Copyright 2004 Adobe Systems Incorporated. All rights reserved.
The information in this document is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Adobe Systems Incorporated. Adobe Systems Incorporated assumes no responsibility or liability for any errors or inaccuracies that may appear in this document. The software described in this document is furnished under license and may only be used or copied in accordance with the terms of such license.

Adobe, Adobe After Effects, Adobe InDesign, Adobe PhotoDeluxe, Adobe Premiere, Adobe Photoshop, Adobe Illustrator, Adobe Type Manager, ATM and PostScript are trademarks of Adobe Systems Incorporated that may be registered in certain jurisdictions. Macintosh and Apple are registered trademarks, and Mac OS is a trademark of Apple Computer, Inc. Microsoft, Windows, Windows 95, Windows 98, and Windows NT are registered trademarks of Microsoft Corporation. All other products or name brands are trademarks of their respective holders.

<table>
<thead>
<tr>
<th>Rev #</th>
<th>Date</th>
<th>Author</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>22 Aug 2003</td>
<td>Rodney Cook</td>
<td>Updated for version CS.</td>
</tr>
<tr>
<td>1.0</td>
<td>03 Sep 2003</td>
<td>Ken Sadahiro</td>
<td>Updated title page image.</td>
</tr>
<tr>
<td>1.1</td>
<td>26 Feb 2004</td>
<td>Rodney Cook</td>
<td>Moved Tools chapter to Programming Guide</td>
</tr>
<tr>
<td>1.2</td>
<td>1 Mar 2004</td>
<td>Ian Paterson</td>
<td>Some final review points and edits.</td>
</tr>
</tbody>
</table>
# Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>User Interface: Introduction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Abstract</td>
</tr>
<tr>
<td></td>
<td>Objectives</td>
</tr>
<tr>
<td></td>
<td>Terminology and definitions</td>
</tr>
<tr>
<td></td>
<td>Introduction</td>
</tr>
<tr>
<td>Key concepts</td>
<td></td>
</tr>
<tr>
<td>Design objectives</td>
<td></td>
</tr>
<tr>
<td>Idioms and naming conventions</td>
<td></td>
</tr>
<tr>
<td>Namespaces for strings and identifiers</td>
<td></td>
</tr>
<tr>
<td>Abstractions and re-use</td>
<td></td>
</tr>
<tr>
<td>Widgets versus platform controls</td>
<td></td>
</tr>
<tr>
<td>Commands, model plug-ins and user interface plug-ins</td>
<td></td>
</tr>
<tr>
<td>Suites and the user interface</td>
<td></td>
</tr>
<tr>
<td>Finding widgets in the API</td>
<td></td>
</tr>
<tr>
<td>Getting started with widgets</td>
<td></td>
</tr>
<tr>
<td>How to get notifications about control events or changes</td>
<td></td>
</tr>
<tr>
<td>Frequently asked questions (FAQs)</td>
<td></td>
</tr>
<tr>
<td>How do I write safe code and get diagnostic information?</td>
<td></td>
</tr>
<tr>
<td>How should I separate logic and UI code?</td>
<td></td>
</tr>
<tr>
<td>How do I develop a plug-in user interface?</td>
<td></td>
</tr>
<tr>
<td>When do I have to subclass widget boss classes and ODFRez custom resource types?</td>
<td></td>
</tr>
<tr>
<td>How do I find the panel that a pop-up menu belongs to?</td>
<td></td>
</tr>
<tr>
<td>How do I iterate over a container's child widgets?</td>
<td></td>
</tr>
<tr>
<td>How do I find a specific widget in the hierarchy?</td>
<td></td>
</tr>
<tr>
<td>How do I show/hide a widget, or enable/disable it?</td>
<td></td>
</tr>
<tr>
<td>How do I create an alert?</td>
<td></td>
</tr>
<tr>
<td>How can I build a basic interface to support a binary or ternary choice?</td>
<td></td>
</tr>
<tr>
<td>How do I create menu-entries?</td>
<td></td>
</tr>
<tr>
<td>How do I create a dialog?</td>
<td></td>
</tr>
<tr>
<td>How do I create a selectable dialog?</td>
<td></td>
</tr>
<tr>
<td>How do I add a panel to a selectable dialog?</td>
<td></td>
</tr>
<tr>
<td>How do I create a previewable dialog?</td>
<td></td>
</tr>
<tr>
<td>How do I add buttons to a dialog?</td>
<td></td>
</tr>
<tr>
<td>How do I add tips to a widget?</td>
<td></td>
</tr>
<tr>
<td>How do I add a check box to a dialog?</td>
<td></td>
</tr>
</tbody>
</table>
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>How do I add a checkbox to a panel?</td>
<td>25</td>
</tr>
<tr>
<td>How do I add radio-buttons?</td>
<td>25</td>
</tr>
<tr>
<td>How do I ensure that buttons in a group have mutually exclusive behaviour?</td>
<td>25</td>
</tr>
<tr>
<td>How do I add a multi-line static text widget that scrolls?</td>
<td>25</td>
</tr>
<tr>
<td>How do I add a text edit box?</td>
<td>26</td>
</tr>
<tr>
<td>How do I get changes associated with an edit box?</td>
<td>26</td>
</tr>
<tr>
<td>How do I override an event handler?</td>
<td>26</td>
</tr>
<tr>
<td>How do I add a drop-down list?</td>
<td>26</td>
</tr>
<tr>
<td>How do I add a specialised combo-box to a panel?</td>
<td>26</td>
</tr>
<tr>
<td>How do I create a widget with a picture?</td>
<td>27</td>
</tr>
<tr>
<td>How do I change the picture displayed in an image widget?</td>
<td>27</td>
</tr>
<tr>
<td>How do I change the font of a static text widget?</td>
<td>27</td>
</tr>
<tr>
<td>How do I know when a dialog repaints?</td>
<td>27</td>
</tr>
<tr>
<td>How do I add a contextual menu to a plug-in?</td>
<td>27</td>
</tr>
<tr>
<td>How do I set the minimum size for a resizable panel?</td>
<td>28</td>
</tr>
<tr>
<td>How do I group widgets?</td>
<td>28</td>
</tr>
<tr>
<td>How do I add a list box to a panel?</td>
<td>28</td>
</tr>
<tr>
<td>How do I vary the set of widgets displayed?</td>
<td>28</td>
</tr>
<tr>
<td>How do I override the default draw behaviour?</td>
<td>28</td>
</tr>
<tr>
<td>How do I use a tree view?</td>
<td>28</td>
</tr>
<tr>
<td>How do I ensure that buttons in a group have mutually exclusive behaviour?</td>
<td>25</td>
</tr>
<tr>
<td>How do I add a drop-down list?</td>
<td>26</td>
</tr>
<tr>
<td>How do I add a specialised combo-box to a panel?</td>
<td>26</td>
</tr>
<tr>
<td>How do I create a widget with a picture?</td>
<td>27</td>
</tr>
<tr>
<td>How do I change the picture displayed in an image widget?</td>
<td>27</td>
</tr>
<tr>
<td>How do I change the font of a static text widget?</td>
<td>27</td>
</tr>
<tr>
<td>How do I know when a dialog repaints?</td>
<td>27</td>
</tr>
<tr>
<td>How do I add a contextual menu to a plug-in?</td>
<td>27</td>
</tr>
<tr>
<td>How do I set the minimum size for a resizable panel?</td>
<td>28</td>
</tr>
<tr>
<td>How do I group widgets?</td>
<td>28</td>
</tr>
<tr>
<td>How do I add a list box to a panel?</td>
<td>28</td>
</tr>
<tr>
<td>How do I vary the set of widgets displayed?</td>
<td>28</td>
</tr>
<tr>
<td>How do I override the default draw behaviour?</td>
<td>28</td>
</tr>
<tr>
<td>How do I use a tree view?</td>
<td>28</td>
</tr>
<tr>
<td>How do I ensure that buttons in a group have mutually exclusive behaviour?</td>
<td>25</td>
</tr>
<tr>
<td>How do I add a drop-down list?</td>
<td>26</td>
</tr>
<tr>
<td>How do I add a specialised combo-box to a panel?</td>
<td>26</td>
</tr>
<tr>
<td>How do I create a widget with a picture?</td>
<td>27</td>
</tr>
<tr>
<td>How do I change the picture displayed in an image widget?</td>
<td>27</td>
</tr>
<tr>
<td>How do I change the font of a static text widget?</td>
<td>27</td>
</tr>
<tr>
<td>How do I know when a dialog repaints?</td>
<td>27</td>
</tr>
<tr>
<td>How do I add a contextual menu to a plug-in?</td>
<td>27</td>
</tr>
<tr>
<td>How do I set the minimum size for a resizable panel?</td>
<td>28</td>
</tr>
<tr>
<td>How do I group widgets?</td>
<td>28</td>
</tr>
<tr>
<td>How do I add a list box to a panel?</td>
<td>28</td>
</tr>
<tr>
<td>How do I vary the set of widgets displayed?</td>
<td>28</td>
</tr>
<tr>
<td>How do I override the default draw behaviour?</td>
<td>28</td>
</tr>
<tr>
<td>How do I use a tree view?</td>
<td>28</td>
</tr>
<tr>
<td>How do I ensure that buttons in a group have mutually exclusive behaviour?</td>
<td>25</td>
</tr>
<tr>
<td>How do I add a drop-down list?</td>
<td>26</td>
</tr>
<tr>
<td>How do I add a specialised combo-box to a panel?</td>
<td>26</td>
</tr>
<tr>
<td>How do I create a widget with a picture?</td>
<td>27</td>
</tr>
<tr>
<td>How do I change the picture displayed in an image widget?</td>
<td>27</td>
</tr>
<tr>
<td>How do I change the font of a static text widget?</td>
<td>27</td>
</tr>
<tr>
<td>How do I know when a dialog repaints?</td>
<td>27</td>
</tr>
<tr>
<td>How do I add a contextual menu to a plug-in?</td>
<td>27</td>
</tr>
<tr>
<td>How do I set the minimum size for a resizable panel?</td>
<td>28</td>
</tr>
<tr>
<td>How do I group widgets?</td>
<td>28</td>
</tr>
<tr>
<td>How do I add a list box to a panel?</td>
<td>28</td>
</tr>
<tr>
<td>How do I vary the set of widgets displayed?</td>
<td>28</td>
</tr>
<tr>
<td>How do I override the default draw behaviour?</td>
<td>28</td>
</tr>
<tr>
<td>How do I use a tree view?</td>
<td>28</td>
</tr>
</tbody>
</table>

## Chapter 3: User Interface Architecture

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>31</td>
</tr>
<tr>
<td>Objectives</td>
<td>31</td>
</tr>
<tr>
<td>User interface example</td>
<td>31</td>
</tr>
<tr>
<td>Type binding.</td>
<td>33</td>
</tr>
<tr>
<td>Type binding and behaviour</td>
<td>36</td>
</tr>
<tr>
<td>Factorisation of the user interface model</td>
<td>38</td>
</tr>
<tr>
<td>Control-views</td>
<td>39</td>
</tr>
<tr>
<td>Control-data models</td>
<td>39</td>
</tr>
<tr>
<td>Attributes</td>
<td>40</td>
</tr>
<tr>
<td>Event handlers</td>
<td>40</td>
</tr>
<tr>
<td>Relevant design patterns</td>
<td>41</td>
</tr>
<tr>
<td>Observer pattern</td>
<td>41</td>
</tr>
<tr>
<td>How the event handlers implement controllers</td>
<td>42</td>
</tr>
<tr>
<td>What role does MVC play in the user interface model?</td>
<td>43</td>
</tr>
<tr>
<td>Chain of responsibility</td>
<td>44</td>
</tr>
<tr>
<td>Facade</td>
<td>44</td>
</tr>
<tr>
<td>Command</td>
<td>44</td>
</tr>
<tr>
<td>Topic</td>
<td>Page</td>
</tr>
<tr>
<td>--------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Design</td>
<td>65</td>
</tr>
<tr>
<td>Widget data model</td>
<td>65</td>
</tr>
<tr>
<td>Capabilities</td>
<td>65</td>
</tr>
<tr>
<td>Notification</td>
<td>66</td>
</tr>
<tr>
<td>Palettes</td>
<td>67</td>
</tr>
<tr>
<td>Description</td>
<td>67</td>
</tr>
<tr>
<td>Design</td>
<td>68</td>
</tr>
<tr>
<td>Capabilities</td>
<td>69</td>
</tr>
<tr>
<td>Advanced issues</td>
<td>70</td>
</tr>
<tr>
<td>Panel widgets</td>
<td>70</td>
</tr>
<tr>
<td>Description</td>
<td>70</td>
</tr>
<tr>
<td>Design</td>
<td>72</td>
</tr>
<tr>
<td>Widget data model</td>
<td>72</td>
</tr>
<tr>
<td>Capabilities</td>
<td>73</td>
</tr>
<tr>
<td>Detail-controlling panels</td>
<td>73</td>
</tr>
<tr>
<td>Dynamic panels</td>
<td>74</td>
</tr>
<tr>
<td>Control strip</td>
<td>74</td>
</tr>
<tr>
<td>Kits</td>
<td>74</td>
</tr>
<tr>
<td>Resizable panels</td>
<td>75</td>
</tr>
<tr>
<td>Static text widgets</td>
<td>76</td>
</tr>
<tr>
<td>Description</td>
<td>76</td>
</tr>
<tr>
<td>Design</td>
<td>77</td>
</tr>
<tr>
<td>Widget data model</td>
<td>77</td>
</tr>
<tr>
<td>Capabilities</td>
<td>78</td>
</tr>
<tr>
<td>Check-boxes and radio buttons</td>
<td>79</td>
</tr>
<tr>
<td>Description</td>
<td>79</td>
</tr>
<tr>
<td>Design</td>
<td>80</td>
</tr>
<tr>
<td>Widget data model</td>
<td>80</td>
</tr>
<tr>
<td>Capabilities</td>
<td>81</td>
</tr>
<tr>
<td>Notification</td>
<td>81</td>
</tr>
<tr>
<td>Controls on panels</td>
<td>81</td>
</tr>
<tr>
<td>Button widgets</td>
<td>82</td>
</tr>
<tr>
<td>Description</td>
<td>82</td>
</tr>
<tr>
<td>Design</td>
<td>83</td>
</tr>
<tr>
<td>Widget data model</td>
<td>84</td>
</tr>
<tr>
<td>Capabilities</td>
<td>84</td>
</tr>
<tr>
<td>UI guidelines</td>
<td>84</td>
</tr>
<tr>
<td>Notification</td>
<td>85</td>
</tr>
<tr>
<td>Buttons on dialogs</td>
<td>85</td>
</tr>
<tr>
<td>Edit boxes</td>
<td>85</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Description</td>
<td>85</td>
</tr>
<tr>
<td>Design</td>
<td>87</td>
</tr>
<tr>
<td>Widget data model</td>
<td>88</td>
</tr>
<tr>
<td>Capabilities</td>
<td>88</td>
</tr>
<tr>
<td>UI guidelines</td>
<td>88</td>
</tr>
<tr>
<td>Notification</td>
<td>88</td>
</tr>
<tr>
<td>Icon and picture widgets</td>
<td>89</td>
</tr>
<tr>
<td>Description</td>
<td>89</td>
</tr>
<tr>
<td>Design</td>
<td>90</td>
</tr>
<tr>
<td>Capabilities</td>
<td>90</td>
</tr>
<tr>
<td>Notification</td>
<td>91</td>
</tr>
<tr>
<td>Working with picture widgets</td>
<td>91</td>
</tr>
<tr>
<td>Drop-down lists and combo-boxes</td>
<td>91</td>
</tr>
<tr>
<td>Description</td>
<td>91</td>
</tr>
<tr>
<td>Design</td>
<td>93</td>
</tr>
<tr>
<td>Widget data model</td>
<td>94</td>
</tr>
<tr>
<td>Capabilities</td>
<td>94</td>
</tr>
<tr>
<td>List-box widgets</td>
<td>94</td>
</tr>
<tr>
<td>Description</td>
<td>94</td>
</tr>
<tr>
<td>Design</td>
<td>96</td>
</tr>
<tr>
<td>Widget data model</td>
<td>96</td>
</tr>
<tr>
<td>Capabilities</td>
<td>96</td>
</tr>
<tr>
<td>User interface guidelines</td>
<td>96</td>
</tr>
<tr>
<td>Notification</td>
<td>96</td>
</tr>
<tr>
<td>Splitter widgets</td>
<td>97</td>
</tr>
<tr>
<td>Description</td>
<td>97</td>
</tr>
<tr>
<td>Design</td>
<td>97</td>
</tr>
<tr>
<td>Capabilities</td>
<td>98</td>
</tr>
<tr>
<td>Scroll-bars and scrolling</td>
<td>98</td>
</tr>
<tr>
<td>Description</td>
<td>98</td>
</tr>
<tr>
<td>Design</td>
<td>98</td>
</tr>
<tr>
<td>Widget data model</td>
<td>99</td>
</tr>
<tr>
<td>Capabilities</td>
<td>99</td>
</tr>
<tr>
<td>Notification</td>
<td>99</td>
</tr>
<tr>
<td>Sliders</td>
<td>99</td>
</tr>
<tr>
<td>Description</td>
<td>99</td>
</tr>
<tr>
<td>Design</td>
<td>101</td>
</tr>
<tr>
<td>Widget data model</td>
<td>101</td>
</tr>
<tr>
<td>Capabilities</td>
<td>101</td>
</tr>
<tr>
<td>Notification</td>
<td>102</td>
</tr>
</tbody>
</table>
Contents

Implementation detail .............................................................. 102
Tree-view widgets .................................................................... 103
Description ................................................................................. 103
Design ......................................................................................... 106
Widget data model ...................................................................... 106
Capabilities ................................................................................ 106
Notification ............................................................................... 109
Implementation specifics ............................................................ 109
User Interface: Introduction

Abstract

This chapter introduces the user-interface programming model and provides a context for the other chapters that describe user-interface programming. The user-interface model is introduced here at a conceptual level and refined in subsequent chapters. We also consider some frequently-asked-questions and provide answers in outline.

Objectives

The objectives of this chapter are:

- Introduce terms needed to understand the user interface programming model of InDesign CS.
- Introduce some key concepts in user interface programming.
- Answer some frequently-asked questions relating to user interfaces.

Terminology and definitions

This section introduces the terms needed to understand the other chapters in this document. The term “widget” is particularly prone to confusion and is always qualified in this document to make its meaning clear. For example, “static text widget” is highly ambiguous in the context of the InDesign CS/InCopy CS API and could refer to one of at least four things: the boss class that provides the behaviour (kStaticTextWidgetBoss), or an instance of this boss class, or the ODFRez custom resource type StaticTextWidget, or an instance of this ODFRez type that is used to specify initial state and properties of a kStaticTextWidgetBoss object.

- **action**: this refers to a parcel of functionality that can be invoked through a menu component or a keyboard shortcut. Plug-ins can implement actions that execute in response to a IActionComponent::DoAction message from the application framework.

- **aggregation**: a boss class is said to aggregate an interface if it provides an implementation of that interface. A boss class may also be said to expose an interface of a specific type. InDesign CS/InCopy CS interfaces are descendants of the type IPMUnknown. There are some abstract classes in the API that do not descend from IPMUnknown, such as IEvent, and these cannot be reference-counted.

- **application core**: this performs functions such as inviting plug-in panels to register, sending IObserver::Update messages to registered observers, or sending...
IActionComponent::DoAction messages when a particular action component has been activated by a shortcut or menu item.

- **client code**: code written, say, by a third-party plug-in developer that exploits the services of the InDesign CS/InCopy CS API. A way to identify client-code; it is typically driving suites and/or processing commands, or wrapper interfaces that in turn process commands (e.g. ITableCommands, IXMLElementCommands).

- **control-data model**: this aspect of a widget boss class represents the data associated with a widget’s state. It can typically be changed by an end-user or through the API. It is associated with an interface named I<data-type>ControlData. For instance, there is an ITextControlData interface associated with text-edit boxes, whose internal state is represented by a PMString.

- **control-view**: this aspect of a widget boss class is concerned with representing the appearance of a widget. It is associated with an interface named IControlView; this interface can also be used to change the visual representation of a given widget, for instance, to vary the resources that are used in representing the states of an iconic button, or to show/hide a widget.

- **detail-control**: the process of varying the resolution of a user-interface, by varying the composition of the widget set or the size of elements in the interface; the capability is represented by IPanelDetailController.

- **dialog protocol**: message sequencing for dialog boss objects (kDialogBoss and descendants). Consists of messages delivered in sequence: IDialogController::InitializeDialogFields, ValidateDialogFields, ApplyDialogFields. A ResetDialogFields may also be sent but only if the dialog has this enabled.

- **extending**: is intended to be synonymous with subclassing. Both boss classes and ODFRez types can be extended or subclassed and the terms may be used interchangeably within this document set.

- **event handler**: is responsible for processing events and changing the data model of a widget-equivalent to a controller. Unlike other APIs, where extending an event-handler is required to get notification about control changes, there are very few circumstances where overriding a widget event handler is required in the InDesign CS/InCopy CS API. Notifications about changes in control state are sent as IObserver::Update messages, and this is true since InDesign CS/InCopy CS update for every keystroke for edit-boxes, if they are configured correctly in the ODFRez data statements.

- **helper class**: may also be described as **partial implementation class**. This is an API class, that can provide most or all of the code required to implement a particular interface. For instance, CObserver can be used to provide a basic implementation of IObserver. CActiveSelectionObserver provides a richer implementation for client code that wishes to observe changes in the active context.

- **interface**: refers to an abstract type that extends IPMUnknown. Interfaces are aggregated by boss classes and represent the capability or services that a boss class provides to client code.

- **low-level event**: refers to something like a window-system occurrence or low-level input, e.g. single keystroke.
message protocol: within the context of the InDesign CS/InCopy CS API, refers to the IID sent along with the IObserver::Update message. An observer ‘listens’ along a particular ‘protocol’; it is merely a way of establishing a filter on the semantic events that an observer might be informed about.

messaging architecture: related to widgets, how client code can be written to receive notifications from the application core when widget data models change. The main architecture of interest uses the Observer pattern.

observer: abstraction which listens for changes in a subject. It is represented in the InDesign CS API by the interface IObserver. Typically implementations will use COobserver or one of its subclasses to implement an IObserver interface, which when wired up correctly, can listen for changes in the control-data model of a subject (ISubject) and respond appropriately.

ODF: OpenDoc Framework. This was a cross-platform software effort initiated by Apple and IBM; it defined an object-oriented extension of the Apple Rez language to specify user interface resources.

ODFRez custom resource type: this refers to the type that is defined in the top-level framework resource file and typically populated in data statements in the localised framework resource files.

ODFRez data statement: an expression in ODFRez other than a type expression. It can be used to define the properties of a given widget or define a key-value pair in a string table.

ODFRez language: the OpenDoc Framework Resource Language that allows cross-platform resources to be defined for plug-ins. The SDK tool DollyXs is able to generate ODFRez.

palettes: these are floating windows that are containers for one or more panels; a palette can be dragged around and its children can be re-ordered.

panel: a container for widgets, such as the Layers panel. They can reside in tabbed palettes.

parent widget: may also be referred to as a container widget, a widget that contains another widget. The parent widget may determine when the child draws, and which of its children receive events.

partial implementation classes: also called helper classes, provided in the API which provide most of the capability required to implement key interfaces. Classes such as COObserver or CControlView fall into this category, as they provide some capability but leave key method implementations to be filled in by the developer.

pattern: a solution to a specific problem in object-oriented software design in a context- it should identify abstractions, along with their responsibilities and interactions; for instance, the Observer pattern includes a Subject abstraction.

persistent interface: a boss class aggregates interfaces, some of which may be persistent. They’re persistent if CREATE_PERSIST_PMINTERFACE is used in the implementation code when defining them, which forces the implementation to provide a ReadWrite method to deserialise/serialise an instance of the class. For instance, the IControlView interface on widget boss objects is always persistent; that is why every widget has at least a CControlView field in the ODFRez data, since it needs to set up its initial state like widget ID, whether it’s enabled, whether it’s visible, which are properties common to all widgets.
Terminology and definitions

- **pseudo-buttons**: buttons that have some button-like facility, being responsive to mouse clicks, but that are intended as image based buttons rather than text-labelled buttons.

- **resources, framework**: these are used to initialise the widgets, and are held in files written in ODFRez. These files also contain locale-specific information to allow user interfaces to be localised. Platform-specific resources (e.g. defined in a .rc, .r or .rsrc file) have a very small role to play in InDesign CS plug-ins, and are typically only used for icons/images in image-based widgets.

- **semantic event**: directly correspond to high level user interactions with a UI component. Clicking of a button is an example of a semantic event; these are communicated to client code through IObserver::Update messages with various ‘protocols’ (e.g. IID_IBOOLEANCONTROLDATA for a button click) and other message parameters.

- **smart pointer**: used to automatically manage the lifetime of the object that it encapsulates. At a minimum, it should provide an implementation of the -> operator that can be used as a pointer; it may also provide an implementation of the * (de-reference) operator. In the context of the InDesign CS API, smart pointers such as InterfacePtr<T> implement the necessary reference-counting behaviour to avoid memory leaks and controlled access to memory. There are other smart pointers in the InDesign CS API that may be encountered such as TreeNodePtr and K2::scoped_ptr.

- **standard buttons**: buttons that descend from kButtonWidgetBoss

- **subject**: abstraction that is the target for the attention of observers. In the user interface API, widget boss objects are the typical subjects.

- **type binding**: an association between a widget boss class and an ODFRez custom resource type. Alternatively, interfaces (by identifier, IID_<whatever>) can be bound to ODFRez custom resource types that compose more complex types. This type binding occurs in the context of an ODFrez type expression. For instance, the type expression defining the ODFRez type ButtonWidget binds it to kButtonWidgetBoss (see the API header file Widgets.fh for many examples of this). The ODFRez type delineates how the data is laid out in a binary resource for the widget boss class to de-serialize itself. That is, the type 'ButtonWidget' specifies the layout for data that allows an instance of kButtonWidgetBoss to read its initial state from the binary resource. When an interface is bound to an ODFRez custom resource type, it is expected that the implementation in the boss class will read its initial state from the fields defined in the ODFRez custom resource type. When the plug-in resource is being parsed for the first time, the type information that is represented in the binary resource provides sufficient information for the application core to make an instance of the correct type of widget, give it the intended widget ID and so, by reading the data in the resource that is effectively a serialised form of the widget boss object.

- **widget boss class**: these are boss classes that derive from the kBaseWidgetBoss. They are typically but not invariably named k<widgetname>WidgetBoss. The only difference between a widget boss class and a ‘normal’ boss class is that a widget boss class may be associated with an ODFRez custom resource type through an associating or “type binding”. The ODFRez type is concerned with the layout of plug-in resource data, so that a widget boss object can be correctly instantiated and initialised by the application framework. The ODFRez type defines the data that is required to correctly de-serialize a particular widget boss object from the plug-in resource.
User Interface: Introduction

Introduction

Plug-in developers may have to implement a user-interface relatively early in their plug-in programming experience. Currently, they may have only a modest grasp on the concepts of boss classes, aggregation and so on. They may become disheartened by encountering an additional layer of complexity when programming user interfaces, where the initial state and properties of the user interface that they are building must be specified in an exotic resource language (ODFRez). The principal objective of this chapter is to de-mystify the user interface programming model and to provide enough background to use existing reference materials to create richly responsive user interfaces for InDesign CS/InCopy CS plug-ins.

Many third-party plug-ins require some form of user interface, to allow end-users to parameterise the plug-in’s behaviour. It is very often the first part of InDesign CS programming that newcomers to the API encounter. However, at first encounter with the InDesign CS user-interface programming model, it may seem relatively complex and frustrating to develop a user interface. This document and the related tech-notes in this series attempt to provide background information and detail to make this process more straightforward. There are only a few key ideas required to understand the user interface programming model, and once these are grasped, programming user interface for InDesign CS plug-ins becomes a straightforward task.

The user interface architecture is fundamentally simple, although initially perplexing. Once you grasp the concept of type binding between boss classes (or API interfaces) and ODFRez types, and how persistent interfaces on the boss classes read their initial state from the plug-in resources, then the whole user interface model becomes transparent. However, since new developers typically are not comfortable with the notion of boss classes, and often developing a plug-in user interface is one of their first tasks, it is plain to see that layering the complexity of an association between the barely-grasped boss-class concept and yet another type system can be a source of pain and confusion for new developers.

The benefit of the ODFRez data format is that it provides a cross-platform resource definition language. The initial geometry of widgets can be defined in ODFRez data, along with other data needed to define the initial state of widgets, such as the labels on buttons. It is rare that a platform-specific resource is required; in the SDK samples, the only occasion a platform-specific resource is needed is for image-based buttons. Some of the benefits of the ODFRez format are considered in the section entitled “Key concepts”, which also explores naming issues and new architecture introduced in the InDesign CS API such as selection suites, which are a clean way to factor user interface and ‘model’ code.

A concrete example of a user interface, taken from the Stroke panel, is introduced to provide a working example for some of the more abstract discussions.

This document also introduces the widget set available within the InDesign CS API. The reader should have some understanding of the user-interface programming model before reading.
The main intent is to provide a guided tour of the InDesign CS widget set and provide a framework for understanding how to use widgets in building a responsive user interface.

This document discusses the user interface programming model in some detail and introduces key concepts such as type-binding, which was the association between a boss class (e.g. kButtonWidgetBoss) and an ODFRez custom resource type (e.g. ButtonWidget). Recapping this information, the ODFRez type specifies initial state and properties of widgets, such as the text that would display on a static text widget. The boss class provides the behaviour, such as the event-handling model for a widget, and provides interfaces that allow the state of the widget to be changed. For example, the text on a ButtonWidget can be initially specified in the ODFRez data statements in the plug-in <project>.fr file, but the text can be dynamically changed by a plug-in using the ITextControlData interface that the boss class (kButtonWidgetBoss) aggregates.

Widget boss classes are generally straightforward to use once the programming model is understood; exceptions are those such as the tree view control (kTreeViewWidgetBoss), which requires a plug-in developer to provide their own custom implementations of interfaces to adapt their data model to that of the API tree-view.

**Key concepts**

**Design objectives**

Some of the design objectives for the user interface model were:

- Enable cross-platform user interface development,
- Create re-usable UI components with rich behaviours,
- Provide a well-factored design that separated data and presentation,

To meet these objectives, the engineering team created a design that provides support for re-use, with the possibility of customising widget behaviour and appearance, and encapsulation of platform dependencies.

Support for re-use is established through the use of widget boss classes, with cross-platform resource data defined in the ODFRez language that can be used to specify initial state and properties of the widgets. ODFRez was an artifact of the OpenDoc movement initiated by companies like Apple and IBM in the mid 1990s.

The key responsibilities of a plug-in developer are to write:

- Boss class definitions; for new boss classes or subclasses of existing widget boss classes, say. These would go in the top-level framework resource file.
- ODFRez custom resource type definitions; if widget boss classes are subclassed, then the ODFRez types associated must be subclassed too. These would also go in the top-level framework resource file.
- ODFRez data statements; this is required for defining the geometry and attributes of the widgets, along with localisation. Some of these would go in the top-level framework
resource file, if they are going to be used across multiple locales, and some would go in localised framework resource files (e.g. <whatever>_enUS.fr, etc.).

Client code may also be responsible for:

- Navigation between boss objects; for instance, finding the panel that a pop-up menu is attached to, and then locating a widget belonging to the panel. Navigation is much more straightforward within the user interface model than within the general InDesign CS document object model. For instance, it may be as simple as using an IWidgetParent interface, present on every widget boss class, and responsible for traversing towards the root of the widget tree, or an IPanelControlData interface, present only on container widgets to allow traversing to the leaves of the widget tree.

- Processing actions when menu items are executed, or keyboard shortcuts executed.

- Handling notifications about changes in widget state sent to interested observers. InDesign CS client code rarely if ever requires writing event-handling code; rather, observers process Update messages that specify how the state of the subject (e.g. a list-box) has changed.

**Idioms and naming conventions**

This section describes some programming idioms and naming conventions that are strongly recommended when writing plug-in user interface code. There are conventions that the plug-in developer should be aware of as they will be encountered within the public API, and those that should be actively followed when it comes to naming to minimise confusion and uncertainty about code intent.

Control-data interfaces on widget boss classes are predictably named. There are many interfaces that are named I<data-type>ControlData that will be encountered when working with controls; notifications of changes in the data model of a control is performed by the implementations of these interface.

It is helpful to be disciplined in defining symbolic constants for identifiers and it is strongly recommended to observe the following conventions:

- Ensure that constants like boss, implementation and widget IDs begin with k,

- Ensure that new interface identifiers begin IID_ and are uppercase throughout, and are declared in the interface ID namespace,

- Write k<name>Boss to define a new ID in the boss namespace, where name is ideally related to the boss class intent, e.g. kMyCustomButtonWidgetBoss.

- Write k<name>Impl to define a new ID in the implementation ID namespace,

- Write k<name>WidgetID to define a new ID in the widget ID namespace,

- Write k<name>Key to define a new string key in the global string-table

- Aim for regular relationships between implementation class names and identifiers, e.g. kMyPluginObserverImpl, associated with a C++ class named MyPluginObserver,

- Make ODFRez custom resource type names indicative of the boss class that they bind to in as regular a fashion, e.g kMyCustomButtonWidgetBoss bound to ODFRez type MyCustomButtonWidget.
Namespaces for strings and identifiers

This section introduces the namespaces that can be used in defining symbolic constants that are necessary for a user interface to function correctly. Along with following the appropriate naming conventions for the domain, it is essential to define symbolic constants in the correct namespace and be aware of the ill-effects of having constants defined which clash numerically.

It is helpful to manage namespaces carefully, since in addition to the boss class IDs and implementation IDs that are used throughout the API, there are also widget identifiers and string-tables consisting of key-value pairs for each locale of interest.

In some cases the namespaces are global- that is, each plug-in must ensure that any identifiers and strings that it creates are unique within the application. In other situations, the requirement for uniqueness is located at the plug-in level; that is, the identifier should be unique within the plug-in. For an additional discussion of namespaces, see the Making Your 1st InDesign Plug-In SDK tech-note.

Abstractions and re-use

This section highlights the major abstractions in the InDesign CS API, and the main strategy for code re-use within the user-interface API -- extending widget boss classes. It also reviews some of the fundamental material connected with InDesign CS programming that is required to understand this document.

There is a very high degree of code re-use within the user-interface domain, and the boss class hierarchies are particularly deep in this area compared to other application domains. Re-use within the widget API typically takes the form of re-use of boss classes by inheritance; in some circumstances, implementations from the API of particular interfaces can be re-used. It is not possible in the general case to predict whether an implementation of an interface on an existing boss class can be safely re-used; there may be implementation dependencies, such as expecting the container widget boss object to expose an IBoolData interface, say and in the general case it is not safe to attempt to re-use just one implementation from a given widget boss class. The recommended re-use policy is to extend an existing widget boss class, and override an interface if required by extending the implementation present on the parent boss class.

Interface identifiers are mapped onto implementation class identifiers through the boss class definition. A boss class at its simplest is a table mapping interfaces (by IID_<whatever>) to implementations (k<whatever>Impl); in this context, the IID_<whatever> is a number rather than a C++ type, although naming conventions used in the API ensure a degree of regularity in the interface to identifier mapping. Other documentation may use the word ‘boss’ freely, but this term can be confusing, since it is overloaded; a ‘boss’ may refer to;

- A type: an expression identifying a set of collaborating implementation classes. In this role it is a specification, a template for creating objects,

- or an object: something that is able to manage the instantiation of the objects that provide its behaviour. In this role, it is a chunk of executable code living within the run-time environment.

Widget boss classes and the user interface API in general can be confusing because there are widgets defined in the ODF resource language (ODFRez) and widgets in the boss class space often with closely similar names. For instance, kButtonWidgetBoss, and ButtonWidget; the
first is a boss class, which provides implementations that are responsible for the behaviour of a
given widget, and the second is an ODFRez custom resource type, which specifies only data for
initial state and properties of a control.

There is another source for confusion; there are helper classes in C++ and ODFRez custom
resource types with occasionally identical names (e.g. CControlView) but different
responsibilities. To avoid confusing these types, it is always necessary to consider whether
entities are from the code domain (C++) or from the data domain (ODFRez).

Widgets versus platform controls

This section describes the relation between InDesign CS widget classes and platform controls.
It is useful to understand that the widget boss classes provide a layer of abstraction over
platform-specific controls and provide additional capability beyond that delivered by the
platform-controls. The user interface model extends widgets to the platform-specific control
set, providing controls such as measure-edit boxes specialised for the domain of print-
publishing.

A widget boss class (potentially) encapsulates a platform control and provides additional
capabilities such as entry validation, a cross-platform API to query and set data values and
change notification. This is the principal benefit of the user interface model; it means that the
same code can be written to develop a plug-in user interface for Macintosh and Windows with
little or no attention required to platform differences.

A widget boss class can be associated with a platform control, as in the case of an edit-box;
however, there are some classes that have no direct platform equivalent or platform peers; for
instance, the iconic push-buttons are not bound to a platform control. It is typically not
necessary when writing client code to be aware of whether there is a platform peer control for
an API widget, and it is recommended that manipulation of the state of InDesign CS widgets
should be performed through the InDesign CS API and not through platform-specific APIs. In
addition, InDesign CS API specific patterns should be used to receive notification about
changes in control state (subject/observer).

It is worth recognising that widget boss classes provide more capability than the platform
controls provide. For instance, there exists a mechanism for controls to persist their state across
instances of the application; this is not default behaviour for platform controls. The integer
edit-box widget (kIntEditTextBoxWidgetBoss) provides additional validation capability not
typically provided by platform controls. Another key point about widget boss classes is that
they expose a cross-platform API and a uniform programming model on both Macintosh and
Windows platforms, providing true cross platform development of user-interfaces, although at
the cost of a modest complexity in the architecture.

Commands, model plug-ins and user interface plug-ins

This section introduces a common factoring in the InDesign CS plug-ins; plug-ins are
decomposed into “model” plug-ins and “user interface” plug-ins. The core capabilities of the
InDesign CS API are delivered by the required plug-ins, the majority of which would be
described as “model” plug-ins. These for instance deliver required pieces of architecture that
every client plug-in would need, or implement the document-object model at the heart of
InDesign CS. Much (but not all) of the application user interface is delivered by plug-ins that
are named <whatever>UI.apln, indicating that they are (a) user-interface specific and (b) not required plug-ins, since they are not ‘rpln’.

The user interface of a plug-in can be regarded as a means of parameterising command sequences which perform functions of benefit to an end-user, such as changing the document object model consistent with their intent. For instance, the “XML” required plug-in provides the core cross-media API (e.g. IXMLElementCommands, a key wrapper interface) and the “XMedia UI” plug-in creates the user-interface and drives the commands delivered through the XML required plug-in. Behind the majority of plug-in user interfaces, a command or command sequence will be executed when a widget receives the appropriate end-user event. Note that the UI plug-ins are typically not required plug-ins, whereas the plug-ins that deliver fundamental commands and suites to change document structure and so on are typically required plug-ins. There are some commands and some suites delivered by UI plug-ins, but in the main commands and suites come from the “model” plug-ins.

Commands provide for a means to encapsulate change, provide undoability and support notification of changes. The Command pattern is a well-known design pattern described in depth in [Gamma et al, 1995].

Commands also make use of the messaging framework, which allows observers (IObserver) to be attached that receive notification of change to the underlying model. It is frequently the case that notifications about commands will be received by observers associated with plug-in user interface components; for instance, the Layers panel receives notifications about documents being opened and closed through command architecture and updates its views accordingly. It is particularly convenient that the same design pattern (subject/observer) is used within the user-interface programming model, since widget boss classes all expose an ISubject interface, that can be observed by another boss object that implements IObserver.

Previewable dialogs are closely connected with commands; commands are processed as part of the preview behaviour. If the user cancels the action, the command sequence is aborted, rolling the publication back to the previous state. Providing preview capability may be an essential requirement in some client code.

Suites and the user interface

Selection suites are a very convenient way to package model-manipulation code that makes it almost trivial to write a user-interface to drive the code. As described above, you should factor the code such that the suite would be delivered by one plug-in, and the client code for your user interface that exercised the methods on the suite in another plug-in. For a more detailed discussion of Suites, see the Selection SDK tech-note.

Suites are particularly germane to writing user-interface code, since they make the process very much simpler. Whenever an end-user is required to manipulate document objects by making active selections, you should attempt to make maximum use of suites.

Finding widgets in the API

There are two main ways to inspect the widget set delivered by the API. One set is composed of the boss classes that are delivered by the Widgets plug-in -- see the API boss class index in the online documentation (and look under Widgets) for more details on these. There are also
ODFRez counterparts for many of these boss classes, and these can be found in the API header file Widgets.fh, which is written in ODFRez. If you do not understand why there are widget boss classes and ODFRez types that are similarly named read about the key concept of type-binding.

Each widget in the API is best understood in terms of the boss class that provides its behaviour. Although there is an ODFRez type named ButtonWidget, it is really the boss class named kButtonWidgetBoss that provides all the significant behaviour, although for simple widgets, you can use only the ODFRez type without worrying about the boss class behind it; a static text widget might be an example of this.

However, it is better to refer to a widget by the boss class than by the ODFRez counterpart. It is also best way to understand what the widget can do for you, since you can examine the interfaces that are aggregated on the boss class and if you know the semantics of how the interfaces are being used on that class, then you can obtain a good handle on what the widget is about. For instance, kButtonWidgetBoss exposes interfaces such as ITextControlData (which represents the label on the button), IControlView (its appearance), IBooleanControlData (its state). The button widget boss class inherits an ISubject from the class kBaseWidgetBoss, and this means that client code can attach an observer (IObserver) to receive notification about changes in button state. When a button is pressed by an end-user, the button widget boss object (kButtonWidgetBoss, say) notifies about the state change by sending attached observers a message along protocol IID_IBOOLEANCONTROLDATA, with some additional information to say whether its being pressed in or not. This is a pretty much complete picture of what a button in the InDesign CS API is about; that is, it can be understood largely in terms of the capabilities exposed by the interfaces aggregated on the boss classes and the semantics of how the interface is used on the specific class.

**Getting started with widgets**

The first time that you encounter the starting array of widgets that are present in the InDesign CS API, you will feel a certain amount of trepidation. If you inspect the API for the boss classes delivered by the Widgets.rpln (which gives the core set of widget boss classes) or view the API header file named Widgets.fh (which lists the ODFRez types corresponding to the widgets delivered by the API) you will find many dozens of widgets, some with long and elaborate names. To confuse matters further, some of the definitions are written in some relatively obscure resource language, named ODFRez (Object Definition Resource Language) that was co-developed by IBM and Apple. However, the widgets in the API break down into a few families that will be familiar to any developer, and some of the differences between widgets in the API are subtle and arcane, and likely to be of relevance only in highly specialised situations. The only things that are required when you are writing client code are choose appropriate widgets for the user interface, define some boss classes or add-ins, write some ODFRez data statements and any necessary ODFRez type expressions to create some code that gets notified when the state of the widgets change, and process those changes appropriately. This may all seem a little overwhelming at first, and if you have never written an InDesign CS user interface before, then the place to get started is the Getting Started material, which is very much geared towards entirely new InDesign CS developers.
The objective of this document is to help the developer understand how to decode widget boss class definitions, understand what the ODFRez counterparts are about, and determine how to receive notification about control events such as someone clicking on a widget on ‘your’ panel.

### How to get notifications about control events or changes

The first thing that you will want to do when writing client code is understand how to get notification about changes in control state; in simpler terms, you would like to know when buttons have been clicked, when keystrokes have been entered into an edit box, when the selection in a list-box has changed and so on. Within the InDesign CS API, the way to do this is to provide your own implementation of an observer (IObserver), that listens for changes in the state of a subject (ISubject). Since all widget boss classes extend kBaseWidgetBoss, which aggregates ISubject, then all widgets are observable by default. The very simplest way to get notification about changes in state, e.g. someone clicking on ‘your’ button, is to add an IObserver interface to the widget boss class of interest (e.g. kButtonWidgetBoss), and subclass the corresponding ODFRez type. This pattern has the virtue of simplicity but if you have any elaboration in your user interface, then the ‘Widget Observer’ pattern is recommended. This is where a single observer listens for changes in multiple widgets, and it is aggregated on the parent of all the widgets whose state changes are of interest.

If you have been accustomed to programming other user-interface APIs, you might expect that you should be extending the event handler interface; however, this is not required for the InDesign CS API and there are very few circumstances in practice where extending IEventHandler would ever be required.

---

### Frequently asked questions (FAQs)

**How do I write safe code and get diagnostic information?**

Methods that are prefixed by ‘Get’, or ‘Find’, such as IPanelControlData::GetWidget or FindWidget, do not increment the reference count, and the pointer returned shouldn’t be used as a constructor argument for an InterfacePtr. Methods that are named ‘Query’<whatever>, such as IDialogController:QueryListControlDataInterface can be used as constructor arguments for an InterfacePtr, since the constructor will try to call AddRef and the destructor Release on the encapsulated pointer.

Encapsulate tests for interface pointers that could be nil in a construct such as a do { ... } while(kFalse); block, breaking when a nil pointer is encountered rather than causing the application to crash, using the pattern below:

```c
do {
    // code here...
    ASSERT(iMyInterfacePtr);
    if(iMyInterfacePtr == nil) {
        break;
    }
    // more code here...
} while(kFalse);
```
Make use of statements such as ASSERT, TRACE model to check your assumptions. For instance if you have an IControlView interface, see the bottom of the page in the HTML-based reference documentation in the SDK to figure out which boss classes aggregate this interface in the core set. To enable Trace when you are debugging, you can set the Old Trace menu item to be checked, and choose to log to Notepad or the Debug Window.

**How should I separate logic and UI code?**

It is recommended to write a plug-in that implements commands and/or suites, and write a user-interface plug-in that drives these commands and/or suites. To make this explicit, suppose that one wishes to add a new feature \(<X>\) and drive this through a panel: it is recommended to write two plug-ins, one named \(<X>\) and the other \(<X>_Panel\) or \(<X>_UI\) say. This particular factorisation is employed widely throughout the application codebase. For instance, in InDesign CS, there is a plug-in named “Tables”, and a user-interface that allows manipulation of the underlying table models in documents, named “TablesUI”. The “Tables” plug-in delivers a combination of commands and suite implementations, whereas the TablesUI plug-in typically is a client of these commands, or works with the suites, rather than having many additional commands. An example of this separation is shown by the TransparencyEffects and TransparencyEffectsUI plug-ins in the SDK.

**How do I develop a plug-in user interface?**

The first recommendation is to take full advantage of the tools that are delivered in the SDK. If you have a good idea of what the user-interface is going to be, DollyXs will generate most of the boilerplate that you need, for menus, dialogs or panels, and can create an arbitrary number of menu items for you with little effort.

To write a user interface for an InDesign CS plug-in is a relatively complex task; it is typically necessary to write some C++ code, define some new boss classes, define new ODFRez custom resource types and define data in ODFRez. There are reasons for this complexity; there is a very strong connection between the user-interface programming model and the persistence model for InDesign CS. In addition, writing a cross-platform user-interface API is a difficult task, and it is difficult to shield developers from the inherent complexity of the task. Not only is the API cross-platform, it operates across multiple locales, and this adds more complexity. The steps that a developer of a plug-in interface will typically perform include:

- Discovering the widget boss classes and ODFRez custom resource types that are needed in the initial analysis phase.
- Working out what must be subclassed (both the widget boss class, and the ODFRez custom resource type) to achieve desired functionality, or if widgets can be used as-is; for example, static text widgets that display invariant text.
- Determining widget hierarchy and geometry; this consists of determining the containment relationships for widgets and their bounding boxes.
- Defining symbolic constants; for instance, constants for boss class IDs if there are subclassed widget boss classes, widget IDs, implementation IDs, and string table keys and translations thereof in the localised string tables for target locales,
- Defining new boss classes and associated ODFRez types if needed.
User Interface: Introduction

Frequently asked questions (FAQs)

- Creating ODFRez data statements to specify the initial states of the user interface elements and localised string-data.
- Implementing required interfaces, for instance, for the tree-view control, there are two interfaces that client code must implement.
- Writing observer implementations; code to handle Update messages from the change manager.

One objective of Developer Technologies is to take away some of the pain from this process, by creating developer tools that ease some of the boilerplate creation in this task list.

**When do I have to subclass widget boss classes and ODFRez custom resource types?**

In many occasions you won’t need to subclass a widget boss class, nor an ODFRez custom resource type. For instance, if all your controls are on a dialog and you are only interested in collecting the state of all the controls at the point when the dialog is being dismissed then you won’t need to subclass. The process of subclassing an existing boss class is typically performed to add an IObserver interface to a subclass of the widget boss class, to enable notification about changes to the data model of the widget boss object to be received. When an existing widget boss class is being subclassed, a new boss class should be defined in the boss type definition file- typically, named <project>.fr. In some more specialised cases, you have to subclass to provide your own implementation of required interfaces; for instance, for a tree-view control. You may also find yourself in the position of wanting to change the drawing behaviour, and wish to override the IControlView interface.

If a widget boss class is subclassed, then there must also be a new ODFRez custom resource type created that is bound to the new boss class by class ID. When existing ODFRez custom resource types are extended, define the new ODFRez custom resource types that should be added to the top-level framework resource file, typically named <project>.fr.

**How do I find the panel that a pop-up menu belongs to?**

IPalettePanelUtils contains a method for locating a given palette given a WidgetID. Note that the relationship between the pop-out menu and the panel has changed since 1.x. There are several SDK samples that show how to navigate from the menu boss object to the panel boss object (look for IPalettePanelUtils).

**How do I iterate over a container’s child widgets?**

A container widget is any that supports IPanelControlData. Given such an interface, it is only necessary to call GetWidget for the widget list of the panel control data; this will navigate over the immediate children of a container.

```c++
// Assume panelControlData is valid ptr
// on a container widget boss object
for(int i=0; i < panelControlData->Length(); i++)
{ IControlView * nextWidget = panelControlData->GetWidget(i);
  ASSERT(nextWidget); // Go ahead and use nextWidget...
}
```
To add new child widgets, given a pointer to an IControlView on the child widget boss object, you can use IPanelControlData::AddWidget.

**How do I find a specific widget in the hierarchy?**

The interface on container widget boss classes named IPanelControlData provides a mechanism to traverse the widget hierarchy in the direction of the leaves to seek for a widget by ID. You can then use the method IPanelControlData::FindWidget to obtain an IControlView pointer referring to the widget boss object sought.

**How do I show/hide a widget, or enable/disable it?**

Use the IControlView interface of a widget boss object; it has methods such as SetVisible and Enable, and their counterparts, which can be used for this purpose. These methods can be used to toggle the state of a widget boss object, such as its visibility, whether enabled, and there are many other methods on this interface that can be used, for instance, to vary its dimensions.

**How do I create an alert?**

An alert can be created by using the API helper class named CAlert, which is appropriate for warnings, information or error-type alerts. It can also be used to solicit a simple choice from the end-user. See “Alerts” on page 59.

**How can I build a basic interface to support a binary or ternary choice?**

Try creating a CAlert with two or three buttons. CAlert::ModalAlert returns the index of the button that was pressed, with ‘1’ for the left-most button, up to ‘3’ for the right-most button of three. See “Alerts” on page 59.

**How do I create menu-entries?**

The easiest way to do this is using DollyXs, with a template such as IfPanelMenu, which will let you create an arbitrary number of menu items. The principal requirement is to create a boss class that promises an implementation of IActionComponent, and some ODFRez data statements (ActionDef and MenuDef). You will also need to define some string-keys for the menu path components, and translations thereof. There are some boilerplate boss classes which the plug-in should deliver, created by DollyXs, which serve to automatically register the action components in a plug-in. The application framework will call the methods on the IActionComponent interface when the menu item is activated and at other points, e.g. if there is custom enabling specified in the ODFRez data statements.

**How do I create a dialog?**

The standard method is to create a new boss class that subclasses kDialogBoss, an exotically named panel boss class. It is conventional also to create other user-interface elements that enable the dialog to be shown- for instance, a menu component, shortcut or a button that brings up the dialog. You should use the DollyXs template Dialog to get a basic dialog; this will
give you code very similar to that in the SDK sample named BasicDialog. See “Dialogs” on page 63.

**How do I create a selectable dialog?**

The standard method to create a selectable dialog is to create a new boss class that is a subclass of kSelectableDialogBoss or kTabSelectableDialogBoss. The principal responsibility is to provide an implementation of the interface IDialogCreator and add an IK2ServiceProvider interface that returns a magic number identifying the boss class as a dialog service provider. See the BasicSelectableDialog SDK example plug-in for more information. See also “Selectable dialogs” on page 67.

**How do I add a panel to a selectable dialog?**

To add a panel to a selectable dialog, it is necessary to add a service that advertises itself as a panel-creator. It is required to provide an implementation of the IPanelCreator interface and make binding in ODFRez code to the dialog that the panel should be added to. See “Selectable dialogs” on page 67.

**How do I create a previewable dialog?**

A previewable dialog involves subclassing kDialogBoss; it should have a check-box widget with the well-known widget ID kPreviewButtonWidgetID. The preview sub-system allows will only commit the commands executed when the dialog is dismissed with a positive confirmation (OK, Done etc). Otherwise, a previewable dialog is an ordinary dialog. It requires no new interfaces to be added to the dialog boss class than would be required for a non-previewable dialog implementation, although some specialised implementation code is required. The SDK sample named TransparencyEffects has an implementation that shows how to make use of this feature. See also “Previewable dialogs” on page 66.

**How do I add buttons to a dialog?**

The typical process of adding buttons to a dialog is to ensure that there are at least OK or Done and Cancel buttons present. These type of buttons should make use of the widget boss classes named kDefaultButtonWidgetBoss and kDefaultCancelButtonWidgetBoss, bound to the ODFRez types named DefaultButtonWidget and DefaultCancelButtonWidget. The buttons should make use of the standard widget identifiers. See “Button widgets” on page 82.

**How do I add tips to a widget?**

There are two ways to add tips. These can either be defined entirely in the resource data, in which case they are static tips. Alternatively a widget boss class can be extended to override the base implementation of the interface ITip (on the boss class named kBaseWidgetBoss) if an entirely custom tip is required.
How do I add a check box to a dialog?

A checkbox can be added to a dialog by adding ODFRez data statements that use the ODFRez type named CheckBoxWidget. The client code should attach to the widget boss object on an AutoAttach message (when the dialog is shown) to the IObserver implementation, and detach on AutoDetach (when the dialog is hidden). Helper methods in CDIalogObserver such as AttachToWidget are useful in this respect. The client code should register for notifications along the IID_ITRISTATECONTROLDATA protocol. Update messages will be sent to this observer when the check-box is clicked. See “Check-boxes and radio buttons” on page 79.

How do I add a checkbox to a panel?

In the case where a check-box is added to a panel, it is recommended to create a new boss class that extends the boss class kCheckBoxWidgetBoss and add an IObserver interface to this new boss class. The AutoAttach message will get sent when the widget is shown, and AutoDetach will be sent when it is hidden.

How do I add radio-buttons?

The behaviour of radio-buttons is provided by kRadioButtonWidgetBoss, and they are bound to the ODFRez custom resource type named RadioButtonWidget.

Notification about changes in the state of the radio-buttons should be requested along the IID_ITRISTATECONTROLDATA protocol. There are numerous SDK samples that use radio-buttons. See “Check-boxes and radio buttons” on page 79.

How do I ensure that buttons in a group have mutually exclusive behaviour?

The answer is to use a cluster-panel widget. For example for a radio-button to inter-operate with other radio-buttons to ensure mutually exclusive selection, a collection of widgets of ODFRez type RadioButtonWidget would be defined as children of an ODFRez ClusterPanelWidget.

This can enforce mutually exclusive behaviour amongst a group of widgets (not only radio-buttons) that expose IBooleanControlData or ITriStateControlData interfaces. See “Check-boxes and radio buttons” on page 79.

How do I add a multi-line static text widget that scrolls?

The process of adding a multi-line static text widget to a panel and associating it with a scroll bar can be achieved almost entirely with ODFRez data statements. The ODFRez custom resource type named MultiLineStaticTextWidget or the ODFRez type DefinedBreakMultiLineStaticTextWidget can be associated with the ODFRez type ScrollBarWidget through widget identifiers. Unless notification of changes associated with the widgets are required, this should suffice to create a multi-line static text widget. See “Static text widgets” on page 76.
How do I add a text edit box?

There are edit boxes that provide highly specialised behaviour; for instance, edit boxes can be created with an associated nudge control, specialised for the display of text-measures such as points, or units such as degrees. If the edit-box is being added to a dialog, it is typically only necessary to use the correct ODFRez type and manipulate the edit-box widget through the utility methods on the helper classes CDialogController and CDialogObserver.

In the case where the edit box is being added to a panel, if Update events associated with Return/Enter being pressed are required, then a subclass of an existing edit box widget boss class is required, which should expose an IObserver interface. In addition, the associated ODFRez custom resource type should also be subclassed and bound to the new boss class. See “Edit boxes” on page 85.

How do I get changes associated with an edit box?

Attaching an observer to a subclass of one of the edit-box widget boss classes will provide for notification about changes when the end-user presses Return or Enter within the edit-control. It is also possible to get notification about every keystroke by setting one of the flags in the ODFRez data statement associated with the edit box definition. Inspect the any sample that uses an edit-box; look for *EditBoxWidgetBoss. See also “Edit boxes” on page 85.

How do I override an event handler?

The question that first would have to be answered is; when would one want to override an event handler? This is only required when adding highly specialised behaviour; for instance, to obtain all the keystrokes associated with an edit control. By default the application framework will notify an observer of an edit-box widget boss object with an Update message if a Return or Enter is pressed with focus in the edit-box control. See “Event handling: advanced topics” on page 56.

How do I add a drop-down list?

The widget boss class required is kDropDownListWidgetBoss; subclass this widget boss class and add an IObserver interface. There is a corresponding ODFRez type named DropDownListWidget. Changes to the data model can be observed by attaching to the interface named IStringListControlData on the widget boss object and waiting for Update messages to be sent to the IObserver interface. See “Drop-down lists and combo-boxes” on page 91.

How do I add a specialised combo-box to a panel?

Suppose that the requirement is to add a combo-box that displays measurements in points. The correct procedure would be to subclass the API widget boss class named kTextMeasureComboBoxWidgetBoss to add an IObserver interface, and attach to the widget boss object in the AutoAttach method of the observer implementation. Similarly, there would be a data statement in ODFRez in the localised framework resource file involving a subclass of the ODFRez custom resource type named TextMeasureComboBoxWidget.
The widget boss observer should listen for changes along the IID_ITEXTCONTROLDATA protocol. Be aware that you will have to subclass the combo-box when adding to a panel to get notifications on change, and be sure that the name you choose for the ODFRez subclass contains the name of the superclass. For instance, the name MyTextMeasureComboBoxWidget would be acceptable; the name MyWidget would not. See “Drop-down lists and combo-boxes” on page 91.

**How do I create a widget with a picture?**

There are several icon and picture widgets in the API. It comes down to a choice of whether an icon-resource has enough image information or whether a bitmap/ PICT image is required. In either case, a platform-specific resource has to be created to hold the image and binding made in ODFRez data statements. If icon resources are sufficient and no button-like behaviour is required, then the ODFRez type IconSuiteWidget can be used. If a bitmap or PICT image is necessary, then the ODFRez type PictureWidget is appropriate. You should inspect the PictureIcon sample for more information on this issue. See “Icon and picture widgets” on page 89.

**How do I change the picture displayed in an image widget?**

A good example of this is in the Layers panel, which changes the image associated with the Eyeball icon depending on whether the layer is visible or not. The trick is to acquire an IControlView interface pointer from an image widget boss object and call SetRsrcID on this interface to change the picture. See “Working with picture widgets” on page 91.

**How do I change the font of a static text widget?**

There is a static text widget that derives its behaviour from the boss class named kInfoStaticTextColorWidget, which is bound to the ODFRez type named InfoStaticTextColorWidget. Initial font commitments can be defined in ODFRez data statements and the font displayed varied through the IUIFontSpec interface on the widget boss object. See “Static text widgets” on page 76.

**How do I know when a dialog repaints?**

One way to discover when a dialog is being repainted is to subclass the IControlView interface on a panel that covers the area of interest. The IControlView::Draw message is sent on repainting. See “Dialogs” on page 63.

**How do I add a contextual menu to a plug-in?**

There are several contextual menus, one for each context that the application recognizes. The process of adding menu items to these context-sensitive menus is similar to that of adding normal menu items; specify in the MenuList resource where the items should go and what the contents of the menu entries should be by giving keys into the string-tables.
How do I set the minimum size for a resizable panel?

Override IControlView::ConstrainDimensions a resizable panel control view. The client code should define the upper and lower dimensions of the panel. See the SDK sample named SnippetRunner, in <sdk>/source/sdksamples/snippetrunner. See also “Resizable panels” on page 75.

How do I group widgets?

A widget is contained within another widget when it is present in the CPanelControlData list of another widget. If the grouping widget should be simply a frame, the ODFRez type GroupPanelWidget is appropriate. In the case of clustering of buttons such as radio-buttons, check-boxes or other buttons that should be mutually exclusive, a ClusterPanelWidget is the correct type. This does not draw a frame, so a group-panel widget would still be required if a visible frame for the collection of widgets were required. See “Panel widgets” on page 70.

How do I add a list box to a panel?

The starting point is to define a new boss class that extends the boss class named kWidgetListBoxWidgetBoss and add an IObserver interface to this boss class to enable the client code to catch notifications from changes to the list-box.

Another step would be to create a new ODFRez custom resource type that extended the ODFRez type WidgetListBoxWidgetN. See the WListBoxComposite sample in the SDK for an example of using a list-box on a panel; see <sdk>/source/sdksamples/wlistboxcomposite. See also “List-box widgets” on page 94.

How do I vary the set of widgets displayed?

The DetailControlListSize example plug-in illustrates how this can be achieved. The initial requirement is to add an IPanelDetailController interface to a panel boss class that will host the variable numbers of elements. See “Detail-controlling panels” on page 73.

How do I override the default draw behaviour?

To create owner-draw controls, it is necessary to override the IControlView::Draw method. However, the implementations of IControlView for the widgets in the widget set are complex and the helper class CControlView provides only a minimal implementation. It is usually necessary to have the implementation header for the existing implementation (and its ancestors) and subclass this implementation class.

How do I use a tree view?

The tree-view control is described in its own section (see “Tree-view widgets” on page 103), but the basic recipe is as follows;

1. Create a new ODFRez custom resource type for tree view widget that extends the ODFRez type TreeViewWidget. Define a resource of this new type and add it to the dialog or panel where the tree-view control will appear.
2. Create a new ODFRez custom resource type for tree node widget that extends the ODFRez type PrimaryResourcePanelWidget. Define a branch node and a leaf node resources based on this tree node widget type.

3. Define a tree view widget boss that extends kTreeViewWidgetBoss, in this boss, at least aggregate ITreeViewHierarchyAdapter and ITreeViewWidgetMgr.

4. Provide implementation for the ITreeViewHierarchyAdapter and ITreeViewWidgetMgr interfaces.

5. Initialize the tree view with ChangeRoot() of ITreeViewMgr also inform InDesign CS any tree-model change through methods of ITreeViewMgr.
User Interface Architecture

Abstract

A concrete example of a user interface is introduced to illustrate key concepts, such as type binding between widget boss classes/interfaces and ODFRez custom resource types. This chapter also describes how the user interface programming model is factored and discusses design patterns relevant to user interface programming. Some of the essential abstractions in the user interface API are introduced and the role of persistence in the user interface model is considered. The basics of writing an observer are discussed; observers are notified when an end-user interacts with the user interface of your plug-in. You need to implement your own observers to receive notification about end-user events; guidance is provided on when and how to implement your own observer.

Objectives

The objectives of this chapter are:

- Provide a concrete example of a user interface to illustrate key concepts.
- Describe the factoring of the user interface model.
- Identify key abstractions in the user interface API.
- Discuss relevant design patterns to user interface programming.
- Define the role played by persistence in the user interface model.
- Identify design patterns relevant to the user interface API.

User interface example

This section introduces a concrete example of a user interface for discussion purposes. The graphic below shows widget boss classes and the table of associated ODFRez resource types that follows shows the behaviour behind an element on the Stroke panel. This element lets an end-user vary the miter-limit associated with a line. It consists of a set of co-operating widgets that have relatively rich and subtle behaviour. There is a text input box (whose behaviour is provided by the boss class kIntEditBoxWidgetBoss), a pair of labels (behaviour provided by kStaticTextWidgetBoss) and a nudge control widget (kNudgeControlWidgetBoss) that increments or decrements values in the edit-control by a specified amount.
This screenshot shows two static text widgets displaying text “Miter Limit:” and “x”; the specific text these widgets display will be different across locales, determined by the localisation subsystem. The label widgets have behaviour that is provided by the boss class named kStaticTextWidgetBoss. The edit box has behaviour provided by the boss class named kIntEditBoxWidgetBoss. The edit-box displays a nudge control with behaviour provided by kNudgeControlWidgetBoss. Given the LocaleIndex, the application core determines which strings to display for each widget by looking in a locale-specific StringTable.

The edit box widget used in the example in Figure 20 is specialised for the input of integers (kIntEditBoxWidgetBoss); a warning message is generated for any other type of input. The nudge control widget (kNudgeControlWidgetBoss) is a cross-platform, API-specific widget that collaborates with the integer edit box. The nudge control widget allows increments or decrements of the value in the edit box.

The integer edit box encapsulates a standard platform native edit-box, but adds additional validation logic when accepting updates; that is, when the end-user presses Return or Enter with focus in the edit-box.

The nudge-control widget does not encapsulate a platform control at all; instead, it consists of two API iconic button-like widgets, although this detail is hidden from the developer of client code. It is an instance of the Facade design pattern.

To receive information about changes to the parameter in the edit-control, the implementation code attaches an observer to the edit-box widget boss object; this observer listens for changes to the data model of the edit-box. There are many examples in the SDK where an observer is attached to listen for changes in a control subject. Note that if an IObserver interface is aggregated on a widget boss class (i.e. added in by your code), then an IObserver::AutoAttach message is sent when the associated widget is shown and IObserver::AutoDetach is sent when the widget is hidden. At this point, you should do things like attach to or detach from the subject (ISubject). See the TableAttributes sample in the SDK, found in <sdk>/source/sdksamples/tableattributes. This sample uses a widget observer on the panel that observes all the widgets on the panel. This is a clean design to use for user-interface code, since it minimises the number of subclasses that need to be created.

The table below shows widget boss classes and associated ODFRez resource types that provide the behavior for our miter-limit example.
**Type binding**

This section describes the binding between widget boss classes or interfaces and ODFRez custom resource types for the example shown in Figure 20.

A key aspect of working with plug-in user interfaces is to understand what the binding between a widget boss class and an ODFRez custom resource type means, or the significance of the binding between an API interface and an ODFRez custom resource type. Once this is understood, it is possible to work with the user interface API quite freely, defining new types and even, in principle, entirely new widgets from the existing set.

Widget boss classes provide the behaviour behind widgets. This includes the capability to draw and manage internal state, and mediate interactions with the end-user. Boss classes implement sets of interfaces. The example in Figure 20 uses an int-edit box (kIntEditBoxWidgetBoss), which is a control specialised for the input of integers. The implementation of the IControlView interface on the boss class named kIntEditBoxWidgetBoss is responsible for drawing the edit-box. The implementation of the interface named ITextDataValidation on this boss class validates that the input is an integer.

---

**Table 18 Example widget boss classes and associated ODFRez resource types**

<table>
<thead>
<tr>
<th>API widget boss class</th>
<th>ODFRez custom resource type</th>
<th>Displaying</th>
</tr>
</thead>
<tbody>
<tr>
<td>kStaticTextWidgetBoss</td>
<td>StaticTextWidget</td>
<td>Miter limit</td>
</tr>
<tr>
<td>kStaticTextWidgetBoss</td>
<td>StaticTextWidget</td>
<td>x</td>
</tr>
<tr>
<td>kIntEditBoxWidgetBoss</td>
<td>IntEditBoxWidget</td>
<td>4</td>
</tr>
<tr>
<td>kNudgeControlWidgetBoss</td>
<td>NudgeControlWidget</td>
<td>(nothing)</td>
</tr>
</tbody>
</table>

Key behaviour of each control comes from the boss classes shown in the first column. The nudge control and the edit box interact in a different way; the nudge control is coupled to the edit box in ODFRez data statements and no additional C++ code need be written to have an edit box with nudge capability. The code behind the nudge control is delivered by the application’s required Widgets plug-in and can be safely re-used through the user interface architecture.

The widget boss objects encapsulate platform differences in controls and provide additional capabilities, such as validation for integer input in the case of the above. The widget boss classes expose interfaces that can be used to set and retrieve data values, register for notification of change, and modify the visual representation of the platform widgets, such as the size or visibility.

Being able to define the control in a cross-platform resource format is useful, but being able to query/set the data and register for notification on changes in the data values with a cross-platform API is tremendously valuable. Anyone who has ever tried to develop cross-platform or maintain a codebase across two or more platforms will recognise the benefit of having a user-interface API that is truly cross-platform.
ODFRez custom resource types are used to define data to initialise widgets and other elements needed for the interface. For instance, the CControlView field in the code example that follows specifies the initial location and dimensions of each widget, along with other properties such as its widget ID and visibility.

When a panel is shown for the first time, a set of widget boss objects are created by the application framework, and the lifetime of these managed by the framework. Each widget boss object is invited to draw itself to the display. A widget boss class provides the capability behind the user interface element drawn to the screen, and implements the user interface model.

For instance, there is a boss class named kStaticTextWidgetBoss which is bound to the ODFRez custom resource type StaticTextWidget. The ODFRez data defines the initial state of a widget the first time it instantiates. Thereafter, the state of the widget is restored from a saved-data database. This enables widgets to persist their state across instances of the application; this is the basis of the end-user being able to save the state of their working areas, in terms of the visible panels, their geometries and so on.

A frequently encountered type expression is an ODFRez custom resource type definition that refers to an interface ID or boss class IDs. An ODFRez expression such as ClassID = xxx (or IID = xxx) establishes a binding between a widget boss class (or interface in the API) and an ODFRez custom resource type.

For our example, some of the type bindings are illustrated below. The figure below shows one of the widget boss classes involved, the class named kIntEditBoxWidgetBoss which provides an API to an integer-specific edit-control. Note how the ODFRez is made up of fields (e.g. CControlView) that map onto interfaces on the widget boss class. There are also other interfaces on the widget boss class which do not map to fields in ODFRez; for instance, IEventHandler. The ODFRez fields specify the appearance of the control; they do not address its behaviour, which is the province of the widget boss class.

**Figure 21**  Binding between classes/interfaces and ODFRez types

![Diagram showing type binding between classes/interfaces and ODFRez types](image)

This diagram shows the type binding between the boss class kIntEditBoxWidgetBoss and the associated ODFRez custom resource type named IntEditBoxWidget, as well as
type bindings for each of the interfaces with persistent data exposed by kIntEditBoxWidgetBoss. Each persistent interfaces is bound to an ODFRez custom resource type that is used to define the initial state stored in the interface by a widget boss object.

The ODFRez fields in IntEditBoxWidget define the initial state for the integer edit box widget, whose behaviour is provided by kIntEditBoxWidgetBoss. Example 21 shows the ODFRez data statements for the widgets in Figure 20.

**EXAMPLE 21 Data statement for widgets in example**

```plaintext
// Miter Limit
StaticTextWidget
{
  // CControlView fields below
  kMiterStaticTextWidgetId, kPMRsrcID_None, // WidgetId, RsrcId
  kBindNone, // Frame binding
  Frame(0,30,58,47) // Frame
  kTrue, kTrue, // Visible, Enabled,
  // StaticTextAttributes fields below
  kAlignRight, // Alignment
  kDontEllipsize, // Ellipsize style
  // CTextControlData field below
  "Miter Limit:",
  // AssociatedWidgetAttributes field below
  kMiterTextWidgetId
},

IntEditBoxWidget
{
  // CControlView fields below
  kMiterTextWidgetId, // WidgetId,
  kSysEditBoxRsrcId, kStrokePanelPluginID, // RsrcId
  kBindNone, // Frame binding
  Frame(73,30,111,47) // Frame
  kTrue, kTrue, // Visible, Enabled
  // CEditBoxAttributes fields below
  kMiterNudgeWidgetId, // widget id of nudge button
  1, 10, // small/large nudge amount
  3, // max num chars( 0 = no limit)
  kFalse, // is read only
  kFalse, // should notify each key stroke
  // TextDataValidation fields below
  kTrue, // range checking enabled
  kFalse, // blank entry allowed
  500, 1, // upper/lower bounds
  "4" // initial value
},

NudgeControlWidget
{
  kMiterNudgeWidgetId, kPMRsrcID_None, // WidgetId, RsrcId
  kBindNone, // Frame binding
  Frame(59,30,73,47) // Frame
  kTrue, kTrue, // Visible, Enabled
```
User Interface Architecture

Type binding

StaticTextWidget
{
  kXTextWidgetId, kPMRsrcID_None, // WidgetId, RsrcId
  kBindNone, // Frame binding
  Frame(115,30,125,47) // Frame
  kTrue, kTrue, kAlignLeft, // Visible, Enabled, Alignment
  kDontEllipsize, // Ellipsize style
  "x",
  kMiterTextWidgetId
},

Note that although there are strings like Miter Limit: present in the ODFRez data, these will always be translated for display in the user interface, so the strings should be regarded as keys rather than as values to be displayed. The SDK string keys are defined with particular care using a scheme to avoid string-key clashes between plug-ins from different third party developers. The approach taken within the application is less structured and not recommended for third party development. The main points to note about the ODFRez data statements are that each is made up of fields of `simpler` types. For instance, CTextControlData has a single field, containing a string-key. The contents of this CTextControlData field are used to initialise the contents of the ITextControlData when the widget boss object is created.

That is, the objects that represent the widgets read their initial state from the compiled version of the ODFRez data. They persist their state to the saved-data database and read it back from this the next time the application starts, if they exist. Otherwise the widgets will read their initial state again from the plug-in binary resource. The tricky part to understand is the relation between persistent interfaces and the representations of widgets in the ODFRez data.

Type binding and behaviour

This section introduces another example of type binding, this time looking at how subclassing an existing widget boss class and adding (or overriding, more generally) an interface can create specialised behaviour.

To illustrate this process of type binding further, consider the following example. There is a very simple type of separator widget that exists in the API, which is used to draw a horizontal or a vertical divider or “separator” between panels. A screenshot of this widget is shown in Figure 22; it is used in contexts such as the Character panel. The behaviour is provided by a widget boss class named kSeparatorWidgetBoss. The main capability of this widget, drawing the line in the correct orientation, is provided by an IControlView implementation aggregated on the kSeparatorWidgetBoss.
This screenshot shows an example of a separator widget (kSeparatorWidgetBoss), which is used as a divider between panels. The most interesting aspect of the behaviour of this widget is drawing a line of the correct orientation, which is the responsibility of its IControlView interface. The implementation of this IControlView interface aggregated by kSeparatorWidgetBoss draws a vertical line if the widget is taller than it is broad.

The boss class named kSeparatorWidgetBoss extends kBaseWidgetBoss and provides an implementation of the interface IControlView. The implementation of the IControlView interface with identifier kSeparatorImpl draws a bevelled line within the frame of the widget. The implementation of this control-view extends the C++ helper class named CControlView. This should not be confused with the ODFRez type of the same name.

An end-user of this widget will typically not need to extend this boss class, so there is no need to define a new ODFRez custom resource type to exploit this application as a plug-in developer. Type expressions relating to the SeparatorWidget are shown in Example 22. These specify how the data in instances of a SeparatorWidget would be laid out.

**Example 22 Type definitions for separator widget**

```c++
// This is a macro so that it makes it easy for the
// Localization engineers to
// write a tool that can replace geometry easy.
/// They simply look for Frame(x).
// And replace x with the new geometry. [amb]
type Frame : PMRect { };
type CControlView : Interface (IID = IID_ICONTROLVIEW) {
    longint;// fWidgetId
    PMRsrcID;// fRsrcId, fRsrcPlugin
    integer;// fFrameBinding
    Frame;// fFrame
    integer;// fVisible
    integer;// fEnabled
};
type SeparatorWidget (kViewRsrcType) : Widget (ClassID = kSeparatorWidgetBoss)
{
    CControlView;
};
```

An instance of a separator widget being defined through ODFRez data statements is shown in Example 23.

**Example 23 Example of a separator widget**
Factorisation of the user interface model

This section describes how the user interface programming model is decomposed into key ‘aspects’ that are shared by most if not all widget boss classes.

Anyone who has programmed a user interface will recognise that it is often tedious and difficult to write a responsive user interface. There are also well-known differences in the platform user interface APIs between Macintosh and Windows. Since one key design objective for the user interface architecture was to provide a cross-platform API and interface definition format, it should not be a surprise that the architecture is relatively elaborate. However, at the core are simple and familiar concepts.

The user interface architecture consists of views, data models, event handlers and observers on the models. There are also attributes associated with the widget boss classes. The event handlers have code which changes the data models. When the widget data models change, the change manager is notified through the default ISubject implementation. It is the change manager which notifies observers of changes to the data model of interest. This abstraction is described elsewhere in connection with Commands and the Notification framework.

Views are connected with the visual appearance of a widget; they can be manipulated to change the visual representation of widgets such as the dimension of the bounding box or the visibility. They also encapsulate the process of rendering a view of the data model.

The only occurrence of an explicit controller abstraction is in the context of dialogs. Widget boss classes have no externally visible controller abstraction; code with an equivalent responsibility is encapsulated in the widget event handlers. It is misleading to assume that the MVC pattern is a complete description of the user interface architecture.

- Control-views: specifying the presentation of a widget, such as whether it is visible, its widget ID, whether enabled or not. The particular implementation that is present on a widget boss class determines how the control draws.
- Control-data models: encapsulating widget state; it is the control-data model implementation that typically notifies when it changes, so that observers on its state can get notified about the changes.
- Event handlers: these are responsible for converting events into changes to the data model,
- Attributes: equivalent to properties rather than user data, such as the point size that a text widget displays its label in; these are properties which may be defined in ODFRez data statements, although can typically also be set through interfaces aggregated on the widget boss class providing the widget behaviour.
There may also be other interfaces that are aggregated on boss classes, often related to details about an implementation of a particular widget. At its simplest, a widget consists of at least a control-view (IControlView). If it has a state that can be changed, then it will have a control-data model (e.g. IBooleanControlData, ITriStateControlData, ITextControlData etc). If the widget is responsive to end-user events, then it will aggregate an event handler (IEventHandler). It may also have some other property-related interfaces. Depending on the widget type, there can be additional interfaces to manipulate the control, and/or perform operations on data. For instance, the combo box widget boss class kIntComboBoxWidgetBoss has additional interfaces such as IDropDownListController (to manipulate the list component of the combo box) and ITextDataValidation, for performing validation on data entered in the edit-box component of the combo-box.

**Control-views**

Control-views are responsible for creating the visual representation of a widget boss class. For instance, the control-view implementation associated with a palette-panel widget draws a drop-shadow. The interface that allows control-views to be manipulated is named IControlView. This interface can, for instance, be used to show or hide a widget, or to vary other visual properties.

The key method for widget drawing is the IControlView::Draw method. A widget boss class implements this method to provide its default visual appearance; this method is called in response to system paint events or explicit requests to redraw. Any owner-draw widgets should override this method to provide a specialised appearance.

Views can also control their own size in response to end-user events. Overriding the ConstrainDimensions method on the IControlView interface allows a panel to resize to its container palette.

**Control-data models**

A control-data model represents the state of a widget and is responsible for notifying the application core of changes in its state. For instance, an edit box has a data model that is represented by a PMString. Changes in this state are likely to be of interest to client code in a plug-in.

The widget boss classes underpinning behaviour of radio buttons (kRadioButtonWidgetBoss) and check-boxes (kCheckBoxWidgetBoss) aggregate an ITriStateControlData interface. This interface can be used to query and modify the state of the check box; the possible states are checked, unchecked or indeterminate (mixed). The event handler interacts with the control-data to set the state. Client code may also be interested to set and query the control-data model state, and register for notification about changes to the state of the control-data model.

Typically, observers associated with widgets will request notification about changes along a protocol that is named IID_1<whatever>CONTROLDATA. This is the standard mechanism for client code to receive messages about changes in the state of a control.

Other APIs require that explicit event-handlers be written to process messages from controls. Coding event handlers is performed relatively infrequently when programming with the user-interface API; far more common is the requirement to implement observers.
Attributes

An attribute is a property of a control that is not an aspect of the data model but that can be used in defining its visual representation and perhaps other non-visual properties. For instance, a multi-line text widget has attributes including:

- The font ID that will be used in rendering text.
- The widget ID of the associated scrollbar.
- Leading between the lines expressed in pixels.
- An instance of a PMPoint specifying the inset of the text from its frame.

These attributes can be defined in ODFRez data statements. The dimensions of a widget (for instance) can also be defined in data statements but this is a property directly associated with the view. The data model represents the widget state that is likely to be varied by an end-user, such as the contents of an edit-box.

Event handlers

Event handlers are not as significant as one might at first assume when programming InDesign CS user interfaces. This is because the main pattern for processing end-user events is the Observer pattern; events are transformed into “semantic events” by the widget boss class’ event handler code, and it is these “semantic events” which are transmitted to the observer, as IObserver::Update messages with informative parameters. An end-user clicking the left mouse button when the pointer is over a button is a low-level event; it becomes a semantic event when interpreted as a “button press” that takes the control into a “button down” state. It is the semantic event rather than the low-level event which is of more interest to client code as it provides a useful level of abstraction over the specifics of how the end-user manipulates the state of different types of controls.

These are responsible for transforming end-user events into changes in the data model. The event handler therefore mediates between the end-user and the control’s internal state. It is important to be clear about the difference between an event handler and an observer. Event handlers are generally of little interest to client code, except in highly specialised circumstances when the standard behaviour of the control needs to be overridden. There is no need to code an event handler to be notified of events within a widget such as mouse clicks on a button if the semantic event of interest is the button press.

When an end-user clicks in a widget, events are transformed by the application core into cross-platform messages; for instance, if an end-user clicks on a button, the button event handler receives an IEvent::LButtonDn message. The responsibility of the event handler is to map these events into changes to the state of the control. It does this through interaction with the control’s data model. For instance, if a check-box is selected, the state is represented by the control-data interface ITriStateControlData as mentioned above. The event handler determines this state and calls ITriStateControlData::Deselect when the check-box is clicked in the selected state.
Relevant design patterns

Design patterns are “simple and elegant solutions to specific problems in object-oriented software design” [Gamma et al, 1995]. The objective of such patterns is to write code that is:

- **Flexible**: the code can be re-used from many different contexts.
- **Well-factored**: responsibilities of classes are not confused or confounded.
- **Easily comprehensible**: low barrier to understanding by new developers.
- **Easy to extend and maintain**.

To understand design patterns, it is necessary to be aware how objects in the pattern interact and how responsibilities are distributed between the classes involved. The intention of this section is to introduce some of the design patterns that can be found in the API related to user interface.

The rationale for introducing these patterns at this stage is to provide a clear outline of the design commitments made in the API. Some of the abstractions may be unfamiliar; however, they will be encountered time and time again in developing plug-ins. It is worth investing some effort in understanding these patterns. Some patterns of relevance and their domain of applicability are as follows:

- **Observer pattern**: event notification framework; for instance, changes in the control-data model (the subject) are notified to observers.
- **Chain of Responsibility**: applicable to the event handling architecture. The application event handler maintains a stack of event handlers, which are invited in turn to process the current event; if one doesn’t handle the event, the next one down on the stack is invited to handle the event. This is not identical with the pattern described by Gamma et al, but the intent is the same.
- **Command pattern**: command processing by the application core is an elaborated implementation of this pattern.

**Observer pattern**

The pattern is also referred to as Publish-Subscribe. This pattern is fundamental to the user interface model; every widget boss class aggregates an ISubject interface and is therefore observable.

The pattern is appropriate when:

- The situation being modelled has two aspects, one dependent on the other and it is desired to represent these in separate, lightly-coupled abstractions.
- Change to one object requires change to another but it is not known a priori how many other objects need to be changed.
- When one object wants to broadcast a state change to other objects registering an interest at that time.
The abstractions in the pattern are the Observer and Subject. The subject abstraction encapsulates state that is of potential interest to client code; for instance, the state of a check-box widget.

The Observer is interested in changes to the Subject; this is the direction of the dependency. The observing abstraction determines when to register with a particular subject and when to un-register.

The observer registers an interest in being informed when the subject changes. For completeness the pattern also specifies the ability for the Observer to send a message to the subject to state that it is no longer interested in being told about changes in the subject.

The Observer pattern defines simple abstractions and a straightforward protocol for communication between the subject and observer. The sequence would be as follows:

- The Observer sends an Attach message to the Subject, equivalent to registering for a mailing list by sending one’s e-mail address.
- A client or owner of the Subject sends it a Notify message to indicate that any registered observers should be informed of a change.
- The Subject sends an Update message to the Observer when a change occurs.
- The Observer can then query the subject for details of the Subject state it is interested in.

Changes in the state (data model) of widgets can be observed by objects derived from a helper C++ class (CObserver); this allows creation of a listener object that is notified when the data model associated with a widget changes.

**How the event handlers implement controllers**

Rather than events leading to observer notifications directly, there is an intermediate step in the user interface model. IEvent types, for example, do not convey the appropriate semantics to allow a listener to determine the meaning of an event- a mouse click on a radio button does not tell a listener enough about the action (selection or de-selection); the listener would be aware only that a particular mouse event had occurred. The missing data is the state of the widget.

The correct process is to attach to the data model of the widget and register for notification on changes in the data model. Rather than each individual observer having to maintain information about the state of the widget, this state is held in the control’s data model, and many observers can listen for changes in this model.

For instance, a radio button being selected or deselected leads to the widget boss object’s event handler changing the control data model, which in turn generates a call to the Update method in the observer. At this point the implementation-specific code can determine what operation to perform based upon the current state and the type of event; the parameters of the Update message can specify to an observer the new state of the control.

Model-view-controller or MVC is a well-known mutation of the observer pattern. The Model plays the role of Subject in the observer pattern. The View is equivalent to the Observer. The only new abstraction is the controller, which is implicit in the Observer pattern- it is the entity that mediates between the end-user (an event source) and the data model. The controller causes views to update after the data model is changed. The responsibilities of the elements of the MVC pattern are as follows;
The controller receives end-user input events, queries or updates the model, and forces the views to refresh with new data.

The model is a data container; it is protected from the end-user by the controller and encapsulates state of interest to the end-user.

The view consists of renderings of data supplied to it by the controller; typically the view cannot actively query the model but passively renders data.

The controller separates out the responsibility of dealing with user interaction from the entities responsible for rendering model information or maintaining model state. It becomes particularly useful when not all end-users have equal rights to change or query the model and it mediates between the users and the data (state).

This pattern is particularly useful when an application is involved in creating multiple renderings of the same data and keeping these synchronised across updates of the data.

**What role does MVC play in the user interface model?**

The application does not, strictly speaking, implement the pattern from SmallTalk’s Model-View-Controller. There are few explicitly-named abstractions in the codebase called <whatever>Controller; for instance, the IDialogController interface. In SmallTalk, there are classes in the class library that are called Model, View and Controller that are subclassed to build a user-interface. There is no direct equivalent in the InDesign CS API; MVC is an approximation to the architecture and is a reasonable conceptual (high-level) description of the architecture but not accurate at a more detailed level.

Event handlers are akin to controllers, since they act upon and change a model (the widget data model). The code that implements the widget data-model in turn sends out notifications via the change manager when the model is changed.

The event handler of a widget changes the data model of a widget boss object directly, without using commands. Note that a command sequence is always needed to change the native document model, which does not involve the user interface widget boss objects. Since the event handler sits between the data-model and the end-user, mediates changes in the model initiated by the end-user, and indirectly triggers notifications about changes in the data-model, it is in the same role as the Controller abstraction in the MVC pattern. That is, within the InDesign CS user-interface architecture, the data-model actively notifies the change manager about its change in state, rather than being a passive abstraction.

The data model of a widget boss object sends a Change message to the change manager through the default ISubject implementation, and attached listeners receive an IObserver::Update message.

This pattern will be encountered in the context of creating dialog interfaces by subclassing the partial implementation classes CDialogController and CDialogObserver. In the behaviour of most widget boss classes, the notion of an explicit controller will not be encountered directly and you can largely forget about MVC when it comes to writing user-interface plug-ins. Think in terms of the Observer pattern, and bear in mind that changes to the control-data model (Subject, represented by ISubject) of a widget will result in change notifications being sent to any registered observers (IObserver) on the abstract subject.
**Chain of responsibility**

This pattern is also called Responder or Event Handler. It is appropriate when:

- Multiple objects may handle a single request or event.
- It is not known a priori which event handler will be used for a specific event.

The key intent of this pattern is to allow multiple objects a chance to handle a request or event. This pattern is useful when writing event handlers for plug-ins, as event handlers are stacked by the application core, and events are chained between the event handlers. If one handler does not signify that an event has been handled, then the next event handler will receive notification of the event. The chaining will stop if one handler claims responsibility for having handled the event and no further event propagation would occur.

The user interface model does not implement the pattern exactly as specified in [Gamma et al, 1995]. In the API, there is an event dispatcher that takes responsibility for propagating the events rather than having each event handler explicitly aware of the next handler in the chain.

**Facade**

The intent of the Facade pattern [Gamma et al, 1995] is to provide a simplified interface to a complex subsystem. The abstractions in a facade pattern are the facade itself and subsystem classes. The facade knows which subsystem classes to delegate particular requests to. The suite architecture uses the Facade pattern; suites provide a simplified API onto potentially complicated selection-format specific code that has detailed knowledge of model structure.

Within the context of the InDesign CS selection architecture, the facade is the abstract interface of the suite itself (e.g. ITableSuite) and the subsystem classes are those such as <whatever>ASB and <whatever>CSB which add in implementations of the suite interface to the abstraction and concrete selection boss classes respectively. In the facade pattern, a client sends requests to the facade, which forwards them to the appropriate object in the subsystem. In the case of suites and the selection architecture, the client would be client code that handled, say, a menu item. The integrator suite would be responsible for choosing the correct implementation given the selection format and delegating the request to the correct implementation.

There are other examples of the facade pattern in the API; for instance, the nudge control widget (kNudgeControlWidgetBoss) and combo-box widgets (kComboBoxWidgetBoss), which are relatively complex widgets that expose a more restricted API to client code to allow it to manipulate their state and properties.

**Command**

The intent of the Command pattern [Gamma et al, 1995] is to encapsulate a request as an object; this allows clients to parameterize requests, enables requests to be queued and executed at different times, and can support an undo protocol. One major benefit of this pattern is that it decouples the object that invokes an operation from the object that knows how to perform it.

The key abstractions in the command pattern are the Client, who creates a Command, a specification of a parameterized operation. The Invoker executes the command subject to its scheduling preferences. A Receiver abstraction knows how to perform the command; for instance, an abstraction that is able to perform a copy on a document page item. The Receiver
is referenced from the Command, to allow the invoker to indirectly execute the required
operation(s).

Within the application, client-code (often user-interface code) takes on the role of the Client;
the Invoker is the command-execution framework of the application core. Client-code may
also provide the Receiver (or delegate to another abstraction within the API). The Command
abstraction at its most basic should support an execute method, and ideally also support an
undo protocol.

---

**Key abstractions in the API**

This section describes some of the most essential abstractions in the InDesign CS user interface
API to be familiar with when developing plug-in user interfaces.

**Interfaces**

**IControlView**

The IControlView interface is used to manipulate views, i.e. the visual representation of a
widget, such as its dimensions or its visibility. Every API widget boss class provides some
implementation of the interface IControlView interface, and the IControlView::Draw method
determines how it renders its appearance; this method is called by the application core after a
view has been invalidated in some way.

In some limited cases, it may be possible to override the implementation of IControlView to
provide a new type of owner-draw widget. Consult the on-line API documentation on
IControlView for more detail on an owner-draw control.

**ISubject**

The ISubject interface makes a widget boss object an observable entity. This means that any
widget boss class exposes an API allowing observer to attach and detach themselves. When
change occurs, typically in the data model of a widget, attached observers are notified by the
change manager. This occurs in the default implementation of ISubject, which is used by
widget boss classes by default.

A client of a widget boss object should call attach and specify which protocol they wish to be
notified along, and on which interface. A client of a check-box widget boss object would be
interested in notification along the protocol IID_ITRISTATECONTROLDATA, for instance.

The parameter named asObserver in the formal parameter list for AttachObserver is relevant
in the condition where a boss class is to be used for observing changes in more than one entity
and wishes to be called by the change manager along a different interface for each change.
Although one observer class can listen for many different changes, there are circumstances
where it is more appropriate to partition responsibility for listening to different types of
changes to different implementation classes. These would all implement IObserver but have a
different interface identifier in the boss class definition and attach with a different asObserver
actual parameter value.
**User Interface Architecture**

**Key abstractions in the API**

**IPanelControlData**
This interface is found on container widget boss classes. The IPanelControlData interface can be used to traverse the widget tree in the direction of the leaves. Navigation through the child widgets on a panel is possible with methods on IPanelControlData interface such as GetWidget or FindWidget. It is aggregated on container widget boss classes (panels, dialogs etc) and is the designator interface for containers.

The widgets are held in the panel control-data in an order that determines how they are drawn by the application core. The widget at index zero is drawn first.

**IWidgetParent**
The IWidgetParent interface allows the widget tree to be traversed in the direction towards the root. The key method on this interface such is QueryParentFor which allows a query for an interface in the direction towards the root of the widget hierarchy. Note that the layout widget boss class named kLayoutWidgetBoss, which provides a view of documents, also exposes this interface. However this is well outside the scope of this documentation.

At the root of a typical widget hierarchy (associated with a panel or dialog) will be a window boss object. Given an interface pointer referring to one widget boss object, it is possible to walk up the widget hierarchy until the root is reached, querying for a particular interface. For instance, an implementation with a dialog might place a custom data interface on a subclassed kDialogBoss class and locate the interface by calling IWidgetParent::QueryParentFor on any widget on the dialog.

**IEventHandler**
The IEventHandler interface exposes an event handling API. The event dispatcher in the application core will dispatch events to a widget's event handler subject to its own logic. Briefly, it exposes methods such as LButtonDn, which is called by the application core when a mouse-button down event occurs in the frame of a widget. The typical responsibility of an event handler in the API widget boss classes are to change the data-model in response to events. Most of the time, client code will be happy to use the observer pattern to receive notification about changes in control state and will not need to delve into the specifics of event handlers. On the rare occasions that overriding an event handler is required, the most useful pattern is to use a “shadow” event handler or proxy, as illustrated in the PanelTreeView sample plug-in, which can be found in <sdk>/source/sdksamples/paneltreeview.

**Partial implementation classes**
Numerous helper classes in the API are relevant to building plug-in user interfaces. Typically they provide a partial implementation of a key interface, although sometimes the implementation is a bare-minimum in terms of functionality. A selection of helper classes are described in Table 18.
### Table 19 User interface helper Classes

<table>
<thead>
<tr>
<th>Class name</th>
<th>Interface implemented</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CActionComponent</td>
<td>IActionComponent</td>
<td>Partial implementation of methods needed to participate in menu subsystem</td>
</tr>
<tr>
<td>CAAlert</td>
<td>(nothing)</td>
<td>Static methods to display alerts, for warning errors or information</td>
</tr>
<tr>
<td>CControlView</td>
<td>IControlView</td>
<td>Minimal implementation of an interface related to appearance</td>
</tr>
<tr>
<td>CCursorProvider</td>
<td>ICursorProvider</td>
<td>Helps in creating custom cursors for widget boss objects</td>
</tr>
<tr>
<td>CEventHandler</td>
<td>IEventHandler</td>
<td>Provides minimal implementation of an event handler, useful in limited circumstances for simple controls</td>
</tr>
<tr>
<td>CDiallogController</td>
<td>IDialogController</td>
<td>Provides core functionality for working with dialogs</td>
</tr>
<tr>
<td>CSelectableDialogController</td>
<td>IDialogController</td>
<td>Adds functionality specific to selectable dialogs, e.g. the global preferences dialog.</td>
</tr>
<tr>
<td>CDiallogObserver</td>
<td>IObserver</td>
<td>Provides helper methods to ease working with and observing collections of widgets on dialogs</td>
</tr>
<tr>
<td>CSelectableDialogObserver</td>
<td>IObserver</td>
<td>Adds functionality for selectable dialogs</td>
</tr>
<tr>
<td>CObserver</td>
<td>IObserver</td>
<td>Minimal implementation of an IObserver interface- usually override Update method</td>
</tr>
<tr>
<td>CDiallogCreator</td>
<td>IDialogCreator</td>
<td>Used in creating selectable dialogs</td>
</tr>
<tr>
<td>CPanelRegister</td>
<td>IPanelRegister</td>
<td>Used for any panel-based plug-in to register panel boss object with application run-time</td>
</tr>
<tr>
<td>CPanelCreator</td>
<td>IPanelCreator</td>
<td>Use in adding panels to selectable dialogs</td>
</tr>
</tbody>
</table>

### Utility classes

There are utility classes such as IPalettePanelUtils and IWindowUtils which are interfaces on kUtilsBoss. Access by writing code like this:

```cpp
IWindow* window = Utils<IWindowUtils>()->GetFrontDocWindow();
```
**kBaseWidgetBoss**

The class named kBaseWidgetBoss is the ancestor for all API widget boss classes. This base widget is effectively abstract - it is ancestor for anything that can be referred to as a widget. Note that it has no IControlView interface, so by default would have no appearance and could not be accessed through an IPanelControlData interface on a container. The semantics of the interfaces aggregated on this key class are explained below

- **ISubject**: client code can attach to the widget boss object on this interface and hence request notification about changes to the data model of the widget boss object.
- **IWidgetParent**: for navigation through the widget tree towards the root as mentioned above,
- **IPMPersist**: this allows the widget to read its state from the plug-in resource/saved-data and write its state back to saved data,
- **ITip**: this allows a tip to be defined in ODFRez data statements.

This set of interfaces defines the basic capabilities of any widget; it is observable, located in a hierarchy, persistent and may have a help tip.

The IPMPersist interface is required because all widgets have some initial state defined in ODFRez data statements. The ReadWrite method on this interface is called during initialisation of widget boss objects, when an object is being created by reading its initial state from a plug-in resource or saved-data database. If there is no saved-data, the widget boss object is initialised from the plug-in resource. If there is saved-data, then any persistent data associated with the widget will be read back from the saved-data. This includes parameters such as position, and size for resizable elements such as resizable panels.

**Boss class kSessionBoss**

The top-level object in the boss object tree is from the class named kSessionBoss. There is a global session object named gSession (ISession*, referring to singleton kSessionBoss), that is the usual starting point in navigating the object hierarchy. The commonest path when working in the UI domain is to obtain a reference to the application boss object, which is an instance of the kAppBoss class. There are many examples in the SDK showing how to navigate between kSessionBoss (gSession) and kAppBoss, and beyond.

**Boss class kAppBoss**

The application boss class named kAppBoss exposes several interfaces relevant to writing user-interface code. An instance of this boss class is accessible through gSession, a global object of class ISession. The navigation diagram below shows how to obtain other key objects from the application (kAppBoss) such as the palette manager (kPaletteManagerBoss).
The dialog manager is an abstraction for working with dialogs (IDialogMgr). Another interface exposed by the kAppBoss class which will be encountered when developing user interfaces is IProgressBarMgr.

The cursor manager (ICursorMgr) is an abstraction that will be encountered when developing custom cursor; it is aggregated on the boss class named kAppBoss.

**kPaletteManagerBoss**

There is an abstraction named the palette manager responsible for maintaining and manipulating collections of palette windows in the application. Its behaviour is provided by a boss class named kPaletteManagerBoss. This exposes interfaces such as IPaletteMgr and IPanelMgr. The panel manager interface (IPanelMgr) can be used to manipulate the collection of panels in the palette windows. Note that multiple panels exist within a palette window container.

### Persistence and widgets

One key to understanding the user interface architecture is being aware of the function of the IPMPersist interface that occurs on all widget boss classes. Widget boss classes that possess this capability can read their initial state from a plug-in resource or saved-data. Briefly, the IPMPersist interface means that a widget boss object can read its initial or stored state for any persistent interfaces exposed by the widget boss class. An interface is persistent if and only if it
is declared with the macro CREATE_PERSIST_PMINTERFACE, in which case it should implement ReadWrite.

The initial state of a widget is created by reading data from the plug-in resource in the very first instance. The persistent interfaces on a widget boss class, such as IControlView, will read their initial state from the plug-in resource.

There is a simple rule to understand which interfaces should be persistent in a widget boss class—at least those which are bound to ODFRez types should be persistent, otherwise there is no way for them to read their initial state from the binary data in the plug-in resource.

Consider the earlier example with the integer edit box and the nudge control that is shown in Figure 20. There are four interfaces bound to ODFRez types (see Figure 21), and other interfaces on this boss class which are not. Table 20 shows some interfaces on the boss class, whether they are persistent, and whether they are bound to an ODFRez type.

**Table 20 Interfaces with persistence and binding info**

<table>
<thead>
<tr>
<th>Interface</th>
<th>Implementation ID</th>
<th>Persistent?</th>
<th>Bound to ODFRez type?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ITextControlData</td>
<td>kEditBoxTextControlDataImpl</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>IEditBoxAttributes</td>
<td>kEditBoxAttributesImpl</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>IControlView</td>
<td>kNudgeEditBoxViewImpl</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ITextDataValidation</td>
<td>kIntTextValidationImpl</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ITextValue</td>
<td>kIntTextValueImpl</td>
<td>Yes **</td>
<td>No</td>
</tr>
<tr>
<td>IEventHandler</td>
<td>kEditBoxEventHandlerImpl</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>IObserver</td>
<td>kCNudgeObserverImpl</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>(other interfaces</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>omitted deliberately)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

At least these four are persistent; there is also an ITextValue interface, marked in the table with **, which is specified as persistent in the implementation code, but is not bound to any specific ODFRez type. This interface provides an API to read and write formatted values. This means that this interface cannot be initialised by ODFRez data statements.

In theory, a third-party developer can create an entirely new widget boss class, deriving from kBaseWidgetBoss and providing the implementation of required interfaces such as IControlView and IEventHandler. Extreme care must be taken to ensure that the specification of the fields in the xxx.fh file matches the order in which the data is read/written in the ReadWrite method of any persistent interfaces on the new widget boss class. The key point to remember is that a binding between an ODFRez type and an interface ID means that the (implementation of) the interface must be persistent.
Plug-in resource roadmap

The typical plug-in consists of:

- Top-level framework resource files (.fr); these contain boss class definitions, new ODFRez custom resource type definitions and other resources, such as view definitions and non-translated string tables.
- Localised framework resource files (postfixed by e.g. _enUS.fr), containing the ODFRez data statements required to define the plug-in user interface on a per-locale basis.
- C++ code that implements the interfaces promised in the boss class definitions.

The framework resources are compiled using the ODF resource compiler (ODFRC). The C++ code is compiled by the appropriate compiler for the platform. Note that the boss class definitions are the starting point to understand a new codebase; these specify the subclasses for the new boss classes that the plug-in adds to the API, and promises implementations of interfaces.

Compiling resources

One important concept to understand is how the files that make up a plug-in project are compiled. The following tools are involved in compiling resources:

- Windows; ODFRC.exe - Windows executable version of the ODF resource compiler that produces Windows binary resources,
- Windows; RC.exe - platform native resource compiler, which compile the RC file present in a plug-in and any platform specific resources such as icon definitions,
- Macintosh; ODFRC - plug-in for Metrowerks CodeWarrior.
- Macintosh; Rez, the platform native compiler, which would compile any platform specific resources such as icons in .rsrc or .r files,

The same (header) file may be compiled with both the C++, ODFRC compiler and the platform native resource compiler; this is achieved by having macros. Some of the key macros can be found in CrossPlatformTypes.h, with core data types defined in CoreResTypes.h.

OpenDoc Framework (ODF) Resources

The ODF resource language or ODFRez was chosen as a cross-platform solution for defining user-interface resources. ODFRez is based upon Rez, Apple’s resource utility that is available with Apple MPW. ODFRez differs from Rez in small respects; it is intended as an object-oriented, cross-platform resource definition format, and it is case-sensitive.

The ODF resource compiler, ODFRC, compiles files written in the ODF Resource Language (ODFRez). The ODFRez language has a very simple grammar; its main intent is to provide a comprehensive way to define resources and not write programs. There is support for other-language code to be embedded, but it is limited; only constant C expressions can be embedded in ODFRez files.
ODF files consist typically of two types of statements, trivially identified by the first word in the expression;

- Type statements.
- Resource data statements.

Examine again the definition of the separator widget, repeated in Example 24 with some annotations. This expression can be read as a definition of the ODFRez custom resource type SeparatorWidget, that extends the ODFRez custom resource type named Widget. As mentioned previously, a ClassID field is initialised to the value of kSeparatorWidgetBoss, specifying the class that provides the behaviour behind this widget.

**Example 24 Annotated type definition for separator widget**

```plaintext
// The ODFRez type expression defining that the SeparatorWidget extends the
// base type, Widget, and has a field of type CControlView.
type SeparatorWidget (kViewRsrcType) :
// note that this is a view-resource type Widget
// superclass for this type - the base ODFRez Widget
(ClassID = kSeparatorWidgetBoss)
// ClassID is a field of Widget, bound here to a boss class kSeparatorWidgetBoss
{   
    CControlView; // field belonging to the SeparatorWidget type
};
```

ODF resources are created in one or more files called ODF resource definition files. These files should have .fr extensions, and are text files. There may also be platform-specific resources associated with a plug-in, such as icons, PICT or Windows -bitmap resources.

---

**Top level framework resources**

This section describes ODFRez custom resource types defined in the top-level framework resource file. The discussion begins with resources related to defining a panel. Panels are containers for widgets that are housed in palettes. Briefly, they enable panels to be ordered, dragged around, shown and hidden. An example of a panel contained in a palette is the Stroke panel, shown below with a tabbed palette containing a single panel, consisting of the part below the tabbed element.

**PanelList resource**

The root panel for a plug-in will be defined in a PanelList resource. The ODFRez custom resource types named PanelList specifies what plug-in ID the panels are associated with, the name of the palette to put the panels in, a resource ID to use in loading the panels, and how each panel interacts with the menu subsystem.

To understand what the PanelList is about, it is important to be able to distinguish between the ODFRez PanelList type (the template, just like a C++ class declaration) and an ODFRez data statement defining an instance of a PanelList. To make this fully explicit;

- The template for PanelList is prefaced by type.
Localising framework resources

A rule-of-thumb is that strings displayed in the user-interface elements, outside of the layout widget (the document view), are likely to require localisation. From a programming perspective, the entities to consider when localising are:

- View resources; the geometry of a panel may change, in for instance a language such as German, where the strings are on average longer than English.
- Strings; for instance, those displayed on dialogs and panels as labels, and on menus as the names for menu items.

For each of these entities there should exist an ODFRez LocaleIndex custom resource, which provides the offsets that the application framework needs to switch in the localised data for a particular locale. There should also be an appropriate ODFRez StringTable or view resource in the localised framework resource files to match those promised in the ODFRez LocaleIndex statements.

The recommended string class to use in the API for most purposes is PMString. The application architecture will attempt to translate all PMStrings for display in the user-interface unless they are explicitly marked as non-translatable, either by including in the non-translate string-table (which should be very small in general) or by calling a SetTranslatable(kFalse) on a PMString before use. A plug-in developer should provide a translation for every string likely to be displayed in the user-interface in locale specific string tables.

Localisation and LocaleIndex resources

The mechanism of localisation is extremely straightforward using the application architecture. The key concept is to manipulate string keys (keys into the StringTable for a locale) rather than thinking in terms of strings as values.

The LocaleIndex resource is a look-up table that specifies how to locate a particular resource given the locale. A LocaleIndex resource should be declared:

- For each view that is to be localised.
- For the localised string table.
- For the non-translated string table.

A common mistake when adding in new resource statements for views is to forget the LocaleIndex associated with the view. The type expressions and the data statements for the
view may be perfectly formed, but without the LocaleIndex resource telling the localization subsystem which view to choose for which locale, the widgets corresponding to the view resource will not appear. The LocaleIndex type is defined in the SDK header file named LocaleIndex.h. The ODFRez LocaleIndex type is a template for defining resources that will enable the application core to choose the correct views and strings given the current locale-setting.

When to subclass a widget boss class and its ODFRez counterpart

A common reason to add a new interface to an existing widget boss class is to create a new widget boss class that exposes an IObserver interface. This is the pattern for obtaining notification about changes to the data model of a particular widget boss object. If you have lots of widgets on a panel, use the 'Widget Observer' pattern described below, otherwise you will risk code bloat and unnecessary proliferation of entities that may perhaps lead to problems later.

For instance, suppose that a developer has written a new type of panel with a single list-box upon it. When an end-user makes a new selection in the list-box, client code should receive an Update message on the IObserver interface; the parameters of the Update message specify the type of change that has occurred. In this instance, the developer would subclass the boss class named kWidgetListBoxWidgetNewBoss, and extend the ODFRez custom resource type WidgetListBoxWidgetN, binding it to the new boss type.

For controls on dialogs, there in general is no need to subclass a widget boss class for each of the controls to get notification about events. There can be single observer for all the controls on the dialog, which can choose to attach to particular controls on the dialog if notifications about changes in data are required before the dialog is dismissed. The helper class CDialogObserver (partial implementation of IObserver) provides an API that is extremely useful for attaching and detaching to controls on a dialog. It should also be used as the basis for an override of the IObserver interface on the class kDialogBoss. If you use DollyXs to generate the boilerplate for a dialog, then you will find that by default you already have an implementation of an observer that gets notification about all the controls. This is really just an implementation of the 'Widget Observer' pattern below, which merely generalises the notion to an arbitrary parent widget type.

Widget observer pattern

This is a specialisation of the Observer pattern. It applies if you have many controls on a panel, and you want to get notifications about changes in their state, but don’t want to subclass every single widget boss class involved just to aggregate an IObserver on each. Rather than having to subclass all the controls that you use on a panel, use a single widget observer that is aggregated on the parent panel boss class; use the observer implementation to attach to and detach from the ISubject interface on the widgets of interest when this widget observer is sent IObserver::AutoAttach and IObserver::AutoDetach messages.
There is an extension of this pattern for use when you also want to observe changes in the active context, which is very common in writing panels that may be constantly present and provide some form of read-out of some subset of application state. This extended pattern for both receiving notification about changes in control state on a panel and notification about changes in the active context might be termed the ‘active selection and widget observer pattern’; it requires you to aggregate a standard API implementation of IControlViewObservers on the panel widget boss class. The corresponding ODFRez data statements are sufficient to create all the wiring for the observer-implancements to be auto-attached and detached. In the old 1.x architecture, there was no way to do this, and it would have been necessary to create a dummy observer that simply called AutoAttach on other observers on a given boss object.

**IControlViewObserver**

Adding a widget observer and active selection observers on a single panel boss class, and using IControlViewObservers mechanism to wire these up in ODFRez data is a very common pattern in the InDesign CS application codebase. For instance, panels such as the Character panel which observe changes in the active context and update states of the widgets, whilst simultaneously monitoring for changes in the state of many combo-box widgets and so on, use a widget observer and an active selection observer, hooked up by IControlViewObservers. Example 25 shows the ODFRez expressions used in the TableAttributes SDK plug-in to add a widget observer, an active-context observer, and control-view observers. This may not seem startling unless you have already tried to have two observers aggregated on the one boss object auto-attached in the 1.x API and you’ll recognise how useful this is.

**Example 25 Defining observers**

```cpp
/** IControlViewObservers is aggregated to attach the panel’s observers. */
IID_ICONTROLVIEWOBSERVERS, kCControlViewObserversImpl,
/** ActiveSelectionObserver implementation */
IID_ITBLATTSELECTIONOBSERVER, kTblAttSelectionObserverImpl,
/** CObserver implementation */
IID_ITBLATTWIDGETOBSERVER, kTblAttWidgetObserverImpl,
```

You then have to make sure that you add the field to the type statement for the ODFRez panel widget: see Example 26.

**Example 26 Subclassing a panel**

```cpp
/* // Type definition. */
type TblAttPanelWidget(kViewRsrcTyp e) :
    PalettePanelWidget(ClassID = kTblAttPanelWidgetBoss)
    {
        CPanelMenuData;
        CControlViewObservers;
    };
```

Right at the bottom of the panel ODFRez data statements where you normally just have the panel-menu name, you add some more fields: see Example 27.
EXAMPLE 27 Defining observers for a panel

kTblAttInternalPopupMenuNameKey // Popup menu name (internal)
{
    // CControlViewObservers - the panel’s observers.
    {
        IID_ITBLATTSELECTIONOBSERVER,
        IID_ITBLATTWIDGETOBSERVER,
    }
}

This means that you have added more than one observer to the panel, without the traditional problem in the 1.x API of having to use one observer’s AutoAttach to have to attach yet another observer (and remember to do the same thing in AutoDetach).

Event handling: advanced topics

Writing a proxy event handler

The Subject/Observer pattern in the user interface model means client code can be notified about changes to state of widgets; mostly, this is sufficient for client code to be responsive to end-user events. This pattern simplifies client code and avoids client code having to turn low-level events (e.g. a mouse-move or button press) into semantic events (a check-box widget becoming checked). Even single keystrokes can be notified to an observer of a text widget, so the granularity of notification is potentially quite fine.

The event-handler implementations in InDesign CS/InCopy CS user-interface may have an inheritance hierarchy that is several levels deep. In most cases, it is not possible to subclass the implementation by straightforward C++ techniques, as the bulk of event-handler implementation headers are private.

There are specialised circumstances when some knowledge of the event-handling model can be useful and the ability to override the default event handling for a control essential. An example might be where there is a requirement to specialise the event handling behaviour of a particular control; for instance, to perform some special function in response to a double-click, say. The challenge is that most of the event handler implementation classes are not public, as these may involve quite deep hierarchies of implementation classes that are not public. It breaks the encapsulation in the user-interface API if client code comes to depend on this code. Fortunately, there is a convenient work-around for this by using a proxy or delegate pattern.

Watching events

The concept of overriding the default event handler for a control using the proxy technique, which is often all that would be appropriate, implies handling a plethora of messages. It is very often the case that only a handful of messages are of interest and a technique with more precision is desirable. There is an abstraction called an event watcher (IEventWatcher) that can target specific events, and a particularly useful partial implementation class, CIdleMouseWatcher, that can provide a convenient source of information about MouseMove events without needing to register every MouseMove event associated with a control.
For instance, the JPEG Export dialog has a feature that allows a description of the widget over which the mouse pointer is hovering, based on this `CIdleMouseWatcher` partial implementation. This implementation class can be used to create a mouse-event watcher, without having to use a proxy event handler pattern.
Abstract

This chapter describes the widgets available and provides details on their design and implementing user interfaces with them.

Objectives

The objectives of this chapter are:

● Build on the material introduced in the User Interface: Introduction and Architecture chapters.
● Identify some common widgets.
● Describe how they are used within the application.
● Provide some information on the widget design.
● Define the widget data model for each widget, if relevant.
● Indicate how a widget notifies observers about changes in its data model, if relevant.

Alerts

Description

Alerts are one of the most straightforward user-interface elements in the API to create from client code. They can provide notifications to an end-user, a warning or a notification about an error. They can also be used to solicit a response from the end-user, such as Yes/No choice or some other response.

The functionality to create and work with alerts comes from the CAlert helper class. The CAlert class is simple to use, involving static methods with many default arguments. Example 28 shows how to create a warning alert from client code.

EXAMPLE 28 Displaying an alert

PMString string(kWarningMessageKey);
// Translation assumed to exist in string-table for kWarningMessageKey
CAlert::WarningAlert(string);

Alerts are an exception to the general rule in the InDesign API because there is no need to be concerned with boss classes or ODFRez types; the alerts are created using methods on the
CAalert API helper class. One caveat is that the user-interface strings are always translated unless the client code explicitly marks them as non-translatable. Only use non-translate strings for ‘internal’ strings or for debugging purposes; otherwise, respect the user-interface architecture and allow strings to have translations for any locale that would likely be encountered.

Capabilities

The CAalert helper class exposes an API that can be used to create several different types of alerts. The InformationAlert, WarningAlert and ErrorAlert static methods can be used to create different simple types. There is also a convenience method named WarningAlertWithDontShowAgain which allows the end-user to specify that a particular alert should not appear again.

The ModalAlert method provides more flexibility in terms of the labels on the buttons, the icons displayed and the ordering of the buttons. It also allows the return value to be queried. ModalAlert enables creation of a dynamically-sized alert, based on the length of the message, which can have an optional icon and up to three buttons. Furthermore, the function lets one specify which button is the default button; that is, the button which is activated if the end-user hits Return or Enter without making an explicit selection.

There should be a PMString title for each button string. Pass in kNullString for no button. It is not necessary to call PMString::Translate on any of the button strings. ModalAlert will automatically translate all of the strings, including the message.

Platform issues

Currently, CAalert relies on the platform alert mechanism to display the dialog. Windows alerts automatically size themselves for the amount of text displayed and automatically lay out the buttons correctly. The CAalert implementation provides code on the Mac to model this same behavior.

Windows is limited in the titles of the buttons permitted in the alerts; only the standard Yes, No, Cancel, and OK buttons are supported. That means that on Windows, it is not possible to use custom button strings, and any custom keyboard shortcut resource is ignored. This appears to be a generally-accepted limitation among Windows users.

Line-breaking algorithms for the text displayed in alerts are dependent on the routines provided by the platform API; there is no control over the composition of text for alerts. The platform breaks lines in the alert where it sees fit. On the Mac this works reasonably well; however, on Windows the algorithm is less predictable and sometimes long, thin alerts are produced. If more control is needed, it is possible to compose strings with explicit carriage returns marking line-breaks in them. This gives the correct line-breaking character sequence for the platform. A sample string in a localised framework resource file is shown in Example 29.

Sample code

To avoid depending on the platform algorithm to break strings for display in an alert, you can pre-segment the text to be displayed by using the constant kLineSeparatorString (defined in
CoreResTypes.h) to specify where lines of text in the alert should break. Example 29 shows how to define a string that would be broken across two lines in a predictable way.

**Example 29 A string with explicit line breaks**

```c
// How to write a string with explicit line breaks for an alert, say
"Here is a string" kLineSeparatorString " that takes up two lines."
//The resource compiler will combine these into one string
// with a carriage return.
```
A common alert application is to display an error message of some kind. Example 30 shows a very common pattern in the InDesign codebase for this situation.

**Example 30 Displaying an error alert**

```c
// do something that may lead to an error condition-
// some other code invoking PMSetGlobalErrorCode()
ErrorCode result = possibleFailure();
// determine the appropriate error message to display
PMString string = ErrorUtils::PMGetErrorString(result);
if (!string.IsNull())
{
    // Display the translated error message
    CAlert::ErrorAlert(string);
    // reset the global error code to clear the error
    ErrorUtils::PMSetGlobalErrorCode(kSuccess);
}
```
Alerts can also determine a user choice in a quick and simple way; the return value from the CAlert::ModalAlert method can be used to find the button that the user clicked on. The button return-codes begin at 1 with the left-most button on the alert.

**Example 31 Soliciting a value with an alert**

```c
PMString alertString = kPerformActionKey;
// Translation assumed to exist in string-table for kPerformActionKey
const int actionAgreedValue = 1;
int32 result = CAlert::ModalAlert(alertString,
    kYesString, // button 1 string
    kNoString, // button 2 string
    kNullString, // button 3 string
    actionAgreedValue, // default button
    CAlert::eQuestionIcon ); // icon to show
if(actionAgreedValue == result) {
    // then user clicked on yes. Code to perform action here.
    // If they clicked on No, result would be 2 (second button from left)
}
User Interface Widgets

Progress bar

Description

A progress bar widget indicates progress on a lengthy task, which frequently occurs on import providers and export providers or may be used during some elaborate conversion. An abstraction called the progress bar manager (IProgressBarManager) mediates creation and interaction with progress bars. A progress bar appears in its own dialog that consists of a progress bar widget and a cancel button to end the current task. The progress bar combines information about the number of tasks and the range of display associated with each task. The default is that the display range is the interval \([0,1]\), and that each task added fills in another division on the progress bar when completed. Table 21 shows some of the boss classes that support a progress bar.

**Table 21 Progress Bar Boss Classes**

<table>
<thead>
<tr>
<th>Widget boss class</th>
<th>ODFRez custom type</th>
<th>Example of use</th>
</tr>
</thead>
<tbody>
<tr>
<td>kProgressBarWidgetBoss</td>
<td>ProgressBarWidget</td>
<td>Used to display activity progress</td>
</tr>
<tr>
<td>kDocProgressBarWidgetBoss</td>
<td>DocProgressBarWidget</td>
<td>When opening a layout window</td>
</tr>
</tbody>
</table>

Design

Figure 24 shows the boss classes that are provide the behaviour of a progress bar in a modal dialog. There is a partial implementation class named BaseProgressBar, with subclasses including RangeProgressBar and TaskProgressBar (see ProgressBar.h). A recommended way to create a progress bar is to use a subclass such as RangeProgressBar and call methods on this class to set the state of the progress bar.

You can also create customized progress bar widgets. You only need detailed knowledge of the boss classes (e.g. kProgressBarWidgetBoss) to implement such a custom progress bar; for instance one, contained in a palette-panel widget, rather than appearing in a modal dialog.

**Figure 24 Progress bar**
This screenshot shows the classes that make up a progress bar; note how there is a specialised cancel button, as well as a dialog boss class.

**Widget data model**

The boss class named kProgressBarWidgetBoss aggregates an IProgressBarControlData interface, which is the data model for the control representing the current position. Normally you don’t call methods on this interface directly, since progress bar helper class methods (for instance, the method BaseProgressBar::SetPosition) are used. Implementation code for the progress bar helper classes calls the methods on these data interfaces, in addition to performing various other checks on state.

**Dialogs**

**Description**

Dialogs are used to solicit input from an end-user, and within InDesign CS, this is a modal process; the end-user has to supply input or dismiss the dialog with a Cancel gesture before they can return to the main application.

*FIGURE 25 Dialog*

The screenshot shows an example of a dialog, the Layer options dialog. Dialogs in the API dialog are all movable, modal windows. Note that there are widget boss classes that provide the behaviour for the OK and Cancel buttons on the dialog. The behaviour of the dialog itself is provided by a descendant of kDialogBoss.

A dialog created with the API is a window (kMovableModalWindowBoss) with a panel inside it; see *Figure 25*. The boss class named kDialogBoss and its descendants provide the behaviour of the panel. Classes named k<whatever>DialogBoss are therefore panel boss classes; kDialogBoss extends kPrimaryResourcePanelWidgetBoss. You should bear this in mind when implementing the dialog; you are defining widgets within a panel that just happens to be housed in a modal, movable window. See Table 22 for a list of dialog boss classes.
Most dialogs should have least two buttons; one (Done or OK in English locales) to accept the choices made, and another (Cancel) to indicate that the choices are to be revoked. The buttons typically derive their behaviour from kButtonWidgetBoss and kCancelButtonWidgetBoss respectively. The default value of the ButtonAttributes member is kTrue, which means that the OK control will grab the input focus when the dialog appears.

Although dialogs and panels appear to be quite different types of widgets, they share much behaviour. The panel widget (e.g. kPalettePanelWidgetBoss) that provides the behaviour for a panel housed within a floating palette uses a lot of the same code as the class kDialogBoss, which provides the behaviour for a panel within a modal, moveable window.

The dialog architecture provides sophisticated features such as preview capability, with the condition that previewable dialogs must also be modal. A modal dialog is one in which the user has to make a set of commitments (optionally previewing the result) and then dismiss the dialog with an affirmation to execute (OK or Done) or a Cancel. The end-user cannot carry on with other activities whilst the modal dialog is present. The end-user is blocked from other tasks, and the behaviour of modal dialogs is therefore synchronous; that is, each end of an exchange of communication waits on the other in turn. The application waits on the end-user finishing with a modal dialog before further processing occurs.

The typical way to create and work with a dialog from client code is to subclass kDialogBoss (or one of its descendants), and provide implementations of IDialogController (methods on this are called e.g. when initialising the dialog or when it is being dismissed) and IObserver by specialising CDDialogObserver, to handle notifications from the child controls. There are several examples in the SDK that involve working with dialogs, such as the BasicDialog SDK plug-in.

ResizeDialogWidget is a Dialog resource that overrides the IControlView interface and replaces it with an ErasablePanelView implementation. The effect is to create a resizable dialog. See, for example, the Map Tags to Styles dialog on the Tags panel.

Although dialogs written with the InDesign API can be declared to be movable and modeless, in practice all dialogs in the application are movable but modal (see kMovableModalWindowBoss).

An abstraction called the dialog manager (IDialogMgr) is used to instantiate new dialogs and query for information about dialogs. Plug-ins that create dialogs such as BasicDialog make use of this interface to instantiate the dialog. There are also selectable dialogs, where a list-control can be used to page through the dialog panels, and a tab-selectable dialog.

### Table 22 Dialog Boss Classes

<table>
<thead>
<tr>
<th>Widget boss class</th>
<th>ODFRez custom type</th>
<th>Example of use</th>
</tr>
</thead>
<tbody>
<tr>
<td>kDialogBoss</td>
<td>DialogBoss</td>
<td>Ancestor for all dialog boss classes.</td>
</tr>
<tr>
<td>kResizeDialogBoss</td>
<td>ResizeDialogWidget</td>
<td>Used to create resizable dialog.</td>
</tr>
<tr>
<td>kSelectableDialogBoss</td>
<td>SelectableDialogBoss</td>
<td>Used to create selectable dialogs</td>
</tr>
<tr>
<td>kTabSelectableDialogBoss</td>
<td>TabSelectableDialogBoss</td>
<td>For tab selectable dialogs</td>
</tr>
</tbody>
</table>

Although dialogs written with the InDesign API can be declared to be movable and modeless, in practice all dialogs in the application are movable but modal (see kMovableModalWindowBoss).
Design

The ancestor for all dialog boss classes is kDialogBoss. It is specialised by many subclasses, e.g. kSelectableDialogBoss, providing behaviour for a selectable dialog. It supports a preview capability, so an end-user can preview the effect of changes they might make without having to commit to changing the document. The usual process to create a dialog is to subclass the kDialogboss class and provide an implementations of the interfaces IDialogController (using the CDialogController helper class) and IObserver (using CDdialogObserver).

The partial implementation class CDdialogObserver has several responsibilities; it attaches observers to the standard OK (which should have the widget identifier kOKButtonWidgetID), Cancel (kCancelButtonWidgetID) and Preview buttons (kPreviewButtonWidgetID).

When creating a new dialog boss class, subclass the ODFrez type DialogBoss and bind it on to the new dialog boss class. It is also necessary to define the view-resources for each locale, in a LocaleIndex resource.

Widget data model

The boss class named kDialogBoss and its descendants (typically named k<whatever>DialogBoss) aggregate the interfaces IDialogController and IObserver. To create and work with a dialog from client code, subclass kDialogBoss (or one of its descendants) and provide your own implementations of IDialogController and IObserver by specialising CDdialogObserver, to handle notifications from the child controls.

Capabilities

The interfaces exposed by boss class kDialogBoss can be seen in the on-line documentation. The responsibilities of some of the interfaces on this boss class are as follows:

1. IPanelControlData: used to access the child widget hierarchy on the dialog.
2. ITextControlData: used to set the dialog name.
3. IControlView: uses the same implementation as panels to draw the appearance of the dialog panel.
4. IDialogController: supports the “dialog protocol” and is unique to dialog boss classes. Helper methods connected with edit-boxes can be used to query and retrieve both strings and typed values, such as real-valued numbers. These avoid having to locate the widget using methods such as IPanelControlData::FindWidget, query for an ITextControlData or ITextValue interface. Helper methods for boolean and tri-state controls provide shortcuts to query and set the state of these controls.
5. IObserver: receives notifications about changes in state of the dialog controls. The partial implementation CDdialogObserver handles the OK, Cancel and Preview buttons by default, and you should subclass CDdialogObserver for the IObserver implementation added to a dialog boss class.
Two key aspects to the messaging associated with dialogs are: receiving notification about the controls on the dialog themselves (through IObserver, based on CDialogObserver), and the messages sent through IDialogController that conform to the “dialog protocol”. The dialog protocol consists of the message sequence sent to the IDialogController:

1. IDialogController::InitializeDialogFields
2. IDialogController::ValidateDialogFields
3. IDialogController::ApplyDialogFields

Note: there is also the possibility that instead of IDialogController::ApplyDialogFields, the last message will be IDialogController::ResetDialogFields. This can happen when a Cancel button changes to a Reset button with the correct keyboard modifiers.

In the event of IDialogController::ResetDialogFields, the IDialogController::InitializeDialogFields message is sent again. Once the dialog is created, the first call made on the dialog is Open on the IDialog interface. Behind the scenes, this calls InitializeFields on a IDialogController interface pointer obtained from the dialog boss object. Typically a dialog will use the CDialogController utility class to provide the bulk of the implementation of the methods of this interface, with an override of at least ApplyDialogFields. The other key methods that can be overridden are:

1. InitializeFields: this should delegate initially to the CDialogController::InitializeDialogFields method, and set up initial values,
2. ValidateDialogFields: this should return kDefaultWidgetId if all fields are valid, else the offending widget ID,
3. ApplyDialogFields: at which point the user has accept the choices and pressed the Done or OK button.

The implementation code behind dialogs provides careful bracketing of multiple commands invoked on ApplyDialogFields in the dialog controller to ensure that a command sequence is run when the OK button is pressed, or that the command sequence is aborted if the Cancel button is pressed. This mechanism works hand-in-hand with the preview capability to ensure that when preview is operating, even if the effects of a given command or command sequence are previewed, that changes will not be committed when the dialog is dismissed with a ‘Cancel’.

Previewable dialogs

Previewable dialogs allow an action to be previewed before it is committed. Previews of parameter effects are found in the context of modal dialogs within applications such as the Transform dialogs (move, rotate, scale, shear). With modeless panels, it is more difficult to define the ‘commit’ semantics that are expressed than when a modal dialog is dismissed with an ‘OK’ or ‘Done’. Consequently, the preview feature is restricted to modal dialogs.

The main requirement is that there be a check-box widget with identifier kPreviewButtonWidgetID; the expectation is that it has an ITriStateControlData interface, which is consistent with the radio and check-box buttons.
**Selectable dialogs**

Selectable dialogs consist of multiple panels, of which only one at a time is visible. This is a mechanism that has been used widely across many applications. To move between panels in a selectable set, the dialog typically provides Prev (previous) or Next buttons (in the English locale). kSelectableDialogBoss and kTabSelectableDialogBoss provide the behaviour for selectable dialogs. See the BasicSelectableDialog SDK sample, which can be found in the folder `<sdk>/source/sdksamples/basicselectabledialog`.

There are two main use cases when working with selectable dialogs:

1. Adding a panel to an existing selectable dialog. This is illustrated in the CustomPrefs SDK sample.
2. Creating a selectable dialog of one's own. This is shown in the BasicSelectableDialog SDK sample. This sample shows how to create a 'standard' selectable dialog, like the Preferences dialog of the application, and how to create a tab-selectable dialog.

---

**Palettes**

**Description**

This section describes the difference between palettes (with behaviour provided by a window boss object, aggregating IWindow) and a panel (which is a dependent of a window boss object, but doesn't itself aggregate IWindow). The terminology in this area can be confusing; for instance, kPalettePanelWidgetBoss is a panel, and not a floating palette widget. Client code may frequently use panel boss classes, but rarely the window boss classes. A discussion of the semantics of IWindow can be found in the Page Item Drawing chapter of the Programming Guide.

A palette is a window that serves as a container for panels; a panel is a container for widgets. This section briefly describes the tabbed palette architecture and abstractions such as the palette manager (IPaletteMgr) that is used to manipulate the collection of palettes in the application. The panel manager (IPanelMgr) is also introduced, since this exposes an API that allow the properties of panels contained in palettes to be queried and set. The PanelMover helper class is also introduced, since this encapsulates some of the complexity of moving panels between palette windows.

Tabbed palettes are windows that can manage multiple panels with tabs that can be used to bring panels to the foreground. Panels can be added to specified palettes by particular plug-ins; for instance, the SDK plug-ins all target one palette. The target palette is named in the panel list resource in the top-level ODFRez framework resource file for each plug-in. A panel can be dragged to another tabbed-palette, so this specifies only the initial location of the panel. There is one “palette” in the application that is not a tabbed palette; this is the Toolbox palette.
Design

Several boss classes are involved in creating the tabbed palette behaviour. There is a window boss class named kTabAwarePaletteBoss which provides the behaviour for palettes and which has a child responsible for managing the individual tabs and children corresponding to the panels that are currently contained within the palette. The class named kTabAwarePaletteBoss derives from kWindowBoss.

There is a class named kDockablePaletteBoss, an instance of which is created by the PanelMover implementation class when a panel is created in a palette not currently in existence. kDockablePaletteBoss manages a set of children, one of which is named kTabAreaWidgetBoss, which is responsible in turn for managing all the tabs visible within the palette. The other children of the kDockablePaletteBoss are the panels that are added to the palette by successive plug-ins; these are likely to be named k<plugin-name>PanelBoss and extend boss classes such as those named kPalettePanelWidgetBoss or kErasablePrimaryResourcePanelWidgetBoss.

To work with collections of palettes, there is an abstraction named the palette manager (IPaletteMgr). To work with panels that reside upon the palettes, there is another abstraction named the panel manager (IPanelMgr).
The diagram above shows relationships between classes involved in a panel. The `->` arrows represent ownership. There is one example of an inheritance relationship, between `kTabAwarePaletteBoss` and `kWindowBoss`.

The diagram above shows the widget class hierarchy of a typical panel. The class named `kPalettePanelWidgetBoss`, which houses the child widgets that client code creates, is a child widget of an instance of a `kDockablePaletteBoss`, which in turn has boss objects of type `kTabAreaWidgetBoss` to provide the behaviour of the tabs via its children.

**Capabilities**

The palette manager (IPaletteMgr) is used to manipulate floating palettes within the application. The panel manager (IPanelMgr) is used to manipulate the panels that live within these palettes. The palette manager methods can return IWindow interface pointers, for instance, IPaletteMgr::FindWindow. In contrast, the panel manager methods would return IControlView interface pointers; the panel manager methods can be used to query for the
existence and visibility of a panel, and change the visibility. Another useful class when working with palettes and panels is named PanelMover, which can be used to move panels between windows. It is used in both the context of working with dialogs and palettes.

The IWindowUtils class provides methods that allow the front window to be determined, to tile or cascade windows, and determine hit testing. The namespace PalettePanelUtils provides static methods that are useful for working with widgets on palettes. There are utility methods for locating the (first, visible) list-box on a panel, as well as methods to hide or destroy panels in the recommended manner.

**Advanced issues**

One thing a developer might wish to know is: when can one tell that the palette manager has started all the palettes required, given the panels added by plug-ins? It is possible to create a boss class that is a palette manager service provider. In this case, the boss object that exposes an IPaletteMgrService interface will be called back after the palette manager has been fully started up, and called when the palette manager is about to shutdown. See the boss class named kBookPanelStartupShutdownBoss for an example.

---

**Panel widgets**

**Description**

This section provides an introduction to some of the panel widgets used in the InDesign API. There are two main distinctions to be made: those which can function as “root panels” in a tabbed-palette widget, and those which are intended to be general purpose widget containers but not necessarily the root panel in a floating palette.

A generic panel widget (kGenericPanelWidgetBoss) can be used as a general-purpose container. It is used in many places within the application. For instance, the character panel consists of a set of generic panel widgets, with separator widgets being used to draw the lines on the panel. The generic panel widget is used on panels such as the paragraph panel, transform panel, tab panel, and library panel. A selection of panel boss classes can be seen in Table 23.

**Table 23 Panel Boss Classes**

<table>
<thead>
<tr>
<th>Widget boss class</th>
<th>ODFRez custom type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kErasablePrimaryResourcePanelWidgetBoss</td>
<td>ErasablePrimaryResourcePanelWidget</td>
<td>Can be used for a root panel in a palette; for instance, the Links panel uses one rather than a PalettePanelWidget.</td>
</tr>
<tr>
<td>kPalettePanelWidgetBoss</td>
<td>PalettePanelWidget</td>
<td>Another choice to use for the root panel in a tabbed-palette.</td>
</tr>
<tr>
<td>Widget boss class</td>
<td>ODFRez custom type</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>-----------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>kPrimaryResourcePanelWidgetBoss</td>
<td>PrimaryResourcePanelWidget</td>
<td>May also be used as a root panel in a tabbed-palette.</td>
</tr>
<tr>
<td>kClusterPanelWidgetBoss</td>
<td>ClusterPanelWidget</td>
<td>Grouping widget required for mutually-exclusive behaviour of radio-buttons (or check-boxes), draws without border.</td>
</tr>
<tr>
<td>kGroupPanelWidgetBoss</td>
<td>GroupPanelWidget</td>
<td>Widget for enclosing a group that draws a border, with an optional title.</td>
</tr>
<tr>
<td>kGenericPanelWidgetBoss</td>
<td>GenericPanelWidget</td>
<td>Workhorse panel widget that can be used as a container for other widgets without border decoration.</td>
</tr>
<tr>
<td>kPanelWithHiliteBorderWidgetBoss</td>
<td>PanelWithHiliteBorderWidget</td>
<td>Border-decorated panel.</td>
</tr>
<tr>
<td>kPanelWithInverseHiliteBorderWidgetBoss</td>
<td>PanelWithInverseHiliteBorderWidget</td>
<td>Border-decorated panel.</td>
</tr>
<tr>
<td>kControlStripWidgetBoss</td>
<td>ControlStripWidget</td>
<td>Basis of control strip panel.</td>
</tr>
<tr>
<td>kDetailKitPanelWidgetBoss</td>
<td>DetailKitPanelWidget</td>
<td>Used if you want to provide your own PanelDetailController.</td>
</tr>
<tr>
<td>kErasableKitPanelWidgetBoss</td>
<td>ErasableKitPanelWidget</td>
<td>Top level Kit. Erases before drawing.</td>
</tr>
<tr>
<td>kKitPanelWidgetBoss</td>
<td>KitPanelWidget</td>
<td>Top level Kit. Does not erase before drawing.</td>
</tr>
<tr>
<td>kKitViewHorzBoss</td>
<td>KitViewHorzWidget</td>
<td>Container for Kit views in the horizontal dock.</td>
</tr>
<tr>
<td>kKitViewVertBoss</td>
<td>KitViewVertWidget</td>
<td>Container for Kit views in the vertical dock.</td>
</tr>
<tr>
<td>kKitViewTabPanelBoss</td>
<td>KitViewTabPanelWidget</td>
<td>Container for Kit views in a standard tabbed palette. Best used if view has a single control set.</td>
</tr>
<tr>
<td>kKitViewTabPanelWithDetailBoss</td>
<td>KitViewTabPanelWithDetailWidget</td>
<td>Container for Kit views in a standard tabbed palette. Allows cycling widget to operate on control sets specified in resource.</td>
</tr>
</tbody>
</table>
**User Interface Widgets**

*Panel widgets*

**Design**

Panels are containers for widgets and/or groups of widgets. Some are suitable as a root panel while others can only be general-purpose containers. Table 23 describes some of the capabilities and intended uses of panel boss classes.

**EXAMPLE 32 Type definition for panel widget**

```
// Type expression for PrimaryResourcePanelWidget
// From Widgets.fh, NOT for use in your code type
PrimaryResourcePanelWidget (kViewRsrcType) :
    Widget(ClassID = kPrimaryResourcePanelWidgetBoss)
{
    ResourceSrcFileInfo;
    CControlView;
    CTextControlData;
    CPanelControlData;
};
```

The type definition for one of the key panel widgets is shown in Example 32 and an instance of the ODFRez data defining a widget of the same type is shown in Example 33.

**EXAMPLE 33 Example panel widget**

```
// Sample data definition for widget
PrimaryResourcePanelWidget
{
    // ResourceSrcFileInfo properties
    PlatformPMString, // fFilename
    longint, // fLineno
    // CControlView properties
    kInvalidWidgetID, // widget ID
    kSysStaticTextPMRsrcId, // PMRsrc ID
    kBindNone, // frame binding
    Frame(0,0,85.0,25.0) // left, top, right, bottom
    kTrue, // visible
    kTrue, // enabled
    // CTextControlData properties
    PlatformPMString, // control label
    // CPanelControlData properties
    {
        // Put your child widgets here.
        // Note: CPanelControlData is defined in Widgets.fh
    },
};
```

**Widget data model**

Panel widgets are typically defined in terms of the relationship that they have with their children. The key data interface is IPanelControlData, which allows access to their child widget hierarchy. Some widgets can have a text label, accessible through ITextControlData.
Capabilities

Navigation through the child widgets on a panel is facilitated by IPanelControlData; this interface is the signature interface for a panel. The IControlView interface for panel widgets is used to manipulate the visual representation. Typically this involves delegating to the children to ensure that they draw after the panel widget has drawn itself and any decorations. The IEventHandler interface for a panel is responsible for locating a child widget to handle an event, delegating events to its children; the container propagates events to its children until one claims to have handled the event. The ITextControlData interface allows the panel name to be set and queried.

Detail-controlling panels

Many of the panels within the application show a feature of detail control, where either the set of widgets displayed or the physical properties of the widgets (such as the height of elements in a list-box) change in some way in response to a user gesture. This feature provides both a simplified user-interface and a more elaborate interface for more expert users. Alternatively, users may not always wish to see the full range of configuration options and this allows the widget set displayed in a panel to be varied by an end-user. SDK samples such as DetailControlListSize show an example of varying list-box height and widget set composition respectively.

FIGURE 27  Detail-controlled panel, showing large and small rows

These screenshots show the Swatches panel with two different levels of detail; the detail level affects the size of the list-box rows and the size of the icons on each of those rows. The higher level of detail may be appropriate for a more experienced end-user, who is more likely to use key-strokes than the mouse in making gestures and may be comfortable with a panel with a higher information content.

The screen-shots in Figure 27 show the Swatches panel at two different levels of detail. IPanelDetailController interface provides capability for switching the level of detail. There are two implementations: for list-box palette row size and set composition respectively. For a detail-controlled set of widgets, override IControlView::ConstrainDimensions which provides the correct behaviour on resize of the detail-controlled panel, because changing the detail control level will typically force a resize on updating the panel.
Dynamic panels

Panels may be created dynamically; the process is modelled on the behaviour of the Asset Library panel manager, which manages a dynamic collection of panels that provide views of asset libraries. The SDK sample named DynamicPanel shows one mechanism to instantiate panels dynamically, and create an arbitrary sequence of panels which can be destroyed in the reverse order. The sample can be found in <sdk>/source/sdksamples/dynamicpanel.

Control strip

The Control Strip is a panel based on ControlStripWidget that uses the new selection architecture to gather different sets of widgets into itself, based on the current selection. Existing widgets from other panels are relatively easy to add to the control strip; observers, etc. are taken care of automatically. New controls can also be added, but with significantly more effort.

The appearance of the Control Strip changes depending on document and selection state and is shown below in a text frame selected state.

Figure 28 Control strip

This shows the Control Strip, which is a context-sensitive widget which displays a widget set that depends on the properties of the current selection.

Kits

Kits are a form of dockable panel that collapse against the right side of the workspace as seen in the Swatches panel. They are tab-selectable and open or collapse based on toggle clicking the tab.

ErasableKitPanelWidget is the top-level Kit. It erases before drawing. Most kits should use this, otherwise they might have redraw issues. KitPanelWidget is also a top level Kit, but does not erase before drawing. KitViewHorzWidget is the container for Kit views that will be in the horizontal dock; can also be used for a tabless floater view. KitViewVertWidget is the container for Kit views that will be in the vertical dock; can also be used for a tabless floater view.

KitViewTabPanelWidget is the container for Kit views that will be in a standard tabbed palette; best used where the view only has a single control set (especially a resizable one, since you have to override the DetailController anyways). KitViewTabPanelWithDetailWidget is the deluxe container for Kit views that will be in a standard tabbed palette. Specialized to allow the cycling widget to operate on the specific control sets specified in resource. Should be used most of the time.
Panel widgets

This screenshot shows a kit-view widget participating in a tabbed palette. The same widget can also participate in a horizontal or vertical dock situation.

Resizable panels

Many panels within the user interface can be resized, such as the Links and Navigator panels. The typical mechanism to create a resizable panel is to attach a window size-box widget to the panel to be resized. The key responsibility of the developer is to ensure that the panel knows how to respond to resize events correctly. In practice, the key method to be implemented is IControlView::ConstrainDimensions. Resize events are generated automatically when the window size-box widget is activated by the end-user; see the sample SnippetRunner for an example of a resizable panel in the SDK (<sdk>/source/sdksamples/snippetrunner).

This screenshot shows a resizable panel, the Paragraph Styles panel from the application. The end-user can change the size with the window size box widget.
Static text widgets

Description

Static text widgets are widely used throughout the application, typically as labels on palettes and dialogs, although they can display longer runs of text, and provide for a scrolling display over multiple lines of text with or without specified line-breaks. An example is shown in Figure 31.

This screenshot shows some static text widgets being used; here, they are used as labels on both the panel widget that frames the combo-boxes and as labels for the combo-boxes.

Static text widgets are non-editable. It is possible to vary the appearance and text of these widgets in response to user events, so they can be considered as dynamic information displays rather than static displays as the name might imply.

There are text widgets provided by the API that can also be used to display large amounts of text that can be scrolled through. There are also fitted text widgets, which make sure that the frame of the widget always allows the text to be displayed with a fixed padding around the text regardless of the chosen font, and a variant that allows the font to be both specified in the ODFRez data statements and potentially dynamically varied through a particular interface on the boss class. Table 24 shows the widget boss classes for static text widgets and the types that they bind to, along with sample use in the application.

<table>
<thead>
<tr>
<th>Widget boss class</th>
<th>ODFRez custom type</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>kStaticTextWidgetBoss</td>
<td>StaticTextWidget</td>
<td>This is used to display a single line of static text. An example can be found on the Stroke panel, in the widget that displays Weight:</td>
</tr>
<tr>
<td>kFittedStaticTextWidgetBoss</td>
<td>FittedStaticTextWidget</td>
<td>Used to display text that always fits inside its frame. Helpful where the full text must always be displayed.</td>
</tr>
<tr>
<td>kGroupPanelTitleTextLabelBoss</td>
<td>GroupPanelTitleTextLabel</td>
<td>Renders text in theme color if WinXP theme is on.</td>
</tr>
</tbody>
</table>
### Design

The ancestor of all static text widget boss classes is named `kStaticTextWidgetBoss`. Text widgets can have a variable font displayed; `kInfoStaticTextWidgetBoss` provides this capability. Text widgets can have a frame that is fitted to the size of the string in the font to display, with a specified padding; `kFittedStaticTextWidgetBoss` provides this capability. There are also multi-line static text widgets; `kStaticMultiLineTextWidgetBoss` for the basic multi-line widget and `kDefinedBreakMultiLineTextWidgetBoss` for the type that accepts pre-defined line-breaks.

Fitted static text widgets will resize their frame to attempt to fit the text in the current drawing font with the specified padding. The boss class named `kFittedStaticTextWidgetBoss` does not add any new interfaces compared with `kStaticTextWidgetBoss`, only overriding the implementation of the interface `IControlView` to ensure that the text fits when drawn.

The principal collaboration of interest is between the multi-line text widget types and the scroll-bar widget type, `kScrollBarWidgetBoss`. An association is established in the ODFRez...
between the multi-line text widgets and a scroll-bar that is responsible for adjusting the views of the text drawn.

The ODFRez custom type associated with the kStaticTextWidgetBoss class is named StaticTextWidget; the type definition can be found in Widgets.fh, and for convenience is shown in Example 34. The set of type definitions for API widgets can be found in Widgets.fh.

**Example 34 Type definition for static text widget**

```c
// Type expression for StaticTextWidget
// From Widgets.fh, NOT for use in your code type
StaticTextWidget (kViewRsrcType) :
    Widget (ClassID = kStaticTextWidgetBoss)
    {
        CControlView;
        StaticTextAttributes;
        CTextControlData;
        AssociatedWidgetAttributes;
    };
```

A sample data definition for a static text widget is shown in Example 35.

**Example 35 Example static text widget**

```c
// Sample data definition for widget
StaticTextWidget
{
    // CControlView properties
    kInvalidWidgetID, // widget ID
    kSysStaticTextPMRsrcId, // PMRsrc ID
    kBindNone, // frame binding
    Frame(5.0,5.0,85.0,25.0) // left, top, right, bottom
    kTrue, // visible
    kTrue, // enabled
    // StaticTextAttributes properties
    kAlignLeft, // Alignment
    kDontEllipsize, // Ellipsize style
    // CTextControlData properties
    "", // control label
    // AssociatedWidgetAttributes properties
    kInvalidWidgetID, // associated widget ID
},
```

**Widget data model**

This control encapsulates data of type PMString; the text for a static text widget can be initialised through the ITextControlData interface using the SetString method. The initial value can be defined in a key-value pair in the StringTable for each locale of interest; the key should be placed in the CTextControlData field in the ODFRez data statement.
User Interface Widgets

Check-boxes and radio buttons

Capabilities

The ITextControlData interface stores the state of the label on the static text widget. IStaticMultiLineTextAttr finds the associated scroll bar widget, to scroll a multi-line text widget. Once a reference to the scroll-bar widget is obtained (use IControlView::FindWidget), query for its ICounterControlData and use the methods on this interface to control the view.

Check-boxes and radio buttons

Description

The API provides check-box buttons (kCheckBoxWidgetBoss) and radio-buttons (kRadioButtonWidgetBoss) that operate like the equivalent platform controls. There are also fitted versions of the radio-buttons and check-boxes provided by the API; these have the additional capability that the button and associated text should always fit within the frame, subject to a padding around the boundary. Radio-button and check-boxes are tri-state controls and aggregate an ITriStateControlData interface to represent their state. An example of these controls being used in the application user interface is shown in Figure 32.

FIGURE 32 Radio-buttons and check-boxes

This screenshot shows a pair of radio-buttons and a pair of check-boxes. The cluster-panel widget that owns the radio-button widgets is not shown but has a bounding box that contains the radio-buttons it owns.

Radio-buttons that operate as a logical unit should be contained within a cluster panel widget (kClusterPanelWidgetBoss), which has a bounding box that contains the union of their individual frames. The ODFRez type named ClusterPanelWidget should be used as an immediate parent. The ODFRez custom resource type named CheckBoxWidget is bound to the boss class named kCheckBoxWidgetBoss. The ODFRez custom resource type RadioButtonWidget is bound to the boss class named kRadioButtonWidgetBoss. The check-box and radio button widgets provided by the API are shown in Table 25.
TABLE 25 Check-box and Radio Button Boss Classes

<table>
<thead>
<tr>
<th>Widget boss class</th>
<th>ODFRez custom type associated</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>kRadioButtonWidgetBoss</td>
<td>RadioButtonWidget</td>
<td>Displays a mutually exclusive set of choices, in conjunction with a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cluster panel widget.</td>
</tr>
<tr>
<td>kFittedRadioButtonWidgetBoss</td>
<td>FittedRadioButtonWidget</td>
<td>A radio button that will always fit within its frame.</td>
</tr>
<tr>
<td>kCheckBoxWidgetBoss</td>
<td>CheckBoxWidget</td>
<td>A standard check box.</td>
</tr>
<tr>
<td>kFittedCheckBoxWidgetBoss</td>
<td>FittedCheckBoxWidget</td>
<td>Check box that fits within frame, as frame resizes, with padding.</td>
</tr>
</tbody>
</table>

Design

The check-box (kCheckBoxWidgetBoss) and radio-button (kRadioButtonWidgetBoss) aggregate similar sets of interfaces, and the fitted variants have the same interface profile, typically just replacing the IControlView implementation by one that draws the fitted text label. The interfaces aggregated are shown in the Web-based documentation. The ODFRez type expressions can be found in Widgets.fh.

Example 36 shows the type expression for the ODFRez type named RadioButtonWidget. Note that it is composed of the ODFRez type CControlView field (bound to IControlView) and the ODFRez type CTextControlData, bound to ITextControlData, which represents the label.

Example 36 Type definition for radio button widget

```csharp
// Type expression from Widgets.fh
type RadioButtonWidget (kViewRsrcType) : Widget (ClassID = kRadioButtonWidgetBoss) {
    CControlView;
    CTextControlData;
};
```

See Example 37 for an instance of a radio button widget being defined in ODFRez data statements.

Example 37 Example radio button widget

```csharp
// would go in an FR file
RadioButtonWidget
{
    // CControlView properties
    kInvalidWidgetID, // widget ID
    kSysRadioButtonPMRsrcId, // PMRsrc ID
    kBIndNone, // frame binding
    Frame(5.0,5.0,100.0,21.0) // left, top, right, bottom
    kTrue, // visible
    kTrue, // enabled
    // CTextControlData properties
```
User Interface Widgets

Check-boxes and radio buttons

Widget data model

The data model for kCheckBoxWidgetBoss and kRadioButtonWidgetBoss is tri-state and can be in one of selected, unselected or 'mixed' states. It is represented by the value stored on the ITriStateControlData of the given widget boss object.

Capabilities

The check-box and radio buttons have a data interface to access their state named ITriStateControlData. The check-box and radio-buttons are said to be tri-state. The states are defined in an enumeration in scope of the definition of ITriStateControlData. The different states that a tri-state control can be in are: unselected, selected and unknown (or mixed). The state of these tri-state controls can be queried and set through this interface.

Notification

When the state of controls such as kRadioButtonWidgetBoss and kCheckBoxWidgetBoss changes, notification is sent along the protocol IID_ITRISTATECONTROLDATA; the messages sent when the data model changes are:

{ kTrueStateMessage, kFalseStateMessage, kUnknownStateMessage }

Messages are sent along the protocol IID_ITRISTATECONTROLDATA. To receive notification of these changes, attach an observer to the button. If your controls are on a dialog, then an implementation derived from CDialogObserver makes it straightforward to attach and detach from the widget of interest. The API on IDialogController also makes it straightforward to initialise the state of the check-boxes or radio-buttons. Note: be aware that it is worth setting the doNotify flag on the calls to change the state to be kFalse to avoid triggering unwanted IObserver::Update messages to one’s own observer brought about by one’s own code rather than end-user events.

Controls on panels

Working with controls on panels is perhaps not quite as straightforward as with controls on dialogs; there are two patterns that can be used.

- Observe the changes directly for each control, by adding in an IObserver interface to the widget boss class of interest, and attaching to the widget to listen for changes. This can require creating many new boss classes and new ODFRez custom resource types, and lead to code bloat. It is only a recommended pattern if you have a small handful of widgets to observe.

- Use the ‘Widget Observer’ pattern described earlier. IControlViewObservers is useful for plug-in client code if you want to observe changes in the active context as well as widget-related changes, since you can add an observer interface for the active selection (CActiveSelectionObserver makes this easier) and one observer interface for all your widgets. This pattern is used widely throughout the InDesign codebase for UI plug-ins.
Button widgets

Description

This section describes some of the button widgets available in the API; here we focus on kButtonWidgetBoss and descendants, and treat the iconic buttons in a later section.

There is also a nudge control widget in the API (kNudgeControlWidgetBoss) which is a composite of two button-like widgets, and is discussed briefly in this section. It is used in conjunction with edit boxes and combo boxes to provide the capability to nudge values up or down.

Buttons are used in many of the dialogs within the application. It is not recommended to use the standard button (kButtonWidgetBoss) or its descendants on palette-panels; instead, use iconic buttons on palette-panels to conform with the application look and feel. Buttons that derive behaviour from kButtonWidgetBoss or a descendant of this boss class are bi-state controls.

![Button widgets](image)

This screenshot shows a selection of button widgets; the OK and Cancel buttons would be present on all dialogs, the other buttons present would depend on the requirements for the particular dialog. The enabling state of the buttons is set up initially by the dialog controller and can be modified by the dialog observer depending on what the end-user gestures are.

A default button is one that by default has keyboard focus; if the user hits return, then the event handler of the default button will process the event. An observer on the default button receives an IObserver::Update message. kButtonWidgetBoss and an ODFRez type DefaultButtonWidget can be used if the default button is an OK or Done button; the button that allows an end-user to confirm that they accept the choices that they have made. The default button need not be an OK or Done button; it can also be a Cancel button (kCancelButtonWidgetBoss), where the cancel button should be the selected button when the dialog first appears.
Design

Inspect the browseable documentation for kButtonWidgetBoss and its descendants to see the interfaces aggregated by this type, and some of the descendants of kButtonWidgetBoss in the required plug-in set. The kButtonWidgetBoss class, in addition to taking the focus when a dialog appears, provides behaviour to dismiss a dialog with an affirmation that the action should be executed or that the commitments have been completed. This class provides the behaviour behind the button with a label such as OK or Done that is intended to dismiss the dialog when it is pressed. kCancelButtonWidgetBoss is the counterpart, which provides behaviour behind a standard 'Cancel' button on a dialog.

Nudge buttons collaborate with edit boxes and combo-boxes to allow incremental changes to be made in the control data model. The association is created by referencing a nudge button by widget ID in the definition of the edit box and combo box. Both these types of widgets have an ancestor that has responsibility for dealing with any associated nudge buttons.

The ODFRez custom resource types corresponding to kButtonWidgetBoss is ButtonWidget or DefaultButtonWidget. The CTextControlData field is used to specify the label on the button. The type expression for ButtonWidget is shown in Example 38.

**EXAMPLE 38 Type definition for a button widget**

```c
// type definition from Widgets.fh, NOT for
// use in your own code type
ButtonWidget (kViewRsrcType) :
  Widget (ClassID = kButtonWidgetBoss )
{
  CControlView;
  ButtonAttributes;
  CTextControlData;
```
User Interface Widgets

Button widgets

Example 39 shows ODFRez data statements defining a button widget. Since the ODFRez type ButtonWidget has not been subclassed, the implication is that changes in the data model of this control are handled by the dialog observer rather than an observer for the individual button widget.

**Example 39 Example button widget**

```c
// Would go in one of your FR files
ButtonWidget
{
    // CControlView properties
    kInvalidWidgetID, // widget ID
    kSysButtonPMRsrcId, // PMRsrc ID
    kBindNone, // frame binding
    Frame(5.0,5.0,100.0,25.0) // left, top, right, bottom
    kTrue, // visible
    kTrue, // enabled
    // ButtonAttributes
    kTrue, // default look true
    // CTextControlData properties
    '', // control label (string key) goes here
},
```

**Widget data model**

The data model for kButtonWidgetBoss and its descendants is a boolean; they are hence bi-state controls. Their state is represented by the value stored on their IBooleanControlData interface, and they notify of changes in this state along the default identifier for this interface, IID_IBOOLEANCONTROLDATA.

**Capabilities**

IBooleanControlData and ITriStateControlData are the key interfaces in working with button widgets. The IBooleanControlData interface provides access to the data model of bi-state buttons. A boolean control such as a standard button can be in one of two states: selected or unselected. The state is set through the IBooleanControlData interface. Update messages are sent along the IID_IBOOLEANCONTROLDATA protocol.

For the bi-state buttons, it is possible to change the state and suppress the notification by calling the mutators with the notifyOfChange parameter set to kFalse.

The ITextControlData interface allows setting the label on buttons. The boss classes share the default implementation used by many of the controls.

**UI guidelines**

The UI guidelines recommend that the widget should be 20 pixels in height and a multiple of 5 in width.
Note that the InDesign application does not make use of standard button widgets (other than iconic varieties) on panels and it is recommended not to put standard button widgets on panels; use the iconic varieties instead.

**Notification**

The button widget boss classes aggregate `I<whatever>ControlData` interfaces to allow the state to be queried and set; usually this results in a message being sent to attached observers that the control state has changed. It is also possible to set the state but suppress the notification about the change to prevent observers performing inappropriate updates when the state is set; that is, the control change can be muted. This is done by setting the `notifyOfChange` to be `kFalse` in the mutator methods on these interfaces. This approach can be used when an observer that is listening for a notification on one or more controls wants to set up the state of a control that it is observing, without receiving another notification about the state-change. There is opportunity for confusion in understanding how the controls work; typically the change parameter is expressed as a message identifier, rather than something more identifiable as a ClassID.

When the state of the controls that descend from `kButtonWidgetBoss` changes, notifications are sent along the protocol `IID_IBOOLEANCONTROLDATA`. Additional information is sent via the ClassID. To receive notification about the state-change (e.g. caused by a button press) attach to the button widget boss object’s `ISubject` interface, and listen along the `IID_IBOOLEANCONTROLDATA` protocol.

**Buttons on dialogs**

Where controls are placed upon a dialog, client code can use the helper partial implementation class `CDialogController` and specialize or implement the appropriate methods. `CDialogObserver` typically implements an `IObserver`. The `IDialogController` interface has utility methods relevant to working with button widgets that can be used to set the initial state.

The recommended superclass for the `IObserver` interface to be added to the dialog boss object is `CDialogObserver`, which provides helper methods such as `AttachToWidget()` and `DetachFromWidget()` which can be used to simplify the process of attaching/detaching to or from a particular widget.

An `IObserver::Update` message will be sent through the `IObserver` interface on the dialog boss object when a button is pressed; the classID of the change should be checked to determine whether it is a `kTrueStateMessage`.

**Edit boxes**

**Description**

Edit boxes are editable displays of information. In the application, they are used to allow an end-user to enter a value. The value can be further constrained to be an integer or to be expressed in units such as picas, points, degrees or some other unit. Edit-boxes are often used
along with nudge button controls. The nudge-button control provides precise, incremental control over the contents of edit boxes, particularly when control over parameters in layout-specific units is a fundamental requirement. This close coupling between the nudge and edit controls helps to explain in part why the edit box boss classes descend from the boss class kNudgeEditBoxWidgetBoss.

The treatment of these edit boxes is a little different when they are used in dialogs rather than panels. In panels, typically a new boss class will be created that derives from one of the <variant>EditTextBoxWidgetBoss classes, and exposes an IObserver interface implemented in client code. This interface will be used to receive notifications of changes in the text edit box data model. An observer attached to a particular control will be notified on every keystroke, if the specific ODFRez field controlling this property is set to be kTrue. A selection of the edit-box related boss classes can be seen in Table 27.

**Table 27 Edit Box Boss Classes**

<table>
<thead>
<tr>
<th>Widget boss class</th>
<th>ODFRez custom type</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>kTextEditBoxWidgetBoss</td>
<td>TextEditBoxWidget</td>
<td>For values that are strictly textual, or where you want to parse a value in your own code.</td>
</tr>
<tr>
<td>kAngleEditBoxWidgetBoss</td>
<td>AngleEditBoxWidget</td>
<td>Display values in angular units. The Rotation dialog has an instance of this widget.</td>
</tr>
<tr>
<td>kIntEditBoxWidgetBoss</td>
<td>IntEditBoxWidget</td>
<td>Display integer values. The Page Set-up dialog has an example.</td>
</tr>
<tr>
<td>kRealEditBoxWidgetBoss</td>
<td>RealEditBoxWidget</td>
<td>Display floating point values. The Scale dialog has an instance.</td>
</tr>
<tr>
<td>kPercentageEditBoxWidgetBoss</td>
<td>PercentageEditBoxWidget</td>
<td>Display values in percentage units. Instance on Scale dialog.</td>
</tr>
<tr>
<td>kTextMeasureEditBoxWidgetBoss</td>
<td>TextMeasureEditBoxWidget</td>
<td>Display values in the current text measurement units. Can be found on the Character panel, e.g. the line weight element.</td>
</tr>
<tr>
<td>kLineWtMeasureEditBoxWidgetBoss</td>
<td>LineWtMeasureEditBoxWidget</td>
<td>Display values in line-weight units. The Stroke panel has an instance, when the Dashed line type is selected.</td>
</tr>
<tr>
<td>kXMeasureEditBoxWidgetBoss</td>
<td>XMeasureEditBoxWidget</td>
<td>Display values in current horizontal measurement units.</td>
</tr>
<tr>
<td>kYMeasureEditBoxWidgetBoss</td>
<td>YMeasureEditBoxWidget</td>
<td>Display values in the current vertical measurement units.</td>
</tr>
</tbody>
</table>
Design

The main responsibility of the edit-box widget boss classes is to provide for managed input and display of text strings and unit-specific values. Unit-specific variants remove the responsibility for parsing the input strings (e.g. 7p5 or 90%) from the developer of client-code. Similarly, the developer of client code is relieved of the necessity to format the output strings when using the unit-specific edit-boxes. The edit-box widget boss classes also provide for validation of data entry, which is conditional on settings made in the ODFRez data statements. The principal collaboration is with the kNudgeControlWidgetBoss class, which provides the behaviour for nudge controls used to bump values up and down by small increments. The mapping between an edit box and a nudge-control is established through the ODFRez data statement for an edit box; no code needs to be written to enforce this collaboration.

The main point to observe about the class hierarchy of the edit-box widget bosses is that there is a common ancestor for all the edit-boxes which is a nudge-edit box widget boss class (kNudgeEditBoxWidgetBoss). There are also a collection of specialised edit-boxes that work in units such as degrees, percent, real or integer values, or in measurements such as line weight, text and the measurement unit currently in force in horizontal or vertical directions. Note that the boss class kNudgeEditBoxWidgetBoss should be treated as if it were an abstract type and not used directly.

There are ODFRez custom resource types that map directly onto each of these boss classes with the usual naming convention that k<variant>/EditBoxWidgetBoss has a related ODFRez custom resource type named <variant>/EditBoxWidget. Example 40 shows ODFRez data statements defining an edit box widget.

**Example 40 Example edit box widget**

IntEditBoxWidget

```c
// CControlView properties
kInvalidWidgetID, // widget ID
kSysEditBoxPMRsrcId, // PMRsrc ID
kBindNone, // frame binding
Frame(4.0,5.0,84.0,25.0) // left, top, right, bottom
kTrue, // visible
kTrue, // enabled
// CEditBoxAttributes
0, // nudgeWidgetId (0 or kInvalidWidgetID if no nudge required)
1, // small nudge amount
5, // large nudge amount
0, // max num chars
kFalse, // read only flag
kFalse, // should notify on each key stroke
// TextDataValidation properties
kTrue, // range checking enabled
kFalse, // blank entry allowed
30, // upper limit
0, // lower limit
// CTextControlData properties
"3", // control label
},
```
Widget data model

An edit box widget boss class has a data model that typically consists of a string. This is accessed through the ITextControlData interface aggregated by the particular edit box widget boss class. The ITextControlData interface provides controlled access to the control's data model for 'simple' text strings. This is appropriate for the boss classes such as the kTextEditBoxWidgetBoss class. For the unit-specialised edit boxes, the key interface to get and set values is the ITextValue interface. This enables access to the measurement data in points, regardless of what measurement is currently being displayed. This eliminates the need to parse the string read back from the control or to format the data for the control on setting a value.

Capabilities

The boss class named kNudgeEditBoxWidgetBoss aggregates interfaces such as ITextControlData, ICursorRegion, INudgeObserver, IEditBoxAttributes, along with other key interfaces such as IControlView and IEventHandler. ITextValue is used to query and set the value for any unit-specific edit box widgets; ITextValue can be used to access the data model of unit-specific edit-boxes without having to parse the input string or format the output string. It allows a unit-specific value to be set or queried in an edit-box.

ITextControlData can be used for untyped edit-box widgets, such as the widget with behaviour provided by kTextEditBoxWidgetBoss. The implementation of ICursorRegion determines how or if the cursor changes when it enters an edit box.

UI guidelines

Adobe UI guidelines specify that buttons should ideally be 20 pixels in height and a multiple of 5 pixels wide.

Notification

When the data model changes, registered observers are sent an Update message with class ID kTextChangeStateMessage along the protocol named IID_ITEXTCONTROLDATA. For an observer on a unit-specific edit box, the client code in the Update message should query for the ITextValue interface and call the GetTextAsValue method to determine the state of the edit-box data model.

Working with edit boxes

The most basic use of an edit box is in conjunction with a dialog. In this circumstance, the value of the edit box may only be queried when the dialog is dismissed, or when the preview check box is selected, in the case of previewable dialogs. When the edit box is used on a panel, it is more practical to attach an observer to the edit control. The observer should receive an
Update message when the content of the edit control changes; all keystrokes within the control result in a notification being sent to registered observers.

The simplest case to consider is where the edit box is attached to a dialog. If the dialog is implemented as recommended by using the helper classes CDialogController and CDialogObserver, setting and getting values from the text control is particularly simple. The application framework sends a sequence of messages to the dialog controller in this order:

- InitializeDialogFields()
- ValidateDialogFields()
- ApplyDialogFields()

This sequence of messages is referred to as the “dialog protocol”. In the case of the layer options dialog described previously, when the InitializeDialogFields message is sent to the dialog controller, the edit box is set up with an initial value through the method CDialogController::SetTextControlItemData. When the user hits the OK button, the dialog controller is sent a ValidateDialogFields message by the application framework, and if this returns kDefaultWidgetId, then an ApplyDialogFields message will be sent.

Icon and picture widgets

Description

This section describes widgets used to display icons or images, which can also have button-like functionality. Image-based widgets are found throughout the application. There are lots of image-widget boss classes in those delivered by the Widgets plug-in of InDesign, but few of these are likely to be relevant when writing client code; at most three would be required for most situations.

Some of the widget boss classes shown in Table 28 do not respond to button clicks. Others are described here as “pseudo-buttons” to differentiate them from the boss classes that descend from kButtonWidgetBoss; they still respond to clicks, and aggregate an IBooleanControlData interface. If you have a more exotic requirement, such as creating an image preview in a widget, there is a PanelTreeView SDK sample which indicates how to do this using a custom-control view for a panel rather than an existing image-based widget per se, which isn’t appropriate for this task. The code can be found in <sdk>/source/sdksamples/paneltreeview.

To simplify the domain, client code should be able to perform most tasks using the following small handful of widgets:

- kIconSuiteWidgetBoss, with its counterpart ODFRez IconSuiteWidget (for icons on panels and dialogs), see PicturIcon: <sdk>/source/sdksamples/picturicon
- kPictureWidgetBoss, and its counterpart ODFRez PictureWidget (for bitmap/PICT based widgets), see PicturIcon
- Subclassing the kRollOverIconButtonBoss, and one of the several ODFRez types it’s bound to (depending on your specific requirements). Note: all the SDK samples that have an About icon use this button.
User Interface Widgets

Icon and picture widgets

### TABLE 28 Icon and Picture Widget Boss Classes

<table>
<thead>
<tr>
<th>Widget boss class</th>
<th>ODFRez custom type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kIconSuiteWidgetBoss</td>
<td>IconSuiteWidget</td>
<td>Used e.g. for warning icons on Links panel, or the top-left icon on the Transform panel. Few examples in 2.0 of it being subclassed; examples in the XMedia user-interface and the pop-out widgets on the Toolbox.</td>
</tr>
<tr>
<td>kIconSuitePopupWidgetBoss</td>
<td>IconSuitePopupWidget</td>
<td></td>
</tr>
<tr>
<td>kPictureWidgetBoss</td>
<td>PictureWidget</td>
<td>Used e.g. for some CJK widgets, and the Toolbox has an instance of boss class being subclassed.</td>
</tr>
<tr>
<td>kIconFieldWidgetBoss</td>
<td>IconFieldWidget</td>
<td>RollOverIconButtonWidget; used in warnings.</td>
</tr>
<tr>
<td>kIconPopupBoss</td>
<td>No corresponding ODFRez custom resource type.</td>
<td>Used on tabbed-palettes as the triangle-icon for the pop-up menu, for internal use only.</td>
</tr>
<tr>
<td>kRollOverIconButtonBoss</td>
<td>RollOverIconButtonWidget</td>
<td>Widely used, for example SDK samples.</td>
</tr>
<tr>
<td>kSimpleIconSuiteButtonWidgetBoss</td>
<td>SimpleIconSuiteButtonWidget</td>
<td>Little used. In Stroke panel.</td>
</tr>
<tr>
<td>kChainButtonBoss</td>
<td>ChainButtonWidget</td>
<td>Used for constrain proportions button.</td>
</tr>
<tr>
<td>kRollOverIconPushButtonBoss</td>
<td>RollOverIconPushButtonWidget</td>
<td>Used in control strip for vertical justification mode buttons.</td>
</tr>
<tr>
<td>kOverPrintIconPushButtonBoss</td>
<td>OverPrintWidget</td>
<td>Not used.</td>
</tr>
</tbody>
</table>

**Design**

Most ODFRez types are composed of nothing more than a CControlView field with an additional flag to specify whether some appearance should be applied.

**Capabilities**

The principal interfaces of interest to client-code developers are the IControlView interface (for switching the image displayed), the IBooleanControlData interface, or at least the role it plays in notification. The IControlView interface can be used to dynamically vary the visual representation of a widget. There may be occasions to switch in a different picture depending on state; the methods on the IControlView interface such as SetPluginRsrcID and SetRsrcID are relevant in this context.
Notification

Clients of the buttons with behaviour provided by boss classes such as kRollOverIconButtonBoss and derivatives can attach to the ISubject interface of an instance of a boss object and request for notifications on protocol IID_IBOOLEANCONTROLDATA. An Update message will be sent dependent on the state of the pseudo-button widget; if selected, then a kTrueStateMessage value will be sent for the ClassID of the Update message with protocol IID_IBOOLEANCONTROLDATA. If the button is un-selected, then the ClassID will be kFalseStateMessage with the same protocol. Usually this change is not particularly of interest.

Working with picture widgets

A common requirement is to use a change in the image displayed to signal some kind of state transition; for instance, to indicate that a layer is locked or unlocked, or visible/invisible, the Layers Panel uses an iconic button; see the WListBoxComposite SDK sample that uses this mechanism. This can be done through the IControlView interface on a particular widget. Switching the icon/picture displayed can be achieved by acquiring an IControlView interface (e.g. through IPanelControlData::FindWidget) and sending a SetRsrcID message to the widget boss object to change the image being displayed.

Drop-down lists and combo-boxes

Description

This section describes two widely used and quite closely related widget groups, drop-down lists and combo-boxes. Drop-down lists are non-editable displays of lists of information. Combo-boxes are also used to display non-editable drop-down lists of information, but the end-user interacts with the list in a different way. The relation between the two is that a combo-box is a composite of an edit-box control (with nudge buttons), and a drop-down list. For instance, the Character panel in the application utilises several different kinds of combo boxes on the one panel; see Figure 34. The widget that displays available fonts is a type-ahead combo box. There are several specialised combo boxes that display information in measurement units; see Table 29 for a selection.
User Interface Widgets
Drop-down lists and combo-boxes

**FIGURE 34** Widgets used in the Character panel

This screenshot shows some of the combo-box widgets that are used in the building the Character panel. Note that the widget boss classes used in the Character panel are subclasses of those shown, so this is an approximation.

A combo-box is a composite user interface element that allows a user to select from a drop-down list or also use a text edit box to enter a new choice. In the API there are a wide selection of combo-box widget boss classes that provide rich behaviour for combo-boxes. A combo-box is a combination of an edit control and a drop-down list. The combo-box can co-operate with a nudge control widget to allow incremental changes in input parameters. Combo-boxes are appropriate for display of units such as measurement units, angles, percentages and so on.

When input is selected from a list of values, the combo-box is more convenient than the drop-down list. Type-ahead combo-boxes allow quick access to a value in a sorted list; the user can enter keystrokes matching items in the list. For instance, the font-family combo-box on the Character panel allows the first few characters of the font name to be entered, which will have the effect of scrolling the list to the desired location. There are many types of combo-boxes available within the API, some of which are shown below in an annotated screenshot of the Character panel.

The measurement units specific to layout include:

- x-measurement units
- y-measurement units
- Line weight measure
- Text measure.

A different measurement system may be in force for each of these, and a combo-box of a specialised type displays the choices in the contextually appropriate units. Below are shown a selection of the combo-boxes available in the API.
**User Interface Widgets**

*Drop-down lists and combo-boxes*

### Table 29 Drop-down List and Combo-box Boss Classes

<table>
<thead>
<tr>
<th>Widget boss class</th>
<th>ODFRez custom type</th>
<th>Example of use</th>
</tr>
</thead>
<tbody>
<tr>
<td>kDropDownListWidgetBoss</td>
<td>DropDownListWidget</td>
<td>Display set of text values rather than numeric values, where no type-ahead or nudge is required. Stroke-type on the Stroke panel is an instance of this widget.</td>
</tr>
<tr>
<td>kTypeaheadTextComboBoxWidgetBoss</td>
<td>TypeaheadTextComboBoxWidget</td>
<td>Allow a user to enter partial string and have control predict value to select. Used in Character panel to display font families.</td>
</tr>
<tr>
<td>kTextComboBoxWidgetBoss</td>
<td>TextComboBoxWidget</td>
<td>For values that are textual. Examples of this widget occur on the Find-Change dialog.</td>
</tr>
<tr>
<td>kAngleComboBoxWidgetBoss</td>
<td>AngleComboBoxWidget</td>
<td>Used to display a set of values in angular units. Instances can be found on Transform panel.</td>
</tr>
<tr>
<td>kIntComboBoxWidgetBoss</td>
<td>IntComboBoxWidget</td>
<td>Display set of integer values. An example is the kerning combo-box on the Character panel.</td>
</tr>
<tr>
<td>kRealComboBoxWidgetBoss</td>
<td>RealComboBoxWidget</td>
<td>Used to display set of floating point values.</td>
</tr>
<tr>
<td>kPercentageComboBoxWidgetBoss</td>
<td>PercentageComboBoxWidget</td>
<td>Display values in percentage units. Instances occur on the Transform panel.</td>
</tr>
<tr>
<td>kTextMeasureComboBoxWidgetBoss</td>
<td>TextMeasureComboBoxWidget</td>
<td>Displays units the current text measurement units. Example on the Character panel, the combo-box widget to specify leading.</td>
</tr>
<tr>
<td>kLineWtMeasureComboBoxWidgetBoss</td>
<td>LineWtMeasureComboBoxWidget</td>
<td>Display values in line-weight units. An instance occurs on the Stroke panel.</td>
</tr>
<tr>
<td>kXMeasureComboBoxWidgetBoss</td>
<td>XMeasureComboBoxWidget</td>
<td>Displays values in the current horizontal measurement units.</td>
</tr>
<tr>
<td>kYMeasureComboBoxWidgetBoss</td>
<td>YMeasureComboBoxWidget</td>
<td>Displays values in the current vertical measurement units.</td>
</tr>
</tbody>
</table>

### Design

Combo boxes are really panel widgets; combo-box widget boss classes descend from kGenericPanelWidgetBoss. When instantiated, a combo-box widget boss object creates two
children, one of which is a drop-down list widget boss object and another which is an edit box widget boss object. There appear to be many widget boss classes relating to combo-boxes in the API, but those relating to the encapsulated drop-down list and edit-box within the combo-panel are really 'implementation' boss classes and not required for writing client code.

**Widget data model**

The data model for combo-boxes consists of a string-list control data model for the drop-down (accessed via the IStringListControlData interface) and the data model for an edit box, represented by the ITextControlData interface. Changes in selection by the end-user are sent to client-code along the IID_ITEXTCONTROLDATA protocol which might be unexpected, as superficially the type is similar to a list-box. The edit box does not have an independent data model to the string-list; it reflects the current selection in the drop-down list.

**Capabilities**

IStringListControlData can be used to populate both drop-down lists and combo-boxes dynamically. IDropDownListController can be used to change the state of the drop-down list or a combo-box drop-down and it can be used to control the current selection of the drop-down list or combo-box; for instance, to specify the item selected, enable or disable the control. The interface ITextControlData exposed by the boss class kComboBoxWidget boss relates to the child edit-box child widget. Similarly the interface IEditBoxAttributes relates to the child edit-box widget. However, note that the changes to selection are notified along the protocol IID_ITEXTCONTROLDATA for a combo box, which is not intuitive; a list control-data protocol might have been expected.

**List-box widgets**

**Description**

There are several list-boxes within the user interface of the application; see Figure 35 for an example list-box. A list-box consists of a set of panels, which can be selected singly or multiply depending on one of the list-box properties. The list-box, as with other widgets, is also capable of supporting drag-and drop. List-boxes derive their behaviour from the API widget boss class named kWidgeListBoxWidgetNewBoss.
User Interface Widgets

List-box widgets

**FIGURE 35 Widget list box**

This screenshot shows an example of a list-box within the application, taken from the Swatches panel.

A list-box is appropriate for displaying lists of information that are interactive and permit the possibility of an end-user clicking on one or more selected items. The list-box is used widely on both dialogs and tabbed palettes; typically a scroll-bar is used in at least the vertical dimension.

The widget list box widget is used in conjunction with a cell-panel widget (kCellPanelBoss) which is the container for the list panels; see Table 30. Many of the list boxes in the application consist of compound list elements, made up of several widgets. The cell-panel widget type is named kCellPanelBoss; note that this derives from kGenericPanelWidgetBoss, although the name departs from the convention of post-fixing with WidgetBoss.

**Table 30 List-box Boss Classes**

<table>
<thead>
<tr>
<th>Widget boss class</th>
<th>ODFRez custom type</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>kWidgetListBoxWidgetNewBoss</td>
<td>WidgetListBoxWidgetN</td>
<td>Used for any list-boxes you create from client code unless you have controls such as edit boxes or drop-down lists, then use a tree-view instead.</td>
</tr>
<tr>
<td>kCellPanelBoss</td>
<td>CellPanelWidget</td>
<td>This manages the child widgets of the list-box</td>
</tr>
</tbody>
</table>

**Known issues**

Some problems, in terms of widget placement and repainting, will be discovered when attempting to have a control with a platform window such as an edit-box or a drop-down list as a child of a list-box widget. A tree-view control (see “Tree-view widgets” on page 103) with the root hidden and all nodes as children of the root should be used instead.
User Interface Widgets

List-box widgets

Design

The kWidgetListBoxWidgetNewBoss boss class provides the behaviour for list-boxes, bound to the ODFRez custom resource type WidgetListBoxWidgetN. This type kWidgetListBoxWidgetNewBoss collaborates with its dependent container type named kCellPanelBoss. The responsibility of the kCellPanelboss class is to manage the children in the list-box, e.g. to help update the views when scrolling occurs by reporting the bounds of the panorama to scroll. The related ODFRez custom resource type is CellPanelWidget.

Widget data model

The widget data model is a collection, represented by IListControlData. This interface can be used to add and remove elements from the collection. It is the interface to the data model of the widget. There is also an class named IListControlDataOf, which is a parameterised type (template class). An observer should attach to the ISubject interface of the widget list box boss object for notification on the protocol IID_ILISTCONTROLDATA.

Capabilities

The IListBoxController interface can be used to specify which item has been selected or is to be selected, to scroll the list up or down, and even to search over the data model. Subclasses of the boss class named kWidgetListBoxWidgetNewBoss can provide an override of IListBoxController::Search to provide a custom search over the list-box elements. The IListBoxAttributes interface allows client code to specify and query whether the list supports multiple selections, support drag-out or drop-into. Usually these attributes are set up through ODFRez data statements, although there may be occasion to wish to vary these programmatically.

User interface guidelines

The relative dimensions of the list-box and the cell-panel widget are specified in the Adobe UI Guidelines; the left boundary of the list-box overhangs the parent left edge by one pixel, but the top under-hangs by one pixel. There is also a one-pixel border at the bottom of the CellPanelWidget, and an 8-pixel border on the right of the CellPanelWidget to accommodate a scroll-bar. See WListBoxBasic and note the geometries in the top-level FR file associated with this plug-in of the list-box and the cell-panel widget.

Notification

The key to working with a list box is to be familiar with the messages that can be sent to an observer. In the boss class that subclasses kWidgetListBoxWidgetNewBoss, the developer should specify that the IObserver interface is being added. This will enable the client code to receive Update messages when the list-box data model changes. The observer should be attached to the ISubject interface of the list-box widget boss object and register for notifications on the protocol IID_ILISTCONTROLDATA. The change manager will send Update messages through the IObserver interface when the data model of the list-box changes. The types of messages that it might receive are defined in WidgetID.h. Messages are sent as ClassID parameter in the IObserver::Update message and are as follows:
User Interface Widgets

Splitter widgets

Description

The splitter widget manages the dimensions of multiple panels within one container. For instance, the Pages panel exploits a splitter widget to create two regions which can be sized co-dependently. The splitter widget can be used to divide both vertically and horizontally. The setting is determined by the bindings, normally specified on the CControlView-related portion of the ODFRez data statements used to define the splitter. Some boss classes that implement splitters are shown in Table 31.

**Table 31 Splitter Boss Classes**

<table>
<thead>
<tr>
<th>Widget boss class</th>
<th>ODFRez custom type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kSplitterWidgetBoss</td>
<td>SplitterWidget</td>
<td></td>
</tr>
<tr>
<td>kSplitterPanelWidgetBoss</td>
<td>SplitterPanelWidget</td>
<td></td>
</tr>
<tr>
<td>kLayoutSplitterPanelWidgetBoss</td>
<td>LayoutSplitterPanelWidget</td>
<td></td>
</tr>
<tr>
<td>kLinkedSplitterPanelWidgetBoss</td>
<td>LinkedSplitterPanelWidget</td>
<td></td>
</tr>
</tbody>
</table>

Design

The boss class kSplitterWidgetBoss and the tracker boss class kSplitterTrackerBossMessage cooperate to provide the behaviour of the splitter widget. The splitter widget can be initialised by ODFRez data statements; see the SplitterWidget in ResizablePanel sample, for instance. There is little need for additional customisation of the splitter widget that is available in the API. The ODFRez custom resource type named SplitterWidget is bound to the boss class kSplitterWidgetBoss. The superclass of kSplitterWidgetBoss is the base widget boss class, kBaseWidgetBoss.
Capabilities

The boss class named kSplitterWidgetBoss exposes interfaces such as ISplitterControlData, which provides access to the list of managed widgets. The ICursorPosition interface is effectively a signature that the widget is bound to a cursor provider. The ISplitterControlData interface provides access to the list of splitter-managed widgets and query methods about the properties of the control.

The boss class named kSplitterTrackerBossMessage provides the behaviour behind the tracker. Indeed, much of the widget’s capability comes from the implementation of ITracker on this widget boss class. The splitter widget boss class event handler serves only to create the tracker and send it a BeginTracking message.

Scroll-bars and scrolling

Description

The API uses a pattern or assembly consisting of three elements to provide for scrolling views within the user interface. The elements are;

1. scroll-bars; these are controls whose behaviour derives from the widget boss class named kScrollBarWidgetBoss

2. panoramas; this is an aspect of a scrollable object that can be accessed through an IPanorama interface. Scrollable objects expose this interface.

3. panorama-syncs; these are abstractions in the API that co-ordinate a scrollable view with the scroll-bar controls, with behaviour provided by one of two boss classes named xxxScrollBarPanoramaSyncBoss. They are really for internal use.

One fundamental area of application of these elements is in the layout widget. Scroll-bars and panoramas occur widely within the application plug-in codebase; panorama-syncs are somewhat more exotic.

Scroll-bars are widgets with behaviour deriving from kScrollBarWidgetBoss. The scroll-bar data model is accessed through the ICounterControlData interface. Panoramas are aspects of objects that scroll; the IPanorama interface can be used to scroll a view. The boss classes in the list below expose an IPanorama interface.

Design

The boss class kScrollBarWidgetBoss extends kBaseWidgetBoss, and is bound to the ODFRez custom resource type ScrollBarWidget. When a scroll-bar is added to a control such as a list-box, it works hand-in-hand with a panorama (IPanorama) to produce the correct scrolling behaviour of the dependent widgets. The scroll-bar receives input from the end-user, and the implementation code manipulates the view through the IPanorama interface. For instance, the class named kCellPanelBoss is normally the only immediate child of a widget list-box widget. It has a panorama interface (IPanorama) used to scroll the cells. This cell-panel widget is needed
because if the panorama was on the list box view itself, the scroll bars and borders would scroll along with the cells.

**Widget data model**

The class named kScrollBarWidgetBoss exposes an ICounterControlData, which is the data-model for a scroll-bar widget.

**Capabilities**

The interface ICounterControlData is the key interface for scroll-bars, used to access the data model of a scroll bar widget boss object and change its state. The IPanorama interface can be used to manipulate a scrollable object’s view.

**Notification**

An observer requesting changes in the data model of a scroll-bar widget boss object should attach to its ICounterControlData interface and request notification on the IID_ICOUNTERCONTROLDATA protocol. When the counter control data changes, registered observers are notified with an Update message with classID parameter of kCounterChangeStateMessage. The client also gets passed a pointer to a CounterControlUpdateParams object as the changedBy parameter of an IObserver::Update message.

---

**Sliders**

**Description**

A slider control is appropriate for situations where an end-user wishes to enter a parameter that can vary continuously over a finite, determinate range. There are several standard sliders (descendant of kStdSliderWidgetBoss) in the application user interface; for instance, on the Color or Navigator panels, as shown below. A popup slider (descendant of kPopupSliderBoss) can be used in the same situation, when the requirement is that the slider should only appear when the user wishes to change a particular value rather than being continuously visible. Examples of the standard slider are shown in Figure 36. A popup slider can be seen in Figure 37.
Figure 36  Some slider widgets

Figure 37  Popup slider combo-box widget

An example of a popup slider can be found in the transparency panel. This slider controls the opacity of the selected page items. There is also a popup slider on the swatches panel, which can be used to vary the tint associated with a particular colour in a swatch. This uses a class of popup slider specialised for the input of real-valued numbers.

This screenshot shows a popup slider from the Transparency panel.

Popup slider combo-box controls consist of a composite of an edit box and a button, which are child widgets of a panel. The popup slider appears only when the end-user clicks on a button to show the slider in its own floating window. This popup slider is a relatively complex control, consisting of a co-operating assembly of an edit box, a button, a floating window and a slider, which are owned and/or managed by a parent boss object that is a descendant of the generic panel widget boss class (kGenericPanelWidgetBoss). It does mean that it is possible to pack more user-interface into a given area of screen “real-estate” than would hitherto have been possible with standard slider controls.

Table 32  Slider Boss Classes

<table>
<thead>
<tr>
<th>Widget boss class</th>
<th>ODFRez custom type</th>
<th>Example of use</th>
</tr>
</thead>
<tbody>
<tr>
<td>kStdSliderWidgetBoss</td>
<td>(none: you create your own)</td>
<td>Slider that is continuously available, e.g. on Color panel</td>
</tr>
<tr>
<td>kRealPopupSliderComboBoxWidgetBoss</td>
<td>RealPopupSliderComboBoxWidget</td>
<td>Swatches panel</td>
</tr>
<tr>
<td>kIntPopupSliderComboBoxWidgetBoss</td>
<td>IntPopupSliderComboBoxWidget</td>
<td>Inks panel</td>
</tr>
<tr>
<td>kPercentPopupSliderComboBoxWidgetBoss</td>
<td>PercentPopupSliderComboBoxWidget</td>
<td>Transparency panel</td>
</tr>
</tbody>
</table>
Design

To use the standard slider widget (kStdSliderWidgetBoss) it is necessary to create a new ODFRez custom resource type which can extend the ODFRez custom resource type CControlView and add an ODFRez SliderControlData field. This new type can then be bound to the subclass of kStdSliderWidgetBoss (e.g. k<whatever>SliderWidgetBoss). This new ODFRez custom resource type would be bound to the boss class delivered by one's own plug-in that extended the boss class named kStdSliderWidgetBoss. There are other subtle dependencies and interfaces required for the standard slider implementation to work correctly.

The first thing to note is that the slider popup widget is a compound widget, consisting of a panel, an edit box, a button widget and a slider. It is described as a combo-box slider, as it functions as if it were a combo-box, with a slider replacing the traditional combo-box drop-down list. A boss class named kPopupSliderComboBoxWidgetBoss provides the core of the behaviour for the combo-box slider. This boss class extends the boss class named kComboBoxWidgetBoss. Note that this boss class named kPopupSliderComboBoxWidgetBoss should not be used directly- rather, use one of its subclasses, which have specified measurement units associated. A popup slider therefore has much functionality in common with combo-box widgets. A popup-up slider has part of its behaviour provided by the boss class kPopupSliderBoss which extends the standard slider widget boss (kStdSliderWidgetBoss).

There is one point that may cause a degree of confusion. The popup slider widget does not at the time of writing require an ODFRez CSliderControlData field to be initialised. The observant reader might be thinking; if that's the case, how does one specify the range of the slider? Internally, the slider widget on the floating window does have an ISliderControlData interface since of course the class kPopupSliderBoss derives from kStdSliderWidgetBoss. However, a popup slider is initialised by the settings obtained by query through the ITextDataValidation interface, which are then used to specify the range through the interior ISliderControlData interface on the kPopupSliderBoss. It is the fields of the ODFRez type TextDataValidation in the ODFRez data statements defining the popup slider which specify the maximum and minimum slider values.

Widget data model

The state of the standard slider is represented by ISliderControlData, and changes in this state are broadcast along IID_ISLIDERCONTROLDATA.

The state of a popup slider is represented by ITextValue, but changes are broadcast with the protocol IID_ISLIDERCONTROLDATA.

Capabilities

One interface aggregated on the boss class named kStdSliderWidgetBoss is the ISliderControlData interface. This encapsulates information about the maximum and minimum value exposed in the control’s range, and the current value of the slider setting. Changes to the data value result in notifications with ClassID equivalent to kRealChangeStateMessage being sent to registered observers of the slider boss object. The boss class aggregates an IEventHandler interface. This should not need to be overridden. The event handler with implementation ID of kCSliderEHImpl is responsible for creating a tracker if
required and calling the tracker methods in appropriate sequence. There is an IControlView interface on the slider control; the default implementation makes use of the ISliderStateData and ISliderControlData interfaces on the widget boss object to determine how to draw the visual representation of the control.

When the data model of a popup slider changes, an IObserver::Update message is sent to attached observers; interested client code should query for the state of the pop-up slider by using the ITextValue interface. This allows the value of the slider state (percent, real or integer) to be queried. Similarly to set the state of the popup slider, the ITextValue interface should be used. Other properties of the popup slider can be queried through the ISliderControlData interface. For instance, this is used to determine the minimum or maximum of the popup slider range.

**Notification**

Notifications about changes in state of a standard slider control are sent along the IID_ISLIDERCONTROLDATA protocol to registered observers. The usual process in processing the IObserver::Update message is to query the data value through ISliderControlData::GetValue to determine the control-state. An IObserver interface should be added to the kStdSliderWidgetBoss, by subclassing. On receiving an AutoAttach message from the application core, the client code should query for the ISubject interface on the widget boss object and request notifications along the IID_ISLIDERCONTROLDATA protocol. The observer should as usual detach in polite fashion in AutoDetach. When changes occur in the slider control’s data, IObserver::Update messages will be sent to this observer. A message with ClassID of kRealChangeStateMessage will be sent to the observer. The ISliderControlData interface can then be used to retrieve the current data value associated with the slider.

Popup slider notifications are also sent along the IID_ISLIDERCONTROLDATA protocol; however, the ClassID of the message is unique to the popup slider and should be one of the small set of messages given below. The messages are as follows:

1. kPopupSliderOpenMessage, sent when the slider opens,
2. kPopupSliderApplyChangeMessage, sent when the value is being committed by the end user,
3. kPopupSliderCloseMessage, sent when the slider is about to close.

When the data model of a popup slider has changed, client code should query for the state of the pop-up slider by using the ITextValue interface. This allows the value of the slider state (percent, real or integer) to be queried. Similarly to set the state of the popup slider, the ITextValue interface should be used. Other properties of the popup slider can be queried through the ISliderControlData interface. For instance, this is used to determine the minimum or maximum of the popup slider range.

**Implementation detail**

The superclass for all the popup sliders is kPopupSliderComboBoxWidgetBoss. The core behaviour of the popup sliders comes from the IControlView implementation aggregated on this boss class. Another boss class named kPopupSliderButtonBoss provides the behaviour
behind the button on the popup slider combo-box widget. The IEventHandler implementation aggregated on this kPopupSliderButtonBoss class is responsible for creating a floating window and adding a child widget which is the slider the user manipulates.

The appearance of the popup slider is determined by the implementation of the IControlView interface with identifier kPopupSliderComboBoxViewImpl. This implementation is responsible for creating the child widgets that provide the behaviour of the popup slider. When the popup slider is restored from persistent data (plug-in binary resource first time, or saved-data database if its representation exists in it), it creates an object of class kPopupSliderButtonBoss to provide the behaviour behind the button that shows the slider. An object of class kComboBoxEditBoxWidgetBoss is also created to support the behaviour of the edit-box in the popup slider combo-box widget. The popup slider itself (kPopupSliderBoss) is created on demand by the class that implements the IEventHandler interface for the composite popup slider combo box widget. The slider is typically shown on a left-button down event. The implementation creates an object of class kWindowBoss and specifies that it is a floating window, then creates an object of class kPopupSliderBoss and adds it as a child of this new window boss object.

**Tree-view widgets**

**Description**

A tree-view widget lets you display hierarchical data; users can expand nodes in the tree to see more detail, and contract nodes to reduce the amount of detail displayed. A tree-view widget can have vertical and/or horizontal scroll-bars to allow end-users to scroll around within the view of the tree-structure. An example of this control can be seen in Figure 38.

**Figure 38** Widgets involved in the tree-view

This screenshot shows a tree-view and indicates the widget boss classes that provide the behaviour for the components of the tree-view. The class kTreeViewWidgetBoss provides the behaviour for the main control; the class kTreeNodeWidgetBoss provides the bulk of behaviour for nodes in the tree, and kTreeExpanderWidgetBoss provides...
User Interface Widgets

Tree-view widgets

the behaviour of the triangle-icon that can be used to show or hide the children shown for a specific node.

The sections following describe how to work with the tree-view control. The control is relatively complex and can be tricky to use. There is an SDK sample that use the tree-view control, PanelTreeView, which shows how to use these controls. this can be found in the folder `<sdk>/source/sdksamples/paneltreeview`.

The API to use a tree-view control is written differently to platform-specific APIs, such as the API to the TreeView control on Windows or the Java/Swing tree-control. Instead of plug-in client code adding nodes to an InDesign tree-view control directly, client code is required to provide the means for navigating through its own tree model. The tree-view control widget boss object (kTreeViewWidgetBoss) handles the navigation, sends queries about the tree-model, and asks for widgets for a particular node when it needs them. This is carried out by the framework sending messages to the client code, asking for, say, a root node in the tree, or a widget that can be used to display a particular uniquely identified node.

**Table 33 Tree View Boss Classes**

<table>
<thead>
<tr>
<th>Widget boss class</th>
<th>ODFRez custom type</th>
<th>Responsibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>kTreeViewWidgetBoss</td>
<td>TreeViewWidget</td>
<td>Provides main behaviour for tree-control</td>
</tr>
<tr>
<td>kTreeNodeWidgetBoss</td>
<td>TreeNodeWidget</td>
<td>Represents node in tree view. Subclass to hear expand/collapse events with an observer or override event handler</td>
</tr>
<tr>
<td>kTreeExpanderWidgetBoss</td>
<td>TreeExpanderWidget</td>
<td>Provides behaviour for widget that expands/collapses nodes of tree with children</td>
</tr>
</tbody>
</table>

The essence of working with the tree view control is to subclass kTreeViewWidgetBoss, and aggregate on it your implementation of the ITreeViewHierarchyAdapter interface, which enables you to *adapt* your data model to the needs of the InDesign tree-view control. You are also required to provide an implementation of the ITreeViewWidgetMgr interface, which allows your client code to specify which user-interface widgets to create to display a given node, and also allows your client code to specify how the widgets should be placed.

Client code is required to notify the application framework when changes in the tree model have taken place, such as nodes being added or deleted. This helps to keep the tree-model (based on your data) and the tree-view synchronised. The key responsibilities of client code are as follows:

1. to return representation of nodes at given locations in the tree's data model, for instance, the root, or at a given index in the children of a given node
2. to manufacture widgets appropriate to render tree nodes given a uniquely identified node
3. to render views of data at given uniquely identified nodes
4. optionally, to provide information about the intended geometry of rendered nodes within
the tree.

Client code should provide methods for returning the root node and also the parent or child
node at a given index from any given node. Each node within the tree can have an associated
data item. This might be a UID, in which case an API class (UIDNodeID) can be used to
represent each node. Alternatively, the data items may be entirely custom items that are unique
to your client code, in which case it may be necessary to subclass an API type. Each node
should be identified uniquely. This enables the application core to render the correct view of
the underlying data model provided by the client code.

On update of the tree view, the application core may pass into the client code a reference to any
node; it is up to the client code to provide an appropriate widget to render the node and
populate the widget with a view of the associated data.

**Adapter pattern**

The tree-view control in the API makes use of the design pattern named Adapter, described in
[Gamma et al, 1995]. At its most basic, the adapter pattern allows two existing but
incompatible classes to work together without changing the public interface of either. The
classes that are required to work together are named in the pattern as Client and Adaptee.
Client here should not be understood to be anything to do with plug-in “client-code”. An
instance of the Client sends particular messages to the Target, an abstract API that is
implemented by the Adapter. The messages sent by the Client are translated by the Adapter
into method calls on an Adaptee. The Adapter then manufactures an appropriate return value,
if required, based on the Adaptee's response. In terms of the tree-control, the roles are occupied
as follows:

1. Client: application core, specifically, the widget run-time subsystem (not plug-in client
code),

2. Target: the abstract API on ITreeViewHierarchyAdapter,

3. Adapter: plug-in code’s implementation of ITreeViewHierarchyAdapter,

4. Adaptee: the data model, which might be for instance, a tree model representing the XML
logical structure of a document.

**Factory method pattern**

The Factory Method pattern described in [Gamma et al, 1995] is used in implementing a
working tree-view control. This pattern can be used when a framework needs to instantiate
classes, but only has knowledge of abstract classes, which cannot be instantiated directly. In
implementing a tree-control, a parameterised factory method is used, where an identifier is
passed in by the framework specifying what type of object to manufacture. There is a factory
method ITreeViewWidgetMgr::CreateWidgetForNode which is parameterised by passing a
reference to the node currently being rendered. The responsibility of client code is to return a
new instance of an IControlView interface on a newly-created widget boss object that can be
used to render the node’s associated data.
Design

The core behaviour for the tree view widget is determined by the class named kTreeViewWidgetBoss. The first responsibility of client code is to subclass this boss class, and create one’s own boss class that provides implementations of ITreeViewHierarchyAdapter and ITreeViewWidgetMgr.

The behaviour of individual nodes in the tree-view control is provided by the boss class named kTreeNodeWidgetBoss. Subclass this where you wish to add an IObserver interface to obtain notification about, for instance, expansion/contraction events associated with tree nodes.

There is a tree expander widget, with behaviour provided by the boss class named kTreeExpanderWidgetBoss. Key boss classes and their responsibilities are listed below.

There is an ODFRez custom resource type named TreeViewWidget. Since it is necessary for client code to subclass kTreeViewWidgetBoss, it will always be necessary to perform a corresponding subclass of TreeViewWidget and bind it to the new widget boss class, say, kMyTreeViewWidgetBoss. The ODFRez custom resource type named TreeExpanderWidget can be used on panels used to render individual nodes in the tree. This widget allows the descendant nodes of a given node to be displayed or collapsed. If there are no children associated with a given node, a correct implementation is to hide this widget.

There should also be a new ODFRez custom resource type created for displaying each node in the tree-view control. For tree-views located on panels, a subclass of PrimaryResourcePanelWidget is appropriate. For tree-view controls located on dialogs, an erasable panel is required, as the tree-view control does not know anything about erasing its own background. The SDK sample PanelTreeView shows how to use a tree-view control on a dialog, and also how to receive notification about several different tree-related events.

Widget data model

There are some APIs where there is an explicit tree-model behind the tree-control (for instance, in Java Swing, say). However, the InDesign API control leaves it up to the client code to determine how the tree data-model is represented and requires only that client code is able to provide certain kinds of information as requested about its data-model when asked; for instance, how many children do you have?

Other controls have a simpler data model, and a simple notification structure that relates pretty directly to this data-model. For instance, an edit box has an ITExtControlData interface which when changed notifies along IID_ITEXTCONTROLDATA. The situation is more complex for tree-view controls and is explored in the “Notification” section below.

Capabilities

ITreeViewHierarchyAdapter

Along with ITreeViewWidgetMgr, this is one of the two interfaces that you should implement to obtain a working tree-view control. The ITreeViewHierarchyAdapter interface provides the means for navigating through your tree model. It is like the adapter pattern in that it provides a way for the TreeView widget to navigate your tree-model without requiring your tree-model to conform to any given interface.
**ITreeViewWidgetMgr**

You should provide an implementation of this interface. ITreeViewWidgetMgr is where you create the tree-view control widgets in your client code and apply node data to the widget. There is a partial implementation of ITreeViewWidgetMgr called CTreeViewWidgetMgr that takes care of the widget placement implementation and simplifies the implementation of this interface.

**ITreeViewMgr**

This interface is aggregated on the boss class kTreeViewWidgetBoss; you are not expected to provide your own implementation of this interface. The principal responsibility of this interface is to keep the tree-view synchronised with changes to the tree-model. Client code should call methods in this interface to let the tree know when changes to the tree have occurred.

For instance, client code should call ITreeViewMgr::ChangeRoot() to initialize the tree-view control. Call this whenever you want to regenerate the tree; for instance when the tree-view control was in a state where it didn't hear about changes to the data model and it wants to start afresh. There is also a debug-only version of ChangeRoot() that can be used to validate an implementation of ITreeViewHierarchyAdapter.

Client code calls ITreeViewMgr::NodeAdded() and ITreeViewMgr::BeforeNodeDeleted() when nodes are added and about to be removed from the tree model, respectively. Failure to call these methods when changes occur in the tree model will cause problems in the tree view, for instance, nodes disappearing unintentionally.

ITreeViewMgr::NodeChanged() should be called when a nodes data has changed that will not affect the nodes height. If the change will affect the node's height, call ITreeViewMgr::BeforeNodeDeleted(), followed by ITreeViewMgr::NodeAdded().

**Uniquely identifying each node**

On examining the ITreeViewHierarchyAdapter interface, you’ll notice a class NodeID which is used extensively. This is best explained by looking at the problem it is trying to solve. When the tree view asks for the Root node, the client needs to uniquely identify that node. In a simple case, a UID may be all that is needed to identify the node. In a simple case, a UID may be all that is needed to identify the node.

However, not all tree-models have UIDs associated with each node; for instance, the SDK sample PanelTreeView represents views onto the local file-system and each node encapsulates a path in the file system and not a UID-based object. That leads us to returning a class that can be specified by the client, but that derives from a class providing the methods required by the tree-view control.

The main issue with that is that we would need to return a pointer to this class, which leads to potential problems with cleanup. To make clean-up easier, the framework uses the NodeID class as a *smart pointer*. NodeID is a class that holds and deletes pointers to NodeIDClass classes.

If your tree is UID-based, there is an existing NodeIDClass based class that you can use called UIDNodeID. If you require different data to identify your tree nodes, you'll need to create your own class based on NodeIDClass.
NodeID and friends

There are several classes that can be found as arguments and return types in the interfaces that should be implemented to create a working tree-view control. These are defined in the API header file named NodeID.h. These have closely similar names and confusion can arise between NodeID, which is akin to a smart pointer for the type NodeIDClass, and other classes such as UIDNodeID, which is a subclass of NodeIDClass. There is also a smart pointer TreeNodePtr which is used for descendants of type NodeIDClass, e.g UIDNodeID. The naming in this area needs careful study before programming.

As discussed earlier, every node need to be identified uniquely. In InDesign, there is a class called NodeIDClass that represents an individual node. There is a sub-class of this class available in the API, which is called UIDNodeID. In reading UIDNodeID, for the moment remember that it should be suffixed -Class.

To muddy the water a little more, there is a class called NodeID is akin to a smart pointer, that is used to manage the lifetime of dynamically created NodeIDClass objects. This class is found as a parameter on methods in ITreeViewHierarchyAdapter and ITreeViewWidgetMgr.

Note that in certain situation, it may be adequate to use a UID to identify a tree node; there is a class in the API called UIDNodeID that can be used for these cases, otherwise you need to implement your own subclass of NodeIDClass that identifies each node uniquely. See PanelTreeView for an example of how this can be done.

### Table 34 C++ Classes relating to the tree-view widget and their responsibility

<table>
<thead>
<tr>
<th>C++ class</th>
<th>Responsibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>NodeIDClass</td>
<td>represents the data associated with each node, and allows each node to be uniquely identified; this type can be subclassed to create custom data nodes</td>
</tr>
<tr>
<td>TreeNodePtr</td>
<td>a smart pointer that is the equivalent of InterfacePtr for NodeIDClass descendants</td>
</tr>
<tr>
<td>NodeID_rv</td>
<td>a class for method return values similar to NodeID, except that gives up control of the encapsulated NodeIDClass rather than copying it</td>
</tr>
<tr>
<td>NodeID</td>
<td>manages lifetime of objects that are of type NodeIDClass.</td>
</tr>
</tbody>
</table>

Node recycling

The tree view widget contains instantiated widgets for the nodes that are visible. When the end-user scroll downs, for example, the node that was on top will no longer be visible and there will be a new visible node at the bottom. The application framework will remove the top widget, and if it is of the same type (based on the WidgetID obtained from ITreeViewWidgetMgr::CreateWidgetForNode()) as the widget that will be needed for the bottom node, the framework will reuse the widget for the bottom node. This is one reason why the method named ITreeViewWidgetMgr::CreateWidgetForNode() and the method named ITreeViewWidgetMgr::ApplyNodeIDToWidget() are distinct. It is only necessary for the application framework to create new widgets when there are no widgets of the right type to recycle.
**Tree view attributes**

The tree-view control also has attributes associated with it which determine scroll information and whether the root item is shown. The attributes are set in the resource for the `TreeView`. Vertical and Horizontal scroll bars are provided by the tree view (if you want them), so don’t add them yourself. You can specify in the resource whether you want no scroll bar, vertical only or horizontal and vertical.

The scroll amounts for both horizontal and vertical scroll bars are also set in the resource. There are two values that you must set for each scroll bar. The scroll button increment is the number of pixels that will scroll when you click on either of the scroll buttons. The thumb scroll increment is the smallest number of pixels that will scroll when an end-user moves the scroll bar thumb. For instance, tree-view control can be configured so that the scroll button would move 20 pixels, but the thumb scroll just one pixel for greater accuracy. The thumb scroll increment must be a factor of the scroll button increment.

Another attribute on the `TreeView` specifies whether or not to display the root element. You should have only one root element, but you don’t necessarily want to show that root element. This would be used, for example, when using the tree-view control in place of a standard list box; for instance, there is a defect in the list-box for InDesign that prevents it correctly displaying dependent controls with a platform window such as edit boxes or drop-down lists, and using a tree-view control would be appropriate for this case.

**Notification**

There are at least two kinds of change which client code might be interested in:

1. the end-user can change which node is selected in the tree-view control. To receive notifications about this, you should attach an observer (IObserver implementation of your own) to the ISubject interface of your `kTreeViewWidgetBoss` subclass, and listen along protocol IID_ITREEVIEWCONTROLLER.

2. a node in the tree can expand or collapse. In this case, you should attach an observer (IObserver) to the ISubject interface of the `kTreeNodeWidgetBoss` subclass, and listen along the protocol IID_ITREEVIEWMGR.

It is also possible to receive notification about events such as double-clicks within the nodes of the tree-view if you use the shadow (proxy) event handler pattern shown in the PanelTreeView SDK sample.

**Implementation specifics**

**Implementing ITreeViewHierarchyAdapter**

The interface named ITreeViewHierarchyAdapter provides the means for navigating your tree. In its `GetRootNode()` method, you should return an instance of a node that represent your root node, with data allowing it to be uniquely identified as such.

The method ITreeViewHierarchyAdapter::GetParentNode() requests the parent node of a given node. If the given node is a root node, there is no parent for a root node, so you return nil, otherwise, dependent on the location of the node in your tree-model, you should return its parent.
In implementing \texttt{ITreeViewHierarchyAdapter::GetNumChildren()}, you will return the number of children given a node that’s in your tree-model. The method \texttt{ITreeViewHierarchyAdapter::GetNthChild()} is the counterpart to \texttt{ITreeViewHierarchyAdapter::GetParentNode()}. Since a node may have multiple children, but only one parent, this method will pass in the index position of the child node it is querying.

In implementing \texttt{ITreeViewHierarchyAdapter::GetChildIndex()}, the index here refers to the index position used in the \texttt{ITreeViewHierarchyAdapter::GetNthChild()} and the range of the index should be from 0 to the (number of children-1).

In implementing the method \texttt{ITreeViewHierarchyAdapter::GetGenericNodeID()}, you can simply return a dummy node that makes a generic node. This method is used primarily for persistence. When something is purged and the application framework has to write out a NodeIDClass, it uses the \texttt{ReadWrite()} method on the NodeIDClass. When the application framework needs to read it back in, it needs to be able to create instances of that NodeIDClass. It uses \texttt{ITreeViewHierarchyAdapter::GetGenericNodeID()} to create an instance and then call \texttt{ReadWrite()} on the NodeIDClass to initialize it.

\textbf{Implementing ITreeViewWidgetMgr}

The next implementation you’ll need to provide is for \texttt{ITreeViewWidgetMgr}. This interface is similar to a widget factory method, because this is where you will create widgets for the nodes in the tree. The two main methods are \texttt{CreateWidgetForNode()} and \texttt{ApplyNodeIDToWidget()}. The purpose of \texttt{ITreeViewWidgetMgr::CreateWidgetForNode()} is to create the right widget for the NodeID passed in. In this method, just create the widget, but don’t change the widget-data to match the node.

Changing the widget-data, e.g., setting the text of a static text widget on a panel to render the data associated with a node, happens in \texttt{ITreeViewWidgetMgr::ApplyNodeIDToWidget()}. These two methods are distinct because the application framework reuses widgets not needed. The framework creates the widget once in \texttt{ITreeViewWidgetMgr::CreateWidgetForNode()}, and then uses it several times by calling \texttt{ITreeViewWidgetMgr::ApplyNodeIDToWidget()} and passing in a different NodeID. The framework is not continually creating and deleting widgets.

The implementation of the method \texttt{ITreeViewWidgetMgr::CreateWidgetForNode()} is where client code creates the widget from the resource given a node. When the application framework wants to reuse a widget, it calls \texttt{ITreeViewWidgetMgr::GetWidgetTypeForNode()} to determine what widgets can be used with that node. \texttt{GetWidgetTypeForNode()} returns a widget id that corresponds to the type of widget the passed given belongs to.

When creating widgets for different types of nodes, the method named \texttt{ITreeViewWidgetMgr::GetWidgetTypeForNode()} becomes significant. In the \texttt{ITreeViewWidgetMgr::CreateWidgetForNode()} method, you can look at the NodeID and determine what type of node to create.

When the framework wants to re-use a widget as part of its optimisation scheme, it will call \texttt{GetWidgetTypeForNode()} to determine what widgets can be used with that node. The framework uses the widget’s WidgetID to discriminate between different node types in the tree. The framework will only call \texttt{ITreeViewWidgetMgr::ApplyNodeIDToWidget()} with a widget that has the WidgetID returned in \texttt{ITreeViewWidgetMgr::GetWidgetTypeForNode()}. For this reason it is important that all widgets of the same type have the same widget ID.
User Interface Widgets

Tree-view widgets