an application needs to add a `MobileDeviceListener` to be informed of these events.

Local features can also be internal or external. External features are only available when a device is closed. Internal features are only available when a device is opened.

MobileDeviceEvent

`MobileDevice` has its own event, `MobileDeviceEvent`, which is sent as a result of device configuration change. An application can add `MobileDeviceListener` to get notified when an event is triggered. The following methods are defined in `MobileDeviceListener`:

- `deviceChanged(MobileDeviceEvent event)`

  Called when a device configuration has been changed, such as opened or closed.
- `inputChanged(MobileDeviceEvent event)`

  Called when the input configuration has been changed.
- `screenChanged(MobileDeviceEvent event)`

  Called when a screen configuration has been changed.

Applications can add and remove `MobileDeviceListener` via the following methods:

- `addMobileDeviceListener(MobileDeviceListener listener)`

  Adds the listener to the collection of listeners who are notified when a device configuration change occurs by calling one of the methods that are defined in the `MobileDeviceListener` interface.
- `removeMobileDeviceListener(MobileDeviceListener listener)`

  Removes the listener from the collection of listeners who are notified when a device configuration change occurs.

Get devices

Application cannot create an instance of `MobileDevice` via `MobileDevice()`. To get an instance of `MobileDevice`, an application should use `getMobileDevice()` to get an *static* instance of `MobileDevice`. Mobile devices are not intended to be changed by an application. `MobileDevice` acts as a static object for an application to query device configuration and get notified when configuration change. The following methods are available:

- `getMobileDevice()`

  Returns singleton instance of `MobileDevice` class.
- `getInputs()`

  Returns an array of Input objects describing the input features available to the device.
- `getScreens()`

  Returns an array of screen objects describing the display features available to the device.

Snippets

Example 4-14 shows how to get an instance of `MobileDevice` and then gets device configuration, including screens and inputs.

Example 4-14 MobileDevice snippets

```
final MobileDevice md = MobileDevice.getMobileDevice();
Screen[] screens = md.getScreens();
Input[] inputs = md.getInputs();
```

4.5.2 Screen

`Screen` represents display screens available for application use.

Types and styles

Table 4-17 lists `Screen` usage types, and Table 4-18 lists orientation style.

Table 4-17 Screen usage types

Types	Description
PRIMARY	One primary screen can be active at a time
SECONDARY	Multiple secondary screens can be active simultaneously
STATUS	Sub type of secondary screen that shows minimal content for notification purposes

Table 4-18 Orientation style

Styles	Description
LANDSCAPE	Indicates text is written normally across the longest width
PORTRAIT	Indicates text is written normally across the shortest width

Events

`Screen` defines its own event, `ScreenEvent`, when screens configuration change. An application can add `ScreenEvent` to get notified when an event is triggered. The following methods are defined in `ScreenListener`:

- `screenActivated(ScreenEvent event)`

  Sent when the screen is activated.
- `screenDeactivated(ScreenEvent event)`

  Sent when the screen is deactivated.
- `screenOrientationChanged(ScreenEvent event)`

  Sent when the screen's orientation is changed.

The application can add and remove `ScreenListener` via the following methods:

- `addEventListener(ScreenListener listener)`

  Adds the listener to the collection of listeners that are notified when screen events occur by sending it one of the messages that are defined in the `ScreenListener` interface.
- `removeEventListener(ScreenListener listener)`

  Removes the listener from the collection of listeners that are notified when screen events occur.

Get screen configuration

`Screen` provides the following methods to get screen configuration:

- `getBounds()`

  Returns the bounds of the screen.
- `getColorDepth()`

  Returns the color depth of the screen in bits per pixel.

- `getLocation()`

  Returns the location of the screen device.
- `getOrientation()`

  Returns the screen orientation.
- `getUsage()`

  Returns the usage type of the screen.

In addition to retrieval of screen configuration, `Screen` also provides one method to change the orientation of `Screen`. The `setOrientation(int orientation)` method sets the screen orientation if supported on this device.

Snippets

Example 4-15 gets an array of `Screen` and outputs the bound of each screen.

Example 4-15 Screen snippet

```
final MobileDevice md = MobileDevice.getMobileDevice();
Screen[] screens = md.getScreens();
for (int i=0; i<screens.length; i++) {
   Rectangle bounds = screens[i].getBounds();
   System.out.println("screen "+i+": width="+bounds.width+", height="+bounds.height);
}
```

4.5.3 Input

`Input` represents key-based input features.

Types

Table 4-19 shows the types of input.

Table 4-19 Table types

Types	Description
FULL_KEYBOARD	Input feature has hardware keys typical in a full keyboard
KEYPAD	Input feature has hardware keys labeled 0 through 9, asterisk (*), and number sign (#)
LIMITED_KEYBOARD	Input feature has more hardware keys than a keypad but fewer than a full keyboard
SOFTKEYS	Input feature has one or more hardware keys whose meaning can be configured

Get input configuration

`Input` provides the following methods to retrieve input configuration:

- `getLocation()`

  Returns the location of the input device.
- `getType()`

  Returns the type of input device.

5

eSWT expanded

This chapter provides a detailed description of the following components found within embedded Standard Widget Toolkit (eSWT) expanded:

- **Layouts**
  - FillLayout
  - RowLayout
  - GridLayout
- **Dialogs**
  - ColorDialog
  - DirectoryDialog
  - FontDialog
- **Controls**
  - Table
  - Tree
- **Browser**
  - Browser

5.1 Layouts

Layouts are used to arrange controls. This section provides details about additional layouts that are supported in the eSWT expanded optional component.

5.1.1 FillLayout

`FillLayout` is the simplest layout class. It controls layout using a single row or column. Available space is distributed evenly between children.

Public Fields

Table 5-1 list the `FillLayout` public fields to assist layout in an eSWT application.

Table 5-1 Public fields of FillLayout

Field	Description
marginHeight	Specifies the number of pixels of vertical margin that are placed along the top and bottom edges of the layout
marginWidth	Specifies the number of pixels of horizontal margin that are placed along the left and right edges of the layout
spacing	Specifies the number of pixels between the edge of one cell and the edge of its neighboring cell
type	Specifies how controls are positioned within the layout

Snippets

`FillLayout` is commonly used to lay out a single child. You can also use it to lay out multiple children. Example 5-1 creates a shell that contains only one button. Example 5-2 creates five buttons.

Example 5-1 Single child in FillLayout

```
shell.setLayout(new FillLayout());
Button button = new Button(shell, SWT.PUSH|SWT.BORDER);
button.setText("One");
```

Example 5-2 Multiple children in FillLayout

```
shell.setLayout(new FillLayout());
Button button1 = new Button(shell, SWT.PUSH|SWT.BORDER);
button1.setText("One");
Button button2 = new Button(shell, SWT.PUSH|SWT.BORDER);
button2.setText("Two");
Button button3 = new Button(shell, SWT.PUSH|SWT.BORDER);
button3.setText("Three");
Button button4 = new Button(shell, SWT.PUSH|SWT.BORDER);
button4.setText("Four");
Button button5 = new Button(shell, SWT.PUSH|SWT.BORDER);
button5.setText("Five");
```

Figure 5-1 and Figure 5-2 shows the results of these examples.

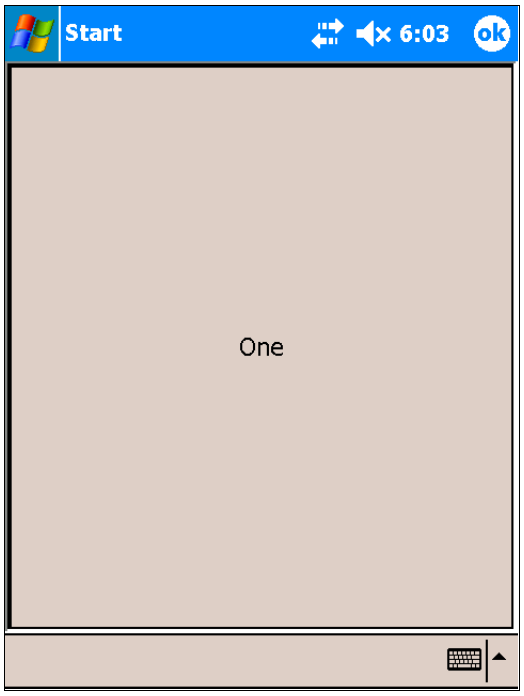


Figure 5-1 One button occupies the entire shell

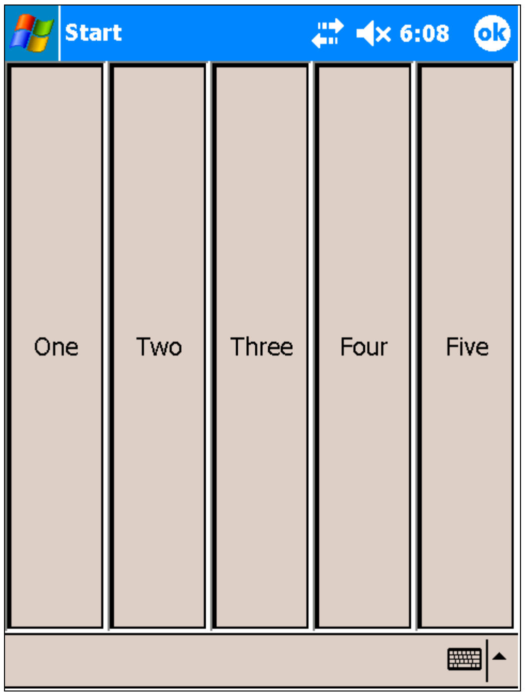


Figure 5-2 Multiple buttons occupy the shell evenly

5.1.2 RowLayout

Unlike `FillLayout`, `RowLayout` lays out its children in a more elegant way, either vertically or horizontally, depending on the type that is specified in `RowLayout`. `RowLayout` also has the ability to wrap and provides configurable margins and spacing. In addition, you can specify the height and width of each control in a `RowLayout` by setting a `RowData` object into the control using `setLayoutData()`.

Public Fields

Table 5-2 lists the public fields to assist in the layout process in an eSWT application.

Table 5-2 Public fields of RowLayout

Field	Description
fill	Specifies whether the controls in a row should be all the same height for horizontal layouts, or the same width for vertical layouts
justify	Specifies whether the controls in a row should be fully justified, with any extra space placed between the controls
marginBottom	Specifies the number of pixels of vertical margin that are placed along the bottom edge of the layout
marginHeight	Specifies the number of pixels of vertical margin that are placed along the top and bottom edges of the layout
marginLeft	Specifies the number of pixels of horizontal margin that are placed along the left edge of the layout
marginRight	Specifies the number of pixels of horizontal margin that are placed along the right edge of the layout
marginTop	Specifies the number of pixels of vertical margin that are placed along the top edge of the layout
marginWidth	Specifies the number of pixels of horizontal margin that are placed along the left and right edges of the layout
pack	Specifies whether all controls in the layout take their preferred size
spacing	Specifies the number of pixels between the edge of one cell and the edge of its neighboring cell
type	Specifies whether the layout places controls in rows or columns; the default is `SWT.HORIZONTAL`
wrap	Specifies whether a control is wrapped to the next row if there is insufficient space on the current row

Snippets

This section includes several examples to demonstrate how `RowLayout` works.

Normal RowLayout

Example 5-3 changes the layout that is used in Example 5-2 on page 90 to `RowLayout`.

Example 5-3 RowLayout snippets

```
shell.setLayout(new RowLayout());
Button button1 = new Button(shell, SWT.PUSH|SWT.BORDER);
button1.setText("One");
Button button2 = new Button(shell, SWT.PUSH|SWT.BORDER);
button2.setText("Two");
Button button3 = new Button(shell, SWT.PUSH|SWT.BORDER);
button3.setText("Three");
Button button4 = new Button(shell, SWT.PUSH|SWT.BORDER);
button4.setText("Four");
Button button5 = new Button(shell, SWT.PUSH|SWT.BORDER);
button5.setText("Five");
```

Figure 5-3 shows the result of this change.

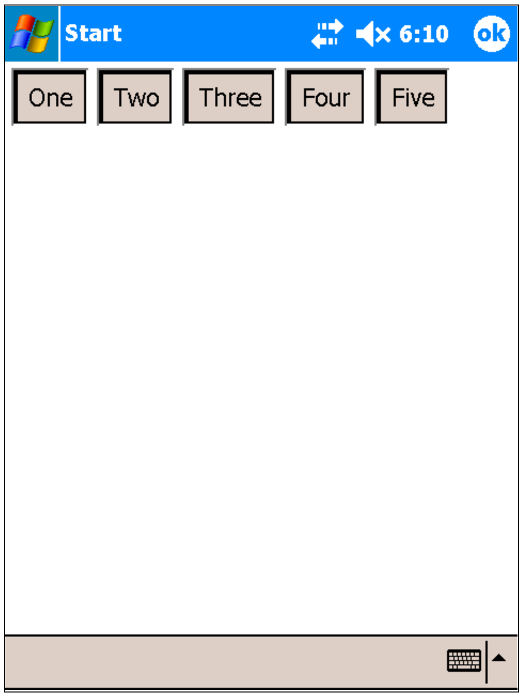


Figure 5-3 Horizontal RowLayout

Figure 5-4 shows how a vertical layout works when the `type` set in `RowLayout` is `SWT.VERTICAL`.

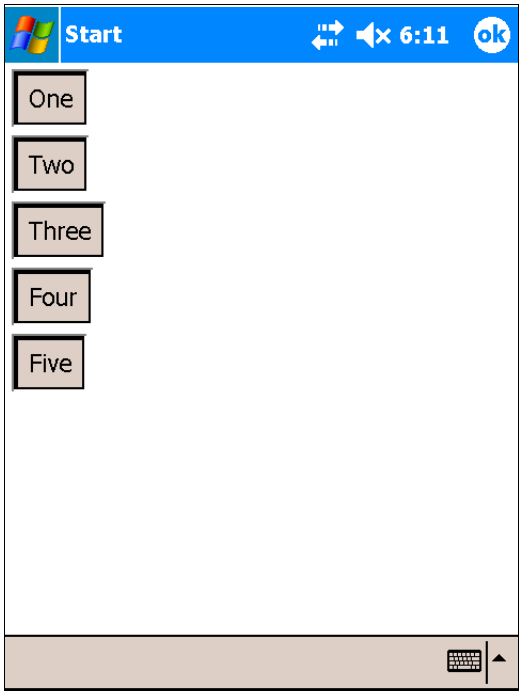


Figure 5-4 Vertical RowLayout

RowLayout with wrap

If the total width of the children plus all spacings over the width of parent composite or shell, `RowLayout` can help to perform wrapping if wrap is set. Example 5-4 shows the same code snippet with and without wrap (Figure 5-5 on page 95 and at Figure 5-6 on page 95).

Example 5-4 RowLayout with wrap

```
RowLayout rowLayout = new RowLayout();
rowLayout.wrap = true;
shell.setLayout(rowLayout);
for (int i=0; i<20; i++) {
   Button button = new Button(shell, SWT.PUSH|SWT.BORDER);
   button.setText("Btn"+i);
}
```

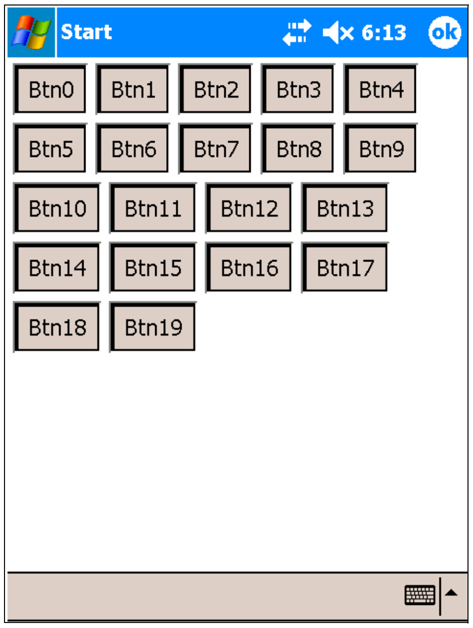


Figure 5-5 RowLayout with wrap

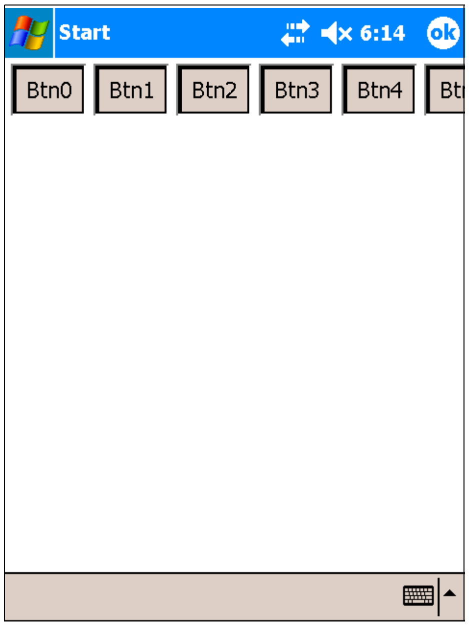


Figure 5-6 RowLayout without wrap

RowLayout with fill

If `RowLayout` is dealing with different types of controls, each control has their own width and height. What `RowLayout` normally does is lay out them in a row (if *type* is set to be `SWT.HORIZONTAL`) with all control aligned at the top but the bottom varies, depending on control's height. In order to make all controls aligned both on the top and bottom (with the same height or width), the `fill` type is used and it works inversely with the `type` of layout.

Example 5-5 shows how fill used in layout for two buttons and a label in a column. All controls are set as the same width as shown at Figure 5-7.

Example 5-5 RowLayout with fill

```
RowLayout rowLayout = new RowLayout(SWT.VERTICAL);
rowLayout.fill = true;
shell.setLayout(rowLayout);
for (int i=0; i<2; i++) {
   Button button = new Button(shell, SWT.PUSH|SWT.BORDER);
   button.setText("Btn"+i);
}
Image image = new Image(display,"\\eSWT\\childs.jpg");
Label label = new Label(shell,SWT.BORDER);
label.setImage(image);
```

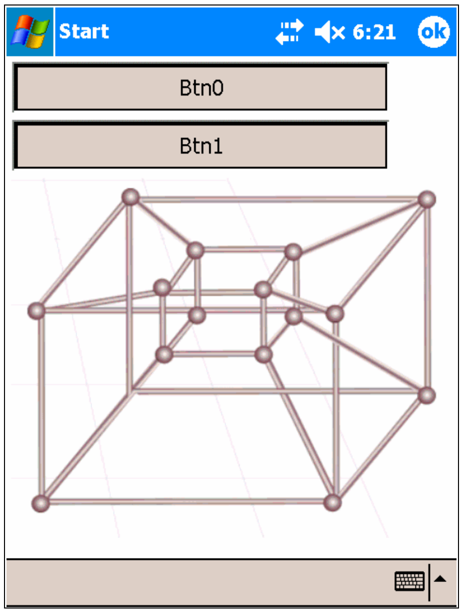


Figure 5-7 RowLayout with fill

RowLayout with RowData

The way `RowLayout` lays out children is to call `Control.computeSize()` to get the preferred size of each control. If the application wants to set the size of control explicitly, it can use `RowData` to override the `RowLayout` default behavior.

Example 5-6 creates four buttons with different `RowData` laid out in a column. The result is shown at Figure 5-8 on page 97.

Example 5-6 RowLayout with RowData

```
RowLayout rowLayout = new RowLayout(SWT.VERTICAL);
shell.setLayout(rowLayout);
for (int i=0; i<4; i++) {
   Button button = new Button(shell, SWT.PUSH|SWT.BORDER);
   button.setText("Btn"+i);
   button.setLayoutData(new RowData(20*(i+1), 10*(i+1)));
}
```

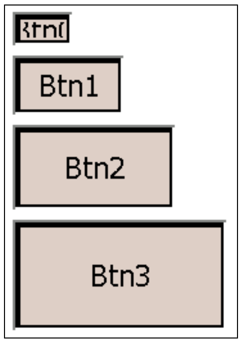


Figure 5-8 Controls with different RowData

5.1.3 GridLayout

`GridLayout` is a powerful layout tool that provides a good means to position each control within a shell. `GridLayout` lays out all children in a row. An important field, `numColumns`, controls how many columns are used in the layout process. The total number of children and number of columns are used by `GridLayout` to determine the number of rows and total cells implicitly. Applications can align each control to any side of the cell. In addition, controls can span cells either horizontally or vertically.

Public fields

Table 5-3 lists the public fields to assist layout in an eSWT application.

Table 5-3 Public fields of GridLayout

Field	Description
horizontalSpacing	Specifies the number of pixels between the right edge of one cell and the left edge of its neighboring cell to the right
makeColumnsEqualWidth	Specifies whether all columns in the layout are forced to have the same width
marginBottom	Specifies the number of pixels of vertical margin that are placed along the bottom edge of the layout
marginHeight	Specifies the number of pixels of vertical margin that are placed along the top and bottom edges of the layout
marginLeft	Specifies the number of pixels of horizontal margin that are placed along the left edge of the layout
marginRight	Specifies the number of pixels of horizontal margin that are placed along the right edge of the layout
marginTop	Specifies the number of pixels of vertical margin that are placed along the top edge of the layout
marginWidth	Specifies the number of pixels of horizontal margin that are placed along the left and right edges of the layout
numColumns	Specifies the number of cell columns in the layout
verticalSpacing	Specifies the number of pixels between the bottom edge of one cell and the top edge of its neighboring cell underneath

GridData

GridData provides an elegant way to configure many layout attributes within a single cell. Table 5-4 lists the public fields that provide flexibility over position and size of control .

Note: Do not reuse GridData objects. Every control in a Composite that is managed by a GridLayout must have a unique GridData object. If the layout data for a control in a GridLayout is null at layout time, a unique GridData object is created for it.

Table 5-4 GridData public fields

Fields	Description
exclude	Informs the layout to ignore this control when sizing and positioning controls
FILL	Value for horizontalAlignment or verticalAlignment
grabExcessHorizontalSpace	Specifies whether the width of the cell changes depending on the size of the parent Composite
grabExcessVerticalSpace	Specifies whether the height of the cell changes depending on the size of the parent Composite
heightHint	Specifies the preferred height in pixels
horizontalAlignment	Specifies how controls will be positioned horizontally within a cell
horizontalIndent	Specifies the number of pixels of indentation that are placed along the left side of the cell
horizontalSpan	Specifies the number of column cells that the control takes up
minimumHeight	Specifies the minimum height in pixels
minimumWidth	Specifies the minimum desired width in pixels
verticalAlignment	Specifies how controls are positioned vertically within a cell
verticalIndent	Specifies the number of pixels of indentation that are placed along the top side of the cell
verticalSpan	Specifies the number of row cells that the control takes up
widthHint	Specifies the preferred width in pixels

Snippets

This section includes examples that demonstrate how GridLayout works.

Normal GridLayout

Example 5-7 creates 20 buttons and uses GridLayout with four columns.

Example 5-7 GridLayout snippet

```
GridLayout gridLayout = new GridLayout();
shell.setLayout(gridLayout);
gridLayout.numColumns = 4;
for (int i=0; i<20; i++) {
   Button button = new Button(shell, SWT.PUSH|SWT.BORDER);
   button.setText("Btn"+i);
}
```

Figure 5-9 shows the result of this example.

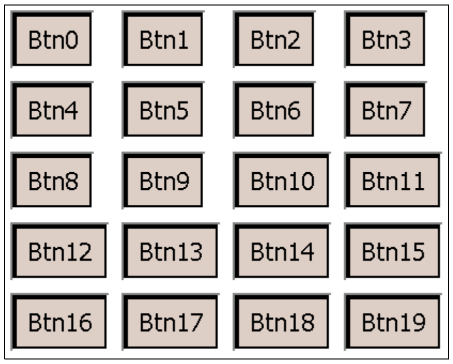


Figure 5-9 GridLayout with four columns

If you change the number of columns from four to three, `GridLayout` uses only three columns to place these controls, as shown in Figure 5-10.

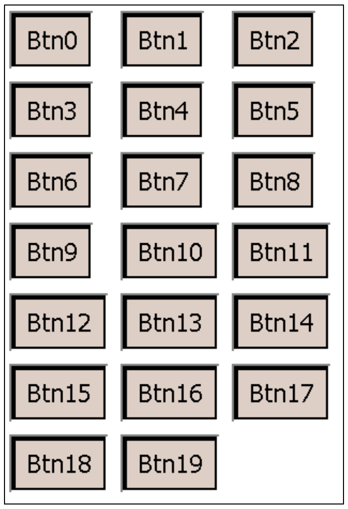


Figure 5-10 GridLayout with three columns

GridLayout with equal width

In `GridLayout`, the width of each column depends on the width of the widest control within one column. Applications can use `makeColumnsEqualWidth` to set each column with the same width.

Figure 5-11 shows when `makeColumnsEqualWidth` is not set, and Figure 5-12 shows when `makeColumnsEqualWidth` is set.

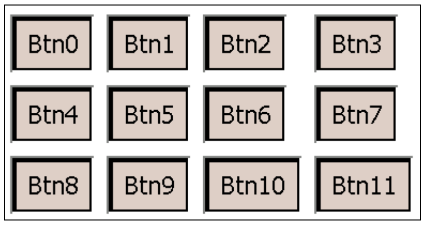


Figure 5-11 makeColumnsEqualWidth is not set

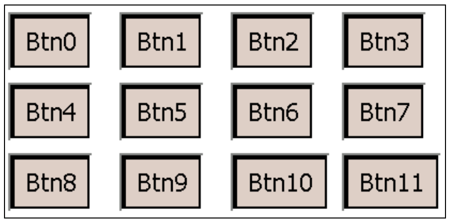


Figure 5-12 makeColumnEqualWidth is set

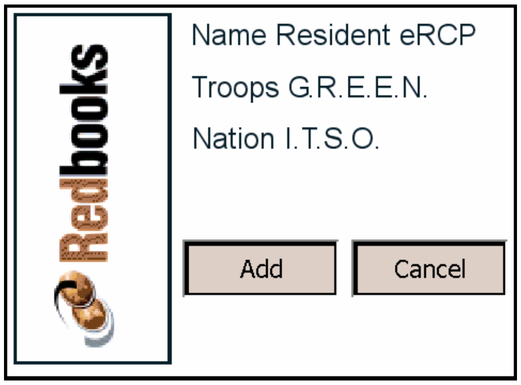
GridLayout with GridData

You can use `GridData` to write a typical profile dialog, along with a `CaptionedControl`. (For information about `CaptionedControl`, see 4.1.1, “CaptionedControl” on page 52.)
Example 5-8 creates a three column `GridLayout` with a photo label that expands to four cells. It then has three `CaptionedControl` controls that contain personal information, including name, troop, and nation (all expand their width to two-cell width). The remaining two buttons use `SWT.Fill` and `SWT.CENTER` to position themselves in their cells.

Example 5-8 GridLayout with GridLayout

```
//Create a shell and set layout
Shell shell = new Shell(display);
Shell smallShell = new Shell(shell,SWT.DIALOG_TRIM|SWT.CLOSE);
Label photo = new Label(smallShell, SWT.NONE);
Image image = new Image(display,
HelloWorldeSWT.class.getResourceAsStream("res/Resident.jpg"));
photo.setImage(image);
CaptionedControl nameCaption = new CaptionedControl(smallShell, SWT.NONE);
nameCaption.setText("Name");
Text nameText = new Text(nameCaption, SWT.SINGLE);
nameText.setText("Resident eRCP");
CaptionedControl troopCaption = new CaptionedControl(smallShell, SWT.NONE);
troopCaption.setText("Troops");
Text troopText = new Text(troopCaption, SWT.SINGLE);
troopText.setText("G.R.E.E.N.");
CaptionedControl nationCaption = new CaptionedControl(smallShell, SWT.NONE);
nationCaption.setText("Nation");
Text nationText = new Text(nationCaption, SWT.SINGLE);
nationText.setText("I.T.S.O.");
Button addButton = new Button(smallShell,SWT.PUSH|SWT.BORDER);
addButton.setText("Add");
Button cancelButton = new Button(smallShell,SWT.PUSH|SWT.BORDER);
cancelButton.setText("Cancel");
smallShell.setSize(240,200);

//set the layout
GridLayout gridLayout = new GridLayout(3, true);
gridLayout.marginWidth = gridLayout.marginHeight = 3;
GridData labelData = new GridData(SWT.FILL, SWT.CENTER, true, true,1,4);
photo.setLayoutData(labelData);
GridData nameCaptionData = new GridData(SWT.FILL, SWT.LEFT, true, false, 2,1);
nameCaption.setLayoutData(nameCaptionData);
GridData troopCaptionData = new GridData(SWT.FILL, SWT.LEFT, true, false, 2,1);
troopCaption.setLayoutData(troopCaptionData);
GridData nationCaptionData = new GridData(SWT.FILL, SWT.LEFT, true, false, 2,1);
nationCaption.setLayoutData(nationCaptionData);
GridData addButtonData = new GridData(SWT.FILL, SWT.CENTER, false, true);
addButton.setLayoutData(addButtonData);
GridData cancelButtonData = new GridData(SWT.FILL, SWT.CENTER, false, true);
cancelButton.setLayoutData(cancelButtonData);
smallShell.setLayout(gridLayout);
```

Figure 5-13 shows the result of this example.


Name Resident eRCP

Troops G.R.E.E.N.

Nation I.T.S.O.

Figure 5-13 GridLayout with GridData

5.1.4 ColorDialog

`ColorDialog` allows the user to select a color from a predefined set of available colors.

Manipulate RGB values

`ColorDialog` provides the following methods to set and get RGB values:

- `getRGB()`

  Returns the currently selected color in the `ColorDialog`.

- `setRGB(RGB rgb)`

  Sets the `ColorDialog` color value.

Snippets

Example 5-9 shows how to open a `ColorDialog`.

Example 5-9 ColorDialog snippet

```
ColorDialog colorDialog = new ColorDialog(shell);
colorDialog.open();
```

Figure 5-14 shows this example running on a Pocket PC.

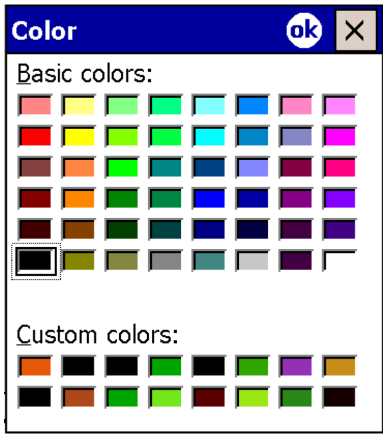


Figure 5-14 ColorDialog example

5.1.5 DirectoryDialog

`DirectoryDialog` allows the user to navigate the file system and select a directory.

FilterPath and Message

Applications can set the initial directory path in `DirectoryDialog` and a message when the dialog opens. The following methods are available:

- `setFilterPath(java.lang.String string)`

  Sets the path that the dialog uses to filter the directories that it shows to the argument, which might be `null`.

- `getFilterPath()`

  Returns the path which the dialog uses to filter the directories that it shows.

- `setMessage(java.lang.String string)`

  Sets the dialog's message, which is a description of the purpose for which it was opened.

- `getMessage()`

  Returns the dialog's message, which is a description of the purpose for which it was opened.

Snippets

Example 5-10 opens a `DirectoryDialog` and set its message.

Example 5-10 DirectoryDialog snippet

```
DirectoryDialog directoryDialog = new DirectoryDialog(shell);
directoryDialog.setMessage("Choose the path...");
directoryDialog.open();
```

5.1.6 FontDialog

`FontDialog` allows the user to select a font from all available fonts in the system.

Font list and RGB values

An application can set or get a font list and RGB with `FontDialog` using the following methods:

- `setFontList(FontData[] fontData)`

  Sets a set of `FontData` objects describing the font to be selected by default in the dialog, or `null` to let the platform choose one.

- `getFontList()`

  Returns a `FontData` set describing the font that was selected in the dialog, or `null` if none is available.

- `setRGB(RGB rgb)`

  Sets the `FontDialog`'s selected color.

- `getRGB()`

  Returns the currently selected color in the `FontDialog`.

Snippets

Example 5-11 opens a `FontDialog`.

Example 5-11 FontDialog snippet

```
FontDialog fontDialog = new FontDialog(shell);
fontDialog.open();
```

5.1.7 Table

`Table` implements a selectable user interface object that displays a list of images and strings and issues notifications when items are selected. The children that might be added to instances of this class must be of type `TableItem`.

Note: Although `Table` is a subclass of `Composite`, it does not make sense to add `Control` children to it or to set a layout on it.

Styles

Table 5-5 shows the styles used in `Table`.

Table 5-5 Table styles

Styles	Description
`SWT.SINGLE`	Creates a single select Table
`SWT.MULTI`	Creates a multiple select Table
`SWT.CHECK`	Creates a table with check boxes
`SWT.FULL_SELECTION`	The selection effect will expand to entire row when selected
`SWT.HIDE_SELECTION`	The selection is hidden when focus out

Events

`Table` has one event, `SelectionEvent`, which can be accessed via the following methods:

- `addSelectionListener(SelectionListener listener)`

  Adds the listener to the collection of listeners who are notified when the control is selected by sending it one of the messages that are defined in the `SelectionListener` interface.

- `removeSelectionListener(SelectionListener listener)`

  Removes the listener from the collection of listeners who are notified when the control is selected.

TableItem and TableColumn

`TableItem` is a fundamental object to represent an item or items in a `Table`. `TableColumn` represents a column in a `Table`. Applications can create table items without creating columns (in this case, a `Table` looks like a list).

`Table` provides the following methods to get and to remove `TableItem` and `TableColumn` objects:

- `getColumn(int index)`

  Returns the column at the given, zero-relative index in the `Table`.

- `getColumnCount()`

  Returns the number of columns contained in the `Table`.

- `getColumns()`

  Returns an array of `TableColumn` which are the columns in the `Table`.

- `getItem(int index)`

  Returns the item at the given, zero-relative index in the `Table`.

- `getItem(Point point)`

  Returns the item at the given point in the `Table` or null if no such item exists.

- `getItemCount()`

  Returns the number of items contained in the `Table`.

- `getItemHeight()`

  Returns the height of the area which would be used to display one of the items in the `Table`.

- `getItems()`

  Returns a (possibly empty) array of `Table`Item which are the items in the `Table`.

- `remove(int index)`

  Removes the item from the `Table` at the given zero-relative index.

- `remove(int[] indices)`

  Removes the items from the `Table` list at the given zero-relative indices.

- `remove(int start, int end)`

  Removes the items from the `Table` which are between the given zero-relative start and end indices (inclusive).

- `removeAll()`

  Removes all of the items from the `Table`.

Selection

`Table` provides several ways to set selection, almost all of which are index-based. The following methods are provided to set or to get selection:

- `setSelection(int index)`

  Selects the item at the given zero-relative index in the `Table`.

- `setSelection(int[] indices)`

  Selects the items at the given zero-relative indices in the `Table`.

- `setSelection(int start, int end)`

  Selects the items in the range specified by the given zero-relative indices in the `Table`.
- `setSelection(TableItem[] items)`

  Sets the `Table`'s selection to be the given array of items.
- `getSelection()`

  Returns an array of `TableItem` that is currently selected in the `Table`.
- `getSelectionCount()`

  Returns the number of items currently selected.
- `getSelectionIndex()`

  Returns the zero-relative index of the item which is currently selected in the `Table`, or -1 if no item is selected.
- `getSelectionIndices()`

  Returns the zero-relative indices of the items which are currently selected in the `Table`.

It also provides several methods to perform deselecting or selecting items:

- `deselect(int index)`

  Deselects the item at the given zero-relative index in the `Table`.
- `deselect(int[] indices)`

  Deselects the items at the given zero-relative indices in the `Table`.
- `deselect(int start, int end)`

  Deselects the items at the given zero-relative indices in the `Table`.
- `deselectAll()`

  Deselects all selected items in the `Table`.
- `select(int index)`

  Selects the item at the given zero-relative index in the `Table.`
- `void select(int[] indices)`

  Selects the items at the given zero-relative indices in the `Table`.
- `void select(int start, int end)`

  Selects the items in the range specified by the given zero-relative indices in the `Table`.
- `void selectAll()`

  Selects all of the items in the `Table`.

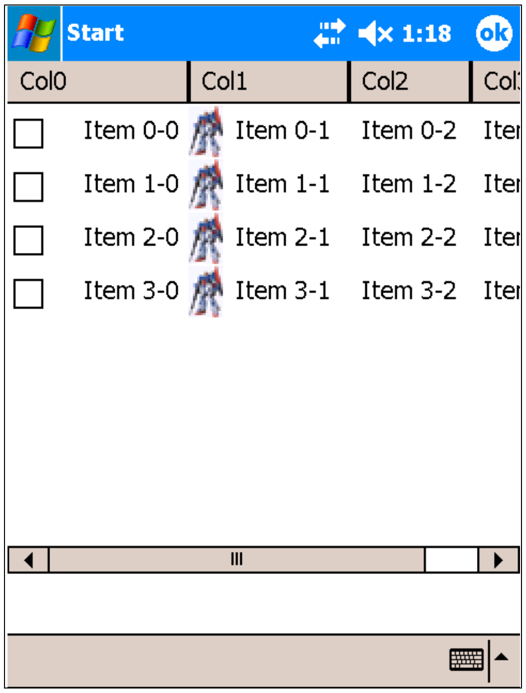
Snippets

Example 5-12 creates a multi-column `Table` with the `SWT.CHECK` style. The second column adds images to the whole column. When the screen width cannot display the whole `Table`, `Table` brings up a scroll bar automatically.

Example 5-12 Table snippets

```
Table table = new Table(shell, SWT.SINGLE|SWT.CHECK);
int COLUMNS = 4, ROWS = 4;
table.setHeaderVisible(true);
for (int i=0; i < COLUMNS; i++) {
   TableColumn column = new TableColumn(table, SWT.NONE);
   column.setText("Col"+i);
}
for (int i=0; i< ROWS; i++){
   TableItem item = new TableItem(table, SWT.NULL);
   for (int j=0; j<COLUMNS;j++){
      item.setText(j,"Item "+i+"-"+j);
      if ( j == 1 ) {
         item.setImage(j, new Image(display,
HelloWorldeSWT.class.getResourceAsStream("res/pgz_2.png")));
}
   }
}
for (int i=0; i<COLUMNS; i++){
   TableColumn column = table.getColumn(i);
   column.pack();
}
table.setSize(240,240);
```

Figure 5-15 shows the results of this example.



Col0	Col1	Col2	Col:
Item 0-0	Item 0-1	Item 0-2	Iter
Item 1-0	Item 1-1	Item 1-2	Iter
Item 2-0	Item 2-1	Item 2-2	Iter
Item 3-0	Item 3-1	Item 3-2	Iter

Figure 5-15 Multi-column table

5.1.8 Tree

`Tree` provides a selectable user interface object that displays a hierarchy of items and issue notifications when an item in the hierarchy is selected. The children that might be added to instances of this class must be of type `TreeItem`.

Note: Although `Tree` is a subclass of `Composite`, it does not make sense to add `Control` children to it or to set a layout on it.

Styles

Table 5-5 shows the styles used in `Tree`.

Table 5-6 Tree styles

Styles	Description
`SWT.SINGLE`	Creates a single select Tree
`SWT.MULTI`	Creates a multiple select Tree
`SWT.CHECK`	Creates a tree with check boxes

Event

Like `Table`, `Tree` has `SelectionEvent`, which can be accessed via the following methods:

- `addSelectionListener(SelectionListener listener)`

  Adds the listener to the collection of listeners who are notified when the control is selected by sending it one of the messages that is defined in the `SelectionListener` interface.

- `removeSelectionListener(SelectionListener listener)`

  Removes the listener from the collection of listeners who are notified when the control is selected.

In addition, `Tree` also has `TreeEvent` to notify application when a node in `Tree` is expanded or collapsed.

- `addTreeListener(TreeListener listener)`

  Adds the listener to the collection of listeners who are notified when an item in the receiver is expanded or collapsed by sending it one of the messages that are defined in the `TreeListener` interface.

- `removeTreeListener(TreeListener listener)`

  Removes the listener from the collection of listeners who are notified when items in the `Tree` are expanded or collapsed.

Operations on TreeItem

Applications can perform several operations on `TreeItem`, such as hierarchical traversing, selection, and scrolling.

Hierarchical traversing

`Tree` provides the following methods to perform hierarchical traversing between tree items:

- `getParentItem()`

  Returns the `TreeItem` parent item, which must be a `TreeItem` or `null` when the `TreeItem` is a root.

- `getItemCount()`

  Returns the number of items contained in the receiver that are direct item children of the `Tree`.

- `getItems()`

  Returns a (possibly empty) array of items contained in the `Tree` that are direct children of it.

Selection

Tree provides the following methods to perform selection:

- `setSelection(TreeItem[] items)`

  Sets the Tree selection to be the given array of items.

- `getSelection()`

  Returns an array of TreeItem that are currently selected in the receiver.

- `getSelectionCount()`

  Returns the number of selected items contained in the Tree.

- `selectAll()`

  Selects all of the items in the Tree.

- `deselectAll()`

  Deselects all selected items in the Tree.

- `showItem(TreeItem item)`

  Shows the item.

- `showSelection()`

  Shows the selection.

Scrolling

An application can scroll the Tree explicitly by getting or setting the *top* item. The top item of Tree is the TreeItem that is shown at the top of the Tree. The following methods are available:

- `setTopItem(TreeItem item)`

  Sets the item that is currently at the top of the Tree.

- `getTopItem()`

  Returns the item that is currently at the top of the Tree.

Snippets

Example 5-13 creates a three-level Tree that includes root, nodes, and leaves. Each level is set with different images.

Example 5-13 Tree snippets

```
Tree tree = new Tree(shell, SWT.CHECK);
for (int i=0; i<3; i++) {
   TreeItem itemI = new TreeItem(tree, SWT.NULL);
   itemI.setText("Root"+i);
   itemI.setImage(new
Image(display,HelloWorldeSWT.class.getResourceAsStream("res/folderRoot.png")));
   for (int j=0; j<4; j++) {
      TreeItem itemJ = new TreeItem(itemI, SWT.NULL);
      itemJ.setText("Node "+i+"-"+j);
      itemJ.setImage(new
Image(display,HelloWorldeSWT.class.getResourceAsStream("res/folder.png")));
      for (int k=0; k<4; k++) {
         TreeItem itemK = new TreeItem(itemJ, SWT.NULL);
         itemK.setText("Leave "+i+"-"+j+"-"+k);
         itemK.setImage(new
Image(display,HelloWorldeSWT.class.getResourceAsStream("res/file.png")));
      }
   }
}
```

Figure 5-16 shows the result of this example.

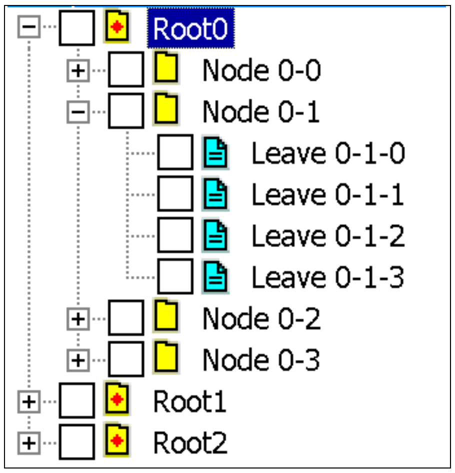


Figure 5-16 Tree example

5.1.9 Browser

Browser implements the Web browser user interface. It allows the user to visualize and navigate through HTML documents.

Events

Browser has four events: `LocationEvent`, `ProgressEvent`, `StatusTextEvent`, and `TitleEvent`. These events can be accessed via the following methods to notify applications when an event is triggered:

- `addLocationListener(LocationListener listener)`

  Adds the listener to the collection of listeners who are notified when the current location has changed or is about to change.

- `removeLocationListener(LocationListener listener)`

  Removes the listener from the collection of listeners who are notified when the current location is changed or about to be changed.

- `addProgressListener(ProgressListener listener)`

  Adds the listener to the collection of listeners who are notified when a progress is made during the loading of the current URL or when the loading of the current URL has been completed.

- `removeProgressListener(ProgressListener listener)`

  Removes the listener from the collection of listeners who are notified when a progress is made during the loading of the current URL or when the loading of the current URL has been completed.

- `addStatusTextListener(StatusTextListener listener)`

  Adds the listener to the collection of listeners who are notified when the status text is changed.

- `removeStatusTextListener(StatusTextListener listener)`

  Removes the listener from the collection of listeners who are notified when the status text is changed.

- `addTitleListener(TitleListener listener)`

  Adds the listener to the collection of listeners who are notified when the title of the current document is available or has changed.
- `removeTitleListener(TitleListener listener)`

  Removes the listener from the collection of listeners who are notified when the title of the current document is available or has changed.

Browser Operations

Applications can tell the `Browser` control to browse a URL, render HTML, move forward, move backward, or reload explicitly. Here are the methods of `Browser` supported in eSWT:

- `setUrl(java.lang.String url)`

  Loads a URL.
- `getUrl()`

  Returns the current URL.
- `setText(java.lang.String html)`

  Renders HTML.
- `forward()`

  Navigate to the next session history item.
- `back()`

  Navigate to the previous session history item.
- `isForwardEnabled()`

  Returns `true` if the receiver can navigate to the next session history item; `false` otherwise.
- `isBackEnabled()`

  Returns `true` if the receiver can navigate to the previous session history item; `false` otherwise.
- `refresh()`

  Refresh the current page.
- `stop()`

  Stop any loading and rendering activity.

Snippets

Example 5-14 creates a fully functional but simple browser to allow a user to set a URL, move forward, move background, and reload. It also adds a `Command` to terminate the application. (For more information about commands, see Chapter 4, “eSWT mobile extensions” on page 51.)

Example 5-14 SimpleBrowser

```
public class BrowserTest {

   static Button prev, reload, next, go;
   static Text url;
   static Browser browser;


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

```
Shell shell = new Shell(display);

//set window title and size
shell.setText("SimpleBrowser");
shell.setSize(240,320);

//previous button
prev = new Button(shell, SWT.PUSH);
prev.setText("<<");
prev.addSelectionListener(new SelectionListener(){
   public void widgetSelected(SelectionEvent e) {
      browser.back();
   }
   public void widgetDefaultSelected(SelectionEvent e) {
   }});

//reload button
reload = new Button(shell, SWT.PUSH);
reload.setText("R");
reload.addSelectionListener(new SelectionListener(){
   public void widgetSelected(SelectionEvent e) {
      browser.refresh();
   }
   public void widgetDefaultSelected(SelectionEvent e) {
   }});

//next button
next = new Button(shell, SWT.PUSH);
next.setText(">>");;
next.addSelectionListener(new SelectionListener(){
   public void widgetSelected(SelectionEvent e) {
      browser.forward();
   }
   public void widgetDefaultSelected(SelectionEvent e) {
   }});

// url text
url = new Text(shell, SWT.SINGLE|SWT.BORDER);
url.setText("http://");

// go button
go = new Button(shell, SWT.PUSH);
go.setText("GO");
go.addSelectionListener(new SelectionListener(){
   public void widgetSelected(SelectionEvent e) {
      // TODO Auto-generated method stub
      browser.setUrl(url.getText());
   }
   public void widgetDefaultSelected(SelectionEvent e) {
   }});

// browser
browser = new Browser(shell, SWT.NONE);
browser.setUrl("http://www.google.com");

FormLayout formLayout = new FormLayout();
shell.setLayout(formLayout);
formLayout.spacing = 1;
formLayout.marginHeight = formLayout.marginWidth = 2;
```

```
      FormData prevData = new FormData();
      prevData.left = new FormAttachment(0);
      prevData.top = new FormAttachment(0);
      prevData.width = 16;
      prev.setLayoutData(prevData);

      FormData reloadData = new FormData();
      reloadData.left = new FormAttachment(prev);
      reloadData.top = new FormAttachment(0);
      reloadData.width = 16;
      reload.setLayoutData(reloadData);

      FormData nextData = new FormData();
      nextData.left = new FormAttachment(reload);
      nextData.top = new FormAttachment(0);
      nextData.width = 16;
      next.setLayoutData(nextData);

      FormData urlData = new FormData();
      urlData.left =  new FormAttachment(next);
      urlData.top = new FormAttachment(0);
      urlData.right = new FormAttachment(89);
      urlData.bottom = new FormAttachment(browser);
      url.setLayoutData(urlData);

      FormData goData = new FormData();
      goData.left = new FormAttachment(url);
      goData.top = new FormAttachment(0);
      goData.width = 24;
      go.setLayoutData(goData);

      FormData browserData = new FormData();
      browserData.top = new FormAttachment(prev);
      browserData.left = new FormAttachment(0);
      browserData.right = new FormAttachment(100);
      browserData.bottom = new FormAttachment(100);
      browser.setLayoutData(browserData);

      shell.open();
      while( !shell.isDisposed() ) {
         if( !display.readAndDispatch() )
            display.sleep();
      }
      display.dispose();
   }
}
```

Figure 5-17 shows how this example runs on Pocket PC.

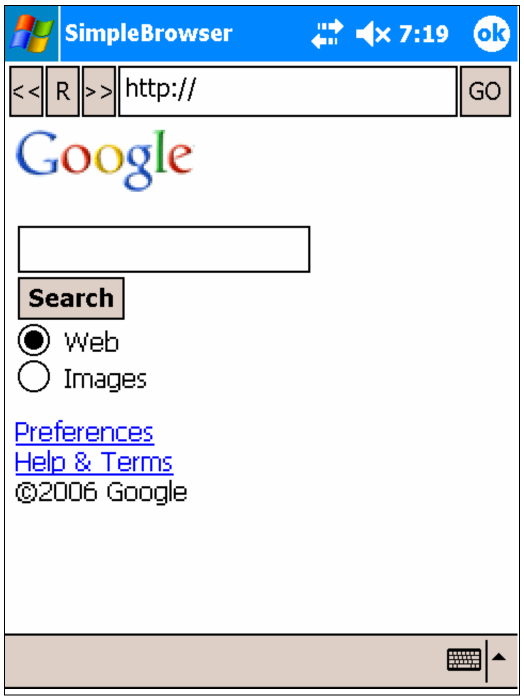


Figure 5-17 SimpleBrowser example

Users can also change to another URL, as shown in Figure 5-18.

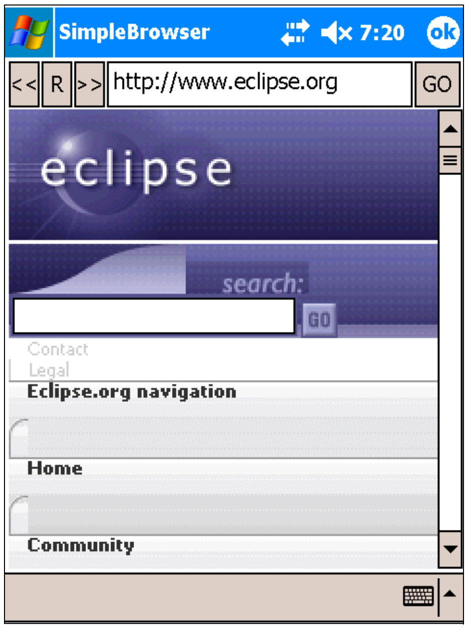


Figure 5-18 SimpleBrowser after URL change

6

eJFace applications

This chapter provides a summary of the functionality found within the org.eclipse.ercp.jface package. It first introduces eJFace and then evaluates the most important facets of the toolkit.

6.1 eJFace fundamentals

To put it simply, eJFace is a platform-independent user interface toolkit that depends upon embedded Standard Widget Toolkit (eSWT). eJFace provides a set of components and help utilities that simplify the development of applications that are based in eSWT, just as JFace does for SWT. In fact, eJFace is a strict subset of JFace, so the two packages share many similarities. eJFace provides support for resource management, viewers, actions, and preference pages.

eJFace is smaller than JFace. Figure 6-1 provides a class diagram of the eJFace packages.

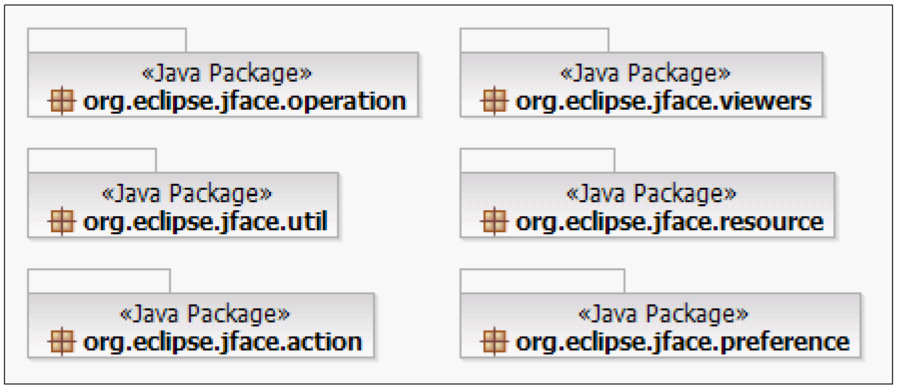


Figure 6-1 eJFace packages

The following list describes these:

- org.eclipse.jface.operation

  Provides support classes for dealing with threading and operations (modal, long-running, and so forth).

- org.eclipse.jface.viewers

  Provides a model-view-controller framework for viewers based off of eSWT widgets such as a `Table` or a `Tree`. It allows for the strict separation of the model that drives the display logic.

- org.eclipse.jface.util

  Provides utility classes pertinent to eJFace (for example, assertions and geometric operations on eSWT classes).

- org.eclipse.jface.resource

  Provides support for managing eSWT resources such as fonts, images, and colors.

- org.eclipse.jface.action

  Provides support for managing actions. Actions represent a command to manipulate a model, commonly found menus.

- org.eclipse.jface.preference

  Provides support for manipulating and viewing preferences.

6.2 Viewers

Viewers are wrappers that simplify development on eSWT widgets. In the simplest terms, viewers provide utilities that allow you to keep the user interface (eSWT widget) synchronized with the model. eJFace plays host to several types of viewers that map to their eSWT counterparts: check box, combo box, list, table, and tree viewers. This section discusses the fundamentals of the viewer framework (which all viewers share) and gives an overview of available viewer types.

6.2.1 Viewer framework

There are three main layers (represented by abstract classes) found within the eJFace framework:

- `Viewer`

  This is the most basic layer in the framework that allows for selection processing via the `IInputSelectionProvider` interface and defines a common interface to handle input via `IInputProvider`.

- `ContentViewer`

  This layer is responsible for defining the relationship between a viewer, content provider and label provider. Content providers share the `IContentProvider` interface and are responsible for working with the input. Label providers share the `IBaseLabelProvider` interface which maps a model element to an optional image and optional string.

- `StructuredViewer`

  This layer adds support for viewer utilities such as filters and sorters.

6.2.2 Viewers

The main use of the `Viewer` abstract class is to identify the input of the model and selection processing. A common input is a file (the input can be anything). The input is manipulated via the following methods:

- `setInput(Object input)`

  Sets the input of the model.

- `getInput()`

  Returns the input of the model.

It is also possible to obtain the selection of the viewer via the method:

- `getSelection()`

  Returns the current selection.

Note: `IStructuredSelection` is used when a selection contains many elements.

6.2.3 Content viewers

The two most important aspects of a content viewer are the concepts of a content provider and a label provider.

Content Providers

Content providers are responsible for manipulating the input. To work with the input you have to set a content provider via the `setContentProvider(IContentProvider provider)` method.

Content providers must adhere to the structure of the widget that they represent. For example, for a list, the content provider represents a flat structure and does not care about parent-child relationships that are found in trees. For this reason, there exists two content providers based on your viewer: `IStructuredContentProvider` and `ITreeContentProvider`. The usage rule for these two content providers is simple: if your viewer is tree-based, use `ITreeContentProvider`; otherwise, use `IStructuredContentProvider`. For example:

- `IStructuredContentProvider`
  - `CheckboxTableViewer`
  - `ComboViewer`
  - `ListViewer`
  - `TableViewer`
- `ITreeContentProvider`
  - `CheckboxTreeViewer`
  - `TreeViewer`

Label providers

Label providers are responsible for adorning model elements with text or an image. They are manipulated via the following the methods:

- `setLabelProvider(IBaseLabelProvider provider)`

  Sets the label provider. Note that label providers are dependent on the type of viewer.
- `getLabelProvider()`

  Returns the label provider

Label providers, similar to content providers, must adhere to the structure of the widget that they represent. The following are the types of label providers and the corresponding viewer that they represent:

- `ILabelProvider`
  - `CheckBoxTreeViewer`
  - `ComboViewer`
  - `ListViewer`
  - `TreeViewer`
- `ITableLabelProvider`
  - `CheckboxTableViewer`
  - `TableViewer`

6.2.4 Structured viewers

Structured viewers allow for users to sort and filter the content in their specified model. Filtering model elements can be a very powerful usability feature for the user and structured viewers allow you to add unlimited filters to your viewer. Structured viewers also provide the ability to add a sorter to the viewer to aid the display of model elements. It is also possible to add label decorators to further enhance the usability of your viewer.

Filtering

All filters must extend the `ViewerFilter` abstract class. Filtering is provided via the following methods:

- `addFilter(ViewerFilter filter)`

  Adds a filter to the list of filters

- `removeFilter(ViewerFilter filter)`

  Removes a filter from the list of filters

Example 6-1 filters out elements that are not of `ISpecialType`.

Example 6-1 Filter: Only allow IType elements

```
ViewerFilter typeFilter = new ViewerFilter() {

            public boolean select(Viewer viewer, Object parentElement, Object element) {
               if(element instanceof IType) { return true; }
               return false;
            }

            public boolean isFilterProperty(Object element, String property) {
               return true;
            }

         };
viewer.addFilter(typeFilter);
```

Sorting

All sorters must extend the `ViewerSorter` abstract class. Sorting is provided via the following methods:

- `setSorter(ViewerSorter sorter)`

  Sets the sorter for the viewer

- `getSorter()`

  Returns the sorter for the viewer

Example 6-2 sorts two types of elements: `IType` and `ISpecialType`. It gives priority to `ISpecialType` instances so that they are displayed first.

Example 6-2 A sorter that gives priority to ISpecialType instances

```
ViewerSorter sorter = new ViewerSorter() {

            public int category(Object element) {
               if(element instanceof ISpecialType)
                  return 0;
               return 1;
            }
            public boolean isSorterProperty(Object element, String property) {
               return true;
            }

         };
viewer.setSorter(sorter);
```

Label decorators

A decorating label provider allows one to combine a nested label provider with an optional label decorator. All decorating label providers must implement the `ILabelDecorator` interface and be instantiated using the `DecoratingLabelProvider` class. Example 6-3 enhances a previous label provider by adding an annotation if the element is modified.

Example 6-3 A label decorator example

```
ILabelDecorator decorator = new ILabelDecorator() {
          ...
           public String decorateText(String text, Object element) {
              if(element instanceof IModifiableElement) {
                 if (((ModifiableElement)element).isModified()) { text = "* " + text; }
              }
           }
          ...
         };
viewer.setLabelProvider(new DecoratingLabelProvider(provider,decorator));
```

6.2.5 Viewer types

Note: A difference between eJFace and JFace is that eJFace lacks a `TableTree`.

The following viewer types are included in eJFace:

- `CheckBoxTableViewer`

  A viewer that allows check boxes to adorn each `TableItem`.
- `CheckBoxTreeViewer`

  A viewer that allows check boxes to adorn each tree node.
- `ComboViewer`

  A simple viewer that is used as an alternative to the `ListViewer`. It displays its content in a combo box rather than a list.
- `ListViewer`

  A simple viewer that is used as an alternative to the `ComboViewer`. It displays its content in a list rather than a combo box.
- `TableViewer`

  A simple viewer that is used in conjunction with the `Table` control.
- `TreeViewer`

  A simple viewer that is used in conjunction with the `Tree` control.

6.3 Operations

There is only one important operation class within eJFace. `ModalContext` is used to facilitate modal operations. The two most important methods in this class are:

- `run(`
  `IRunnableWithProgress operation,`
  `boolean fork,`
  `IProgressMonitor monitor,`
  `Display display)`

  Runs a modal operation with an associated progress monitor

- `runInCurrentThread(`
  `IRunnableWithProgress runnable,`
  `IProgressMonitor progressMonitor)`

  Runs a modal operation with associated progress monitor in the same thread.

6.4 Resource management

eJFace provides a method to manage resources via a convenience class, `JFaceResources`. Fonts, colors, and images are managed by this class using the concept of a registry.

Note: Normally, when creating a resource with eSWT, you are responsible for disposing it. By using `JFaceResources` for your resource management, you save the trouble of having to dispose your resources.

The following methods are used to manage these resources:

- `JFaceResources.getColorRegistry()`

  Returns the color registry. If no registry is present, one is created.

- `JFaceResources.getFontRegistry()`

  Returns the font registry. If no registry is present, one is created.

- `JFaceResources.getImageRegistry()`

  Returns the image registry. If no registry is present, one is created.

Tip: For convenience reasons, it is possible initialize default values into the image registry by using the `initializeImageRegistry(ImageRegistry reg)` method found in `AbstractUIPlugin`.

The usage of these registries are similar through the use of getters and setter methods. Example 6-4 stores an image into a `ImageRegistry` and retrieves it.

Example 6-4 `ImageRegistry` Usage

```
JFaceResources.getImageRegistry().put(“key”,image); // store image into registry
Image myImage = JFaceResources.getImageRegistry().get(“key”);
```

Note: If you need further management of your resources, you can use `LocalResourceManager` to share local resources with a global registry.

6.5 Preferences

Preferences provide a method to manage information that is associated with an application. They are commonly used in dialogs as a way to persist information. The preference infrastructure also provides the ability to listen to preference changes and to react to them appropriately.

6.5.1 Preference storage

Preferences are manipulated through a preference store. The plug-in class manages the preference store. Example 6-5 manipulates the preference store by adding a value.

Example 6-5 Manipulating the preference store

```
Activator.getDefault().getPreferenceStore().putValue(“download.dir”, “C:\downloads”);
String home = Activator.getDefault().getPreferenceStore().getString(“home.dir”);
```

If you want to listen to value changes, the first step is to develop a listener that implements the `IPropertyChangeListener` interface. The final step is to then attach that listener to the preference store via the `addPropertyChangeListener(IPropertyChangeListener listener)` method.

It is also possible to set default values on properties. The preference store provides a series of `setDefault()` methods to handle each type of persisted value.

6.5.2 Preference dialogs

Preference dialogs allow you to manipulate data that is found within a set of preference pages. Preference pages represent information found within a preference store. In order to use preference dialogs, you need to first create and define preference pages using the `org.eclipse.ui.preferencePages` extension point. Example 6-6 lists a sample preference page extension definition.

Example 6-6 An example preference page definition

```
<extension
        point="org.eclipse.ui.preferencePages">
      <page
           class="org.eclipse.ercp.sample.preferences.DefaultPage"
           id="org.eclipse.ercp.sample.preferences.defaultPage"
           name="Default Preference Page"/>
      <page
           category="org.eclipse.ercp.sample.preferences.ExtPage"
           class="org.eclipse.ercp.sample.preferences.ExtPage"
           id="org.eclipse.ercp.sample.preferences.defaultPage"
           name="Additional Preference Page"/>
   </extension>
```

After your preference pages are created and defined in the preference page extension point, you can launch them by using the `PreferencesUtil` convenience class (Example 6-7).

Example 6-7 Launching a preference dialog

```
PreferenceDialog dialog = PreferencesUtil.createPreferenceDialogOn(
   shell,
   “org.eclipse.ercp.sample.preferences.DefaultPage”,
   null,
   null);
dialog.open(); // open the preference dialog and wait for user input
```

7

eRCP eWorkbench

This chapter provides a detailed description of embedded Rich Client Platform (eRCP) eWorkbench and how to develop applications targeted for it.

7.1 Introduction

In Eclipse, the concept of a workbench was defined as the feature that controls where applications (views) are displayed. In eRCP eWorkbench mimics Eclipse but makes the modifications that are needed for mobile devices. eWorkbench extends plug-ins to contribute views based on common device display types. The conceptual difference between eWorkbench and the Eclipse workbench is that eWorkbench has no concept of perspectives.

This chapter focuses on how to develop an application for the eWorkbench.

7.2 Developing for the eWorkbench

There are three basic steps that you need to follow in order to develop an eWorkbench application:

- Creating your plug-in
- Defining your views
- Defining your application

7.2.1 Creating your plug-in

The first step of defining a proper eWorkbench application (or any Eclipse-based application) is to create your plug-in. The most common way to do this is to use the plug-in wizard in Eclipse's plug-in development environment (**File** → **New** → **Plug-in Project**). This wizard generates the needed files for you automatically.

The key point is that your plug-in class extends either `Plugin` or `AbstractUIPlugin`, depending on the functionality that required, and it must be defined properly in the MANIFEST.MF (Example 7-1). In this case, our plug-in class (also referred to as a *bundle*) has to be defined in the `Bundle-Activator` attribute of the manifest.

Example 7-1 A sample MANIFEST.MF with a properly defined plug-in

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: IBM  eRCP Sample Application
Bundle-SymbolicName: com.ibm.ercp.application;singleton:=true
Bundle-Version: 1.0.0
Bundle-ClassPath: .
Bundle-Activator: com.ibm.ercp.application.Activator
...
```

7.2.2 Defining you views

eWorkbench allows you to define a few types of views that eWorkbench will use automatically, depending on the device on which your application is running. The different types of views are:

- Normal

  The default view that is used for your application that is required for all eWorkbench applications.
- Large (optional)

  The view used for larger display devices. It is optional.
- Status (optional)

  The view that is used for small display devices (such as the status screen of a cell-phone). It is optional.

After you have decided which views that you would like to support, the next step is to implement these views. All the views extend `org.eclipse.ui.ViewPart` as their base class (this holds true for views defined within the normal Eclipse workbench). Example 7-2 shows a normal view that serves as a Hello eWorkbench application.

Example 7-2 A sample normal view for eWorkbench

```
public class NormalView extends ViewPart {

   public void createPartControl(Composite parent) {
      Composite composite = new Composite(parent, SWT.NONE);
      composite.setLayout(new FillLayout());

      Label label = new Label(composite, SWT.CENTER);
      label.setText("Hello eWorkbench");
   }
...
}
```

After you implement the views that you need, the next step is to define these views in the plugin.xml so that eRCP can be aware of them. You can implement these views via the org.eclipse.ui.views extension point. Example 7-3 shows a view that is defined using the common contribution mechanism of Eclipse extension points.

Example 7-3 A sample eRCP view definition (plugin.xml)

```
<extension
        point="org.eclipse.ui.views">
     <view
          allowMultiple="false"
          category="org.eclipse.ercp.eworkbench.viewCategory"
          class="com.ibm.ercp.application.views.NormalView"
          icon="icons/sample.gif"
          id="com.ibm.ercp.application.normalView"
          name="Sample eWorkbench Normal View">
     </view>
</extension>
```

After these steps are complete, everything is in place to let eWorkbench know how to launch your application.

7.2.3 Defining your application

In order for eWorkbench to launch applications, it needs to be aware of them first. The process to accomplish this task is very similar to how an application is defined in Rich Client Platform (RCP). All eWorkbench applications must contribute to the org.eclipse.ercp.eworkbench.applications extension point. The structure of the extension point (for each eWorkbench application) is as follows:

- id

  The unique identifier that represents your application.
- name

  The name of your application that is displayed in the eWorkbench application list.
- views (normal, large, status)

  The views that your application supports.

Example 7-4 shows a sample eWorkbench application definition.

Example 7-4 A sample eWorkbench application definition (plugin.xml)

```
<extension
        point="org.eclipse.ercp.eworkbench.applications">
        <application
        id="com.ibm.ercp.application"
        name="IBM Sample Application"
        singleton="true">
        <views normal="com.ibm.ercp.application.normalView"/>
        </application>
</extension>
```

Note: The views that are defined in the application extension point represent the IDs of the views that are defined in the org.eclipse.ui.views extension point.

After all these steps are complete, a subsequent deployment and launch of the eWorkbench should reveal your new application (Figure 7-1).

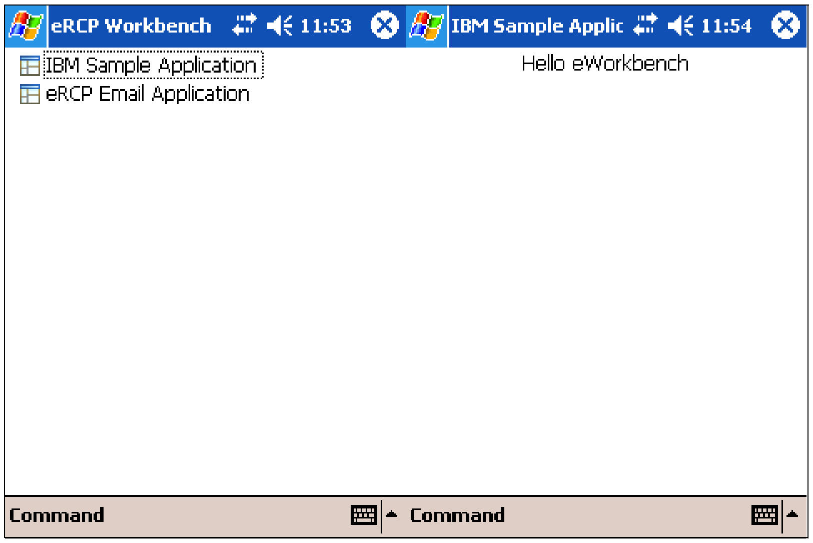


Figure 7-1 A sample eWorkbench application

eRCP sample scenario

This chapter provides a sample scenario to illustrate how you develop an embedded Rich Client Platform (eRCP) application.

8.1 Preparing the environment

This section provides the steps needed to set up the environment for eRCP application development. It shows you how to add the required plug-ins for database connectivity. For this scenario, the target platforms are Win32® and Windows Mobile 2003, although the steps are very similar for other platforms.

8.1.1 The eRCP development environment

The minimum eRCP development environment consists of the following components:

- Eclipse SDK 3.1.x
- A supported JRE
- The eRCP runtime

In this scenario, we used Eclipse SDK 3.1.2 for Win32 and the Java JRE 1.4.2 as the Eclipse JRE. You can download Eclipse SDK 3.1.2 for Win32 from the Eclipse site at:

`http://www.eclipse.org`

You can download Java JRE 1.4.2 as the Eclipse JRE from the Java site at:

`http://java.sun.com`

Regarding the eRCP runtime, eRCP Milestone 7 (M7) is used. You can download the runtimes from the eRCP site at:

`http://www.eclipse.org/ercp`

For the M7 version, there are three supported platforms:

- Windows Mobile 2003 (WM2003)
- Windows Desktop (Win32)
- Nokia Series 80

Notice that eRCP support is available for Windows Desktop systems. This is very useful because it allows you to run and test your application on a Windows system without the need to use a WM2003 device. However, for application deployment you use the WM2003 runtime.

You are probably wondering what Java Runtime Environment you use on the WM2003 device. For small devices there is an entirely different specification of the Java platform called Java 2 Micro Edition (J2ME). Because small devices capabilities vary greatly, they have divided J2ME into *configurations* and *profiles*, which are basically subsets of the Java 2 Standard Edition (J2SE) specification, based on the device capabilities. A detailed discussion about J2ME is out of the scope of this document. For more information, refer to the following:

`http://java.sun.com/j2me`

For WM2003, you use a Foundation Profile implementation part of the IBM Workplace™ Client Technology, Micro Edition for Windows product.

Note: You can download a trial version of Workplace Client Technology™, Micro Edition from the product site at:

`http://www.ibm.com/software/wireless/wctme/`

To summarize this scenario, you need the following software:

- Eclipse SDK 3.1.2 + JRE 1.4.2
- J2ME Foundation Profile (part of Workplace Client Technology, Micro Edition) for Win32 and WM2003
- eRCP runtime for Windows Desktop and WM2003

Why do I need the J2ME Foundation profile for Win32?

As you have seen, you have the JRE1.4.2 for Windows in place, so why do you need the Foundation Profile for Win32? J2ME profiles are subsets of the J2SE specification. If you use J2SE (and the JRE 1.4.2 is a J2SE implementation) to develop the application in the desktop, you are at the risk of using classes that are not actually part of the selected J2ME profile, without being aware of it. So, if you are wondering why the application that was working perfectly on the desktop is throwing `ClassNotFoundException` errors when running on the device, it is recommended to use the Foundation Profile in both environments, desktop and device. Workplace Client Technology, Micro Edition provides implementations for both.

8.1.2 Installing the Eclipse SDK

Before installing the Eclipse SDK 3.1.2, you need to install JRE V1.4.2. You can download it from:

http://java.sun.com

Follow the instructions to install the JRE in your machine. You can find information about suggested JRE for Eclipse 3.1.2 at:

http://download.eclipse.org/eclipse/downloads/drops/R-3.1.2-200601181600//java-runtimes.html

You can download the Eclipse SDK from the Eclipse site at:

http://www.eclipse.org

The Eclipse SDK 3.1.2 for Windows is packaged as a zipped file named eclipse-SDK-3.1.2-win32.zip. To install this file, follow these steps:

1. Unzip the file to a directory, for example c:\eclipse.
2. Locate the eclipse.ini file on the root directory, c:\eclipse in the example. Modify the file to fit best your environment. You might modify the maximum amount of memory that Eclipse takes, as denoted by the `Xmx` parameter. In general, you adjust this value to give Eclipse enough memory to play without reaching the system limits. For example, in your development environment with 1 GB RAM machines, 512 MB for Eclipse would be fine. There are other parameters to adjust, but they are only worth modifying if you have performance problems with your development environment.
3. By default, if you launch Eclipse using eclipse.exe, it uses the system JRE. A better approach when you have multiples JREs is to specify which one you want Eclipse to use. You can do this by creating a Windows shortcut for the eclipse.exe launcher and adding the `-vm` parameter to it. For example:

   c:\eclipse\eclipse.exe -vm C:\Program Files\Java\j2re1.4.2_11\javaw

   You can also place the shortcut in your Windows desktop for easy access.

4. Launch the Eclipse SDK. Click **OK** to accept the default workspace location. If everything is fine, you have the screen shown in Figure 8-1.

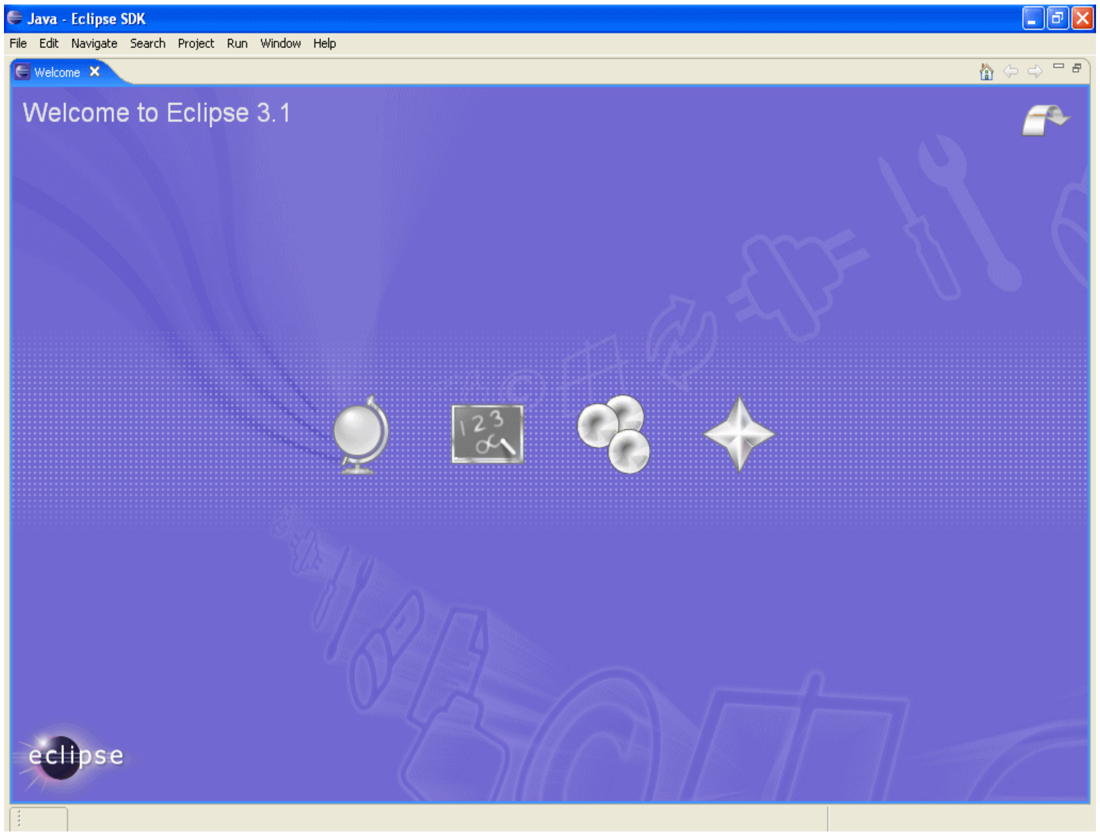


Figure 8-1 Eclipse SDK installed

8.1.3 Installing Workplace Client Technology Micro Edition for Windows

You need to install the Foundation Profile for Win32 and WM2003. They come as part of Workplace Client Technology Micro Edition for Windows. You can download a trial copy from:

`http://www.ibm.com/software/wireless/wctme`

Note: In the trial download Web page there are several components. You must download the complete package. It appears under the Workplace Client Technology Micro Edition for Windows header (which contains all component products and tools) . The file to download is windows_cd.zip.

To install the product to get the Foundation Profile libraries, follow these steps:

1. Unzip the zipped file to a temporary folder, for example c:\temp. Start launchpad.exe.
2. Click **Install WebSphere Studio Device Developer**. Wait while the installer loads.
3. Click **Next** on the Welcome screen.
4. Select **I accept the terms in the license agreement** and click **Next**.
5. The installer presents the location to install the program, for example C:\Program Files\Device Developer. Write down this location. You will need it later. Click **Next**.
6. Click **Next** in the summary window. Wait while the product is installed.
7. Click **Finish** to end the installation.

The Foundation profiles for each platform are located under the WebSphere Studio Device Developer directory (the one that you wrote down in the previous steps). For example:

- \wsdd5.0\ive-2.2\runtimes\win32\x86\foundation10 for Win32
- \wsdd5.0\ive-2.2\runtimes\wm2003\arm\foundation10 for WM2003

8.1.4 Installing the eRCP runtimes

You can download the eRCP runtime packages from the eRCP site at:

http://www.eclipse.org/ercp

In this book, we use the Milestone 7 (M7) version. They are packaged as zipped files. For eRCP M7, you have to download the following:

- eRCP-v20060118-1017.win32-x86.zip (Win32 runtime)
- eRCP-v20060118-1017.wm2003.zip (WM2003 runtime)

For this scenario, you should unzip the runtimes in a structure similar to the one shown in Figure 8-2. This structure is used for the remainder of this chapter.

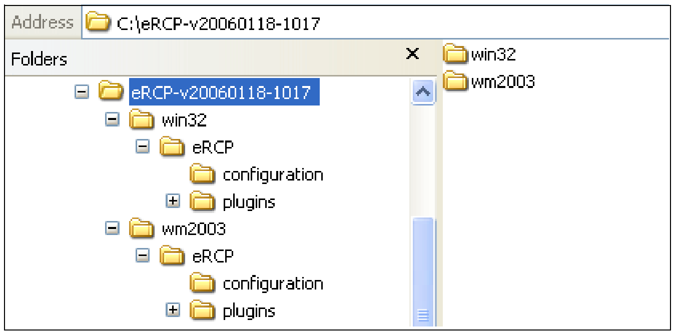


Figure 8-2 eRCP runtimes

8.1.5 Database support for small devices

To develop database-oriented applications on mobile devices, you need the following components:

- DB2 Everyplace database engine
- DB2 Everyplace JDBC™ drivers
- JDBC Optional Package for Foundation Profile (JSR 169)

DB2 Everyplace database engine

The DB2 Everyplace database engine is composed of a native library for each supported platform. For Win32 and WM2003 the library is the DB2e.dll file. The native library must be available to the application on the java.library.path location.

DB2 Everyplace JDBC drivers

The DB2 Everyplace JDBC drivers are composed of two files: a native library called DB2EJDBC.dll that, similar to the database engine, is specific for each platform and a JAR library called DB2e.jar.

The native library must be available to the application on the java.library.path location. Depending how the DB2e.jar file is packaged, it has to be listed in the application CLASSPATH or it has to be available as an Eclipse bundle. For this application, you use the bundle version.

JDBC optional package for Foundation Profile (JSR 169)

You have seen that the J2ME Foundation Profile is a subset of J2SE and does not include the java.sql and javax.sql packages. Because many applications for small devices need to use a database, a JSR was created that specifies a subset of the JDBC 3.0 standard. The final result is named *JDBC Optional Package for Foundation Profile* or JSR 169 for short. So, you need an implementation of JSR 169.

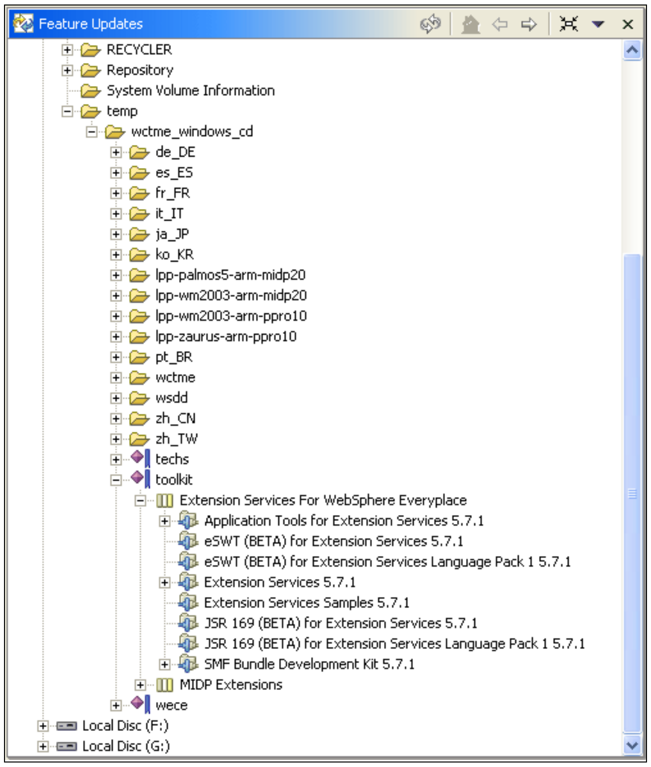
8.1.6 Installing Workplace Client Technology, Micro Edition database components

There are several places to obtain the components that we have mentioned. If you have downloaded the complete Workplace Client Technology, Micro Edition package as recommended, you already have all the components in place. They are part of the Workplace Client Technology, Micro Edition Toolkit for Device Developer.

Note: You can also download a trial version of DB2 Everyplace that comes with the Synchronization Server. We recommend that you do so if you want to try the database synchronization capabilities of DB2 Everyplace.

To have the components available, you need to install the Workplace Client Technology, Micro Edition Toolkit on Device Developer. Follow these steps:

1. Start WebSphere Studio Device Developer. Click **OK** to accept the default location for the workspace.
2. Select **Help** → **Software Updates** → **Update Manager**.
3. In the Update Manager perspective, go to the Features Updates view and expand the directory where you have extracted the Workplace Client Technology, Micro Edition package (for example D:\temp\wctme_windows_cd). The toolkit should be there.
4. Expand the toolkit to see the toolkit's components as shown in Figure 8-3.

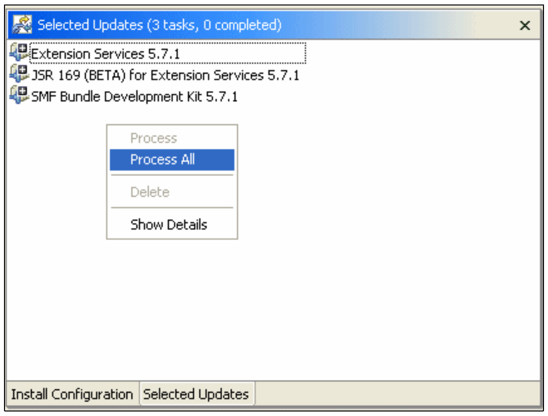

Feature Updates

- RECYCLER
- Repository
- System Volume Information
- temp
  - wctme_windows_cd
    - de_DE
    - es_ES
    - fr_FR
    - it_IT
    - ja_JP
    - ko_KR
    - lpp-palmos5-arm-midp20
    - lpp-wm2003-arm-midp20
    - lpp-wm2003-arm-ppro10
    - lpp-zaurus-arm-ppro10
    - pt_BR
    - wctme
    - wsdd
    - zh_CN
    - zh_TW
    - techs
    - toolkit
      - Extension Services For WebSphere Everyplace
        - Application Tools for Extension Services 5.7.1
        - eSWT (BETA) for Extension Services 5.7.1
        - eSWT (BETA) for Extension Services Language Pack 1 5.7.1
        - Extension Services 5.7.1
        - Extension Services Samples 5.7.1
        - JSR 169 (BETA) for Extension Services 5.7.1
        - JSR 169 (BETA) for Extension Services Language Pack 1 5.7.1
        - SMF Bundle Development Kit 5.7.1
      - MIDP Extensions
    - wece
- Local Disc (F:)
- Local Disc (G:)

Figure 8-3 Workplace Client Technology, Micro Edition Toolkit

5. Select **Extension Services For WebSphere Everyplace** → **SMF Bundle Development Kit 5.7.1**.
6. In the Preview view on the left, select **Install by adding to the Selected Updates.** Note the feature has been placed in the Selected Updates view.
7. Repeat the outlined procedure for the Extension Services 5.7.1 and JSR 169 (BETA) for Extension Services 5.7.1 features.

   You should have three features listed in the Selected Updates view.
8. Right-click any place inside the view and select **Process All**, as shown in Figure 8-4.


Selected Updates (3 tasks, 0 completed)

- Extension Services 5.7.1
- JSR 169 (BETA) for Extension Services 5.7.1
- SMF Bundle Development Kit 5.7.1

Figure 8-4 Installing the selected updates at once

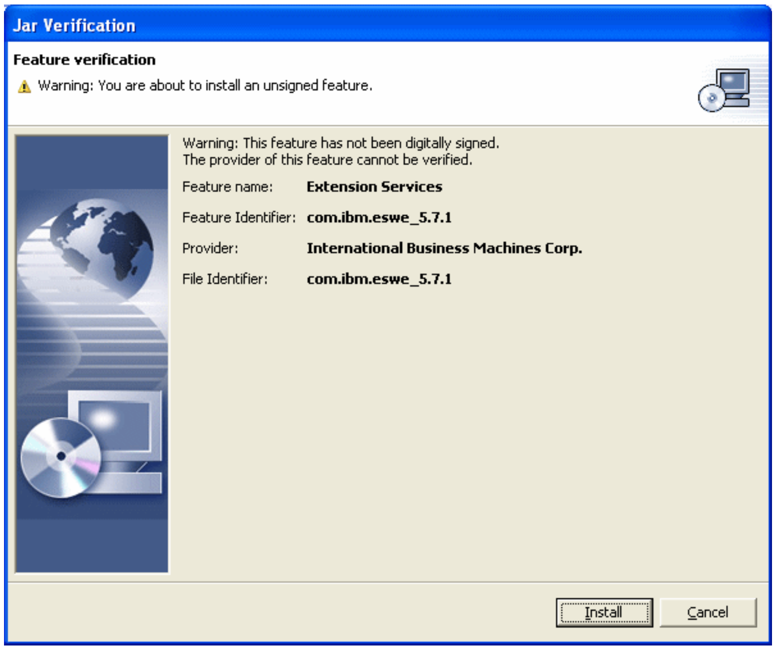
9. The Update task dialog lists the features to be installed. Click **Next**.

10. Select **I accept the terms in the license agreements and** click **Next**.

    A summary of the features to be installed is shown.

11. Click **Finish** to begin the install.

12. For each feature, a warning dialog indicates that you are about to install an unsigned feature, as shown in Figure 8-5. This is fine. Click **Install** to install the feature.



Warning: You are about to install an unsigned feature.

Warning: This feature has not been digitally signed.
The provider of this feature cannot be verified.

Feature name: Extension Services

Feature Identifier: com.ibm.eswe_5.7.1

Provider: International Business Machines Corp.

File Identifier: com.ibm.eswe_5.7.1

Figure 8-5 Warning about unsigned features

13. Wait while the features are installed. At the end, click **Yes** to restart the workbench. After the workbench is restarted, close it.

8.1.7 Configuring the environment to use the database components

Now that you have all the components installed in the development machine, you need to place the database components in the right place.

Configuring the JRE to find the DB2 Everyplace native libraries

You need to include the DB2 Everyplace engine and the JDBC driver native libraries in the java.lib.path location. The bin directory of the JRE that is used to run the application is added by default to the java.lib.path, so you can copy the natives libraries there.

Note: Because you have two different platforms — Win32 to develop and test the application in the desktop and WM2003 to test the application in the device — you need to update two JREs:

- The Win32 JRE that is used by Eclipse.
- The WM2003 JRE that is used by the mobile device.

This section describes how to update the Win32 JRE. The steps to update the WM2003 JRE are described in 8.8.3, "Copying the database libraries to the device" on page 220.

To update the Win32 JRE, follow these steps:

1. Locate the DB2 Everyplace natives libraries for the Win32 platform. They should be located in the Device Developer directory. For example:

   C:\Program Files\IBM\DeviceDeveloper\wsdd5.0\technologies\eswe\files\db2e\win32\x86

   They are:

   – DB2e.dll, the DB2e engine
   – DB2EJDBC.dll, the DB2e JDBC driver native support

   Note: There are another two files in the DB2e native libraries directory: CryptoPlugin and DB2eODBC. CryptoPlugin is necessary if you want to use the DB2e database local encryption feature. DB2eODBC is the DB2e ODBC driver and is not intended to be used in Java environments unless you want to use the JDBC-ODBC bridge to connect to the database. We strongly discourage this.

2. Copy the libraries to the bin directory of the JRE used by Eclipse (for example at C:\Program Files\Java\j2re1.4.2_11\bin).

Configuring eRCP to use the database Java libraries

The DB2e JDBC driver and the JDBC Optional Package for Foundation Profile (JSR 169) that are provided with Workplace Client Technology, Micro Edition are already packaged as OSGi bundles. So, you can copy them directly to the eRCP plug-ins directory. The libraries are the same for WM2003 and Win32. Follow these steps for both platforms:

1. Locate the DB2 Everyplace Java and JSR169 libraries. They should be located in the Device Developer directory. For example:

   C:\Program Files\IBM\DeviceDeveloper\wsdd5.0\technologies\eswe\bundlefiles

   They are:

   – DB2e.jar, the DB2e JDBC driver
   – jdbc.jar, the JSR169 package

2. Copy the libraries to the plug-ins directory of each eRCP platform. For example copy the libraries to:

   – C:\eRCP-v20060118-1017\win32\eRCP\plugins, for Win32
   – C:\eRCP-v20060118-1017\wm2003\eRCP\plugins, for WM2003

8.2 Configuring Eclipse and creating the eRCP project

The configuration procedure that is needed to prepare the environment for eRCP with database application development is not trivial; however, in most cases, you have to do this task only once.

This section explains how to configure Eclipse and to create the project necessary to develop and to test an eRCP application using the configured environment.

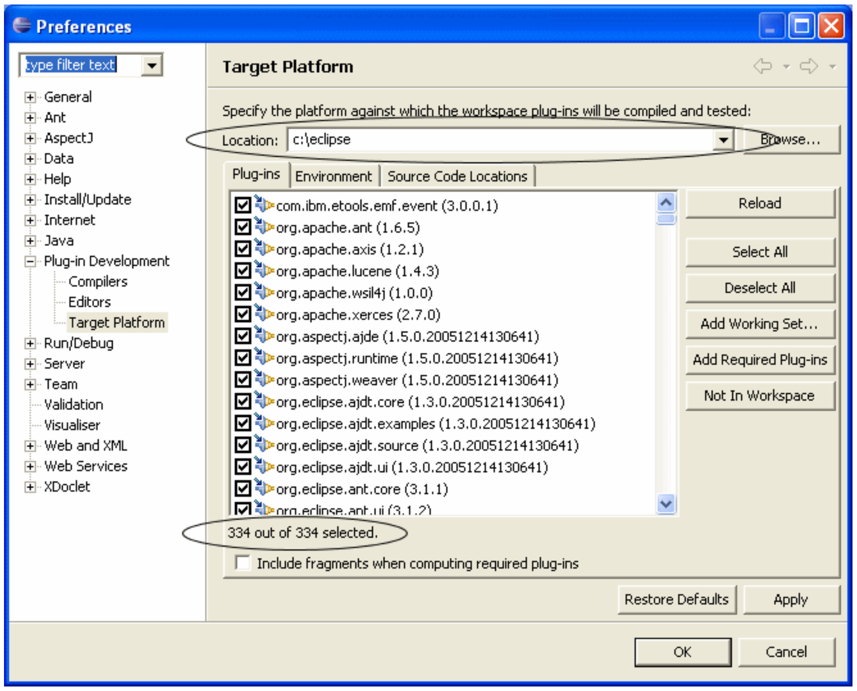
Note: Most of the configuration tasks that you perform in this section are *workspace-scoped*, which means that if you change the workspace to a different location, you will need to repeat the tasks in this section.

8.2.1 Configuring eRCP as the Eclipse target platform

First, you need to set eRCP as the Eclipse target platform. By default the Eclipse target platform is Eclipse itself. So, you need to tell Eclipse that the target platform is eRCP. Follow these steps:

1. Start Eclipse if not already started. Choose a location for the workspace and click **OK**.
2. Select **Window** → **Preferences** from the menu bar.
3. In the Preferences window, select **Plug-in Development** → **Target Platform**. Because by default the target platform is Eclipse itself, the Location entry has the Eclipse directory root (for example c:\eclipse), as shown in Figure 8-6. In this environment, the Eclipse SDK loads around 334 plug-ins.

Note: The number of plug-ins in a Eclipse installation varies in function of the number of additional features installed.



Specify the platform against which the workspace plug-ins will be compiled and tested:

Location: c:\eclipse

- ☑ com.ibm.etools.emf.event (3.0.0.1)
- ☑ org.apache.ant (1.6.5)
- ☑ org.apache.axis (1.2.1)
- ☑ org.apache.lucene (1.4.3)
- ☑ org.apache.wsil4j (1.0.0)
- ☑ org.apache.xerces (2.7.0)
- ☑ org.aspectj.ajde (1.5.0.20051214130641)
- ☑ org.aspectj.runtime (1.5.0.20051214130641)
- ☑ org.aspectj.weaver (1.5.0.20051214130641)
- ☑ org.eclipse.ajdt.core (1.3.0.20051214130641)
- ☑ org.eclipse.ajdt.examples (1.3.0.20051214130641)
- ☑ org.eclipse.ajdt.source (1.3.0.20051214130641)
- ☑ org.eclipse.ajdt.ui (1.3.0.20051214130641)
- ☑ org.eclipse.ant.core (3.1.1)
- ☑ org.eclipse.ant.ui (3.1.2)

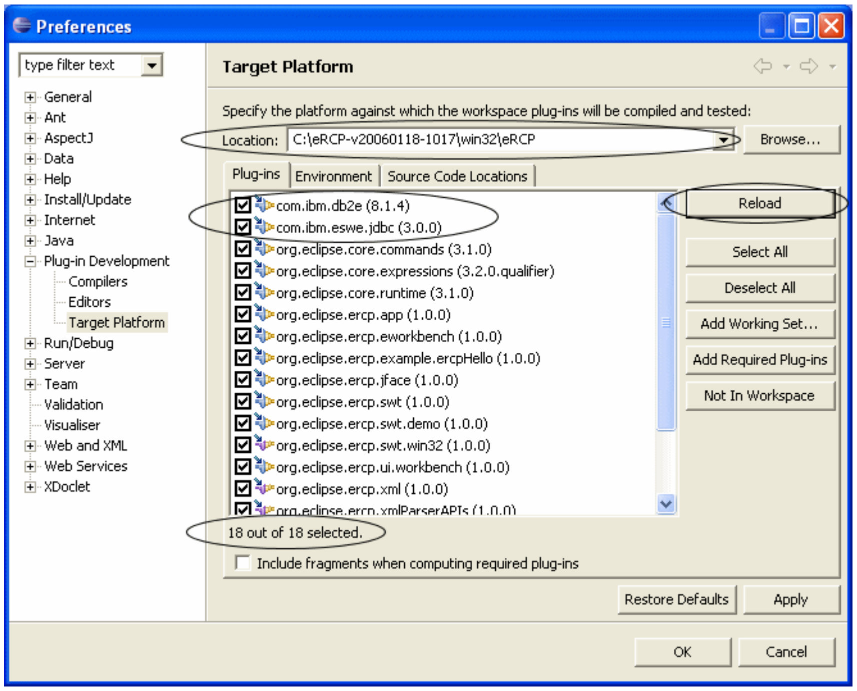
334 out of 334 selected.

☐ Include fragments when computing required plug-ins

Figure 8-6 Eclipse Target Platform

4. Click **Browse** to change the target platform to eRCP. Navigate to the eRCP for Win32 root directory (for example C:\eRCP-v20060118-1017\win32\eRCP). Note that the number of plug-ins has been reduced to only 18 plug-ins, as shown in Figure 8-7 on page 139. Note also that the DB2e JDBC and JSR169 have been loaded.

Tip: The Reload button, shown in Figure 8-7 on page 139, is useful to make effective changes immediately into the eRCP platform (for example, when you have added new plug-ins). Also, it is useful as a general troubleshooting technique, together with the **Project** → **Clean** option from the Eclipse menu bar, when you get obscure compilation errors about classes that could not be found.



Specify the platform against which the workspace plug-ins will be compiled and tested:

Location: C:\eRCP-v20060118-1017\win32\eRCP

- [✓] com.ibm.db2e (8.1.4)
- [✓] com.ibm.eswe.jdbc (3.0.0)
- [✓] org.eclipse.core.commands (3.1.0)
- [✓] org.eclipse.core.expressions (3.2.0.qualifier)
- [✓] org.eclipse.core.runtime (3.1.0)
- [✓] org.eclipse.ercp.app (1.0.0)
- [✓] org.eclipse.ercp.eworkbench (1.0.0)
- [✓] org.eclipse.ercp.example.ercpHello (1.0.0)
- [✓] org.eclipse.ercp.jface (1.0.0)
- [✓] org.eclipse.ercp.swt (1.0.0)
- [✓] org.eclipse.ercp.swt.demo (1.0.0)
- [✓] org.eclipse.ercp.swt.win32 (1.0.0)
- [✓] org.eclipse.ercp.ui.workbench (1.0.0)
- [✓] org.eclipse.ercp.xml (1.0.0)
- [✓] org.eclipse.ercp.xmlParserAPIs (1.0.0)

18 out of 18 selected.

[ ] Include fragments when computing required plug-ins

Figure 8-7 eRCP Runtime as the Eclipse Target Platform

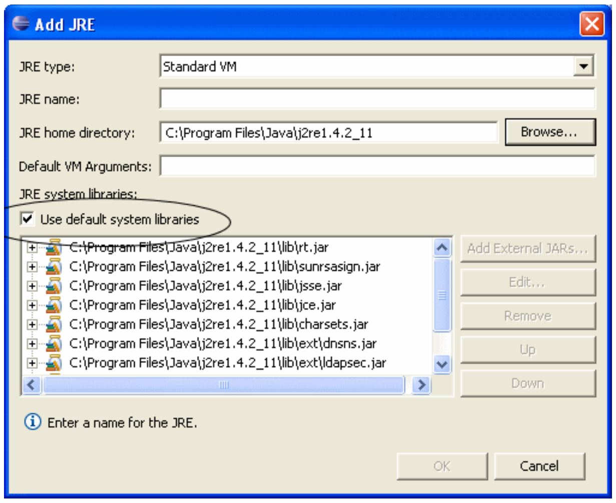
5. Click **OK** to accept the changes.

8.2.2 Adding Foundation Profile as an Eclipse JRE

When developing mobile applications, it is good practice to use the same JRE in both the development machine and the target device. Because, for WM2003 devices, you are using the Foundation Profile, you also need to add the Foundation Profile as an additional Eclipse JRE. Follow these steps:

1. Start Eclipse if not already started. Select the workspace location and click **OK**. Select **Window** → **Preferences** from the menu bar.
2. In the Preferences window, select **Java** → **Installed JREs**. The Installed JREs window lists the JREs already configured to work with Eclipse. You need to add Foundation Profile as a new one. Click **Add**.

3. Click **Browse** to search for the location of the *Eclipse JRE*, for example C:\Program Files\Java\j2re1.4.2_11. The default JRE libraries are loaded, because the Use default system libraries option is selected, as shown in Figure 8-8. You need to remove them and to specify that you want to use the libraries that are provided by the Foundation Profile.


Add JRE

JRE type: Standard VM

JRE name:

JRE home directory: C:\Program Files\Java\j2re1.4.2_11

Default VM Arguments:

JRE system libraries:

☑ Use default system libraries

C:\Program Files\Java\j2re1.4.2_11\lib\rt.jar
C:\Program Files\Java\j2re1.4.2_11\lib\sunrsasign.jar
C:\Program Files\Java\j2re1.4.2_11\lib\jsse.jar
C:\Program Files\Java\j2re1.4.2_11\lib\jce.jar
C:\Program Files\Java\j2re1.4.2_11\lib\charsets.jar
C:\Program Files\Java\j2re1.4.2_11\lib\ext\dnsns.jar
C:\Program Files\Java\j2re1.4.2_11\lib\ext\ldapsec.jar

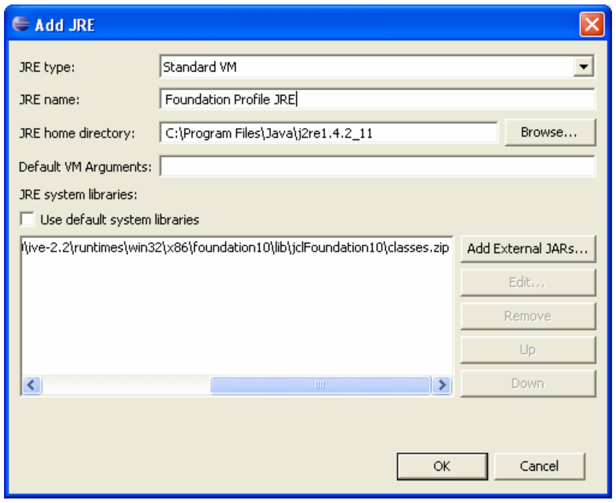
Enter a name for the JRE.

Figure 8-8 Adding a new JRE

4. Deselect **Use default system libraries.** Select all the libraries that are loaded by default and click **Remove.**
5. Click **Add External JARs.** Select the Foundation profile library for Win32 at the Device Developer directory, for example:

   C:\Program Files\IBM\DeviceDeveloper\wsdd5.0\ive-2.2\runtimes\win32\x86\foundation10\lib\jclFoundation10

   The library is called classes.zip.
6. Enter a name for the JRE, for example `Foundation Profile JRE`. The window should look similar to the one shown in Figure 8-9. Click **OK.**


Add JRE

JRE type: Standard VM

JRE name: Foundation Profile JRE

JRE home directory: C:\Program Files\Java\j2re1.4.2_11

Default VM Arguments:

JRE system libraries:

☐ Use default system libraries

\ive-2.2\runtimes\win32\x86\foundation10\lib\jclFoundation10\classes.zip

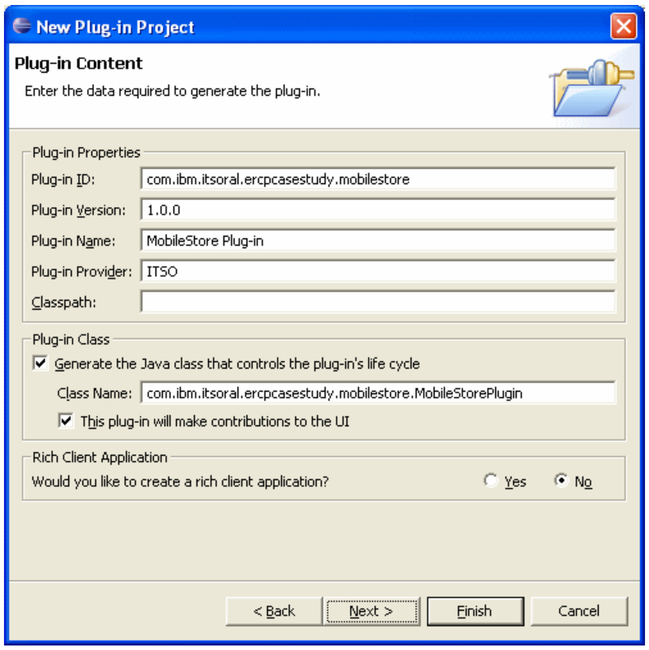
Figure 8-9 Foundation Profile JRE added

7. Click **OK** to close the Installed JREs list.

8.2.3 Creating the ITSO Mobile Store project

Now that you have the eRCP target platform loaded and the Foundation Profile as an Eclipse JRE, you can begin creating the application project. Because an eRCP application contributes to the eRCP platform, it needs to be developed as an Eclipse plug-in. Therefore, all the development work is done in the Eclipse plug-in Development perspective. Follow these steps:

1. Start Eclipse. Select the location for the configured workspace and click **OK**.
2. Select **Window** → **Open Perspective** → **Other** to show all the perspectives available. Select **Plug-in Development** and click **OK**.
3. Select **File** → **New** → **Project** from the menu bar. Select **Plug-in Development** → **Plug-in Project** and click **Next**.
4. Enter a name for the project, for example `com.ibm.itsoral.ercpcasestudy.mobilestore`. Leave all the other options on its default values and click **Next**.
5. Enter the information for the new plug-in. Use the information that is provided in Figure 8-10. Click **Finish**.


New Plug-in Project

Plug-in Content

Enter the data required to generate the plug-in.

Plug-in Properties

Plug-in ID: com.ibm.itsoral.ercpcasestudy.mobilestore

Plug-in Version: 1.0.0

Plug-in Name: MobileStore Plug-in

Plug-in Provider: ITSO

Classpath:

Plug-in Class

[✓] Generate the Java class that controls the plug-in's life cycle

Class Name: com.ibm.itsoral.ercpcasestudy.mobilestore.MobileStorePlugin

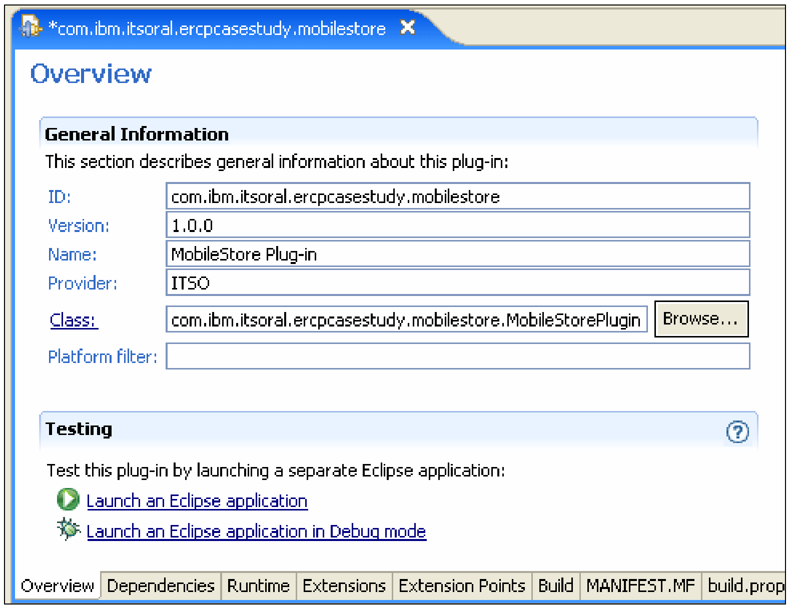
[✓] This plug-in will make contributions to the UI

Rich Client Application

Would you like to create a rich client application? Yes [✓ No]

Figure 8-10 Creating the ITSO Mobile Store project

6. The wizard creates the project and opens the plug-in descriptor.


*com.ibm.itsoral.ercpcasestudy.mobilestore

Overview

General Information

This section describes general information about this plug-in:

ID: com.ibm.itsoral.ercpcasestudy.mobilestore
Version: 1.0.0
Name: MobileStore Plug-in
Provider: ITSO
Class: com.ibm.itsoral.ercpcasestudy.mobilestore.MobileStorePlugin
Platform filter:

Testing

Test this plug-in by launching a separate Eclipse application:

Launch an Eclipse application
Launch an Eclipse application in Debug mode

Figure 8-11 Plug-in descriptor editor

The Overview section of the descriptor holds general information about the plug-in such as its name, provider, activator class, and so on, as shown in Figure 8-11. There are other sections in the descriptor that hold information about dependencies with other plug-ins, extensions contributed to the platform, build information, and so on. The descriptor is divided in two the files, the META-INF/MANIFEST.MF file and the plugin.xml file.

Note: The plugin.xml file is created automatically by Eclipse when an extension is added using the plug-in descriptor editor, as shown in Figure 8-11.

The plugin.xml file holds the extensions contributed for the plug-in to the running platform (eRCP in this case) and the MANIFEST.MF file holds all other information, except the build information that is kept in the build.properties file.

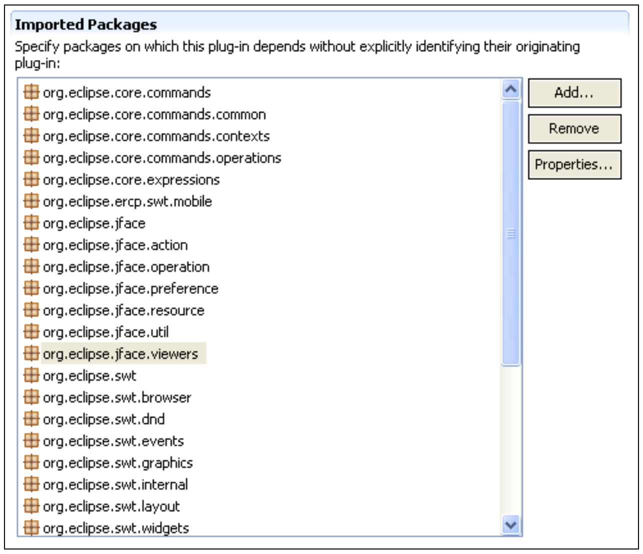
You have probably noticed that there are several error messages in the Problems view that refer to classes that cannot be resolved. This situation occurs because the wizard does not add all the required dependencies for the plug-in when the project is created. You can fix this by following these steps:

1. Open the MANIFEST.MF file if not already open. Select the Dependencies tab. You need to add the required dependencies here. You can add them by importing the entire required plug-in or by selecting the packages that they export.

   Note: Generally, you should use the `Imported Package` method instead the `Required Plug-in` method if you want to isolate your application from changes in how the plug-in provider packages its classes. For example, when you are working with a *Milestone Version* of the eRCP platform, you could expect that the packaging will be changed in a future release. Therefore, it is a good practice to import the eRCP classes using the `Imported Package` method. The DB2e and JSR 169 bundles are not expected to change. Therefore, it is preferred to use the Required Plug-in method in this case.

2. On the Imported Packages section, click **Add**. In the Package Selection window select all the packages that begin with org.eclipse.ui, org.eclipse.core, org.eclipse.ercp,

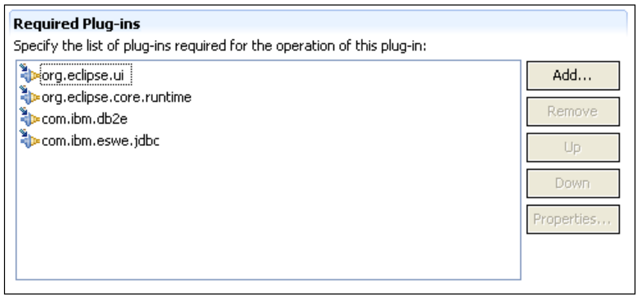
org.eclipse.jface, and org.eclipse.swt, *except* the ones that contain the internal word, and click **OK**. The packages should be listed in the Imported Packages section, as shown in Figure 8-12.


Imported Packages
Specify packages on which this plug-in depends without explicitly identifying their originating plug-in:

org.eclipse.core.commands
org.eclipse.core.commands.common
org.eclipse.core.commands.contexts
org.eclipse.core.commands.operations
org.eclipse.core.expressions
org.eclipse.ercp.swt.mobile
org.eclipse.jface
org.eclipse.jface.action
org.eclipse.jface.operation
org.eclipse.jface.preference
org.eclipse.jface.resource
org.eclipse.jface.util
org.eclipse.jface.viewers
org.eclipse.swt
org.eclipse.swt.browser
org.eclipse.swt.dnd
org.eclipse.swt.events
org.eclipse.swt.graphics
org.eclipse.swt.internal
org.eclipse.swt.layout
org.eclipse.swt.widgets

Figure 8-12 Importing the eRCP packages

3. On the Required Plug-ins section, click **Add**. In the Plug-in Selection window, select **com.ibm.db2e (8.1.4)** and **com.ibm.eswe.jdbc (3.0.0)** and click **OK**. The plug-ins should be listed in the Required Plug-ins as shown in Figure 8-13.


Required Plug-ins
Specify the list of plug-ins required for the operation of this plug-in:

org.eclipse.ui
org.eclipse.core.runtime
com.ibm.db2e
com.ibm.eswe.jdbc

Figure 8-13 Adding the DB2e bundles as required plug-ins

4. Save the file. The errors that are related to the missing classes are gone.

8.3 Designing the ITSO Mobile Store application

In this section, the sample application design is described as well as what patterns and usage are useful for a specific situation or problem.

8.3.1 ITSO Mobile Store architecture

ITSO Mobile Store is a data-driven application that is not so different to the one that you develop for the desktop. Although the application is designed with a mobile device in mind — and mobile devices have limited resources such as memory, CPU power, screen size, different input methods and so on — we think that many software design patterns still apply. So the first pattern that we applied to the application design is the *Layer* pattern. The application is divided into in several layers, implemented as Java packages, grouping classes with similar responsibilities according to the *high cohesion* pattern and decoupling them from the other layers using interfaces according to the *low coupling* pattern. Figure 8-14 shows the layer division as well their dependencies.

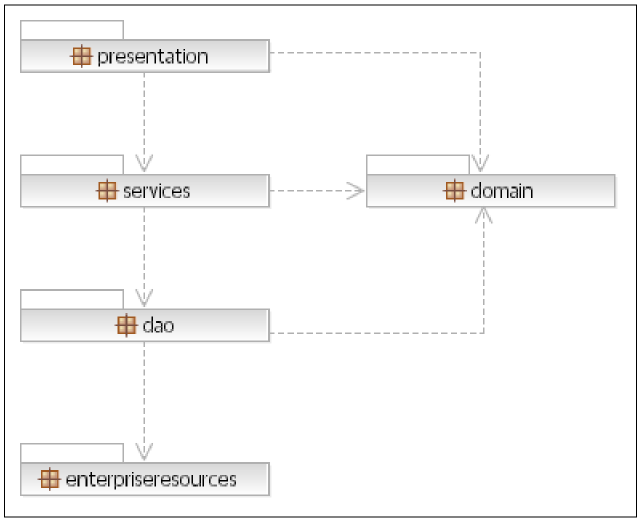


Figure 8-14 Application layers

The application is divided into several layers and the dependencies between them are well defined. This package division is similar to the ones found in larger applications. The good thing about patterns is that they are applicable to several application types. We describe each layer in the sections that follow.

8.3.2 Enterprise Resources layer

The Enterprise Resource layer is composed of all the classes that allow connectivity to the enterprise resource systems, such as databases, messaging systems, and so on. The only external resource in the application is the database. For the ITSO Mobile Store, there is only one class in charge of the local database connectivity: `DatabaseManager`. This class centralizes the connection to the database and keeps the current local database location. This class follows the Singleton pattern.

8.3.3 Domain layer

The Domain layer contains the classes that represent the problem domain. Because this a store application, you would expect to find things such as orders, products, customers, and so on. Figure 8-15 shows the classes that compose this layer and their relationships.

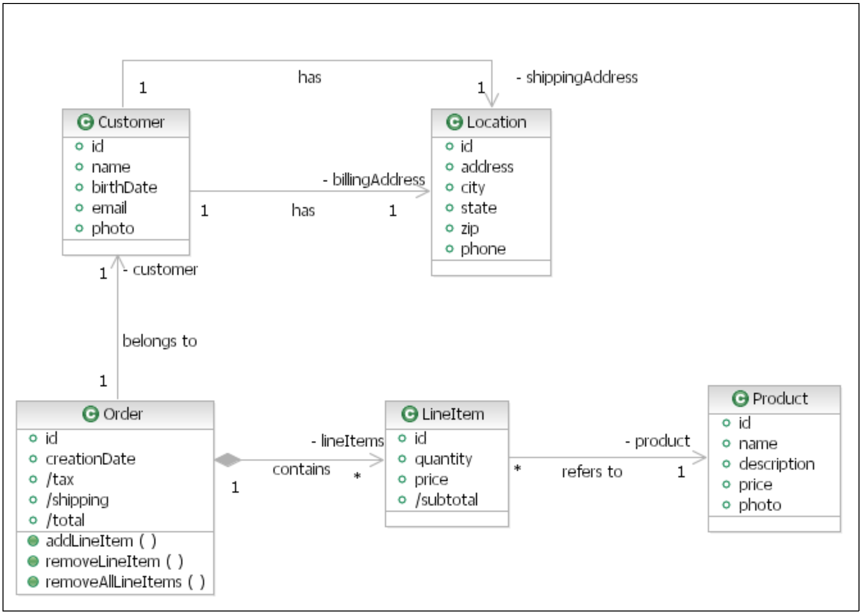


Figure 8-15 Domain layer

Note: Not every attribute and methods are shown in this diagram. For example, the accessor methods (setters and getters) are not included. Also notice that in this diagram you see attributes with public or protected visibility, but it does not mean that in the code they are instance variables with public or protected access (a *very bad* practice by the way). Instead, it means that these attributes have public or protected *accessors* (getters and setters). Also note that the attributes with a slash character (/) as a prefix are calculated according to the UML standard. Using this type of UML style reduces the clutter in the diagram and, at the same time, shows the important aspects of each class.

The classes for this layer are as follows:

- Customer

  Holds information about a customer, including name, birth date, and so on. Each customer can have up to two addresses as depicted in Figure 8-15.

- Location

  Holds information about a particular location inside the U.S. and is used primarily to keep the business and shipping addresses of a particular customer.

- Order

  Holds information for orders. Each order belongs to a customer and contains several line items for each product in the order. It has derived attributes to calculate the subtotal, tax, shipping, and total for the order. In addition, it has the responsibility of taking care of the its line item objects.

- LineItem

  Holds information for a particular line item inside an order. Note that a line item keeps track of the product price when the order was created.

- Product

  Holds information about the products available for purchasing using the ITSO Mobile Store application.

8.3.4 Data Access Objects layer

The Data Access Objects (DAO) layer provides several objects with create, retrieve, update, and delete (CRUD) operations that close the gap between the domain model and its persistent representation, database tables. The persistency is implemented using direct JDBC calls to the database. Figure 8-16 shows the interfaces and classes in this layer.

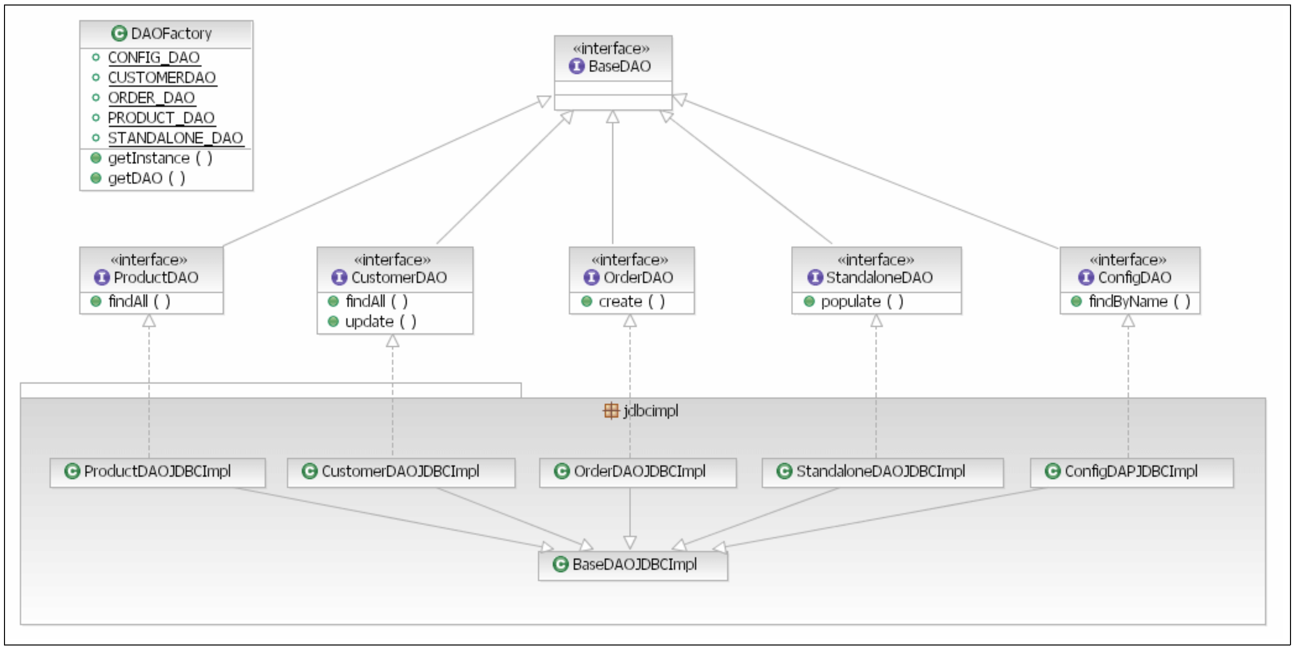


Figure 8-16 DAO Layer

The DAO layer exposes its functionality as interfaces. The `jdbcimpl` package contains the classes that implement the interfaces using JDBC calls. The mapping between the interface and a particular implementation is done by a factory class called `DAOFactory`. The main interfaces and classes of this layer are as follows:

- ProductDAO

  Declares one method, `findAll()`, which returns all the products that are available for shopping.

- CustomerDAO

  Declares two methods:

  – `findAll()` returns all the customers that can place orders
  – `update()` updates the customer information in the database

- OrderDAO

  Exposes a `create()` method that adds a new order and all its associated line items to the database.

- StandaloneDAO

  Exposes a `populate()` method that is used to create the underlaying database and populates the tables with data.

- ConfigDAO

  Exposes a `findByName()` method that is used to get values for a particular name. For example, give the `STATE` name provides values such as `NC`, `NY`, and so on.

- BaseDAO

  Provides a common base for all the interfaces in the DAO layer.

- DAOFactory

  Follows the Singleton pattern and maps the interfaces with their implementations. Each DAO implementation is mapped as a constant in the class. The `getDAO()` is the constant as a parameter and provides an instance of the DAO. The return type for `getDAO()` is BaseDAO.

Note: For more details about the implementation, refer to the source code that is provided in Appendix A, "Additional material" on page 225. For information about how to import this code, see 8.3.7, "Importing the ITSO Mobile Store code" on page 149.

8.3.5 Services layer

The Business Services layer provides interfaces that contain methods that carry on the application business logic. Also, it serves as a facade between the presentation layer and lower level layers, isolating the presentation layer from the complexities of the underlaying layers, achieving low coupling, and making possible to change the technology used in the presentation layer, if it is required.

Figure 8-17 shows the main interfaces and classes for this layer. Notice that as part of the layer, we provide an implementation on the interfaces in the `impl` package. If you review the code of the implemented services, you note that we are using the *dependency injection* pattern, also called *inversion of control*, to provide the required DAO instances to the services classes. For ITSO Mobile Store, the dependencies are injected in the `ServiceFactory` class using code.

Note: A better approach could be to declare the dependencies in some kind of application descriptor. In fact, there are several good Inversion of Control frameworks available that perform that task and that provide several other services. You could use them provided that you could tailor them to the J2ME environment. For mobile devices, you can use the Spring framework, using properties files (instead of the more popular XML files) to declare the dependencies. You can find more information about the dependency injection pattern in the Martin Flower's site at:

    http://www.martinfowler.com/articles/injection.html

You can find more information about the Spring framework at:

    http://www.springframework.org/

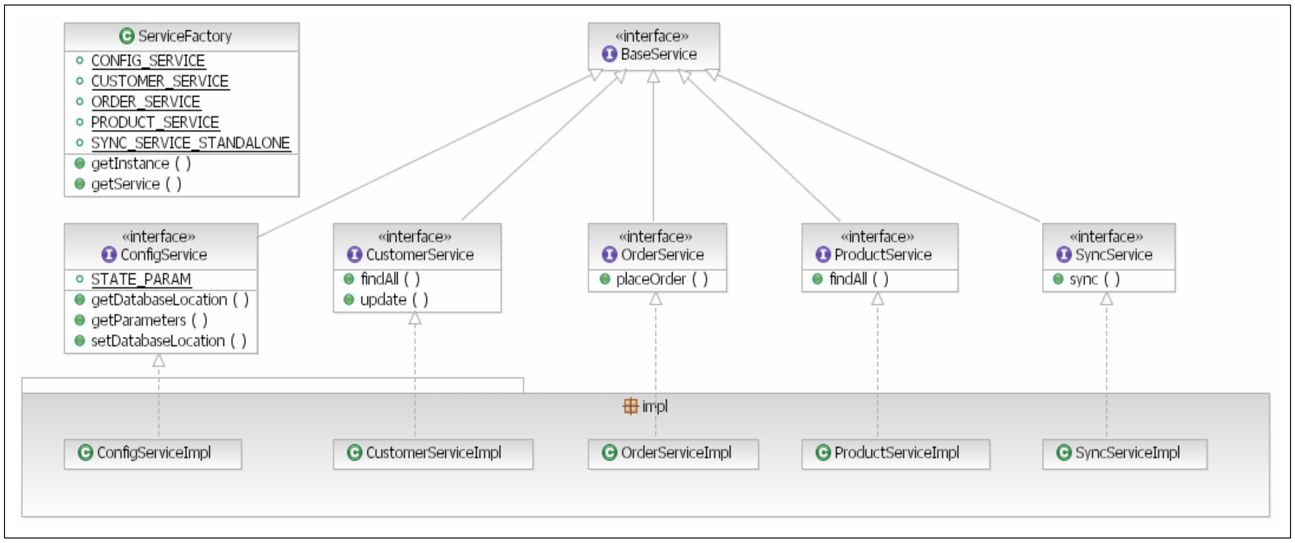


Figure 8-17 Service layer

The main interfaces of this layer are as follows:

- `ConfigService`

  Holds the methods that are needed to access to the configuration information and the parameters for the application.

- `CustomerService`

  Holds the methods that are related to managing customer information.

- `OrderService`

  Provides methods that are used to manage orders.

- `ProductService`

  Holds the methods that are related to retrieving customer information.

- `SyncService`

  Holds the methods that are related to retrieving information from the remote database and syncing information with the local database. For the sake of simplicity and to avoid the need to use a DB2 Everyplace Sync Server to test the application, the current implementation calls the `StandaloneDAO` to recreate the database using local resources each time the `sync()` method is called. Feel free to replace the current implementation with one that performs a real database synchronization. You can find more information about the synchronization features of DB2 Everyplace in *IBM WebSphere Everyplace Deployment V6 Handbook for Developers and Administrators Volume I: Installation and Administration*, SG24-7141.

- `BaseService`

  Provides a common base to the different services interfaces.

- `ServiceFactory`

  This factory class, that follows the Singleton pattern, maps the interfaces with an actual implementation of them. The `getService()` method takes a parameter with the service required and returns a BaseService instance that must be casted to the service requested.

8.3.6 Presentation layer

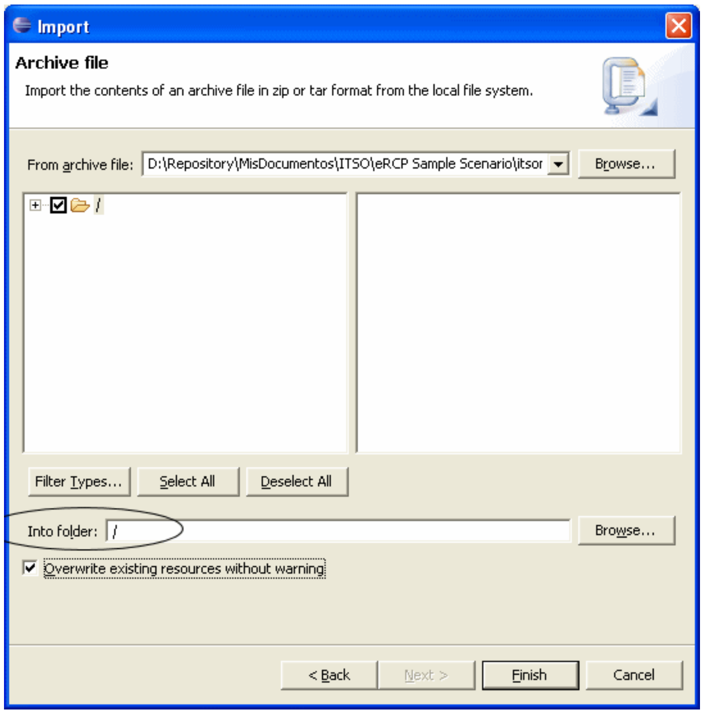
The Presentation layer contains the GUI that allows the user to interact with the application. It has been developed using eRCP technologies, eSWT, eJFace, and eWorkbench. Therefore it is the main focus of this chapter. You see this layer in detail in 8.4, “Developing the presentation layer using eRCP” on page 150.

8.3.7 Importing the ITSO Mobile Store code

Now, you need to import the code for the layers, except the presentation one.

Locate the itsomobilestore_src_no_presentation.zip file in the \ercpcasestudy directory in additional materials and follow these steps to import it:

1. Start Eclipse. Open the ITSO Mobile Store project created in 8.2.3, “Creating the ITSO Mobile Store project” on page 141.
2. Right click the project and select **Import**. Select **Archive** and click **Next**.
3. Click **Browse** in the From archive file field and locate the itsomobilestore_src_no_presentation.zip file.
4. In the Into folder field enter a backslash (/), select **Overwrite existing resources without warning**, and click **Finish**, as shown in Figure 8-18.


Import

Archive file

Import the contents of an archive file in zip or tar format from the local file system.

From archive file: D:\Repository\MisDocumentos\ITSO\eRCP Sample Scenario\itsor

Into folder: /

☑ Overwrite existing resources without warning

Figure 8-18 Importing the ITSO Mobile Store code

5. Take a look at the code, it implements the design that is outlined in 8.3, “Designing the ITSO Mobile Store application” on page 143.

8.4 Developing the presentation layer using eRCP

Now that you have the code for all the layers in place, you can begin developing the presentation layer using eRCP. However, first we explain how this layer is divided as illustrated in Figure 8-19.

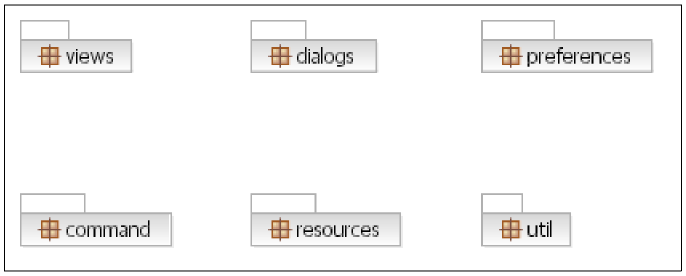


Figure 8-19 Presentation packages

The views package contains all the view objects. The dialogs package contains the dialogs that are used in the application. The preferences package contains the page used to contribute to the eWorkbench preferences subsystem. The command package contains classes used to manage the eSWT Mobile Extension Command objects used in the application. The resources package contains some images and finally the util package contains utility classes that are used in all the other packages. This section discusses these artifacts in great detail.

8.4.1 The command package

A command represents an action that an user can trigger in the user interface. Each command is associated to an specific element in the user interface and is visible when the bond user interface element or its parent has the focus. In Win32 and WM2003 devices, the commands are implemented as menu elements.

For ITSO Mobile Store, we designed two classes to manage the application commands, as shown in Figure 8-20.

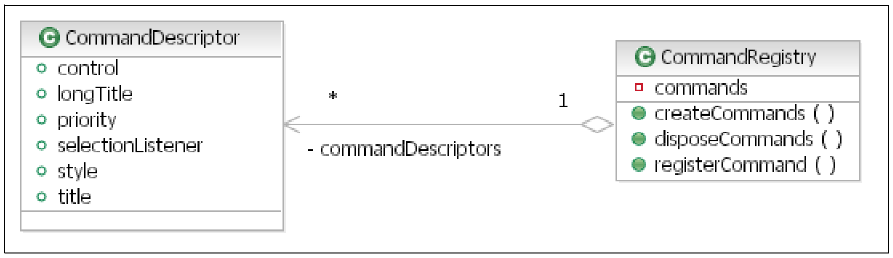


Figure 8-20 Command package

CommandDescriptor

The `CommandDescriptor` class holds information about a command. This class has the control property that indicates the element to which the command belongs, the priority property that indicates the menu order for the command and the `selectionListener` property that is the handler to be invoked when the command is selected.

Follow these steps to create the `CommandDescriptor` class:

1. Select **File** → **New** → **Class** from the menu bar. Enter `com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.command` as the package

name and `CommandDescriptor` as the class name. The package is created automatically. Click **Finish**.

2. Replace the newly generated class with the code in Example 8-1. Save the changes.

Example 8-1 CommandDescriptor.java

```
package com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.command;

import org.eclipse.swt.events.SelectionListener;
import org.eclipse.swt.widgets.Control;

/**
 * Holds information about a command
 *
 */
public class CommandDescriptor {

   private Control control;

   private int style;

   private int priority;

   private String title;

   private String longTitle;

   private SelectionListener selectionListener;

   /**
    * @return Returns the control.
    */
   public Control getControl() {
      return control;
   }

   /**
    * @param control The control to set.
    */
   public void setControl(Control control) {
      this.control = control;
   }

   /**
    * @return Returns the longTitle.
    */
   public String getLongTitle() {
      return longTitle;
   }

   /**
    * @param longTitle The longTitle to set.
    */
   public void setLongTitle(String longTitle) {
      this.longTitle = longTitle;
   }

   /**
    * @return Returns the priority.
    */
   public int getPriority() {
      return priority;
   }

   /**
    * @param priority The priority to set.
    */
   public void setPriority(int priority) {
      this.priority = priority;
   }

   /**
    * @return Returns the selectionListener.
    */
   public SelectionListener getSelectionListener() {
      return selectionListener;
   }

   /**
    * @param selectionListener The selectionListener to set.
    */
   public void setSelectionListener(SelectionListener selectionListener) {
      this.selectionListener = selectionListener;
   }

   /**
    * @return Returns the style.
    */
   public int getStyle() {
      return style;
   }

   /**
    * @param style The style to set.
    */
   public void setStyle(int style) {
      this.style = style;
   }

   /**
    * @return Returns the title.
    */
   public String getTitle() {
      return title;
   }

   /**
    * @param title The title to set.
    */
   public void setTitle(String title) {
      this.title = title;
   }

}
```

CommandRegistry

The `CommandRegistry` maintains all the registered `CommandDescriptors` and allows you to create and dispose the `Command` objects at your convenience, as shown in Example 8-2.

Example 8-2 Creating commands from CommandDescriptors

```
...
   CommandDescriptor descriptor = (CommandDescriptor) iter.next();
   Command newCommand = new Command(descriptor.getControl(), descriptor.getStyle(),
      descriptor.getPriority());
   newCommand.setText(descriptor.getTitle());
   newCommand.setLongLabel(descriptor.getLongTitle());
   newCommand.addSelectionListener(descriptor.getSelectionListener());
   commands.add(newCommand);
...
```

Each view in the ITSO Mobile Store application has its own command registry. Follow these steps to create the `CommandRegistry` class:

1. Select **File** →**New** → **Class** from the menu bar. Enter `com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.command` as the package name and `CommandRegistry` as the class name. Click **Finish**.
2. Replace the newly generated class with the code in Example 8-3. Save the changes.

Example 8-3 CommandRegistry.java

```
package com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.command;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import org.eclipse.ercp.swt.mobile.Command;
import org.eclipse.swt.events.SelectionListener;
import org.eclipse.swt.widgets.Control;

/**
 *  Mantains the registered CommandDescriptors and allows to create and
 *  dispose the Command objects at convenience. Each view in the ITSO Mobile
 *  Store application has its own command registry.
 *
 */
public class CommandRegistry {

   /**
    * The command descriptors for this registry
    */
   private List commandDescriptors;

   /**
    * The generated commands
    */
   private List commands;

   /**
    * Default constructor. Initialize the containers for the commands and
    * command registry
    *
    */
   public CommandRegistry() {
```

```
   commandDescriptors = new ArrayList();
   commands = new ArrayList();
}

/**
 * Register a command provided a CommandDescriptor
 *
 * @param descriptor Descriptor for the command
 */
public void registerCommand(CommandDescriptor descriptor) {

   commandDescriptors.add(descriptor);
}

/**
 * Register a command provided the command information
 *
 * @param commandStyle Style for the command
 * @param priority Command priority
 * @param title Label for the command
 * @param longTitle Long label for the command
 * @param selectionListener Code to eexecute when the command is selected
 */
public void registerCommand(Control control, int commandStyle,
      int priority, String title, String longTitle,
      SelectionListener selectionListener) {
   CommandDescriptor descriptor = new CommandDescriptor();
   descriptor.setStyle(commandStyle);
   descriptor.setPriority(priority);
   descriptor.setTitle(title);
   descriptor.setLongTitle(longTitle);
   descriptor.setSelectionListener(selectionListener);
   descriptor.setControl(control);
   registerCommand(descriptor);
}

/**
 * Create the command objects from the command descriptors.
 *
 */
public void createCommands() {

   for (Iterator iter = commandDescriptors.iterator(); iter.hasNext();) {
      CommandDescriptor descriptor = (CommandDescriptor) iter.next();
      Command newCommand = new Command(descriptor.getControl(),
            descriptor.getStyle(), descriptor.getPriority());
      newCommand.setText(descriptor.getTitle());
      newCommand.setLongLabel(descriptor.getLongTitle());
      newCommand.addSelectionListener(descriptor.getSelectionListener());
      commands.add(newCommand);
   }
}

/**
 * Dispose the commands for this registry
 *
 */
public void disposeCommands() {
        for (Iterator iter = commands.iterator(); iter.hasNext();) {
            Command command = (Command) iter.next();
            command.dispose();
        }

        commands = new ArrayList();
    }
}
```

8.4.2 The util package

The util package provides convenience classes that ar used by the other classes. Figure 8-21 shows the classes that compose this package.

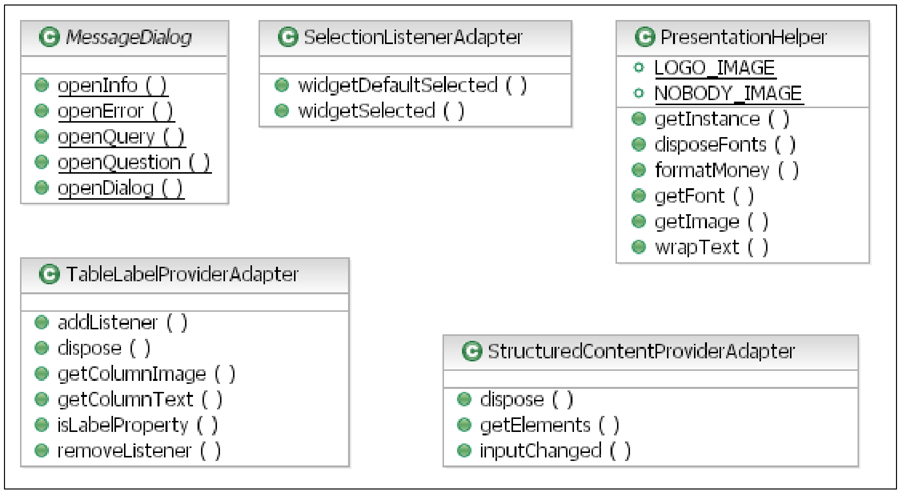


Figure 8-21 The util package

MessageDialog

This class provides several static methods to open common used dialogs, such as information dialogs, confirmation dialogs, and so on. They use the `MessageBox` class that is described in 3.1.7, “MessageBox” on page 41, as illustrated in Example 8-4.

Example 8-4 Using MessageBox

```
...
public static int openDialog(Shell shell, String title, String message, int style) {

        MessageBox messageBox = new MessageBox(shell, style);
        messageBox.setText(title);
        messageBox.setMessage(message);
        return messageBox.open();
}
...
```

This class is similar to the one that is provided in the normal JFace library. Unfortunately, it is not available in the current M7 eJFace library. Follow these steps to create the MessageDialog class:

1. Select **File** →**New** → **Class** from the menu bar. Enter `com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.util` as the package name and `MessageDialog` as the class name. The package is created automatically. Click **Finish**.

2. Replace the newly generated class with the code in Example 8-5. Save the changes.

Example 8-5 MessageDialog.java

```
package com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.util;

import org.eclipse.ercp.swt.mobile.QueryDialog;
import org.eclipse.swt.SWT;
import org.eclipse.swt.widgets.MessageBox;
import org.eclipse.swt.widgets.Shell;

/**
 * This class provides several static methods to open common used dialogs, like
 * OK dialogs, confirmation dialogs and so on. They use the MessageBox class.
 * This class is similar to the one provided in the complete JFace library,
 * sadly it is not available in the current M7 eJFace library.
 *
 */
public final class MessageDialog {

   /**
    * Opens a dialog with an information icon and an OK button
    *
    * @param shell
    *           The parent shell
    * @param title
    *           The dialog title
    * @param message
    *           The message to show
    * @return The index of the button clicked
    */
   public static int openInfo(Shell shell, String title, String message) {

      return openDialog(shell, title, message, SWT.OK | SWT.ICON_INFORMATION);
   }

   /**
    * Opens a dialog with an error icon and an OK button
    *
    * @param shell
    *           The parent shell
    * @param title
    *           The dialog title
    * @param message
    *           The message to show
    * @return The index of the button clicked
    */
   public static int openError(Shell shell, String title, String message) {

      return openDialog(shell, title, message, SWT.OK | SWT.ICON_ERROR);

   }

   /**
    * Opens a dialog with an question icon and two buttons YES and NO
    *
    * @param shell
    *           The parent shell
    * @param title
    *           The dialog title
    * @param message
```

```
	 *            The message to show
	 * @return The index of the button clicked
	 */
	public static int openQuestion(Shell shell, String title, String message) {

	   return openDialog(shell, title, message, SWT.YES | SWT.NO
	         | SWT.ICON_QUESTION);
	}

	/**
	 * Opens a dialog provided its style parameter
	 *
	 * @param shell
	 *            The parent shell
	 * @param title
	 *            The dialog title
	 * @param message
	 *            The message to show
	 * @param style
	 *            The style for the dialog
	 * @return The index of the button clicked
	 */
	public static int openDialog(Shell shell, String title, String message,
	      int style) {

	   MessageBox messageBox = new MessageBox(shell, style);
	   messageBox.setText(title);
	   messageBox.setMessage(message);
	   return messageBox.open();
	}

	/**
	 * Opens a dialog used to collect input from the user.
	 *
	 * @param shell
	 *            The parent shell
	 * @param style
	 *            The dialog style
	 * @param prompt
	 *            The prompt for the value entered
	 * @param defaultValue
	 *            A default value shown when the dialog opens
	 * @return The value entered as an String
	 */
	public static String openQuery(Shell shell, int style, String prompt,
	      String defaultValue) {

	   QueryDialog queryDialog = new QueryDialog(shell, SWT.NONE, style);
	   queryDialog.setPromptText(prompt, defaultValue);
	   return queryDialog.open();
	}
}
```

PresentationHelper

It seems that every application needs at least one helper class. This class provides several utility methods that are used by the other classes in the presentation layer. It provides a point to register fonts and static images that are bundled with the application as shown in the code snippet provided in Example 8-6.

Example 8-6 Accessing to images and fonts

```
...
   // Accessing to the JFace default ImageRegistry
   Image image = JFaceResources.getImageRegistry().get(imageName);
   if (image == null) {

      image = new Image(display, imageSource);
      JFaceResources.getImageRegistry().put(imageName, image);
   }
   return image;
...
   // Accesing to a custom Font registry
   Font font = (Font) fontRegistry.get(new Integer(style));
   if (font == null) {
      font = new Font(display, new FontData(DEFAULT_FONT_NAME,
            DEFAULT_FONT_SIZE, style));
      fontRegistry.put(new Integer(style), font);
   }
   return font;
...
```

In addition, this class provides formatting methods. The `PresentationHelper` class is implemented using the *Singleton* pattern. Follow these steps to create the PresentationHelper class:

1. Select **File →New → Class** from the menu bar. Enter `com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.util` as the package name and `PresentationHelper` as the class name. Click **Finish**.
2. Replace the newly generated class with the code in Example 8-7. Save the changes.

Example 8-7 PresentationHelper.java

```
package com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.util;

import java.io.InputStream;
import java.text.DecimalFormat;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.StringTokenizer;

import org.eclipse.jface.resource.JFaceResources;
import org.eclipse.swt.graphics.Font;
import org.eclipse.swt.graphics.FontData;
import org.eclipse.swt.graphics.Image;
import org.eclipse.swt.widgets.Display;

/**
 * This class provides several utility methods used by the other classes in the
 * presentation layer. Provides a point to register fonts and static images
 * bundled with the application, formatting methods and so on. It has been
 * implemented using the Singleton pattern.
 *
 */
public final class PresentationHelper {

   private final static String PLUGIN_ROOT
      = "/com/ibm/itsoral/ercpcasestudy/mobilestore";

   public final static String LOGO_IMAGE = PLUGIN_ROOT
         + "/presentation/resources/logo.png";

   public final static String NOBODY_IMAGE = PLUGIN_ROOT
         + "/presentation/resources/nobody.png";

   private static final String DEFAULT_FONT_NAME = "MS Shell Dlg";

   private static final int DEFAULT_FONT_SIZE = 10;

   private DecimalFormat moneyFormat;

   /**
    * The container for the fonts used in the application
    */
   private Map fontRegistry = new HashMap();

   private static PresentationHelper instance;

   /**
    * Default constructor.
    *
    */
   protected PresentationHelper() {

      moneyFormat = new DecimalFormat("$ ###,###,##0.00");
   }

   /**
    * @return The unique instance for this class
    */
   public static PresentationHelper getInstance() {

      if (instance == null) {
         instance = new PresentationHelper();
      }

      return instance;
   }

   /**
    * Format a double using the $ ###,###,##0.00 pattern
    *
    * @param amount
    *           The amount to format
    * @return the formatted amount
    */
   public String formatMoney(double amount) {

      return moneyFormat.format(amount);

   }
```

```
/**
 * Wrap a string to the number of characters specified in the maxLength
 * parameter.
 *
 * @param text
 *            The text to wrap
 * @param maxLength
 *            The maximum length for the line
 * @return The wrapped text
 */
public String wrapText(String text, int maxLength) {

   StringTokenizer tokenizer = new StringTokenizer(text, " ");
   int lineLength = 0;
   StringBuffer buffer = new StringBuffer(text.length());
   while (tokenizer.hasMoreTokens()) {
      String token = tokenizer.nextToken();
      if (lineLength + token.length() < maxLength) {
         lineLength += token.length();
      } else {
         buffer.append("\n");
         lineLength = token.length();
      }
      buffer.append(token + " ");
   }

   return buffer.toString();
}

/**
 * Get an static image from the JFace default registry. The image names are
 * provided as constant in this class.
 *
 * @param display
 *            The SWT display
 * @param imageName
 *            An image defined as a constant in this class
 * @return The image object
 */
public Image getImage(Display display, String imageName) {

   return getImage(display, imageName, getClass().getResourceAsStream(
         imageName));
}

/**
 * Get an image from the JFace default registry. If the image has not been
 * registered before is registered and returned. If the image was previously
 * registered, it's returned.
 *
 * @param display
 *            The SWT display
 * @param imageName
 *            The image name under which the image will be registered and
 *            /or retrieved
 * @param imageSource
 *            The image source
 * @return The image object
 */
public Image getImage(Display display, String imageName,
            InputStream imageSource) {

        Image image = JFaceResources.getImageRegistry().get(imageName);
        if (image == null) {

            image = new Image(display, imageSource);
            JFaceResources.getImageRegistry().put(imageName, image);
        }

        return image;
    }

    /**
     * Get a font from the application custom registry. If the font has not been
     * registered before, it's registered under the provided name. If it was
     * registered before it is returned.
     *
     * @param display
     *            The SWT display
     * @param style
     *            The font style
     * @return The font object
     */
    public Font getFont(Display display, int style) {

        Font font = (Font) fontRegistry.get(new Integer(style));
        if (font == null) {
            font = new Font(display, new FontData(DEFAULT_FONT_NAME,
                    DEFAULT_FONT_SIZE, style));
            fontRegistry.put(new Integer(style), font);
        }

        return font;
    }

    /**
     * Dispose the fonts registered in the custom application registry
     *
     */
    public void disposeFonts() {

        for (Iterator iter = fontRegistry.entrySet().iterator(); iter.hasNext();) {
            Font font = (Font) iter.next();
            font.dispose();
        }
    }

}
```

SelectionListenerAdapter

This class is an adapter for the `org.eclipse.swt.events.SelectionListener` interface, and it is provided for convenience. The `SelectionListener` interface is part of the SWT API for event handling that is explained in Chapter 5, "eSWT expanded" on page 89. In general the SWT/JFace design philosophy is to try to keep the API simple and compact, so that any convenience methods and classes must be provided by the application developer.

Follow these steps to create the `SelectionListenerAdapter` class:

1. Select **File** →**New** → **Class** from the menu bar. Enter `com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.util` as the package name and `SelectionListenerAdapter` as the class name. Click **Finish**.
2. Replace the newly generated class with the code in Example 8-8. Save the changes.

Example 8-8 SelectionListernerAdapter.java

```
package com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.util;

import org.eclipse.swt.events.SelectionEvent;
import org.eclipse.swt.events.SelectionListener;

/**
 * Adapter for the org.eclipse.swt.events.SelectionListener interface
 *
 */
public class SelectionListenerAdapter implements SelectionListener {

   /*
    *  (non-Javadoc)
    * @see org.eclipse.swt.events.SelectionListener#widgetSelected(
    * org.eclipse.swt.events.SelectionEvent)
    */
   public void widgetSelected(SelectionEvent arg0) {


   }

   /*
    *  (non-Javadoc)
    * @see org.eclipse.swt.events.SelectionListener#widgetDefaultSelected(
    * org.eclipse.swt.events.SelectionEvent)
    */
   public void widgetDefaultSelected(SelectionEvent arg0) {


   }
}
```

TableLabelProviderAdapter

This class is an adapter for the `org.eclipse.jface.viewers.ITableLabelProvider` interface. The `ITableLabelProvider` interface is part of the JFace Viewers API. It allows you to work with several SWT widgets using an MVC design pattern approach. More information about the JFace Viewers API can be found in Chapter 6, “eJFace applications” on page 115. Follow these steps to create the `TableLabelProviderAdapter` class:

1. Select **File** →**New** → **Class** from the menu bar. Enter `com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.util` as the package name and `TableLabelProviderAdapter` as the class name. Click **Finish**.
2. Replace the newly generated class with the code in Example 8-9. Save the changes.

Example 8-9 TableLabelProviderAdapter.java

```
package com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.util;

import org.eclipse.jface.viewers.ILabelProviderListener;
import org.eclipse.jface.viewers.ITableLabelProvider;
import org.eclipse.swt.graphics.Image;

/**
 * Adapter for the org.eclipse.jface.viewers.ITableLabelProvider interface
 *
 */
public class TableLabelProviderAdapter implements ITableLabelProvider {

   /*
    *  (non-Javadoc)
    * @see org.eclipse.jface.viewers.ITableLabelProvider#getColumnImage(
    * java.lang.Object, int)
    */
   public Image getColumnImage(Object arg0, int arg1) {

      return null;
   }

   /*
    *  (non-Javadoc)
    * @see org.eclipse.jface.viewers.ITableLabelProvider#getColumnText(
    * java.lang.Object, int)
    */
   public String getColumnText(Object arg0, int arg1) {

      return null;
   }

   /*
    *  (non-Javadoc)
    * @see org.eclipse.jface.viewers.IBaseLabelProvider#addListener
    * (org.eclipse.jface.viewers.ILabelProviderListener)
    */
   public void addListener(ILabelProviderListener arg0) {

   }

   /*
    *  (non-Javadoc)
    * @see org.eclipse.jface.viewers.IBaseLabelProvider#dispose()
    */
   public void dispose() {

   }

   /*
    *  (non-Javadoc)
    * @see org.eclipse.jface.viewers.IBaseLabelProvider#isLabelProperty(
    * java.lang.Object, java.lang.String)
    */
   public boolean isLabelProperty(Object arg0, String arg1) {

      return false;
   }

   /*
    *  (non-Javadoc)
    * @see org.eclipse.jface.viewers.IBaseLabelProvider#removeListener(
    * org.eclipse.jface.viewers.ILabelProviderListener)
    */
   public void removeListener(ILabelProviderListener arg0) {

   }
}
```

StructuredContentProviderAdapter

This class is an adapter for the `org.eclipse.jface.viewers.IStructuredContentProvider` interface. The `IStructuredContentProvider` interface is also part of the JFace Viewers API. Follow these steps to create the `StructuredContentProviderAdapter` class:

1. Select **File** →**New** → **Class** from the menu bar. Enter `com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.util` as the package name and `StructuredContentProviderAdapter` as the class name. Click **Finish**.
2. Replace the newly generated class with the code in Example 8-10. Save the changes.

Example 8-10 StructuredContentProviderAdapter.java

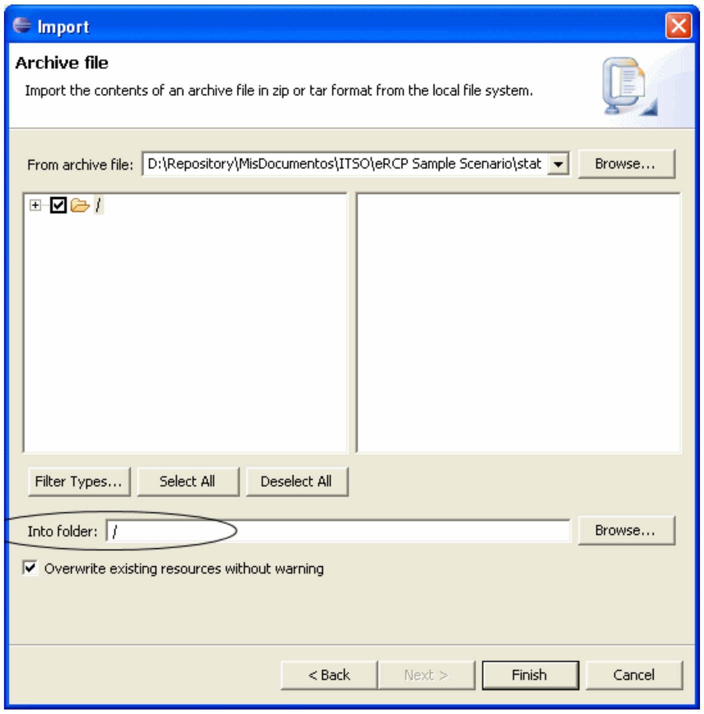
```
package com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.util;

import org.eclipse.jface.viewers.IStructuredContentProvider;
import org.eclipse.jface.viewers.Viewer;

/**
 * Adapter for the org.eclipse.jface.viewers.IStructuredContentProvider interface
 *
 */
public class StructuredContentProviderAdapter implements IStructuredContentProvider {
   /*
    *  (non-Javadoc)
    * @see org.eclipse.jface.viewers.IStructuredContentProvider#getElements(
    * java.lang.Object)
    */
   public Object[] getElements(Object arg0) {

      return null;
   }

   /*
    *  (non-Javadoc)
    * @see org.eclipse.jface.viewers.IContentProvider#dispose()
    */
   public void dispose() {

   }

   /*
    * (non-Javadoc)
    * @see org.eclipse.jface.viewers.IContentProvider#inputChanged(
    * org.eclipse.jface.viewers.Viewer, java.lang.Object, java.lang.Object)
    */
   public void inputChanged(Viewer arg0, Object arg1, Object arg2) {

   }
}
```

8.4.3 The resources package

The resource package contains the static images used in the application. They are accessible by using the `PresentationHelper` class described in “PresentationHelper” on page 158. You can import the images from the \ercpcasestudy\static_images.zip file from additional materials. For more information, see Appendix A, “Additional material” on page 225.

Follow these steps to import the images into the project:

1. Right-click the project name and select **Import**. Select **Archive File** and click **Next**.
2. Click **Browse** in the From archive file field and select the static_images.zip file.
3. Enter a backslash (/) in the Into folder field as shown in Figure 8-22. Click **Finish**.


Import

Archive file

Import the contents of an archive file in zip or tar format from the local file system.

From archive file: D:\Repository\MisDocumentos\ITSO\eRCP Sample Scenario\stat

Into folder: /

Overwrite existing resources without warning ✓

Figure 8-22 Importing the graphical resources

8.4.4 The preferences package

This package contains preferences support for the application. eRCP provides support for storing preferences and showing them to the user on pages in the eWorkbench Preferences dialog. Preferences are key/value pairs, where the key describes the name of the preference, and the value is one of several different types. To contribute preferences to the eWorkbench you have to add an extension to the plug-in descriptor.

The org.eclipse.ui.preferencePages extension

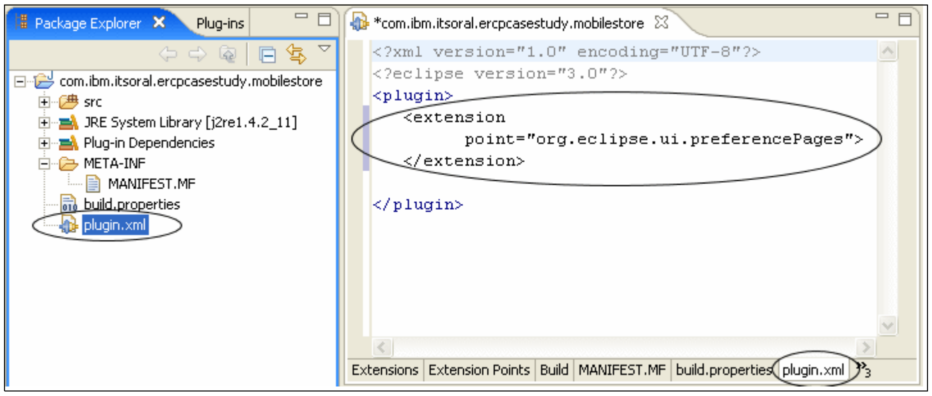
This extension allows you to contribute a preferences page to the eWorkbench preferences dialog. By design, all the contributed extensions to the platform are put in the plug-in XML descriptor, the plugin.xml file. This file is created automatically when you add an extension using the Manifest Editor. Example 8-11 depicts the structure for this extension.

Example 8-11 org.eclipse.ui.preferencePages extension

```
<extension point="org.eclipse.ui.preferencePages">
      <page class="<class_name>" id="<id>" name="<page_name>"/>
</extension>
```

The class pointed in the class attribute must be a subclass of `org.eclipse.jface.preference.PreferencePage`. This class define the preferences page user interface and how the preferences must be stored. The `id` attribute identified uniquely the preference page. The `name` attribute indicates under which name the preferences page should appear in the eWorkbench Preferences dialog. To create this extension for ITSO Mobile Store, follow these steps:

1. Open the Manifest Editor by double-clicking the MANIFEST.MF file and selecting the **Extensions** tab. Click **Add**.
2. Select **org.eclipse.ui.preferencesPage** and click **Finish**. Note that the editor creates the plugin.xml file automatically, as shown in Figure 8-23.
3. In the Manifest Editor, select the plugin.xml tab. The extension is added to the plugin.xml file, as shown in Figure 8-23.



```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin>
   <extension
         point="org.eclipse.ui.preferencePages">
   </extension>

</plugin>
```

Figure 8-23 Adding the preferencesPage extension.

4. Replace the empty extension point with the content shown in Example 8-12. The class for this preferences page is `MainPreferencesPage`, that you will create shortly. The ID has to be unique, generally the fully qualified name of the class is used for this purpose. Finally, the name attribute is the page label in the eWorkbench Preferences dialog.

Example 8-12 Preferences page extension for ITSO Mobile Store

```
<extension point="org.eclipse.ui.preferencePages">
      <page
class="com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.preferences.MainPreferencesPa
ge"
id="com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.preferences.MainPreferencesPage"
name="Mobile Store Configuration"/>
</extension>
```

5. Save the changes and close the editor.

MainPreferencesPage

This class defines the user interface for the preferences page. For ITSO Mobile Store, there are two preferences to be kept: the local database location and if the application should work in stand-alone mode.

The `MainPreferencesPage` inherits from `org.eclipse.jface.preference.PreferencePage`. This `PreferencesPage` define an abstract method, `createContents()`, that the preferences page class must implement. This method creates the page contents using the usual SWT widgets, as shown in Example 8-13, and is invoked when the preferences page is opened.

Example 8-13 Preferences page createContents() method

```
...
   Composite composite = new Composite(parent, SWT.NONE);
   composite.setLayout(new GridLayout(2, true));

   new Label(composite, SWT.LEFT).setText("Standalone Mode: ");
   standaloneButton = new Button(composite, SWT.CHECK);
   standaloneButton.setLayoutData(new GridData(GridData.FILL_HORIZONTAL));
   standaloneButton.setSelection(preferencesStore
         .getBoolean(MainPreferencesPage.P_STANDALONE_MODE));

   new Label(composite, SWT.LEFT).setText("Database Location: ");
   databaseLocationText = new Text(composite, SWT.LEFT | SWT.BORDER);
   databaseLocationText.setLayoutData(new GridData(
         GridData.FILL_HORIZONTAL));
   databaseLocationText.setText(preferencesStore
         .getString(MainPreferencesPage.P_DATABASE_LOCATION));
....
```

Note: The readers familiar with SWT/JFace might be wondering why we are using SWT widgets directly instead the more usual *field editors*. To put it simply, the field editor API is not available in the M7 eJFace package.

Another interesting method is the `performOK()` method, which is invoked when the user clicks OK in the preferences dialog. This method should store the modified preferences and notify the changes to any class that is interested in the preferences settings, as shown in Example 8-14.

Example 8-14 Preferences page performOK() method

```
...
   preferencesStore.setValue(MainPreferencesPage.P_STANDALONE_MODE,
      standaloneButton.getSelection());
   preferencesStore.setValue(MainPreferencesPage.P_DATABASE_LOCATION,
      databaseLocationText.getText());

   // Notify the DatabaseManager that the database location has changed
   ConfigService service = (ConfigService) ServiceFactory.getInstance()
         .getService(ServiceFactory.CONFIG_SERVICE);

   service.setDatabaseLocation(databaseLocationText.getText());
...
```

The class uses the plug-in's default preferences store to store the preferences that are accessible via the plug-in activator class, `MobileStorePlugin`, as shown in Example 8-15.

Example 8-15 Obtaining the plug-in default preferences store

```
...
public MainPreferencesPage() {
   preferencesStore = MobileStorePlugin.getDefault().getPreferenceStore();
}
...
```

Finally, it is possible to set some default values for the preferences. They are set in the preferences store that the preferences page will use. The default values should be set when the application starts, so a good place to put this code is in the `start()` bundle activator method of MobileStorePlugin. You will review this in 8.6, “The Mobile Store plug-in” on page 212.

Follow these steps to create the `MainPreferencesPage` class:

1. Select **File →New → Class** from the menu bar. Enter `com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.preferences` as the package name and `MainPreferencesPage` as the class name. Click **Finish.**
2. Replace the newly generated class with the code in Example 8-16. Save the changes.

Example 8-16 MainPreferencesPage.java

```
package com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.preferences;

import org.eclipse.jface.preference.IPreferenceStore;
import org.eclipse.jface.preference.PreferencePage;
import org.eclipse.swt.SWT;
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Button;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Control;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.widgets.Text;

import com.ibm.itsoral.ercpcasestudy.mobilestore.MobileStorePlugin;
import com.ibm.itsoral.ercpcasestudy.mobilestore.services.ConfigService;
import com.ibm.itsoral.ercpcasestudy.mobilestore.services.ServiceFactory;

/**
 * Preferences page class for ITSO Mobile Store. It holds two properties: the
 * local database location and if the application should work in standalone
 * mode. The class uses the plugin's default preferences store to store the
 * preferences, that it's accesible via the plugin activator class,
 * MobileStorePlugin.
 *
 */
public class MainPreferencesPage extends PreferencePage {

   public static final String P_STANDALONE_MODE = "P_STANDALONE_MODE";

   public static final String P_DATABASE_LOCATION = "P_DATABASE_LOCATION";

   private IPreferenceStore preferencesStore;

   private Button standaloneButton;

   private Text databaseLocationText;

   /**
    * Default constructor. Sets the plugin's default preferences store as the
    * applicartion preferences store.
    *
    */
   public MainPreferencesPage() {

      preferencesStore = MobileStorePlugin.getDefault().getPreferenceStore();
   }

   /*
```

```
	 * (non-Javadoc)
	 *
	 * @see org.eclipse.jface.preference.PreferencePage#createContents(
	 * org.eclipse.swt.widgets.Composite)
	 */
	protected Control createContents(Composite parent) {

		Composite composite = new Composite(parent, SWT.NONE);
		composite.setLayout(new GridLayout(2, true));

		new Label(composite, SWT.LEFT).setText("Standalone Mode: ");
		standaloneButton = new Button(composite, SWT.CHECK);
		standaloneButton.setLayoutData(new GridData(GridData.FILL_HORIZONTAL));
		standaloneButton.setSelection(preferencesStore
				.getBoolean(MainPreferencesPage.P_STANDALONE_MODE));

		new Label(composite, SWT.LEFT).setText("Database Location: ");
		databaseLocationText = new Text(composite, SWT.LEFT | SWT.BORDER);
		databaseLocationText.setLayoutData(new GridData(
				GridData.FILL_HORIZONTAL));
		databaseLocationText.setText(preferencesStore
				.getString(MainPreferencesPage.P_DATABASE_LOCATION));

		return composite;
	}

	/*
	 * (non-Javadoc)
	 *
	 * @see org.eclipse.jface.preference.IPreferencePage#performOk()
	 */
	public boolean performOk() {

		preferencesStore.setValue(MainPreferencesPage.P_STANDALONE_MODE,
				standaloneButton.getSelection());
		preferencesStore.setValue(MainPreferencesPage.P_DATABASE_LOCATION,
				databaseLocationText.getText());

		// Notify the DatabaseManager that the database location has changed
		ConfigService service = (ConfigService) ServiceFactory.getInstance()
				.getService(ServiceFactory.CONFIG_SERVICE);

		service.setDatabaseLocation(databaseLocationText.getText());

		return true;
	}

}
```

8.4.5 The views package

You have reached one of the main sections of this chapter, where the user interface for the application is built. The views package contains the eWorkbench `ViewPart` classes that are the main windows for the application. However, before we discuss the inner-workings of these classes, we will review how they have been organized.

Figure 8-24 shows that all the views are children of the `BaseView` class. This section reviews each class in detail.

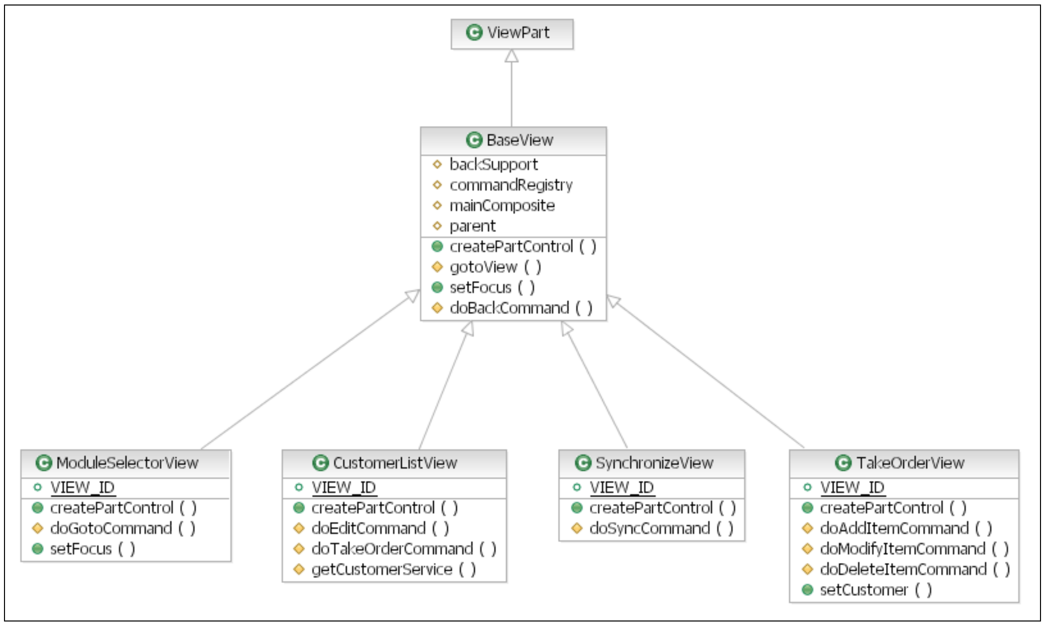


Figure 8-24 The views package

BaseView

This abstract class is the father for all the application views. It inherits from `org.eclipse.ui.part.ViewPart`, allowing that its children can plug into the workbench. The BaseView class provides common functionality that is used in the descendent views as follows:

- Provides a command registry instance per view and a method to access it, called `getCommandRegistry()`. You can find more details about commands and the application `CommandRegistry` class in 8.4.1, “The command package” on page 150.
- Provides a method to switch between views given its view ID, called `gotoView()`. This method also keeps track of the invoker view, so it is easy to return to the previous view.
- Provides a Back command that returns to the previous view. By default, the views add the Back command to their own commands unless you specify the contrary by setting the `supportBack` property to `false`.
- Provides a composite instance to place the view widgets and a method to access it, the `getMainComposite()` method. By default the `setFocus()` method set the focus to the `mainComposite`. This means that any command that you add using `mainComposite` as the associated control is available when the view is shown unless you explicitly override the `setFocus` method in a `BaseView` subclass.
- Provides access to the parent composite for the view that can be used in other methods of the view classes.

Follow these steps to create the `BaseView` class:

1. Select **File** →**New** → **Class** from the menu bar. Enter `com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.views` as the package name and `BaseView` as the class name. Click **Finish**.
2. Replace the newly generated class with the code in Example 8-17. Save the changes.

Example 8-17 BaseView.java

```
package com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.views;

import org.eclipse.ercp.swt.mobile.Command;
import org.eclipse.swt.SWT;
import org.eclipse.swt.events.SelectionEvent;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.ui.PartInitException;
import org.eclipse.ui.part.ViewPart;

import com.ibm.itsoral.ercpcasestudy.mobilestore.ExceptionManager;
import com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.command.CommandRegistry;
import
com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.util.SelectionListenerAdapter;

/**
 * This abstract class is the antecesor for all the application views. It
 * inherits from org.eclipse.ui.part.ViewPart allowing that its children can
 * plug into the workbench.
 *
 */
public abstract class BaseView extends ViewPart {

   private String previousViewId;

   private Composite parent;

   private CommandRegistry commandRegistry;

   private boolean backSupport;

   private Composite mainComposite;

   /**
    * Default constructor. By default the views will have a Back command.
    *
    */
   public BaseView() {

      backSupport = true;
   }

   /*
    * (non-Javadoc)
    *
    * @see
org.eclipse.ui.IWorkbenchPart#createPartControl(org.eclipse.swt.widgets.Composite)
    */
   public void createPartControl(Composite parent) {

      this.parent = parent;

      mainComposite = new Composite(parent, SWT.NONE);

      commandRegistry = new CommandRegistry();

      if (backSupport) {

         commandRegistry.registerCommand(mainComposite, Command.SELECT, 1,
               "Back", "Back", new SelectionListenerAdapter() {
                public void widgetSelected(SelectionEvent arg0) {

                    doBackCommand();
                }
            });
    }
}

/**
 * Goes back to the previous view.
 *
 */
protected void doBackCommand() {

    gotoView(this.previousViewId, false);

}

/**
 * Goes to a view given its id. The target view will remember its previous
 * visited view.
 *
 * @param viewId
 *            The view id
 * @return The target view
 */
protected ViewPart gotoView(String viewId) {

    return gotoView(viewId, true);
}

/**
 * Goes to a view given its id. It takes another parameter to indicate if
 * the new view should remember the previous visited view.
 *
 * @param viewId The view id
 * @param rememberPreviousView Remember the previous view?
 * @return The target view
 */
private ViewPart gotoView(String viewId, boolean rememberPreviousView) {

    BaseView view = null;

    try {

        view = (BaseView) getSite().getWorkbenchWindow().getActivePage()
                .showView(viewId);
        if (rememberPreviousView) {
            view.previousViewId = getConfigurationElement().getAttribute(
                    "id");
            System.out.println("Setting " + view.previousViewId
                    + " as the previous view of " + viewId);
        }

    } catch (PartInitException e) {

        ExceptionManager.getInstance().handleException(this, e,
                getParent().getShell(), "The view cannot be shown");
```

```
      }

      return view;
   }

   /**
    * Sets the focus to the main composite
    */
   public void setFocus() {

      getMainComposite().setFocus();
   }

   /**
    * Returns the parent composite for this view. Useful when you need to
    * access the parent from anonymous inner classes, i.e. event handlers for
    * widgets and so on
    *
    * @return The parent composite
    */
   protected Composite getParent() {
      return parent;
   }

   /**
    * @return The command registry for this view
    */
   protected CommandRegistry getCommandRegistry() {
      return commandRegistry;
   }

   /**
    * @return The main composite for this view
    */
   protected Composite getMainComposite() {
      return mainComposite;
   }

   /**
    * Sets if the view will have a Back command
    *
    * @param backSupport
    *            The backSupport to set.
    */
   protected void setBackSupport(boolean backSupport) {
      this.backSupport = backSupport;
   }
}
```

ModuleSelectorView

This view is the first window that you see when the application is started. It allows you to select a particular module to work with. Each module has its own starting view. For the ITSO Mobile Store sample scenario, there are two modules:

- The Mobile Store module that allows you to manage customers and orders.
- The Synchronize module, that allows you to sync the local database with a remote one. For the scenario the synchronization is simulated by actually recreating the entire local database.

Figure 8-25 shows a sketch with the desired appearance for the view.

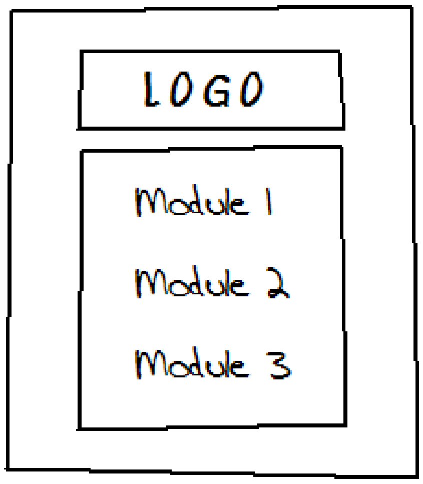


Figure 8-25 ModuleSelectorView sketch

There are two main objects in this view. In the upper side, we want to put the company logo. Below the logo, we want to show some kind of selector for the modules available. If you have reviewed the eSWT widgets available in Chapter 2, "eSWT fundamentals" on page 9, you understand that the logo should be implemented as a `Label` and the module selector could be implemented as a `List`. The layout could be a `GridLayout` with just one column. All the widgets will be placed in the main composite that the view inherits from `BaseView`.

In Example 8-18, the view extends from the `BaseView` class. The view ID is a constant that must be unique, so the fully qualified class name is generally used. Also, this ID is used to declare the view in the plugin.xml descriptor. We have decided to use a list, but instead of using the SWT List directly, we use the JFace List Viewer. (See Example 8-21 on page 175 to see how to set up the model for this viewer.) Information about the start views for each modules are kept in the `modulesMap` property. As decided previously, the logo is implemented as a SWT `Label`. Because this view is the first one for the application, it must set the `backSupport` property to false.

Example 8-18 ModuleSelectorView - Snippet 1

```
....
public class ModuleSelectorView extends BaseView {

   public static final String VIEW_ID =
"com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.views.ModuleSelectorView";

   private ListViewer modulesViewer;
   private Map modulesMap;
   private Label logoLabel;

......
   public ModuleSelectorView() {

      // The only view that doesn't require/support back
      setBackSupport(false);
      initModulesMap();
   }
....
```

The `initModulesMap()` method shown in Example 8-19 initializes the map with the starting views for each module. This map is the model object for the `ListViewer`.

Example 8-19 ModuleSelectorView - Snippet 2

```
...
private void initModulesMap() {

      modulesMap = new HashMap();
      modulesMap.put("Mobile Store", CustomerListView.VIEW_ID);
      modulesMap.put("Synchronization", SynchronizeView.VIEW_ID);
   }
...
```

The widgets that compose the view user interface are created in the `createPartControl()` method. First, you have to call the parent implementation. Then, you have to set the layout for the main composite. Finally, you have to create the controls, as shown in Example 8-20. Note how we use the `PresentationHelper` class to set the image and how we use layout data to fine-tune the control position.

Example 8-20 ModuleSelectorView - Snippet 3

```
...
public void createPartControl(Composite parent) {

      super.createPartControl(parent);
      getMainComposite().setLayout(new GridLayout());

      logoLabel = new Label(getMainComposite(), SWT.NONE);
      logoLabel.setImage(PresentationHelper.getInstance().getImage(
            getParent().getDisplay(), PresentationHelper.LOGO_IMAGE));
      logoLabel.setLayoutData(new GridData(GridData.FILL_HORIZONTAL));
      ((GridData) logoLabel.getLayoutData()).horizontalAlignment = GridData.CENTER;
...
```

After we create the label control, we create the `ListViewer`, as shown in Example 8-21. Note that the input for the viewer is the modules map. Then, the input object, the map in this case, is passed to the content provider's `getElements()` method, that in turn returns an Object array, representing the list elements. Finally, when the list is displayed, each element in the Object array is passed to the label provider's `getText()` method, that return the text to be displayed for that element in the SWT List widget that is associated to the viewer.

Each time an element in the List is selected, the `gotoCommand()` is called. Note that we have used the adapter classes that you saw in 8.4.2, "The util package" on page 155 to make the code clear by avoiding spurious do-nothing methods.

Example 8-21 ModuleSelectorView - Snippet 4

```
...
   modulesViewer = new ListViewer(getMainComposite(), SWT.SINGLE
            | SWT.BORDER);
      modulesViewer.getList().setLayoutData(new GridData(GridData.FILL_BOTH));

      modulesViewer
            .setContentProvider(new StructuredContentProviderAdapter() {

               public Object[] getElements(Object input) {

                  return ((Map) input).keySet().toArray();
            }
         });

      modulesViewer.setLabelProvider(new LabelProvider() {

         public String getText(Object rowElement) {

            return (String) rowElement;
         }
      });

      modulesViewer.setInput(modulesMap);

      modulesViewer.getList().addSelectionListener(
            new SelectionListenerAdapter() {

               public void widgetSelected(SelectionEvent arg0) {

                  doGotoCommand();
               }
            });
...
```

Additionally, we registered a command, associated to the List widget, to perform the go to functionality, as shown in Example 8-22. It is a bit redundant because the selection triggers the `gotoCommand()` also, but we decided to put it here to show how to add commands to a view.

Note: You must always call the CommandRegistry `createCommands()` method at the end of `createPartControl()` to create the command objects. Also, you must call the layout method to draw all the objects in the screen.

Example 8-22 ModuleSelectorView - Snippet 5

```
...
   getCommandRegistry().registerCommand(modulesViewer.getList(),
         Command.SELECT, 1, "Go to", "Go to",
         new SelectionListenerAdapter() {

            public void widgetSelected(SelectionEvent event) {

                  doGotoCommand();
            }
         });

   getCommandRegistry().createCommands();

   parent.layout();
...
```

The `gotoCommand()` method takes the current selection from the `ListViewer`, obtains the view ID, and then use the inherited `gotoView()` method to switch to the selected view, as shown in Example 8-23. The `setFocus()` method sets the focus to the module selector list.

Example 8-23 ModuleSelectorView - Snippet 6

```
...
protected void doGotoCommand() {

      String viewId = (String) modulesMap
            .get(((StructuredSelection) modulesViewer.getSelection())
                  .getFirstElement());

      if (viewId == null) {

         MessageDialog.openInfo(getParent().getShell(), "Info",
               "This option is not available yet");
         return;
      }

      gotoView(viewId);
   }

   public void setFocus() {
      modulesViewer.getList().setFocus();
   }
...
```

Follow these steps to create the `ModuleSelectorView` class and to register it with eWorkbench:

1. Select **File** → **New** → **Class** from the menu bar. Enter `com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.views` as the package name and `ModuleSelectorView` as the class name. Click **Finish**.
2. Replace the newly generated class with the code in Example 8-24. Save the changes.

Example 8-24 ModuleSelectorView.java

```
package com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.views;

import java.util.HashMap;
import java.util.Map;

import org.eclipse.ercp.swt.mobile.Command;
import org.eclipse.jface.viewers.LabelProvider;
import org.eclipse.jface.viewers.ListViewer;
import org.eclipse.jface.viewers.StructuredSelection;
import org.eclipse.swt.SWT;
import org.eclipse.swt.events.SelectionEvent;
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Label;

import com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.util.MessageDialog;
import
com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.util.SelectionListenerAdapter;
import
com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.util.StructuredContentProviderAdapte
r;
```

```
import com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.util.PresentationHelper;

/**
 * Allows you to select a particular module to work with. Each module will have
 * its own starting view.
 *
 */
public class ModuleSelectorView extends BaseView {

   public static final String VIEW_ID =
"com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.views.ModuleSelectorView";

   private ListViewer modulesViewer;

   private Map modulesMap;

   private Label logoLabel;

   /**
    * Default constructor. Sets the backSupport to false and initialize the
    * modules map.
    *
    */
   public ModuleSelectorView() {

      // The only view that doesn't require/support back
      setBackSupport(false);
      initModulesMap();
   }

   /**
    * Initialize the modules map with the starting views for each module.
    *
    */
   private void initModulesMap() {

      modulesMap = new HashMap();
      modulesMap.put("Mobile Store", CustomerListView.VIEW_ID);
      modulesMap.put("Synchronization", SynchronizeView.VIEW_ID);
   }

   /*
    * (non-Javadoc)
    *
    * @see org.eclipse.ui.IWorkbenchPart#createPartControl(
      org.eclipse.swt.widgets.Composite)
    */
   public void createPartControl(Composite parent) {

      super.createPartControl(parent);
      getMainComposite().setLayout(new GridLayout());

      logoLabel = new Label(getMainComposite(), SWT.NONE);
      logoLabel.setImage(PresentationHelper.getInstance().getImage(
            getParent().getDisplay(), PresentationHelper.LOGO_IMAGE));
      logoLabel.setLayoutData(new GridData(GridData.FILL_HORIZONTAL));
      ((GridData) logoLabel.getLayoutData()).horizontalAlignment = GridData.CENTER;

      modulesViewer = new ListViewer(getMainComposite(), SWT.SINGLE
            | SWT.BORDER);
   modulesViewer.getList().setLayoutData(new GridData(GridData.FILL_BOTH));

   modulesViewer
         .setContentProvider(new StructuredContentProviderAdapter() {

            public Object[] getElements(Object input) {

               return ((Map) input).keySet().toArray();
            }

         });

   modulesViewer.setLabelProvider(new LabelProvider() {

      public String getText(Object rowElement) {

         return (String) rowElement;
      }
   });

   modulesViewer.setInput(modulesMap);

   modulesViewer.getList().addSelectionListener(
         new SelectionListenerAdapter() {

            public void widgetSelected(SelectionEvent arg0) {

               doGotoCommand();
            }
         });

   getCommandRegistry().registerCommand(modulesViewer.getList(),
         Command.SELECT, 1, "Go to", "Go to",
         new SelectionListenerAdapter() {

            public void widgetSelected(SelectionEvent event) {

               doGotoCommand();
            }
         });

   getCommandRegistry().createCommands();

   parent.layout();
}

/**
 * Switchs to the selected module
 *
 */
protected void doGotoCommand() {

   String viewId = (String) modulesMap
         .get(((StructuredSelection) modulesViewer.getSelection())
               .getFirstElement());

   if (viewId == null) {

      MessageDialog.openInfo(getParent().getShell(), "Info",
            "This option is not available yet");
            return;
        }

        gotoView(viewId);
    }

    /**
     * Sets the focus to the List widget
     */
    public void setFocus() {
        modulesViewer.getList().setFocus();
    }
}
```

Note: You will notice some compilation errors in the code provided in Example 8-24. These errors are because the code is referencing classes that you will add later. Ignore the compilation errors until all the application code is in place.

3. To register the view with eWorkbench, add the extension that is shown in Example 8-25 to the plugin.xml file.

Example 8-25 ModuleSelectorView view extension

```
<extension point="org.eclipse.ui.views">
    <view
        allowMultiple="false"
        category="org.eclipse.ercp.eworkbench.viewCategory"
    class="com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.views.ModuleSelectorView"
        icon="icons/sample.gif"
        id="com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.views.ModuleSelectorView"
        name="Module Selector"/>
</extension>
```

SynchronizeView

The synchronize view allows you to create and to populate the local database. It takes the data from a remote database using the DB2 Everyplace sync capabilities. For this ITSO Mobile Store version, the database is created and populated using local resources. Figure 8-26 presents a sketch for this view.

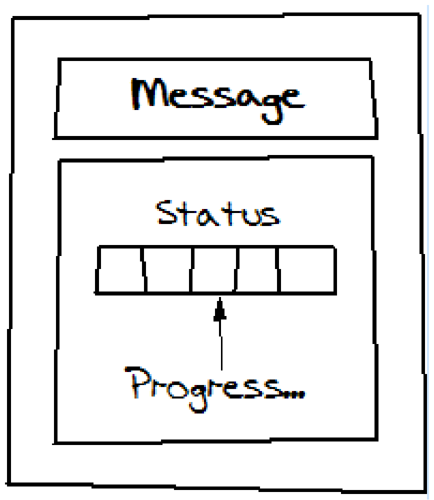


Figure 8-26 SynchronizeView sketch

For the message, divisor line, and status, you can use a SWT `Label` widget. For the progress bar, we used the SWT `ProgressBar` widget, explained in 3.1.11, “ProgressBar” on page 47. We used a `GridLayout` to order the widgets in the screen.

The class inherits from `BaseView` and implements the `createPartControl()` method to create each widget in the main composite. It calls to the `createCommands()` and `layout()` methods at the end of the `createPartControl()` method.

Example 8-26 shows how the progress bar widgets is created. We used the `SWT.INDETERMINATE` style to indicate that the progress bar should be always show progress.

Example 8-26 SynchronizeView - Snippet 1

```
...
   progressBar = new ProgressBar(getMainComposite(), SWT.INDETERMINATE
            | SWT.HORIZONTAL);
   progressBar.setVisible(false);
...
```

When the user selects the **Sync** command, the `doSyncCommand()` method is called. This method starts a worker thread to perform the sync to avoid freezing the user interface thread, as shown in Example 8-27.

Note: You must use the `Display.asyncExec()` method inside the worker thread when access the user interface widgets to avoid concurrent access problems.

Example 8-27 SynchronizeView - Snippet 2

```
protected void doSyncCommand() {
...
   new Thread() {

         public void run() {

            getParent().getDisplay().asyncExec(new Runnable() {

               public void run() {

                  statusLabel.setVisible(true);
                  progressBar.setVisible(true);
               }
            });

...
            boolean syncStatus = false;
            try {
               syncService.sync(syncProperties);
               syncStatus = true;

            } catch (RuntimeException re) {

               ExceptionManager.getInstance().handleException(this, re);
            }

            final boolean success = syncStatus;
            getParent().getDisplay().asyncExec(new Runnable() {

               public void run() {
                  statusLabel.setVisible(false);
                  progressBar.setVisible(false);

                  ...
```

```
                }
            });
        }

    }.start();
...
```

Follow these steps to create the `SynchronizeView` class and to register the view with eWorkbench:

1. Select **File** → **New** → **Class** from the menu bar. Enter `com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.views` as the package name and `SynchronizeView` as the class name. Click **Finish**.
2. Replace the newly generated class with the code in Example 8-28. Save the changes.

Example 8-28 SynchronizeView.java

```
package com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.views;

import java.util.Properties;

import org.eclipse.ercp.swt.mobile.Command;
import org.eclipse.swt.SWT;
import org.eclipse.swt.events.SelectionEvent;
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.widgets.ProgressBar;

import com.ibm.itsoral.ercpcasestudy.mobilestore.ExceptionManager;
import com.ibm.itsoral.ercpcasestudy.mobilestore.MobileStorePlugin;
import
com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.preferences.MainPreferencesPage;
import com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.util.MessageDialog;
import
com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.util.SelectionListenerAdapter;
import com.ibm.itsoral.ercpcasestudy.mobilestore.services.ConfigService;
import com.ibm.itsoral.ercpcasestudy.mobilestore.services.ServiceFactory;
import com.ibm.itsoral.ercpcasestudy.mobilestore.services.SyncService;

/**
 * Creates and populates the local database.
 *
 */
public class SynchronizeView extends BaseView {

   public static final String VIEW_ID =
"com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.views.SynchronizeView";

   private Label titleLabel;

   private Label statusLabel;

   private ProgressBar progressBar;

   /*
    * (non-Javadoc)
    *
```

```
     * @see
org.eclipse.ui.IWorkbenchPart#createPartControl(org.eclipse.swt.widgets.Composite)
     */
    public void createPartControl(Composite parent) {

        super.createPartControl(parent);

        getMainComposite().setLayout(new GridLayout());
        GridData gridData = null;

        titleLabel = new Label(getMainComposite(), SWT.WRAP | SWT.LEFT);
        titleLabel
                .setText("To synchronize select Command > Sync from the menu...");
        gridData = new GridData();
        gridData.horizontalAlignment = SWT.FILL;
        gridData.widthHint = getMainComposite().getSize().x;
        titleLabel.setLayoutData(gridData);

        gridData = new GridData();
        gridData.horizontalAlignment = SWT.FILL;
        new Label(getMainComposite(), SWT.SEPARATOR | SWT.HORIZONTAL
                | SWT.SHADOW_OUT).setLayoutData(gridData);

        gridData = new GridData();
        gridData.horizontalAlignment = SWT.FILL;
        new Label(getMainComposite(), SWT.NONE).setLayoutData(gridData);

        statusLabel = new Label(getMainComposite(), SWT.CENTER);
        statusLabel.setText("Synchronizing...");
        statusLabel.setVisible(false);

        gridData = new GridData();
        gridData.horizontalAlignment = SWT.CENTER;
        statusLabel.setLayoutData(gridData);

        progressBar = new ProgressBar(getMainComposite(), SWT.INDETERMINATE
                | SWT.HORIZONTAL);
        progressBar.setVisible(false);
        gridData = new GridData();
        gridData.horizontalAlignment = SWT.FILL;
        progressBar.setLayoutData(gridData);

        getCommandRegistry().registerCommand(getMainComposite(),
                Command.SELECT, 1, "Sync", "Sync",
                new SelectionListenerAdapter() {

                    public void widgetSelected(SelectionEvent event) {
                        doSyncCommand();
                    }

                });

        getCommandRegistry().createCommands();

    }

    /**
     * Executes the synchronization
     */
    protected void doSyncCommand() {

        boolean standaloneMode = MobileStorePlugin.getDefault()
                .getPreferenceStore().getBoolean(
                        MainPreferencesPage.P_STANDALONE_MODE);

        if (!standaloneMode) {
            MessageDialog.openInfo(getParent().getShell(), "Info",
                  "DB2eSync is not yet implemented");
            return;
         }

         new Thread() {

            public void run() {

               getParent().getDisplay().asyncExec(new Runnable() {

                  public void run() {
                     statusLabel.setVisible(true);
                     progressBar.setVisible(true);
                  }
               });

               SyncService syncService = ((SyncService) ServiceFactory
                     .getInstance().getService(
                           ServiceFactory.SYNC_SERVICE_STANDALONE));

               ConfigService configService = (ConfigService) ServiceFactory
                     .getInstance()
                     .getService(ServiceFactory.CONFIG_SERVICE);

               Properties syncProperties = new Properties();
               syncProperties.setProperty(SyncService.SYNC_DBLOCATION,
                     configService.getDatabaseLocation());

               boolean syncStatus = false;
               try {
                  syncService.sync(syncProperties);
                  syncStatus = true;

               } catch (RuntimeException re) {

                  ExceptionManager.getInstance().handleException(this, re);
               }

               final boolean success = syncStatus;
               getParent().getDisplay().asyncExec(new Runnable() {

                  public void run() {

                     statusLabel.setVisible(false);
                     progressBar.setVisible(false);

                     if (success)
                        MessageDialog.openInfo(getParent().getShell(),
                              "Info", "Sync successfully completed");
                     else
                        MessageDialog.openError(getParent().getShell(),
                              "Error", "Sync failed");

                  }
               });
            }

         }.start();
      }
   }
```

3. To register the view with eWorkbench, add the extension shown in Example 8-29 to the plugin.xml file. Add the view entry to the extension `org.eclipse.ui.views` that you added in Example 8-26 on page 181.

Example 8-29 SynchronizeView extension

```
<view
   allowMultiple="false"
   category="org.eclipse.ercp.eworkbench.viewCategory"
   class="com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.views.SynchronizeView"
   icon="icons/sample.gif"
   id="com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.views.SynchronizeView"
   name="Synchronize"/>
```

CustomerListView

This view presents a list of customers for whom the user can place orders. The list includes the customer's photo and name for identification. If the user selects a customer, the application allows the user to edit customer information such as the address in a separate window. It also allows the user to start an order for the selected customer. Figure 8-27 presents a sketch for this view.

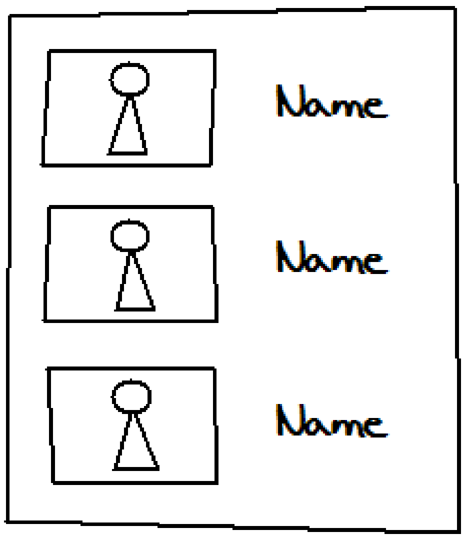


Figure 8-27 CustomerListView sketch

Given that we need to include two columns of information — the customer's photo and name — we decided to use an SWT `Table` widget. The table fills the entire application screen. The view has two commands available: one to edit the customer information and one to take an order for the selected customer.

As the others views, the `CustomerListView` inherits from `BaseView`, has a view ID, and implements the `createPartControl()` method. In Example 8-30, the `Table` widgets are not used directly. Instead, the eJFace `TableViewer` class is used. After you create the table viewer, you need to create the table columns and set its properties, such as title, size, alignment, and so on.

Example 8-30 CustomerListView - Snippet 1

```
...
   tableViewer = new TableViewer(getMainComposite(), SWT.BORDER
            | SWT.FULL_SELECTION | SWT.SINGLE);

   tableViewer.getTable().setLayoutData(new GridData(GridData.FILL_BOTH));

   new TableColumn(tableViewer.getTable(), SWT.CENTER);
   new TableColumn(tableViewer.getTable(), SWT.CENTER);

   tableViewer.getTable().getColumn(0).setWidth(100);
   tableViewer.getTable().getColumn(0).setAlignment(SWT.CENTER);
   tableViewer.getTable().getColumn(1).setWidth(100);
   tableViewer.getTable().getColumn(1).setAlignment(SWT.CENTER);
...
```

As its counterpart for SWT `Lists`, the `TableViewer` needs a content provider, a label provider, and an input object. The input object for the customer viewer is the `customerService` service object. Another interesting thing is that the label provider can take images. This feature is used to show the customer photo, as shown in Example 8-31.

Example 8-31 CustomerListView - Snippet 2

```
      tableViewer.setContentProvider(new StructuredContentProviderAdapter() {

         public Object[] getElements(Object input) {

            return ((CustomerService) input).findAll().toArray();
         }
      });

      tableViewer.setLabelProvider(new TableLabelProviderAdapter() {

         public Image getColumnImage(Object element, int col) {

            if (element instanceof Customer) {

               Customer customer = (Customer) element;

               if (col == 0) {

                  Image photo = null;
                  if (customer.getPhoto() == null) {
                     photo = PresentationHelper.getInstance().getImage(
                           getParent().getDisplay(),
                           PresentationHelper.NOBODY_IMAGE);

                  } else {
                     photo = PresentationHelper.getInstance().getImage(
                           getParent().getDisplay(),
                           "customer" + customer.getId(),
                           new ByteArrayInputStream(customer
                                 .getPhoto()));
                  }

                  return photo;
```

```
                }
            }

            return null;
        }

        public String getColumnText(Object element, int col) {

        .....

        }
    });

    tableViewer.setInput(getCustomerService());
```

The `doEditCommand()` method, shown in Example 8-32, gets the selected customer from the table viewer. The selection is a Customer object — an object from the model. Then, the customer details dialog is called with information that is provided by the customer object. (You can find information about the `CustomerDetailsDialog` in 8.4.6, "The dialogs package" on page 200.) Finally, the updated customer objects are passed to the service layer to update them in the database.

Example 8-32 CustomerListView - Snippet 3

```
...
  protected void doEditCommand() {
    Customer customer = (Customer) ((IStructuredSelection) tableViewer
         .getSelection()).getFirstElement();
    if (customer == null) {
      MessageDialog.openError(getParent().getShell(), "Error",
           "Please select a customer");
      return;
    }

    CustomerDetailsDialog dialog = new CustomerDetailsDialog(getParent());
    dialog.setCustomerInfo(customer);
    dialog.open();
    getCustomerService().update(dialog.getCustomerInfo());
  }
...
```

The `doTakeOrderCommand()` method takes the selected customer from the viewer and switches to the take order view, as shown in Example 8-33.

Example 8-33 CustomerListView - Snippet 4

```
...
protected void doTakeOrderCommand() {

    Customer customer = (Customer) ((IStructuredSelection) tableViewer
         .getSelection()).getFirstElement();
    if (customer == null) {
      MessageDialog.openError(getParent().getShell(), "Error",
           "Please select a customer");
      return;
    }

    TakeOrderView takeOrderView = (TakeOrderView) gotoView(TakeOrderView.VIEW_ID);
    takeOrderView.setCustomer(customer);
  }
...
```

Follow these steps to create the `CustomerListView` class and to register the view with the eWorkbench:

1. Select **File** → **New** → **Class** from the menu bar. Enter `com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.views` as the package name and `CustomerListView` as the class name. Click **Finish**.
2. Replace the newly generated class with the code in Example 8-34. Save the changes.

Example 8-34 CustomerListView.java

```
package com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.views;

import java.io.ByteArrayInputStream;

import org.eclipse.ercp.swt.mobile.Command;
import org.eclipse.jface.viewers.IStructuredSelection;
import org.eclipse.jface.viewers.TableViewer;
import org.eclipse.swt.SWT;
import org.eclipse.swt.events.SelectionEvent;
import org.eclipse.swt.graphics.Image;
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.widgets.TableColumn;

import com.ibm.itsoral.ercpcasestudy.mobilestore.domain.Customer;
import
com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.dialogs.CustomerDetailsDialog;
import com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.util.MessageDialog;
import
com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.util.SelectionListenerAdapter;
import
com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.util.StructuredContentProviderAdapte
r;
import
com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.util.TableLabelProviderAdapter;
import com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.util.PresentationHelper;
import com.ibm.itsoral.ercpcasestudy.mobilestore.services.CustomerService;
import com.ibm.itsoral.ercpcasestudy.mobilestore.services.ServiceFactory;

/**
 * This view presents a list of customers for whom the user can place orders.
 * The list includes the customer’s photo and name to identify her. If the user
 * selects a customer, the application allows to edit customer’s information
 * like address and so on, in a separate window; also allows to start an order
 * for the selected customer.
 *
 */
public class CustomerListView extends BaseView {

   public final static String VIEW_ID =
"com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.views.CustomerListView";

   private TableViewer tableViewer;

   private CustomerService customerService;

   /*
    * (non-Javadoc)
```

```
     *
     * @see
org.eclipse.ui.IWorkbenchPart#createPartControl(org.eclipse.swt.widgets.Composite)
     */
    public void createPartControl(Composite parent) {

        super.createPartControl(parent);

        getMainComposite().setLayout(new GridLayout());

        new Label(getMainComposite(), SWT.LEFT)
                .setText("Please select a customer:");

        tableViewer = new TableViewer(getMainComposite(), SWT.BORDER
                | SWT.FULL_SELECTION | SWT.SINGLE);

        tableViewer.getTable().setLayoutData(new GridData(GridData.FILL_BOTH));

        new TableColumn(tableViewer.getTable(), SWT.CENTER);
        new TableColumn(tableViewer.getTable(), SWT.CENTER);

        tableViewer.getTable().getColumn(0).setWidth(100);
        tableViewer.getTable().getColumn(0).setAlignment(SWT.CENTER);
        tableViewer.getTable().getColumn(1).setWidth(100);
        tableViewer.getTable().getColumn(1).setAlignment(SWT.CENTER);

        tableViewer.setContentProvider(new StructuredContentProviderAdapter() {

            public Object[] getElements(Object input) {

                return ((CustomerService) input).findAll().toArray();
            }
        });

        tableViewer.setLabelProvider(new TableLabelProviderAdapter() {

            public Image getColumnImage(Object element, int col) {

                if (element instanceof Customer) {

                    Customer customer = (Customer) element;

                    if (col == 0) {

                        Image photo = null;
                        if (customer.getPhoto() == null) {
                            photo = PresentationHelper.getInstance().getImage(
                                    getParent().getDisplay(),
                                    PresentationHelper.NOBODY_IMAGE);

                        } else {
                            photo = PresentationHelper.getInstance().getImage(
                                    getParent().getDisplay(),
                                    "customer" + customer.getId(),
                                    new ByteArrayInputStream(customer
                                            .getPhoto()));
                        }

                        return photo;
                    }
```

```
         }

         return null;
      }

      public String getColumnText(Object element, int col) {

         if (element instanceof Customer) {
            Customer customer = (Customer) element;
            if (col == 1)
               return customer.getName();

         }

         return null;
      }
   });

   tableViewer.setInput(getCustomerService());

   tableViewer.getTable().addSelectionListener(
         new SelectionListenerAdapter() {

            public void widgetDefaultSelected(SelectionEvent arg0) {

               doTakeOrderCommand();
            }
         });

   getCommandRegistry().registerCommand(tableViewer.getTable(),
         Command.SELECT, 1, "View/Edit", "View/Edit",
         new SelectionListenerAdapter() {

            public void widgetSelected(SelectionEvent arg0) {

               doEditCommand();
            }
         });

   getCommandRegistry().registerCommand(tableViewer.getTable(),
         Command.SELECT, 1, "&Take Order", "&Take Order",
         new SelectionListenerAdapter() {

            public void widgetSelected(SelectionEvent arg0) {

               doTakeOrderCommand();
            }
         });

   getCommandRegistry().createCommands();

   parent.layout();
}

/**
 * Invokes the CustomerDetailsDialog with the selected customer for
 * viewing/editing and updates the database with the newly customer
 * information
 *
 */
```

```
    protected void doEditCommand() {

        Customer customer = (Customer) ((IStructuredSelection) tableViewer
                .getSelection()).getFirstElement();
        if (customer == null) {
            MessageDialog.openError(getParent().getShell(), "Error",
                    "Please select a customer");
            return;
        }

        CustomerDetailsDialog dialog = new CustomerDetailsDialog(getParent());
        dialog.setCustomerInfo(customer);
        dialog.open();
        getCustomerService().update(dialog.getCustomerInfo());

    }

    /**
     * Switchs to the Take Order view with the selected customer
     *
     */
    protected void doTakeOrderCommand() {

        Customer customer = (Customer) ((IStructuredSelection) tableViewer
                .getSelection()).getFirstElement();
        if (customer == null) {
            MessageDialog.openError(getParent().getShell(), "Error",
                    "Please select a customer");
            return;
        }

        TakeOrderView takeOrderView = (TakeOrderView) gotoView(TakeOrderView.VIEW_ID);
        takeOrderView.setCustomer(customer);

    }

    /**
     * @return The CustomerService service
     */
    protected CustomerService getCustomerService() {

        if (customerService == null)
            customerService = (CustomerService) ServiceFactory.getInstance()
                    .getService(ServiceFactory.CUSTOMER_SERVICE);

        return customerService;
    }
}
```

3. To register the view with eWorkbench, add the extension shown in Example 8-35 to the plugin.xml file. Add the view entry to the extension `org.eclipse.ui.views`, that you added in Example 8-26 on page 181.

Example 8-35 CustomerListView view extension

```
<view
   allowMultiple="false"
   category="org.eclipse.ercp.eworkbench.viewCategory"
   class="com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.views.CustomerListView"
   icon="icons/sample.gif"
   id="com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.views.CustomerListView"
   name="Customer List"/>
```

TakeOrderView

The take order view is similar to the common shopping carts that you can find in many Web sites. It allows the user to add, modify, or remove products to the order, automatically calculating the tax amount, total order amount, and so on. Figure 8-28 depicts a sketch of the layout for the take order view.

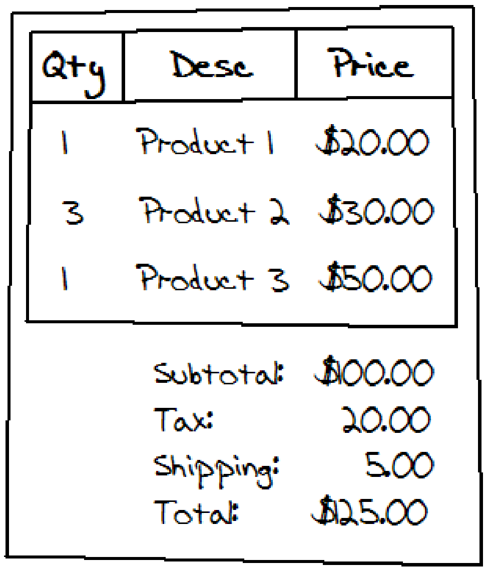


Figure 8-28 TakeOrderView sketch

Each line item holds the product name, the quantity, price, and subtotal. Clearly, a table is appropriate here. For the tax, shipping, subtotal, and total fields, labels are adequate. With more fields than in previous views, the `GridLayout` is the appropriate choice here.

Follow these steps to create the `TakeOrderView` class:

1. Select **File** → **New** → **Class** from the menu bar. Enter `com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.views` as the package name and `TakeOrderView` as the class name. Click **Finish**.
2. Replace the newly generated class with the code in Example 8-36. Save the changes.

Example 8-36 TakeOrderView.java

```
package com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.views;

import org.eclipse.ercp.swt.mobile.Command;
import org.eclipse.ercp.swt.mobile.QueryDialog;
import org.eclipse.jface.viewers.IStructuredSelection;
import org.eclipse.jface.viewers.TableViewer;
import org.eclipse.swt.SWT;
import org.eclipse.swt.events.SelectionEvent;
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.widgets.TableColumn;

import com.ibm.itsoral.ercpcasestudy.mobilestore.domain.Customer;
import com.ibm.itsoral.ercpcasestudy.mobilestore.domain.LineItem;
import com.ibm.itsoral.ercpcasestudy.mobilestore.domain.Order;
import com.ibm.itsoral.ercpcasestudy.mobilestore.domain.Product;
import com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.dialogs.ProductDialog;
import com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.util.MessageDialog;
import
com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.util.SelectionListenerAdapter;
import
com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.util.StructuredContentProviderAdapte
r;
import
com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.util.TableLabelProviderAdapter;
import com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.util.PresentationHelper;
import com.ibm.itsoral.ercpcasestudy.mobilestore.services.OrderService;
import com.ibm.itsoral.ercpcasestudy.mobilestore.services.ServiceFactory;

/**
 * Allows the user to add/modify/delete products from an order and allows to put
 * the order in the database
 *
 */
public class TakeOrderView extends BaseView {

   public static final String VIEW_ID =
"com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.views.TakeOrderView";

   private Order order;

   private TableViewer orderViewer;

   private Label tax;

   private Label shipping;

   private Label total;
   private Label subTotal;

   private Label customerLabel;

   /**
    * Default constructor. Initialize an empty order
    *
    */
   public TakeOrderView() {

      order = new Order();
   }

   /*
    * (non-Javadoc)
    *
    * @see
org.eclipse.ui.IWorkbenchPart#createPartControl(org.eclipse.swt.widgets.Composite)
    */
   public void createPartControl(Composite parent) {
      super.createPartControl(parent);

      getMainComposite().setLayout(new GridLayout(2, false));

      Label label = new Label(getMainComposite(), SWT.LEFT);
      label.setText("Customer: ");
      label.setFont(PresentationHelper.getInstance().getFont(
            getParent().getDisplay(), SWT.BOLD));

      customerLabel = new Label(getMainComposite(), SWT.SINGLE);
      customerLabel.setLayoutData(new GridData(GridData.FILL_HORIZONTAL));

      orderViewer = new TableViewer(getMainComposite(), SWT.BORDER
            | SWT.SINGLE | SWT.FULL_SELECTION);
      orderViewer.getTable().setLayoutData(new GridData(GridData.FILL_BOTH));
      ((GridData) orderViewer.getTable().getLayoutData()).horizontalSpan = 2;

      TableColumn column = new TableColumn(orderViewer.getTable(), SWT.CENTER);
      column.setText("Qt.");
      column.setWidth(30);

      column = new TableColumn(orderViewer.getTable(), SWT.CENTER);
      column.setText("Name");
      column.setWidth(120);

      column = new TableColumn(orderViewer.getTable(), SWT.CENTER);
      column.setText("Price");
      column.setWidth(50);

      column = new TableColumn(orderViewer.getTable(), SWT.CENTER);
      column.setText("Subtotal");
      column.setWidth(70);

      orderViewer.getTable().setHeaderVisible(true);

      orderViewer.setLabelProvider(new TableLabelProviderAdapter() {

         public String getColumnText(Object element, int colIndex) {

            String columnText = "";
            LineItem item = (LineItem) element;
            switch (colIndex) {
            case 0:
               columnText = String.valueOf(item.getQuantity());
               break;
            case 1:
               columnText = String.valueOf(item.getProduct().getName());
               break;
            case 2:
               columnText = PresentationHelper.getInstance().formatMoney(
                     item.getPrice());
               break;
            case 3:
               columnText = PresentationHelper.getInstance().formatMoney(
                     item.getSubtotal());
               break;
            default:
               break;
            }

            return columnText;
         }
      });

      orderViewer.setContentProvider(new StructuredContentProviderAdapter() {

         public Object[] getElements(Object input) {

            return ((Order) input).getLineItemsArray();
         }
      });

      orderViewer.setInput(order);

      orderViewer.getTable().addSelectionListener(
            new SelectionListenerAdapter() {

               public void widgetDefaultSelected(SelectionEvent arg0) {
                  doModifyItemCommand();
               }

            });

      label = new Label(getMainComposite(), SWT.RIGHT);
      label.setText("SubTotal: ");
      label.setLayoutData(new GridData(GridData.HORIZONTAL_ALIGN_END
            | GridData.FILL_HORIZONTAL));

      subTotal = new Label(getMainComposite(), SWT.RIGHT);
      subTotal.setText("$ 0.00");
      subTotal.setLayoutData(new GridData(GridData.HORIZONTAL_ALIGN_END));

      label = new Label(getMainComposite(), SWT.RIGHT);
      label.setText("Tax: ");
      label.setLayoutData(new GridData(GridData.HORIZONTAL_ALIGN_END
            | GridData.FILL_HORIZONTAL));

      tax = new Label(getMainComposite(), SWT.RIGHT);
      tax.setText("$ 0.00");
      tax.setLayoutData(new GridData(GridData.HORIZONTAL_ALIGN_END));
```

```
label = new Label(getMainComposite(), SWT.RIGHT);
label.setText("Shipping: ");
label.setLayoutData(new GridData(GridData.HORIZONTAL_ALIGN_END
      | GridData.FILL_HORIZONTAL));

shipping = new Label(getMainComposite(), SWT.RIGHT);
shipping.setText("$ 0.00");
shipping.setLayoutData(new GridData(GridData.HORIZONTAL_ALIGN_END));

label = new Label(getMainComposite(), SWT.RIGHT);
label.setText("Total: ");
label.setFont(PresentationHelper.getInstance().getFont(
      getParent().getDisplay(), SWT.BOLD));
label.setLayoutData(new GridData(GridData.HORIZONTAL_ALIGN_END
      | GridData.FILL_HORIZONTAL));

total = new Label(getMainComposite(), SWT.RIGHT);
total.setText("$ 0.00");
total.setFont(PresentationHelper.getInstance().getFont(
      getParent().getDisplay(), SWT.BOLD));
total.setLayoutData(new GridData(GridData.HORIZONTAL_ALIGN_END));

getCommandRegistry().registerCommand(orderViewer.getTable(),
      Command.SELECT, 1, "Add item", "Add item",
      new SelectionListenerAdapter() {

         public void widgetSelected(SelectionEvent arg0) {

            doAddItemCommand();
         }

      });

getCommandRegistry().registerCommand(orderViewer.getTable(),
      Command.SELECT, 1, "Modify item", "Modify item",
      new SelectionListenerAdapter() {

         public void widgetSelected(SelectionEvent arg0) {

            doModifyItemCommand();
         }
      });

getCommandRegistry().registerCommand(orderViewer.getTable(),
      Command.SELECT, 1, "Delete item", "Delete item",
      new SelectionListenerAdapter() {

         public void widgetSelected(SelectionEvent arg0) {

            doDeleteItemCommand();
         }
      });

getCommandRegistry().registerCommand(getMainComposite(),
      Command.SELECT, 1, "Place order", "Place order",
      new SelectionListenerAdapter() {

         public void widgetSelected(SelectionEvent arg0) {
                doPlaceOrderCommand();
            }
        });

    getCommandRegistry().registerCommand(getMainComposite(),
        Command.SELECT, 1, "Reset order", "Reset order",
        new SelectionListenerAdapter() {

            public void widgetSelected(SelectionEvent arg0) {

                doResetOrderCommand();
            }
        });

    getCommandRegistry().createCommands();
    parent.layout();
}

/**
 * Reset an order by removing all these elements
 *
 */
protected void doResetOrderCommand() {

    if (order.getLineItemsArray().length == 0) {
        MessageDialog.openError(getParent().getShell(), "Empty order",
            "Your order is empty!");
        return;
    }

    if (MessageDialog.openQuestion(getParent().getShell(),
        "Confirm deletion", "Are you sure?") == SWT.YES) {

        order.removeAllLineItems();
        refresh();
    }

}

/**
 * Puts the order in the database
 *
 */
protected void doPlaceOrderCommand() {

    try {
        if (order.getLineItemsArray().length == 0) {
            MessageDialog.openError(getParent().getShell(), "Empty order",
                "Your order is empty!");
            return;
        }

        long orderID = getOrderService().placeOrder(order);
        MessageDialog.openInfo(getParent().getShell(), "Info",
            "Your order has been successfully placed. Your order ID is "
                + orderID);
        order.removeAllLineItems();
        refresh();
    } catch (RuntimeException e) {
        MessageDialog
```

```
            .openError(getParent().getShell(), "Error",
                  "There's a problem with your order. Please try again later");
   }
}

/**
 * Delete an item from the current order
 *
 */
protected void doDeleteItemCommand() {

   LineItem item = (LineItem) ((IStructuredSelection) orderViewer
         .getSelection()).getFirstElement();
   if (item == null) {
      MessageDialog.openInfo(getParent().getShell(), "Info",
            "Please select an item");
      return;
   }

   order.removeLineItem(item);
   refresh();

}

/**
 * Updates the quantity of the selected item to a values entered by the user
 *
 */
protected void doModifyItemCommand() {

   LineItem item = (LineItem) ((IStructuredSelection) orderViewer
         .getSelection()).getFirstElement();
   if (item == null) {
      MessageDialog.openInfo(getParent().getShell(), "Info",
            "Please select an item");
      return;
   }

   int quantity = Integer.parseInt(MessageDialog.openQuery(getParent()
         .getShell(), QueryDialog.NUMERIC, "Quantity: ", ""
         + item.getQuantity()));

   if (quantity > 0) {

      item.setQuantity(quantity);
   } else {
      order.removeLineItem(item);
   }

   refresh();
}

/**
 * Adds an item to the current order.
 *
 */
protected void doAddItemCommand() {

   ProductDialog dialog = new ProductDialog(getParent());
   dialog.open();
```

```
      Product product = dialog.getSelectedProduct();
      if (product != null) {

         int quantity = Integer.parseInt(MessageDialog.openQuery(getParent()
               .getShell(), QueryDialog.NUMERIC, "Quantity: ", "1"));
         order.addLineItem(product, quantity);
         refresh();
      }
   }

   /**
    * Sets the current customer for the order
    *
    * @param customer
    */
   public void setCustomer(Customer customer) {

      order.setCustomer(customer);

      customerLabel.setText(customer.getName());
      getMainComposite().layout();

   }

   /**
    * @return The order service
    */
   protected OrderService getOrderService() {

      return (OrderService) ServiceFactory.getInstance().getService(
            ServiceFactory.ORDER_SERVICE);
   }

   /**
    * Refresh the window controls
    *
    */
   protected void refresh() {

      // Update listing
      orderViewer.refresh();

      // Update totals
      subTotal.setText(PresentationHelper.getInstance().formatMoney(
            order.getSubTotal()));
      tax.setText(PresentationHelper.getInstance()
            .formatMoney(order.getTax()));
      shipping.setText(PresentationHelper.getInstance().formatMoney(
            order.getShipping()));
      total.setText(PresentationHelper.getInstance().formatMoney(
            order.getTotal()));

      getMainComposite().layout();
   }
}
```

3. To register the view with the eWorkbench, add the extension shown in Example 8-37 to the plugin.xml file. Add the view entry to the extension `org.eclipse.ui.views` that you added in Example 8-26 on page 181.

Example 8-37 TakeOrderView view extension

```
<view
   allowMultiple="false"
   category="org.eclipse.ercp.eworkbench.viewCategory"
   class="com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.views.TakeOrderView"
   icon="icons/sample.gif"
   id="com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.views.TakeOrderView"
   name="Take Order"/>
```

8.4.6 The dialogs package

The dialogs package contain the dialogs that are used by the views reviewed previously. Generally, these dialogs are supportive elements for the views and, when properly designed, can be reused by other applications. Figure 8-29 shows the classes that compose this package.

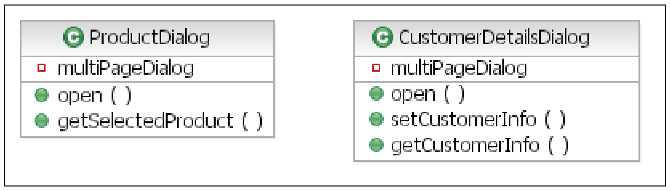


Figure 8-29 The dialogs package

The core element in the dialogs package, is the `MultiPageDialog` class, part of the eSWT Mobile Controls. Briefly, this class is a tabbed dialog that is very appropriate for mobile devices because it saves space in the window. Refer to 4.3.1, “MultiPageDialog” on page 77 for more details about the `MultiPageDialog` class.

ProductDialog

The product dialog allows the user to select an available product. The dialog returns the selected product to the caller. The dialog has two pages: one for the product listing and one for the product detail. Figure 8-30 illustrates a sketch for the dialog.

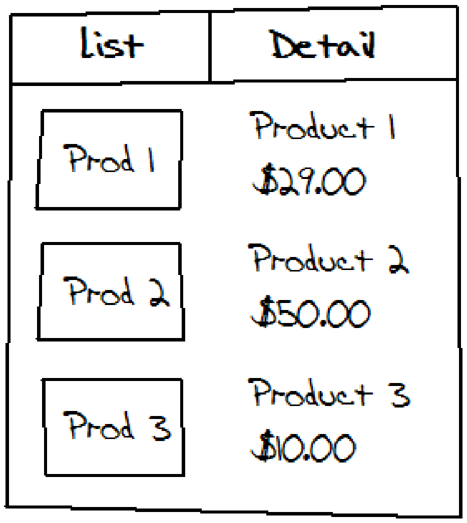


Figure 8-30 ProductDialog sketch

The first tab displays the product photo, name, and price. The second one displays the same information as the first one plus a description.

Follow these steps to create the `ProductDialog` class:

1. Select **File** → **New** → **Class** from the menu bar. Enter `com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.dialogs` as the package name and `ProductDialog` as the class name. Click **Finish.**
2. Replace the newly generated class with the code in Example 8-38. Save the changes.

Example 8-38 ProductDialog.java

```
package com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.dialogs;

import java.io.ByteArrayInputStream;

import org.eclipse.ercp.swt.mobile.ListBox;
import org.eclipse.ercp.swt.mobile.ListBoxItem;
import org.eclipse.ercp.swt.mobile.MultiPageDialog;
import org.eclipse.jface.resource.JFaceResources;
import org.eclipse.swt.SWT;
import org.eclipse.swt.events.SelectionEvent;
import org.eclipse.swt.graphics.Image;
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Label;

import com.ibm.itsoral.ercpcasestudy.mobilestore.domain.Product;
import
com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.util.SelectionListenerAdapter;
import com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.util.PresentationHelper;
import com.ibm.itsoral.ercpcasestudy.mobilestore.services.ProductService;
import com.ibm.itsoral.ercpcasestudy.mobilestore.services.ServiceFactory;
```

```
/**
 * The product dialog allows the user to select an available product. The dialog
 * will return the selected product to the caller. The dialog has two pages,
 * one for the product listing and one for the product detail.
 *
 */
public class ProductDialog {

   private Object[] productArray;

   private Product selectedProduct;

   private Composite parent;

   private MultiPageDialog multiPageDialog;

   private Composite productListPage;

   private Composite productDetailPage;

   private ListBox productListBox;

   private Label productImage;

   private Label productName;

   private Label productDescription;

   private Label productPrice;

   private Label priceLabel;

   /**
    * Constructor for ProductDialog
    *
    * @param parent The parent composite for the associated MultiPageDialog
    */
   public ProductDialog(Composite parent) {

      this.parent = parent;

      initialize();
   }

   /**
    * Initialize the dialog's controls
    *
    */
   private void initialize() {

      multiPageDialog = new MultiPageDialog(parent.getShell());

      productListPage = createProductListPage();
      productDetailPage = createProductDetailPage();

   }

   /**
    * Creates the product list page
 *
 * @return The page composite
 */
private Composite createProductListPage() {

   Composite page = multiPageDialog.createPage("Product List", null);

   page.setLayout(new GridLayout());
   productListBox = new ListBox(page, SWT.SINGLE | SWT.V_SCROLL,
         ListBox.LB_MOD_SHOW_HEADING_ICONS | ListBox.LB_STYLE_2LINE_ITEM);
   productListBox.setLayoutData(new GridData(GridData.FILL_BOTH));
   productListBox.setDataModel(createProductListDataModel());
   productListBox.addSelectionListener(new SelectionListenerAdapter() {

      public void widgetSelected(SelectionEvent e) {

         doSelectProduct();

      }
   });
   new Label(page, SWT.NONE).setLayoutData(new GridData(
         GridData.FILL_HORIZONTAL));

   page.layout();
   return page;
}

/**
 * Updates the product details on the product details page
 */
private void doSelectProduct() {

   selectedProduct = (Product) productArray[productListBox
         .getSelectionIndices()[0]];
   System.out.println("Selected " + selectedProduct.getName());

   productImage.setImage(JFaceResources.getImageRegistry().get(
         "product" + selectedProduct.getId()));
   productName.setText(selectedProduct.getName());
   productDescription.setText(selectedProduct.getDescription());
   productPrice.setText(PresentationHelper.getInstance().formatMoney(
         selectedProduct.getPrice()));
   priceLabel.setVisible(true);
   productDetailPage.layout();
}

/**
 * Creates the data model for the product list control
 *
 * @return The ListBoxItem array
 */
private ListBoxItem[] createProductListDataModel() {

   productArray = getProductService().findAll().toArray();
   ListBoxItem[] dataModel = new ListBoxItem[productArray.length];

   for (int i = 0; i < productArray.length; i++) {

      Product product = (Product) productArray[i];
```

```
        Image productImage = PresentationHelper.getInstance().getImage(
              parent.getDisplay(), "product" + product.getId(),
              new ByteArrayInputStream(product.getPhoto()));
        dataModel[i] = new ListBoxItem("Price: "
              + PresentationHelper.getInstance().formatMoney(
                    product.getPrice()), null, PresentationHelper
              .getInstance().wrapText(product.getName(), 25),
              productImage);
     }

     return dataModel;
  }

  /**
   * Creates the product detail page
   *
   * @return The page composite
   */
  private Composite createProductDetailPage() {

     Composite page = multiPageDialog.createPage("Product Detail", null);
     page.setLayout(new GridLayout(3, true));

     productImage = new Label(page, SWT.CENTER);
     productImage.setLayoutData(new GridData());
     ((GridData) productImage.getLayoutData()).verticalSpan = 3;
     ((GridData) productImage.getLayoutData()).verticalAlignment =
        GridData.VERTICAL_ALIGN_BEGINNING;

     productName = new Label(page, SWT.LEFT | SWT.WRAP);
     productName.setLayoutData(new GridData(GridData.FILL_HORIZONTAL));
     ((GridData) productName.getLayoutData()).horizontalSpan = 2;
     System.out.println(productName.getFont().getFontData()[0].getName());
     productName.setFont(PresentationHelper.getInstance().getFont(
           parent.getDisplay(), SWT.BOLD));

     productDescription = new Label(page, SWT.LEFT | SWT.WRAP);
     productDescription
           .setLayoutData(new GridData(GridData.FILL_HORIZONTAL));
     ((GridData) productDescription.getLayoutData()).horizontalSpan = 2;
     ((GridData) productDescription.getLayoutData()).widthHint = (int) (page
           .getSize().x * 0.75);

     priceLabel = new Label(page, SWT.LEFT);
     priceLabel.setText("Price: ");
     priceLabel.setFont(PresentationHelper.getInstance().getFont(
           parent.getDisplay(), SWT.BOLD | SWT.ITALIC));
     priceLabel.setVisible(false);

     productPrice = new Label(page, SWT.LEFT);
     productPrice.setAlignment(SWT.RIGHT);

     page.layout();
     return page;
  }

  /**
   * @return The ProductService service
   */
  private ProductService getProductService() {
      return (ProductService) ServiceFactory.getInstance().getService(
            ServiceFactory.PRODUCT_SERVICE);
   }

   /**
    * Opens the associated MultiPageDialog for this class.
    *
    */
   public void open() {
      multiPageDialog.open();

   }

   /**
    * @return The selected product
    */
   public Product getSelectedProduct() {

      return selectedProduct;
   }
}
```

CustomerDetailsDialog

This dialog allows the user to view and edit information for a customer. It has three pages: one for personal information, one for a shipping address, and one for a billing address. Figure 8-31 shows a sketch for this dialog.

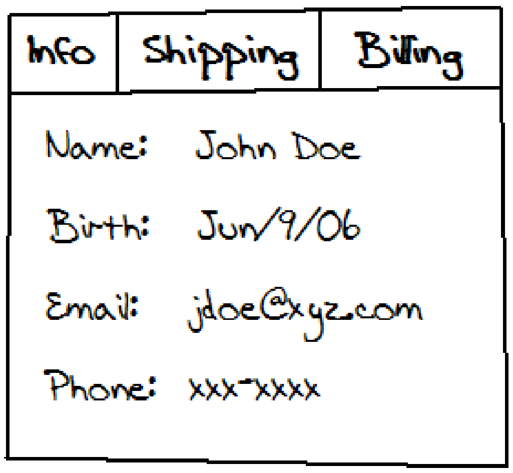


Figure 8-31 CustomerDetailsDialog sketch

Follow these steps to create the `CustomerDetailsDialog` class:

1. Select **File** → **New** → **Class** from the menu bar. Enter `com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.dialogs` as the package name and `CustomerDetailsDialog` as the class name. Click **Finish**.
2. Replace the newly generated class with the code in Example 8-39. Save the changes.

Example 8-39 CustomerDetailsDialog.java

```
package com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.dialogs;

import java.util.Iterator;

import org.eclipse.ercp.swt.mobile.CaptionedControl;
import org.eclipse.ercp.swt.mobile.ConstrainedText;
import org.eclipse.ercp.swt.mobile.DateEditor;
import org.eclipse.ercp.swt.mobile.HyperLink;
import org.eclipse.ercp.swt.mobile.MultiPageDialog;
import org.eclipse.swt.SWT;
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.Combo;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.widgets.Text;

import com.ibm.itsoral.ercpcasestudy.mobilestore.domain.Customer;
import com.ibm.itsoral.ercpcasestudy.mobilestore.services.ConfigService;
import com.ibm.itsoral.ercpcasestudy.mobilestore.services.ServiceFactory;

/**
 * This dialogs allow to view and edit information for a customer. It has three
 * pages, one for editing personal information and the other two for editing the
 * shipping and billing address  respectively.
 *
 */
public class CustomerDetailsDialog {

   private static final int BILLING_ADDRESS = 0;

   private static final int SHIPPING_ADDRESS = 1;

   private Composite parent;

   private Customer customerInfo;

   private MultiPageDialog multiPageDialog;

   private Composite personalInfoPage;

   private Composite shippingAddressPage;

   private Composite billingAddressPage;

   private Label nameLabel;

   private DateEditor birthDateEditor;

   private HyperLink emailLink;
private HyperLink phoneLink;

private Text[] addressTextArray;

private Text[] cityTextArray;

private Combo[] stateComboArray;

private ConstrainedText[] zipTextArray;

private ConstrainedText[] phoneTextArray;

/**
 * Constructor for CustomerDetailsDialog
 *
 * @param parent
 *            The parent composite for the MultiPageDialog
 *
 */
public CustomerDetailsDialog(Composite parent) {

   this.parent = parent;

   initialize();

}

/**
 * Initialize the dialog’s controls
 *
 */
private void initialize() {

   addressTextArray = new Text[2];
   cityTextArray = new Text[2];
   stateComboArray = new Combo[2];
   zipTextArray = new ConstrainedText[2];
   phoneTextArray = new ConstrainedText[2];

   multiPageDialog = new MultiPageDialog(parent.getShell());
   personalInfoPage = createPersonalInfoPage();
   shippingAddressPage = createAddressPage("Shipping Info",
         SHIPPING_ADDRESS);
   billingAddressPage = createAddressPage("Billing Info", BILLING_ADDRESS);
}

/**
 * Creates an customer address page
 *
 * @param title
 *            Title for the page
 * @param type
 *            Page type. Could be SHIPPING_ADDRESS or BILLING_ADDRESS
 * @return The page composite
 */
private Composite createAddressPage(String title, int type) {

   Composite page = multiPageDialog.createPage(title, null);

   page.setLayout(new GridLayout(2, false));
```

```
      new Label(page, SWT.LEFT).setText("Address: ");
      addressTextArray[type] = new Text(page, SWT.BORDER | SWT.MULTI
            | SWT.WRAP);
      addressTextArray[type].setLayoutData(new GridData(
            GridData.FILL_HORIZONTAL));
      ((GridData) addressTextArray[type].getLayoutData()).heightHint = 40;

      new Label(page, SWT.LEFT).setText("City: ");
      cityTextArray[type] = new Text(page, SWT.BORDER | SWT.SINGLE);

      new Label(page, SWT.LEFT).setText("State: ");
      stateComboArray[type] = new Combo(page, SWT.SINGLE | SWT.READ_ONLY);

      for (Iterator it = getConfigService().getParameters(
            ConfigService.STATE_PARAM).iterator(); it.hasNext(); stateComboArray[type]
            .add((String) it.next()))
         ;

      new Label(page, SWT.LEFT).setText("Zip: ");
      zipTextArray[type] = new ConstrainedText(page, SWT.BORDER,
            ConstrainedText.NUMERIC);
      zipTextArray[type].setTextLimit(5);

      new Label(page, SWT.LEFT).setText("Phone: ");
      phoneTextArray[type] = new ConstrainedText(page, SWT.BORDER,
            ConstrainedText.PHONENUMBER);
      phoneTextArray[type].setTextLimit(10);
      page.layout();

      return page;
   }

   /**
    * Creates the page for editing customer personal information
    *
    * @return The page composite
    */
   private Composite createPersonalInfoPage() {

      Composite page = multiPageDialog.createPage("Personal Info", null);

      page.setLayout(new GridLayout());

      CaptionedControl cc = new CaptionedControl(page, SWT.RIGHT_TO_LEFT);
      cc.setText("Name: ");
      nameLabel = new Label(cc, SWT.SINGLE);

      cc = new CaptionedControl(page, SWT.LEFT_TO_RIGHT);
      cc.setText("Birth Date: ");
      birthDateEditor = new DateEditor(cc, DateEditor.DATE);

      cc = new CaptionedControl(page, SWT.LEFT_TO_RIGHT);
      cc.setText("Email: ");
      emailLink = new HyperLink(cc, SWT.BORDER, HyperLink.EMAIL);

      cc = new CaptionedControl(page, SWT.LEFT_TO_RIGHT);
      cc.setText("Phone: ");
      phoneLink = new HyperLink(cc, SWT.BORDER, HyperLink.PHONE);
    page.layout();

    return page;
}

/**
 * Opens the associated MultiPageDialog
 *
 */
public void open() {

    multiPageDialog.open();
}

/**
 * @return The modified customer information
 */
public Customer getCustomerInfo() {

    customerInfo.setBirthDate(birthDateEditor.getDate());

    customerInfo.getBillingAddress().setAddress(
            addressTextArray[BILLING_ADDRESS].getText());
    customerInfo.getBillingAddress().setCity(
            cityTextArray[BILLING_ADDRESS].getText());
    customerInfo.getBillingAddress().setZip(
            zipTextArray[BILLING_ADDRESS].getText());
    customerInfo.getBillingAddress().setPhone(
            phoneTextArray[BILLING_ADDRESS].getText());
    customerInfo.getBillingAddress().setState(
            stateComboArray[BILLING_ADDRESS].getText());

    customerInfo.getShippingAddress().setAddress(
            addressTextArray[SHIPPING_ADDRESS].getText());
    customerInfo.getShippingAddress().setCity(
            cityTextArray[SHIPPING_ADDRESS].getText());
    customerInfo.getShippingAddress().setZip(
            zipTextArray[SHIPPING_ADDRESS].getText());
    customerInfo.getShippingAddress().setPhone(
            phoneTextArray[SHIPPING_ADDRESS].getText());
    customerInfo.getShippingAddress().setState(
            stateComboArray[SHIPPING_ADDRESS].getText());

    return customerInfo;
}

/**
 * Sets the customer information to be modified in the dialog
 *
 * @param customerInfo
 *            The original customer information
 */
public void setCustomerInfo(Customer customerInfo) {

    this.customerInfo = customerInfo;

    nameLabel.setText(customerInfo.getName());
    birthDateEditor.setDate(customerInfo.getBirthDate());
    emailLink.setText(customerInfo.getEmail());
    phoneLink.setText(customerInfo.getPhone());

    addressTextArray[SHIPPING_ADDRESS].setText(customerInfo
            .getShippingAddress().getAddress());
    cityTextArray[SHIPPING_ADDRESS].setText(customerInfo
          .getShippingAddress().getCity());
    stateComboArray[SHIPPING_ADDRESS]
          .select(stateComboArray[SHIPPING_ADDRESS].indexOf(customerInfo
                .getShippingAddress().getState()));
    zipTextArray[SHIPPING_ADDRESS].setText(customerInfo
          .getShippingAddress().getZip());
    phoneTextArray[SHIPPING_ADDRESS].setText(customerInfo
          .getShippingAddress().getPhone());

    addressTextArray[BILLING_ADDRESS].setText(customerInfo
          .getBillingAddress().getAddress());
    cityTextArray[BILLING_ADDRESS].setText(customerInfo.getBillingAddress()
          .getCity());
    stateComboArray[BILLING_ADDRESS]
          .select(stateComboArray[BILLING_ADDRESS].indexOf(customerInfo
                .getBillingAddress().getState()));
    zipTextArray[BILLING_ADDRESS].setText(customerInfo.getBillingAddress()
          .getZip());
    phoneTextArray[BILLING_ADDRESS].setText(customerInfo
          .getBillingAddress().getPhone());

    personalInfoPage.layout();
}

/**
 * @return The ConfigService service
 */
private ConfigService getConfigService() {

    return (ConfigService) ServiceFactory.getInstance().getService(
          ServiceFactory.CONFIG_SERVICE);

}
```

8.5 Managing exceptions

Error handling is a common task in application development and one that you cannot avoid. For ITSO Mobile Store, the exceptions are managed in a centralized way using the `ExceptionManager` class. This class allows the application to handle exceptions simply and consistently.

8.5.1 The ExceptionManager class

The `ExceptionManager` class provides a centralized point to deal with error conditions. For critical exceptions, it provides a method that logs the message and rethrows the exception as a `RuntimeException`. For non-critical exceptions, it provides a method to log the exact exception message and to show a human-readable friendly message to the user. This strategy to manage the exceptions was chosen because of its simplicity.

Follow these steps to create the `ExceptionManager` class:

1. Select **File** → **New** → **Class** from the menu bar. Enter `com.ibm.itsoral.ercpcasestudy.mobilestore` as the package name and `ExceptionManager` as the class name. Click **Finish**.
2. Replace the newly generated class with the code in Example 8-40. Save the changes.

Example 8-40 ExceptionManager.java

```
package com.ibm.itsoral.ercpcasestudy.mobilestore;

import java.sql.SQLException;

import org.eclipse.swt.widgets.Shell;

import com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.util.MessageDialog;

/**
 * Manages the exceptions thrown by the application in a centralized way.
 */
public class ExceptionManager {

   private static ExceptionManager instance;

   /**
    * Constructor
    *
    */
   protected ExceptionManager() {

   }

   /**
    * @return The unique instance for this class
    */
   public static ExceptionManager getInstance() {

      if (instance == null) instance = new ExceptionManager();

      return instance;
   }

   /**
    * Handles an exception rethrowing it as a RuntimException
    *
    * @param source The object that throws the exception
    * @param e The exception thrown
    */
   public void handleException(Object source, Throwable e) {

      handleException(source, e, true);

   }

   /**
    * Handles an exception by log it to the standard ouput.
    *
    * @param source The exception originator
    * @param e The exception thrown
    */
   public void handleException(Object source, Throwable e, boolean rethrowAsRuntime) {
      String errorMessage = e.getMessage();

      if (e instanceof SQLException) {
         SQLException sqle = (SQLException) e;
         errorMessage = "SQLSTATE: "+sqle.getSQLState()+
            ". Error code: "+sqle.getErrorCode();
      }

      System.out.println("Error: "+errorMessage);
      System.out.flush();
      e.printStackTrace();

      if (rethrowAsRuntime)
         throw new RuntimeException(errorMessage);

   }


   /**
    * Handles an exception by log it in the standard output and by showing a dialog
    * with an user-friendly message
    *
    * @param source The exception source
    * @param e The exception thrown
    * @param shell The application shell window
    * @param message The message to show
    */
   public void handleException(Object source, Throwable e, Shell shell, String message) {

      handleException(source, e, false);

      MessageDialog.openError(shell, "Error", message);

   }

}
```

8.6 The Mobile Store plug-in

Now that you have all the application classes in place, you need to modify the `MobileStorePlugin` class to add some initialization and cleanup code. Also, you need to declare that the ITSO Mobile Store application is an eWorkbench application.

8.6.1 The MobileStorePlugin class

In 8.4.4, “The preferences package” on page 165, we discussed some default values for the preferences values that you need to set. The natural place to put this initialization code is in the plug-in activator `start()` method. The plug-in activator class for the application is `MobileStorePlugin`. Also, in “PresentationHelper” on page 158, we discussed that the fonts need to be disposed when the application is closed to free the associated system resources. The natural place to put cleanup code at the application level is in the plug-in activator `stop()` method.

Follow these steps to modify the `MobileStorePlugin` class:

1. Open the generated `MobileStorePlugin` that is located at the `com.ibm.itsoral.ercpcasestudy.mobilestore` package.
2. Replace the existing class code with the code in Example 8-41. Save the changes.

Example 8-41 MobileStorePlugin.java

```
package com.ibm.itsoral.ercpcasestudy.mobilestore;

import org.eclipse.jface.resource.ImageDescriptor;
import org.eclipse.ui.plugin.AbstractUIPlugin;
import org.osgi.framework.BundleContext;

import
com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.preferences.MainPreferencesPage;
import com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.util.PresentationHelper;
import com.ibm.itsoral.ercpcasestudy.mobilestore.services.ConfigService;
import com.ibm.itsoral.ercpcasestudy.mobilestore.services.ServiceFactory;

/**
 * The main plugin class to be used in the desktop.
 */
public class MobileStorePlugin extends AbstractUIPlugin {

   // The shared instance.
   private static MobileStorePlugin plugin;

   /**
    * The constructor.
    */
   public MobileStorePlugin() {
      plugin = this;
   }

   /**
    * This method is called upon plug-in activation
    */
   public void start(BundleContext context) throws Exception {
      super.start(context);

      initApplication();
   }

   /**
    * Initialiaze the application default preferences values.
    * Also sets up the location for the local database
    *
    */
   private void initApplication() {

      try {
         // Setup the default for preferences in case they aren't set by the user
         getPreferenceStore().setDefault(
               MainPreferencesPage.P_DATABASE_LOCATION, "\\sampledb\\");
         getPreferenceStore().setDefault(
               MainPreferencesPage.P_STANDALONE_MODE, true);

         // Set the database location for the application
         ConfigService service = (ConfigService) ServiceFactory
               .getInstance().getService(ServiceFactory.CONFIG_SERVICE);
         service.setDatabaseLocation(getPreferenceStore().getDefaultString(
               MainPreferencesPage.P_DATABASE_LOCATION));

      } catch (Throwable t) {
         ExceptionManager.getInstance().handleException(this, t);
      }
   }

   /**
    * This method is called when the plug-in is stopped
    */
   public void stop(BundleContext context) throws Exception {
      super.stop(context);
      plugin = null;

      PresentationHelper.getInstance().disposeFonts();
   }

   /**
    * Returns the shared instance.
    */
   public static MobileStorePlugin getDefault() {
      return plugin;
   }

   /**
    * Returns an image descriptor for the image file at the given plug-in
    * relative path.
    *
    * @param path
    *           the path
    * @return the image descriptor
    */
   public static ImageDescriptor getImageDescriptor(String path) {
      return AbstractUIPlugin.imageDescriptorFromPlugin(
            "com.ibm.itsoral.ercpcasestudy.mobilestore", path);
   }
}
```

8.6.2 The eWorkbench application extension

Every application that runs in eWorkbench has to declare an extension in the plugin.xml file. Example 8-42 shows the format for the extension.

Example 8-42 eWorkbench application extension

```
   <extension point="org.eclipse.ercp.eworkbench.applications">
      <application id="<APPLICATION_ID>" name="<APPLICATION_NAME>">
         <views normal="<STARTING_VIEW_ID>"/>
      </application>
   </extension>
```

For the ITSO Mobile Store application, Example 8-43 shows the extension. Add this extension to the application plugin.xml file.

Example 8-43 ITSO Mobile Store application extension

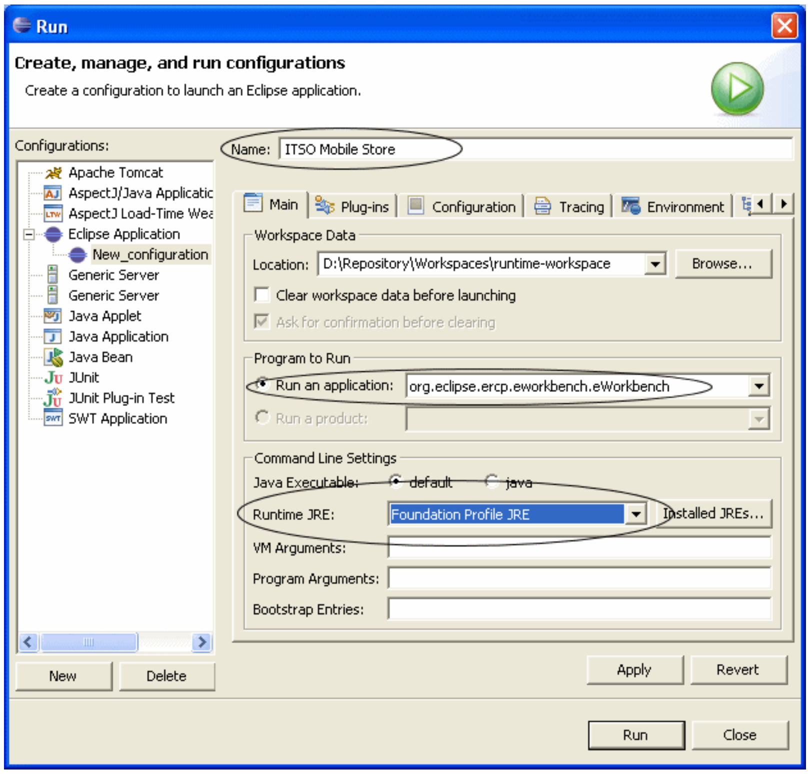
```
<extension point="org.eclipse.ercp.eworkbench.applications">
      <application
            id="com.ibm.itsoral.ercpcasestudy.mobilestore"
            name="ITSO Mobile Store">
         <views
normal="com.ibm.itsoral.ercpcasestudy.mobilestore.presentation.views.ModuleSelectorView"/>
      </application>
   </extension>
```

8.7 Running the ITSO Mobile Store application

Now it is time of some testing. You need to create a new Eclipse Application launch configuration. Make sure that you have followed all the steps to set up the environment as described in 8.1, “Preparing the environment” on page 130.

Follow these steps to create the launch configuration and test the application on Win32:

1. Select **Run** → **Run** from the menu bar. Select **Eclipse Application** and click **New.**
2. Enter ITSO Mobile Store as the configuration name. Note that the `org.eclipse.ercp.eworkbench.eWorkbench` application is selected, as shown in Figure 8-32.



Name: ITSO Mobile Store

Workspace Data
Location: D:\Repository\Workspaces\runtime-workspace
Clear workspace data before launching
Ask for confirmation before clearing

Program to Run
(●) Run an application: org.eclipse.ercp.eworkbench.eWorkbench
( ) Run a product:

Command Line Settings
Java Executable: (●) default ( ) java
Runtime JRE: Foundation Profile JRE
VM Arguments:
Program Arguments:
Bootstrap Entries:

Figure 8-32 Creating the launch configuration

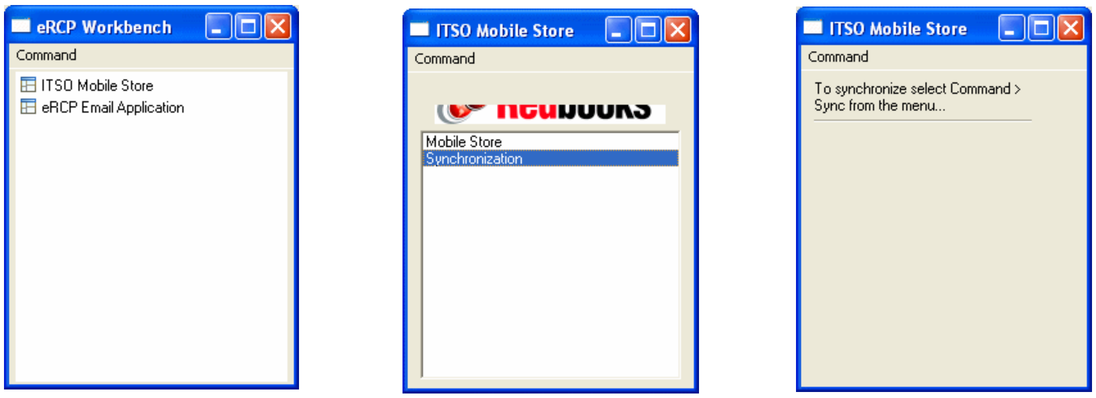
3. Select Foundation profile JRE as the Runtime JRE. Click **Apply** and then **Run**.

The eWorkbench window opens.

4. Select **ITSO Mobile Store**. The ITSO Mobile Store main view (`ModuleSelectorView`) opens.

   You need to create and populate the database first.

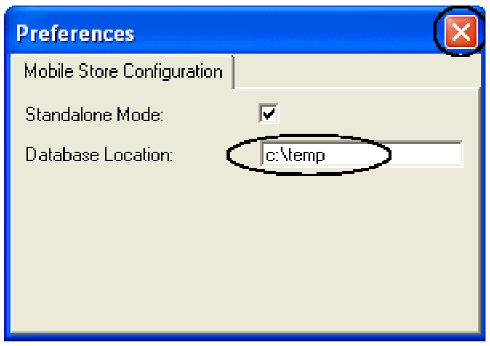
5. Select **Synchronize**. The synchronization window (`SynchronizeView`) opens. Figure 8-33 shows the windows sequence.



To synchronize select Command > Sync from the menu...

Figure 8-33 Testing the application

6. Select **Command** → **Preferences** to see the application preferences (`MainPreferencesPage`). Set a local directory for the database files as shown in Figure 8-34. The directory must exist. Click the *X* located in the upper, left corner to accept the changes and close the dialog.



Standalone Mode: ✓

Database Location: c:\temp

Figure 8-34 Setting the preferences on the desktop

7. Select **Command** → **Sync** from the menu. The database creation begins. At the end of the process, an information message informing the synchronization status displays. Click **OK**. Figure 8-35 shows the windows sequence.

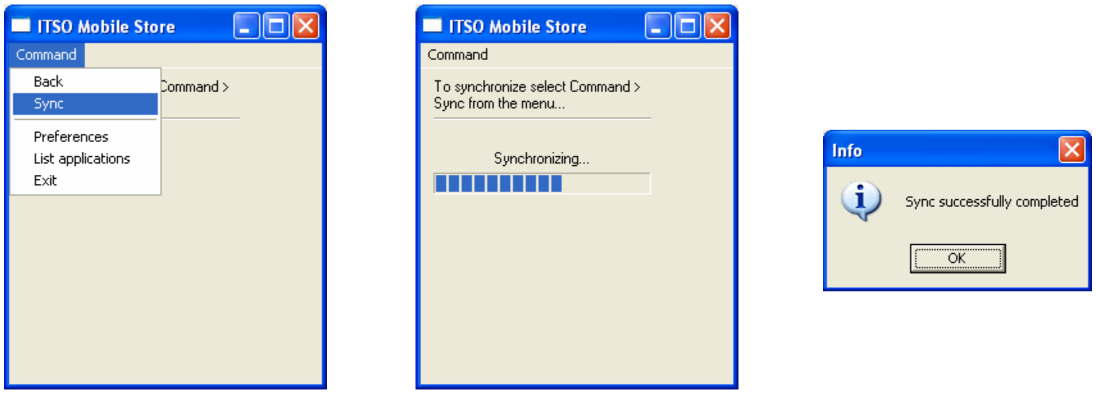


Figure 8-35 Testing the sync capabilities

8. Select **Command** → **Back** to return to the module selector view. Select **Mobile Store**. The `CustomerList` view opens. You can browse the window to the see all the registered customers. Figure 8-36 shows the windows sequence.

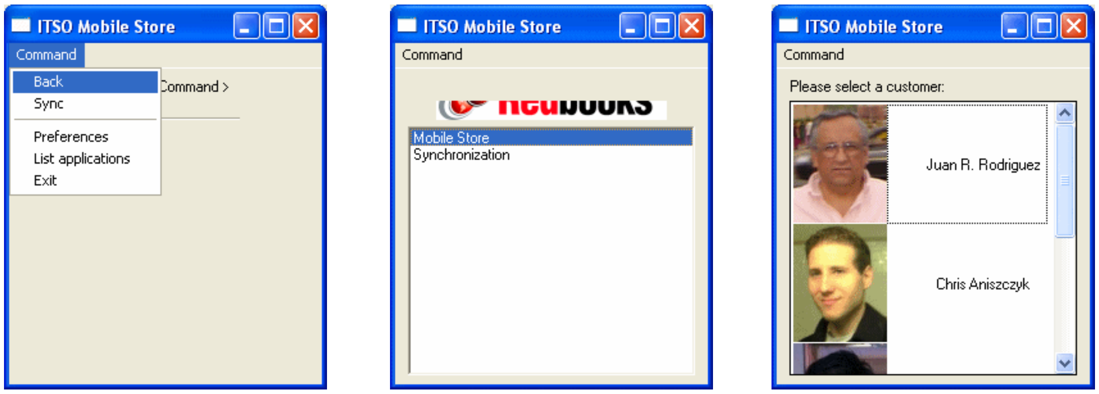


Figure 8-36 Testing the customer list

9. Select a customer from the list and select **Command** → **View / Edit** to open the customer details dialog. The dialog has three pages, or tabs, for personal information, shipping address, and billing address.

10. Select the billing address page. Modify some values. Click the *X* in the upper, left corner to close the dialog. The changes are saved automatically. Figure 8-37 shows the windows sequence.

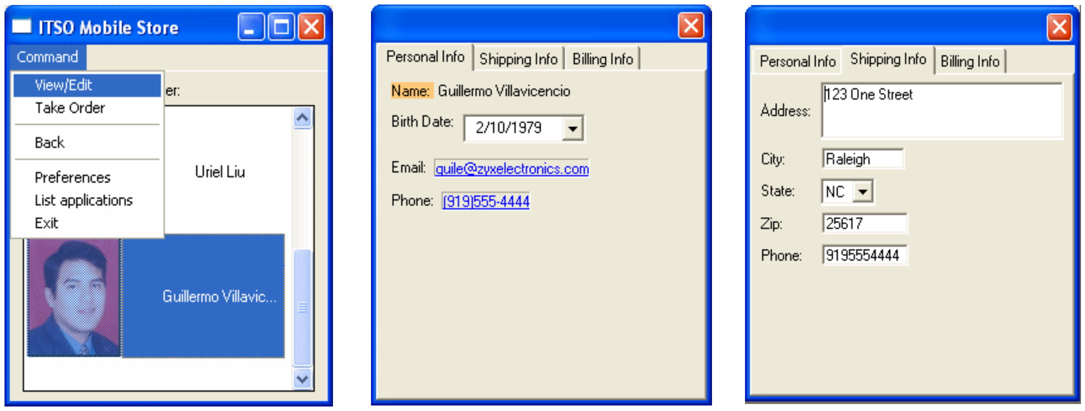


Figure 8-37 Testing the customer details dialog

11. Select another customer and select **Command** → **Take Order**. The Take Order view opens (`TakeOrderView` class). The order belongs to the customer that you previously selected. Note that the order is empty.

12. Add a product to the order. Select **Command** → **Add item.** Figure 8-38 shows the windows sequence.

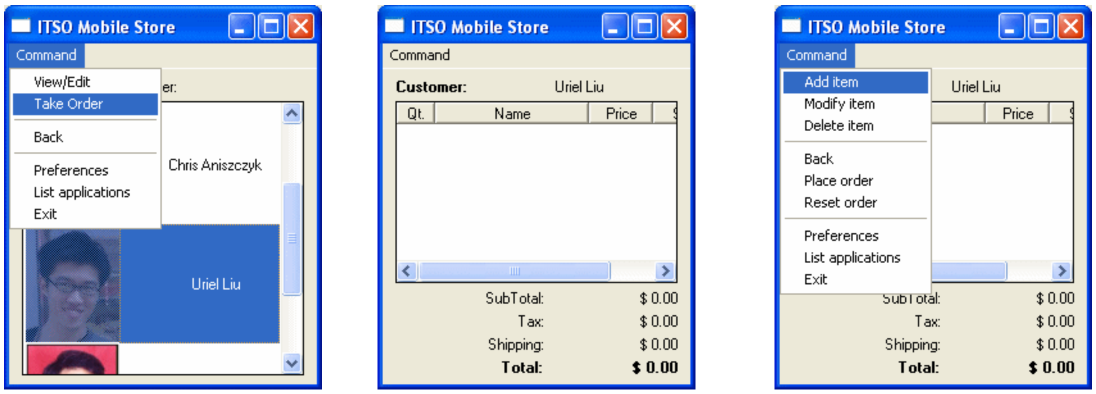


Figure 8-38 Testing the take order view

13. The product list dialog opens (`ProductDialog` class). Select a product from the product list. Go to the product details page. You see a brief description of the product. Click the *X* located in the upper, left corner to close the dialog. A dialog prompts you for the quantity. Enter a number or accept the default, and click **OK**. Figure 8-39 shows the windows sequence.

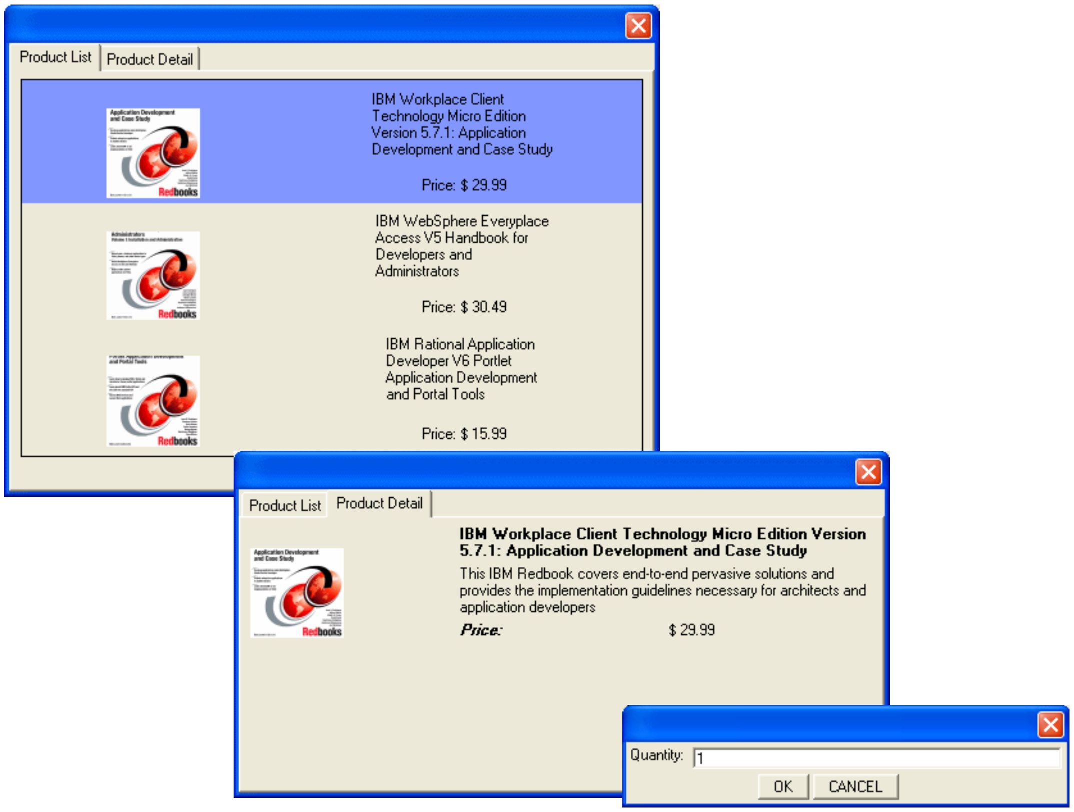
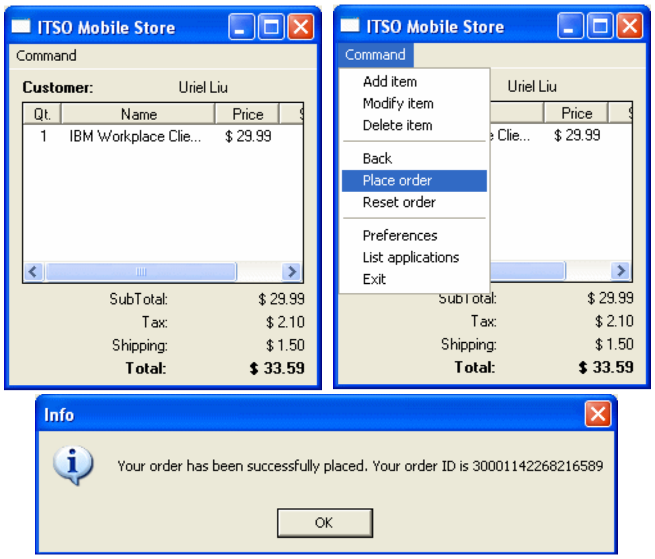

Product List
Product Detail
IBM Workplace Client Technology Micro Edition Version 5.7.1: Application Development and Case Study
Price: $ 29.99
IBM WebSphere Everyplace Access V5 Handbook for Developers and Administrators
Price: $ 30.49
IBM Rational Application Developer V6 Portlet Application Development and Portal Tools
Price: $ 15.99
IBM Workplace Client Technology Micro Edition Version 5.7.1: Application Development and Case Study
This IBM Redbook covers end-to-end pervasive solutions and provides the implementation guidelines necessary for architects and application developers
Price:
$ 29.99
Quantity:
1
OK
CANCEL

Figure 8-39 Testing the product list dialog

14. Note that the order now has an item on it and that the fields reflect this fact. Add other items if you want and test some of the other available commands, such as modify item, delete item, and so on.

15. Select **Command** → **Place order** to register the order in the database. An order ID is generated.

16. Click **OK** to close the information dialog. Close the application. Figure 8-40 shows the windows sequence.



Your order has been successfully placed. Your order ID is 30001142268216589

Figure 8-40 Testing the place order feature

8.8 Deploying and running the application in the mobile device

At this time, you can run the application in the mobile device. Make sure that you have followed all the steps to set up the environment as described in 8.1, "Preparing the environment" on page 130.

Follow these steps to deploy and to test the application in your WM2003 device:

1. Deploy the JRE for WM2003.
2. Deploy the eRCP Runtime for WM2003.
3. Copy the database libraries to the device.
4. Deploy the application to the eRCP runtime.
5. Test the application.

8.8.1 Deploying the JRE for WM2003

You need to deploy the JRE in the WM2003 device. The JRE for is provided as part of the Workplace Client Technology, Micro Edition environment that you installed in 8.1.3, "Installing Workplace Client Technology Micro Edition for Windows" on page 132. For WM2003 the JRE is conveniently packaged as a CAB file. Follow these steps to install the JRE for WM2003.

To deploy the JRD for WM2003:

1. Make sure that your device is connected to your desktop using the ActiveSync® software. Refer to your device vendor documentation for more information about this step.

2. Locate the JRE Foundation Profile for WM2003 CAB file. It should be located at the Workplace Client Technology, Micro Edition Toolkit installation directory, for example:

   C:\Program Files\IBM\DeviceDeveloper \wsdd5.0\ive-2.2\runtimes\wm2003\arm\foundation10\cab

   The CAB file to install is named j9-foun10-wm2003-arm_22.cab.
3. Copy the CAB file to a folder in your device, for example \Temp and tap it. Wait while the Foundation Profile is installed. The Foundation Profile should has been installed in \Program Files\J9.

8.8.2 Deploying the eRCP runtime for WM2003

If you have followed the steps to prepare the eRCP environment as described in 8.1.4, “Installing the eRCP runtimes” on page 133, deploying the eRCP runtime to the mobile devices involves no more than copying the files to the root folder on the device. Follow these steps:

1. Make sure that your device is connected to your desktop using the ActiveSync software. Refer to your device vendor documentation for more information about this step.
2. Copy the eRCP directory from the desktop location (for example, C:\eRCP-v20060118-1017\wm2003) to the root directory of the mobile device.

8.8.3 Copying the database libraries to the device

As discussed in 8.1.7, “Configuring the environment to use the database components” on page 136, you need to copy certain native libraries to the JRE bin directory. Follow these steps:

1. Locate the DB2 Everyplace natives libraries for the WM2003 platform. They should be located in the Device Developer directory, for example:

   C:\Program Files\IBM \DeviceDeveloper\wsdd5.0\technologies\eswe\files\db2e\wince\wce400\arm

   They are:

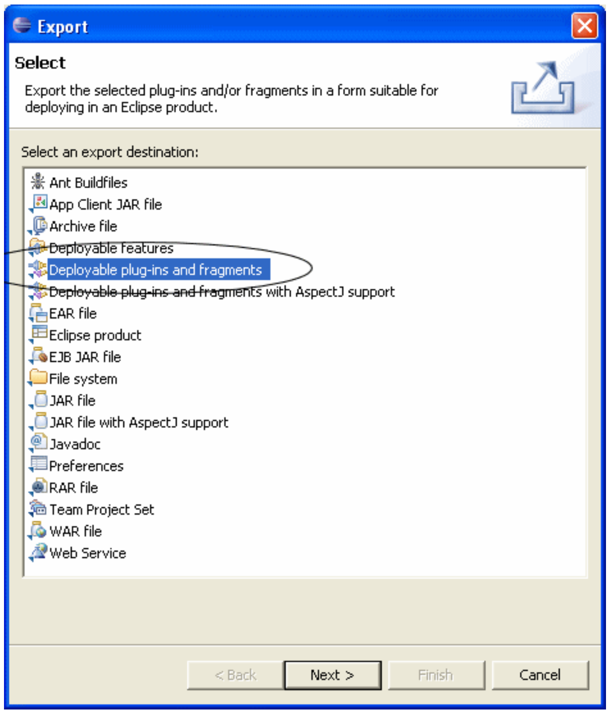
   – DB2e.dll, the DB2e engine
   – DB2EJDBC.dll, the DB2e JDBC driver native support
2. Copy the libraries in the JRE bin directory on the device (for example, \Program Files \J9\FOUN10\bin).
3. Copy the jdbc.jar library, located in the Workplace Client Technology, Micro Edition Toolkit directory (for example C:\Program Files\IBM\DeviceDeveloper\wsdd5.0\technologies\eswe\bundlefiles) to the ext directory of the device JRE located for example at \Program Files\J9\FOUN10\lib \jclFoundation10 on the device. Create the directory if it does not exists. Remember that you also copied this library to the eRCP Runtime plug-ins directory for Win32 and WM2003.

Note: By default the JRE looks for any java.* classes in libraries that are located in the ext directory. Also, the DB2e.jar has a declared dependency with the jdbc.jar bundle in its deployment descriptor. So, it is necessary to copy the library to the eRCP runtime plug-ins directory as described in 8.1.7, “Configuring the environment to use the database components” on page 136.

8.8.4 Deploying the application

To deploy the application, you need to export the application as a plug-in. Then, you need to copy this plug-in to the eRCP Runtime plug-ins directory. Follow these steps:

1. Start the Eclipse SDK, if it is not already started. Right-click the project name and select **Export**. Select **Deployable plug-ins and fragments** as shown in Figure 8-41. Click **Next**.



Export

Select

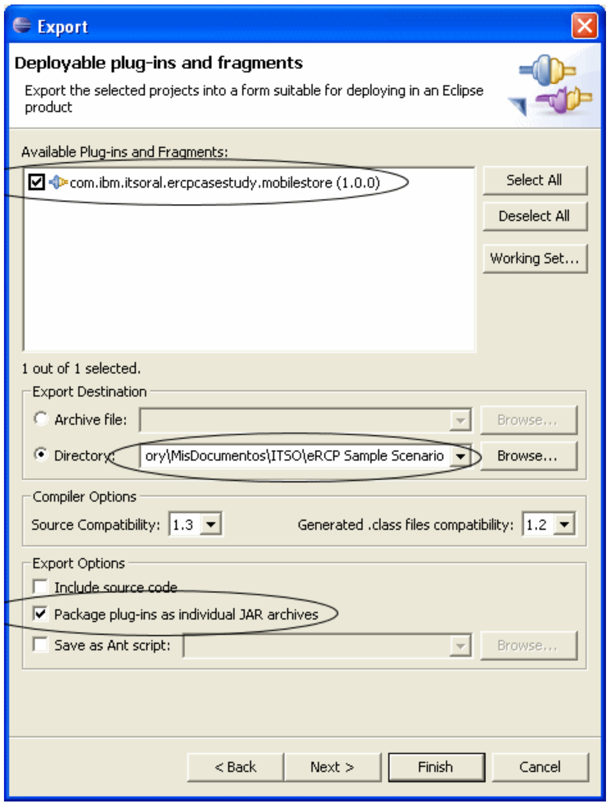
Export the selected plug-ins and/or fragments in a form suitable for deploying in an Eclipse product.

Select an export destination:

- Ant Buildfiles
- App Client JAR file
- Archive file
- Deployable features
- Deployable plug-ins and fragments
- Deployable plug-ins and fragments with AspectJ support
- EAR file
- Eclipse product
- EJB JAR file
- File system
- JAR file
- JAR file with AspectJ support
- Javadoc
- Preferences
- RAR file
- Team Project Set
- WAR file
- Web Service

Figure 8-41 Deploying the application as a plug-in

2. Make sure that the application plug-in is selected. Click **Browse** and select a directory in which to place the plug-in file. Note that the plug-in is deployed as a JAR file, as shown in Figure 8-42. Click **Finish**.



Export

Deployable plug-ins and fragments

Export the selected projects into a form suitable for deploying in an Eclipse product

Available Plug-ins and Fragments:

☑ com.ibm.itsoral.ercpcasestudy.mobilestore (1.0.0)

1 out of 1 selected.

Export Destination

Archive file:

Directory: ory\MisDocumentos\ITSO\eRCP Sample Scenario

Compiler Options

Source Compatibility: 1.3

Generated .class files compatibility: 1.2

Export Options

☐ Include source code

☑ Package plug-ins as individual JAR archives

☐ Save as Ant script:

Figure 8-42 Specifying the option for plug-in exporting

3. Wait while the plug-in is exported. The resulting plug-in should be a file located in the directory that you selected in the previous step (for example, plugins\com.ibm.itsoral.ercpcasestudy.mobilestore_1.0.0.jar). Copy this file to the eRCP Runtime plug-ins directory, located at \eRCP\plugins in your device.

8.8.5 Running the application

Now that you have all the components in place, you can run the application in the mobile device. Follow these steps:

1. Enter to the \eRCP directory in your device and tap the j9foun-eworkbench icon. Wait while the eWorkbench loads. Select **ITSO Mobile Store** and then **Synchronization**. Figure 8-43 shows the windows sequence.

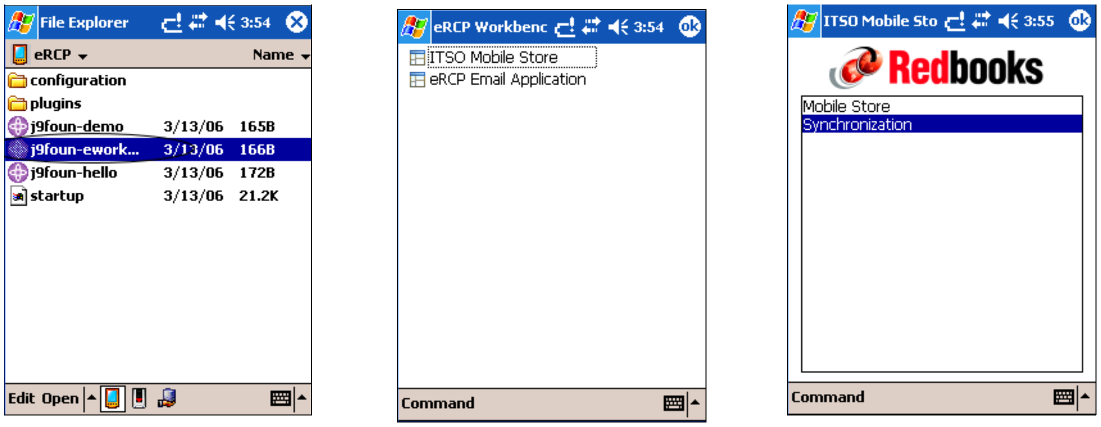
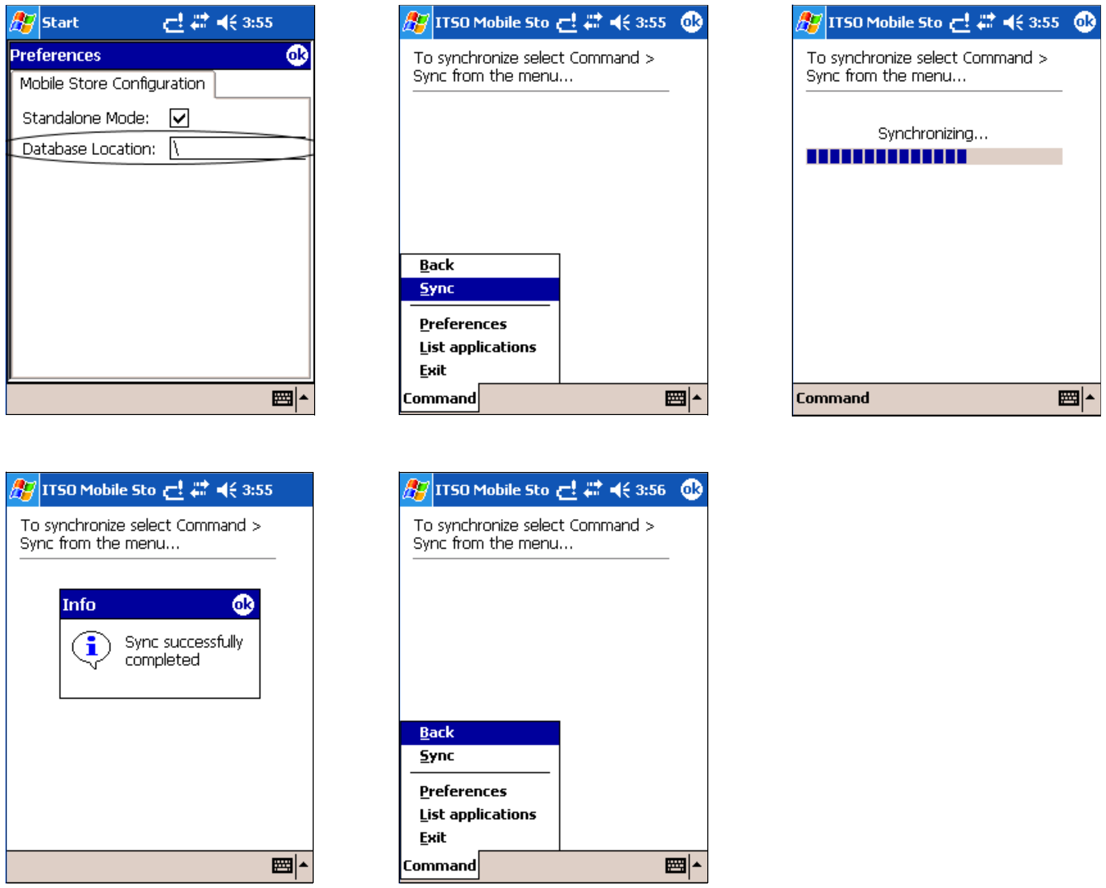


Figure 8-43 Testing the application on the WM2003 device

2. Select **Command** → **Preferences** from the menu bar, enter a location for the local database, and click **OK.**
3. Select **Command** → **Sync** and wait while the database is created and populated. Click **OK** in the sync status dialog.

4. Select **Command** → **Back** to return to the Module Selector view. Figure 8-44 shows the windows sequence.



To synchronize select Command > Sync from the menu...

Sync successfully completed

Figure 8-44 Testing the preferences and synchronization features on the WM2003 device

5. Select **Mobile Store** in the module selector view. The customer list information displays, as shown in Figure 8-45.

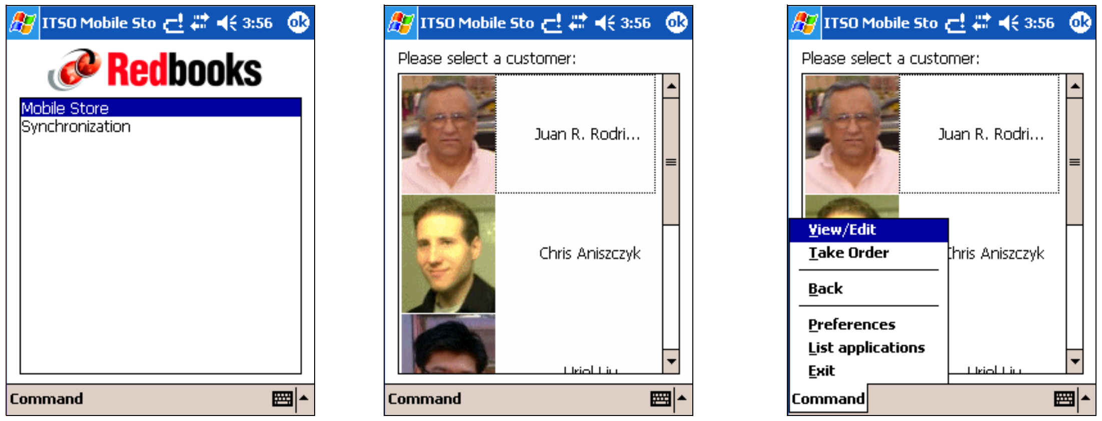


Figure 8-45 Testing the customer list on the WM2003 device

You can keep testing the application, using the steps that are described in 8.7, “Running the ITSO Mobile Store application” on page 215 as a guide.

A

Additional material

This appendix describes how to download additional material from the Internet as described throughout this Redpaper.

Locating the Web material

The Web material that is associated with this Redpaper is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser to:

`ftp://www.redbooks.ibm.com/redbooks/REDP4118`

Alternatively, you can go to the IBM Redbooks Web site at:

`ibm.com/redbooks`

Select **Additional materials** and open the directory that corresponds with the Redpaper form number, REDP4118.

Using the Web material

The additional Web material that accompanies this Redpaper includes the following files:

File name	*Description*
redp4118.zip	Sample code

System requirements for downloading the Web material

The following system configuration is recommended:

Hard disk space:	40 GB minimum
Operating System:	Windows
Processor:	1 GH or higher
Memory:	1 GB

How to use the Web material

Create a subdirectory (folder) on your workstation, and unzip the contents of the Web material zipped file into this folder.

Related publications

This section lists the publications that are considered particularly suitable for a more detailed discussion of the topics that we cover in this Redpaper.

IBM Redbooks

For information about ordering these publications, see “How to get IBM Redbooks” on page 227. Note that some of the documents referenced here might be available in softcopy only.

- *IBM WebSphere Everyplace Deployment V6 Handbook for Developers and Administrators Volume II: Smart Client Application Development*, SG24-7183

Other publications

These publications are also relevant as further information sources:

- *Java Developer's Guide to Eclipse*, 0321159640
- *Eclipse: Building Commercial-Quality Plug-ins*, 032142672X

Online resources

These Web sites and URLs are also relevant as further information sources:

- Eclipsepedia

  `http://wiki.eclipse.org`

- Eclipse Web site

  `http://www.eclipse.org`

- Eclipse eRCP project site

  `http://www.eclipse.org/proposals/eclipse-ercp/`

How to get IBM Redbooks

You can search for, view, or download Redbooks, Redpapers, Hints and Tips, draft publications and Additional materials, as well as order hardcopy Redbooks or CD-ROMs, at this Web site:

ibm.com/redbooks

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

The Eclipse embedded Rich Client Platform

A Graphical User Interface for Small Devices

Redpaper

Understand eRCP applications for embedded mobile devices

Learn about the embedded Standard Widget Toolkit (eSWT)

Develop eJFace applications using the eWorkbench application model

The embedded Rich Client Platform (eRCP) is an open source project under the Eclipse Technology Project using OSGi standards. This IBM Redpaper covers the eRCP Extension Point Framework as the basis for creating embedded client platforms and implementing eSWT, eJFace, and eWorkbench for embedded devices having fewer resources and smaller screen sizes than a desktop computer.

In this redpaper, you will find information about what APIs, as a subset of SWT for desktops, are available and apply to Eclipse eSWT basic components, including Core eSWT and Expanded eSWT. You will also find information to better use JFace as a means for providing model, view, and controller mobile application paradigms.

This redpaper includes also a sample scenario chosen to illustrate how eRCP applications are developed to access a local database using DB2 Everyplace. You will find the elements of the business processes being put into place, and understand what technologies were chosen to solve the different parts of the implementation. The sample scenario includes step by step guidelines to use the proper tools, runtimes, and APIs needed to build, test and deploy an eRCP application for small devices.

A basic knowledge of Java programming and Java technologies is required.