User Interface Programming Model

Technical Note #10050

Version InDesign 2.0

20 Aug 2002

ADOBE SYSTEMS INCORPORATED

Corporate Headquarters

345 Park Avenue

San Jose, CA 95110-2704

(408) 536-6000

Rev #	Date	Author	Comments
0.1	20 Aug 2002	Ian Paterson	Brought content in from the old uihowtos and integrated the architecture content into this document to make a single, coherent piece of documentation.

Contents

#10050 User Interface Programming Model

Overview

This document introduces the user interface programming model; it is intended to provide a context for the other resources that describe user-interface programming. It is intended to provide some insights into user interface programming that complement and extend those offered in the introductory tech-note #10045 on writing your first InDesign plug-in.

This document describes other key aspects of the user interface architecture, such as type binding between widget boss classes or API interfaces and ODFRez custom resource types, and explains the role played by ODRez resources in defining plug-in user interfaces. Another key concept explained is how widgets read their initial state and how they persist their state, and how this relates to the representation in ODFRez.

Who should read this document

You should read this document if you need to create user interfaces for InDesign plug-ins and want to understand the programming model. The target audience is experienced programmers, who nevertheless may be comparatively new to InDesign programming.

Before you begin

This document assumes that the reader is an experienced programmer who is comfortable with object-oriented programming and has created user interfaces with other programming models. However, it is assumed that the reader may only have a limited experience of InDesign programming and be looking at this document for more detail to help them understand the user interface programming model after having read a "Getting Started" text. It would be extremely helpful if the reader is familiar with the following resources:

- "Making your first InDesign plug-in" (resource #10045)
- Browseable API documentation that ships with the SDK or the on-line version on http://partners.adobe.com under "SDKs >> InDesign >> Exploded SDK".
- References such as IObjectModel_<whatever>.xls and/or InterfaceList.txt.

After reading this document

This document describes the user interface in theoretical terms and with one or two examples. Once you have read this document you should be familiar with some of the key principles required to create rich plug-in user interfaces.

Goals

This document has the following objectives:

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

Terminology and definitions

This section introduces the terms needed to understand this document. The term “widget” is particularly prone to confusion and is always qualified in this document to make its meaning unambiguous. For example, “static text widget” is highly ambiguous in the context of the InDesign API and could refer to one of at least four things. This expression does not specify whether one is referring to the boss class that provides the behaviour (kStaticTextWidgetBoss), 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. Action components (IActionComponent) in InDesign 2 have completely replaced menu components (obsoleting IMenuComponent) in the x API. An ActionDef ODFRez custom resource specifies what action component is responsible for handling an action by ID, and a MenuDef ODFRez custom resource binds a menu item onto a specific action component.
- *aggregation*; a boss class is said to aggregate an *interface* if is provides an implementation of that interface. A boss class may also be said to expose an interface of a specific type. InDesign 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.
- *boss class*; A boss class is a compound type expression. A boss class declaration promises implementations of interfaces and associations implementations with these interfaces. The complete class definition, by strict analogy with C++, consists of both the interfaces promised, along with the C++ implementation code. A convenient way of thinking about a boss class is that it is like (but not identical with) a C++ class that has multiple mix-in base classes; however, bear in mind that this is only an analogy, and the implementation of the object model in the API is quite different. It implements a specified set of interfaces and derives its own particular characteristics from both the semantics of those interfaces, and the specific implementation that it exploits. For instance, widgets all have an IControlView interface; the precise appearance of a given widget depends on the underlying IControlView implementation, which will be different for a static text widget versus a list-box widget. At its simplest, it is a table associating interfaces (by identifier, IID_<whatever>) with implementations (by identifier, k<whatever>Impl). The object model allows boss classes to be instantiated, indirectly through factory methods rather than the programmer having to write an explicit constructor.
- *boss object*; an instance of a boss class. For instance, a dialog might be an instance of kDialogBoss (or a subclass). A boss object can be manipulated through any interface it aggregates, although in the user interface domain, the commonest way to acquire a reference to a boss object is through its IControlView interface.
- *client code*; code written, say, by a third-party plug-in developer that exploits the services of the InDesign 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 th e 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.

- *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 API. Notifications about changes in control state are sent as IObserver::Update messages, and this is true since InDesign 2.0 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 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 API by the interface IObserver. Typically implementations will use CObserver 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 tools DollyWizard and Freddy

are able to generate ODFRez, and Freddy can manipulate snippets of ODFRez data defining widgets.

- *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 CObserver 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.
- *resources, framework*; these are used to initialise the widgets, and are held in files written in ODFRez. These files also contains 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 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.
- *subclassing*; this is used to refer to the process of extending both a C++ class, typically a partial implementation such as CActionComponent , deriving from a boss class or extending an ODFRez custom resource type.
- *subject*; abstraction that is the target for the attention of observers. In the user interface API, widget boss objects are the typical subjects.
- *superclass*; the immediate ancestor in the inheritance hierarchy of a particular boss class, ODFRez custom resource type or C++ class.
- *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 ODRez 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 required to correctly de-serialise a particular widget boss object from the plug-in resource.
- *widget boss object*; an instantiated widget boss class. Perhaps the commonest way to manipulate widget boss objects from client code is through an IControlView pointer, since this is aggregated on any widget boss object that has a visual representation.

Introduction

This section provides a context for the discussion of the user interface programming model and provides a brief roadmap for the document. Plug-in developers may have to implement a user-interface relatively early in their InDesign programming experience. At this point they may have only a modest grasp on the concepts of boss classes, aggregation and so on. They can be further disheartened by encountering an additional layer of complexity when programming user interfaces, where 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 document is to de-mystify the user interface programming model and provide enough background to use existing reference materials to create richly responsive user interfaces for InDesign 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 programming that newcomers to the API encounter- however, at first encounter with the InDesign user-interface programming model, it may seem relatively complex and frustrating to develop a user interface. This document and 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 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, say. It is very 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 2 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 worked example for some of the more abstract discussions.

Key concepts

This section introduces some of the key concepts in the user-interface that are an essential basis to a deeper understanding of how to program user interfaces of InDesign plug-ins.

Design objectives

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

- enable cross-platform user interface development,
- create re-useable UI components with rich behaviours,
- provide a well-factored design that separated data and presentation,

To meet these objectives, the engineering team created a sophisticated user interface design that provides high support for re-use, with the possibility of customising widget behaviour and appearance, and that encapsulates platform dependencies to a high degree.

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 artefact 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, e.g. TblSort.fr is the top-level framework resource file for the plug-in described in TableSorterDesign.
- 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 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- allows 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 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 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. This helps tools such as Freddy, a basic ODFRez snippet editor, described in resource #10026, function more reliably.

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.

Some namespaces that are commonly used in defining identifiers are shown below; you will find ample examples of use of the macros to define identifiers (DECLARE_PMID) in any of the SDK sample plug-ins.

Namespace	Scope	Use
kClassIDSpace	global	defining boss class identifiers
kInterfaceIDSpace	global	defining interface identifiers or IIDs
kImplementationIDSpace	global	defining implementation identifiers for boss class definitions

Namespace	Scope	Use
kWidgetIDSpace	global	defining the widget ID associated with a widget boss object's IControlView
kServiceIDSpace	global	for registering services
kErrorIDSpace	global	for naming unique error codes
kPlugInIDSpace	global	uniquely naming plug-ins loaded by an application
string-table key space	global	defining keys for mapping into a localised string for output in a view
plug-in resource ID space	plug-in scope	defining indices in plug-in resource tables

Abstractions and re-use

This section highlights the major abstractions in the InDesign API, and 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 programming that is required to understand this document.

Key abstractions within the API take two forms; boss classes and interface types. An interface in the context of the API is an abstract C++ class that extends IPMUnknown. The abstract C++ class named IPMUnknown is at the root of the inheritance hierarchy for interfaces defined in the API. There are a small handful of abstract C++ types in the API that will be encountered, the most frequently occurring of which is IDataBase. Interfaces that are ancestors of IPMUnknown support the reference-counting mechanism at the heart of InDesign's memory management, by analogy with the Microsoft COM architecture and the role of IUnknown. The smart pointer type InterfacePtr should only be used to encapsulate the 'true' interfaces that are descendants of IPMUnknown.

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. I n 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 for one or two reasons; for instance, 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).

InDesign widgets versus platform controls

This section describes the relation between InDesign 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 widgets should be performed through the InDesign API and not throught platform-specific APIs. In addition, InDesign 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 (kIntEditBoxWidgetBoss) 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 one of the key factorings in the InDesign plug-in set; plug-ins are decomposed into “model” plug-ins and “user interface” plug-ins. The core capabilities of the InDesign 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. 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; see PreviewableDialogDesign for a worked example within the SDK sample set.

Suites and the user interface

This section describes selection suites, which 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. If writing code which manipulates the “model” underlying an active selection made by an end-user, you should write a suite. The new selection architecture that brought in suites is described in resource #10006.

Suites were added to InDesign at version 2.0, and represented a major improvement in the factorisation of the API. These allow client code to be insulated from the precise details of both the selection format and the underlying details of the model associated with each selection format. Client code needs only to care whether the abstract selection supports a particular type of capability; for instance, does the abstract selection let me apply a markup-tag to it? If the abstract selection does support this capability, I can go ahead and do it, without caring whether the selection is a text frame, a graphics frame, a range of text or some other document object. In the 1.x architecture, client code had to be very closely coupled to the model and typically required quite different code paths for different selection formats. For instance, ITableSuite and IXMLTagSuite are two suite interfaces that (respectively) support changing tables and the markup properties of document objects in an abstract selection.

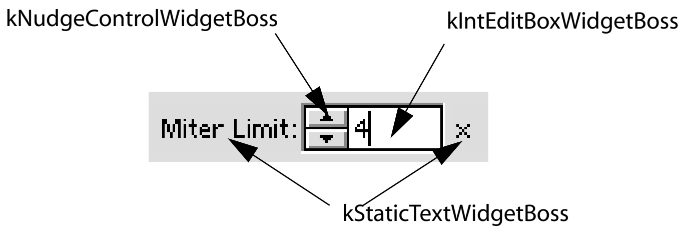
Suites are particular germane to writing user-interface code, since they make the process very much simpler. Whenever an end-user is required to manipulate document objects by making active selections, you should attempt to make maximum use of suites. If the suites do not already exist, then ideally you should factor your code to write your own suites; to he cleanest design would be that you'd have a minimum of two plug-ins, one of which is a UI plug-in, and one that delivers the suites but that is devoid of user-interface functionality. It then becomes straightforward to write the user-interface component.

When implementing suites using the templates provided in the API, each of the methods in suite implementation are bracketed in a command sequence, so there is no need to worry about whether to implement commands to make direct changes to the model.

Example of a user interface

This section introduces a concrete example of a user interface, in an attempt to illustrate some of the key concepts introduced above- it specifies which boss classes are responsible for the behaviour of the user interface elements, and the nature of the type binding for the example, which is one of the main concepts to grasp to become successful in programming the user interface API.

The graphic below shows widget boss classes and associated ODFRez resource types that provide 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 can be used to increment or decrement values in the edit-control by a specified amount.



There are two static text widgets displaying text “Miter Limit:” and “x”; the text that these display can be different across locales. The text to display will be determined by the localisation subsystem; given the LocaleIndex, the application core will determine what strings to display by looking in an appropriate StringTable given the LocaleIndex.

The edit box widget in this example is one specialised for the input of integers (kIntEditBoxWidgetBoss); it allows only input of integer values; a warning message will be 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; in reality, 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 to 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 for instance the plugin described in TableAttributesDesign, which 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.

API widget boss class	ODFRez custom resource type	Displaying
kStaticTextWidgetBoss	StaticTextWidget	Miter limit
kStaticTextWidgetBoss	StaticTextWidget	x

API widget boss class	ODFRez custom resource type	Displaying
kIntEditBoxWidgetBoss	IntEditBoxWidget	4
kNudgeControlWidgetBoss	NudgeControlWidget	(nothing)

The key part of the 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 Widgets required plug-in and can be 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, 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 that 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.

Type binding for the example

This section describes the binding between widget boss classes or interfaces and ODFRez custom resource types for the example. Once you understand the concept of type binding, then it becomes very straightforward to start creating your own rich plug-in user interfaces.

The most fundamental 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 outwith 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. For the example at the start with an integer-edit box, the implementation of the IControlView interface on the boss class named kIntEditBoxWidgetBoss provides the behaviour to render the edit box correctly. The implementation of the interface named ITextDataValidation on this boss class validates that the input is an integer.

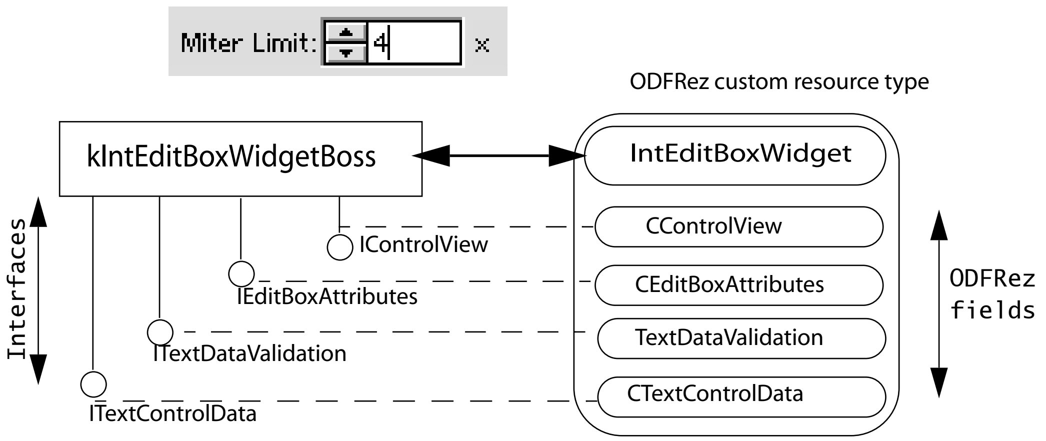
ODFRez custom resource types are used to define data to initialise widgets and other elements needed for the interface. For instance, the CControlView field in the example shown below specifies the initial location and dimensions of each widget, along with other properties such as its widget ID and visibility.

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 that is 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 the example introduced previously, some of the type bindings are illustrated below. The figure 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.



The ODFRez fields in IntEditBoxWidget define the initial state for the integer edit box widget, whose behaviour is provided by kIntEditBoxWidgetBoss. Below is shown the ODFRez data statements for the widgets above. One point to note is 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 arguably a little more cavalier 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.

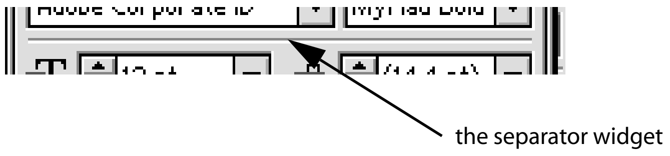
That is, the objects that represent the widgets read their initial state from the compiled version of the ODFRez data. They persist their state to the saved-data database and read it back from this the next time the application starts, if they exist. Otherwise the widgets will read their initial state 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.

```
// 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
),

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
),
```

Another example of type binding

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 below. 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 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 require 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, and a sample data statement are shown below.

```
// 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 };

// A separator widget from the Character panel.
SeparatorWidget (
     kCharVSeparatorWidgetId, kPMRsrcID_None, // WidgetId, RsrcId
     kBindNone,
     Frame(5,0,202,2) // Frame
     kTrue, kTrue, // Visible, Enabled
),
```

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 the Programming Guide 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, say) has additional interfaces such as IDropDown ListController (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 of its 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; for instance, the SDK contains a sample showing how to implement a resizeable panel; see ResizablePanelDesign. This demonstrates overriding the ConstrainDimensions method on the IControlView interface to allow 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 al so 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_I<whatever>CONTROLDATA. This is the standard mechanism for client code to receive messages about changes in the state of a control.

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

Attributes

An attribute is 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,
- 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 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 st ate.

Relevant design patterns

Design patterns are “simple and elegant solutions to specific problems in object-oriented software design” [Gamma et al, 1995]. The objective in using 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; 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 i s 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 that 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?

There are few explicitly named abstractions in the codebase called <whatever>Controller- for instance, the IDialogController interface. In practice, event handlers are the controllers, since they act upon and change a model, then the data-model in turn sends out notifications via the change manager. The event handler can change the data model of the widget boss object directly; a command sequence is initiated to change other data models such as document page items or the number of layers in a document, say. 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 user-interface architecture, the data-model actively notifies the change manager about its change in state, rather than being some 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 (see tech-note #10006) 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 2.0 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. The command architecture is described in tech-note #10001.

Key abstractions in the API

This section describes some of the most essential abstractions in the InDesign 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 and see CustomCntlViewDesign for more detail on writing 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 partitition 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.

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 outwith 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, illustrated in PanelTreeViewDesign.

Partial implementation classes

There are numerous helper classes in the API that 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.

Class name	Interface it implements	Description
CActionComponent	IActionComponent	Partial implementation of methods needed to participate in menu subsystem
CAlert	(nothing)	Static methods to display alerts, for warning errors or information
CControlView	IControlView	Minimal implementation of an interface related to appearance
CCursorProvider	ICursorProvider	Helps in creating custom cursors for widget boss objects
CEventHandler	IEventHandler	Provides minimal implementation of an event handler, useful in limited circumstances for simple controls
CDialogController	IDialogController	Provides core functionality for working with dialogs
CSelectableDialogController	IDialogController	Adds functionality specific to selectable dialogs, e.g. the global preferences dialog.
CDialogObserver	IObserver	Provides helper methods to ease working with and observing collections of widgets on dialogs
CSelectableDialogObserver	IObserver	Adds functionality for selectable dialogs

Class name	Interface it implements	Description
CObserver	IObserver	Minimal implementation of an IObserver interface- usually override Update method
CDialogCreator	IDialogCreator	Used in creating selectable dialogs
CPanelRegister	IPanelRegister	Used for any panel-based plug-in to register panel boss object with application run-time
CPanelCreator	IPanelCreator	Use in adding panels to selectable dialogs

Utility classes

There are utility classes such as PalettePanelUtils and WindowUtils which provide static methods relevant to working with palettes and widgets.

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.

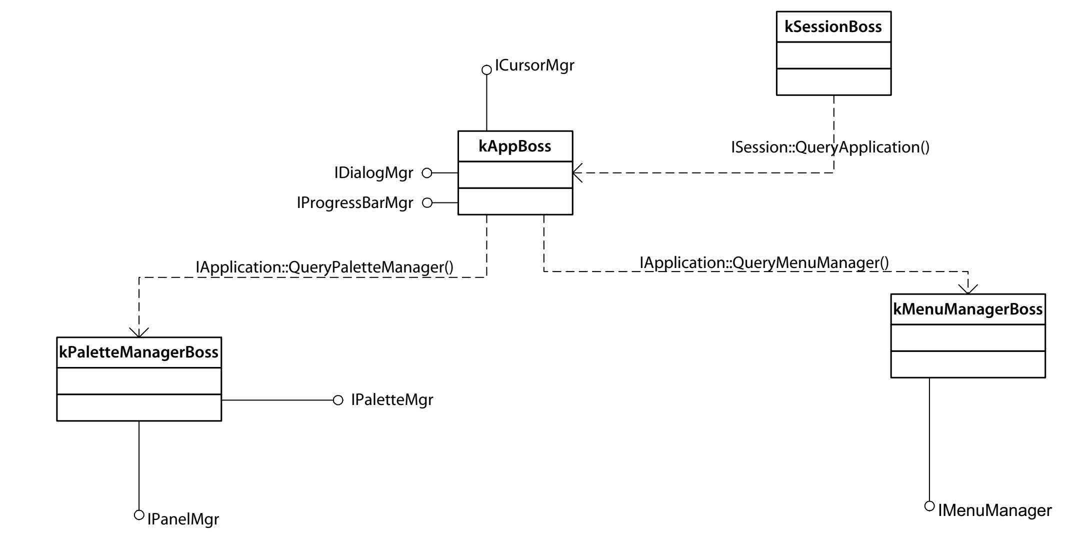
The IPMPersist interface is required because all widgets have some initial state defined in ODFRez data statements. The ReadWrite method on this interface is called during initialisation of widget boss objects, when an object is being created by reading its initial state from a plug-in resource or saved-data 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 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.

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. There is a document of the Programming Guide devoted to persistence. 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.

For instance, consider the example with the int edit box and the nudge control. There are four interfaces bound to ODFRez types, and many interfaces on this boss class which are not. See below for a table showing some interfaces on the boss class, whether they are persistent and whether they are bound to an ODFRez type or not.

Interface	Implementation ID	Persistent?	Bound to ODFREz type?
ITextControlData	kEditBoxTextControlDataImpl	Yes	Yes
IEditBoxAttributes	kEditBoxAttributesImpl	Yes	Yes
IControlView	kNudgeEditBoxViewImpl	Yes	Yes
ITextDataValidation	kIntTextValidationImpl	Yes	Yes
ITextValue	kIntTextValueImpl	Yes **	No
IEventHandler	kEditBoxEventHandlerImpl	No	No
IObserver	kCNudgeObserverImpl	No	No
(other interfaces ommitted deliberately)			

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 interface must be persistent.

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 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.

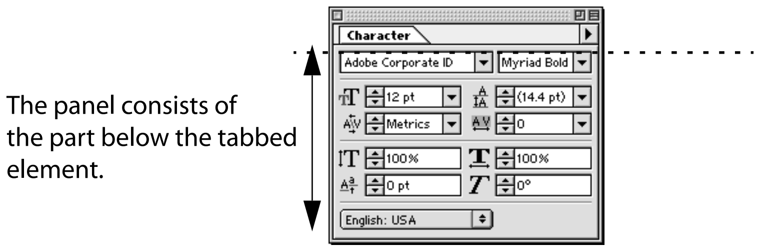
Examine again the definition of the separator widget, repeated below with some annotations. This expression can be read as a definition of the ODFRez custom resource type SeparatorWidget, that extends the ODFRez custom resource type named Widget. As mentioned previously, a ClassID field is initialised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

```
// 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 that are 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 Character panel, shown below; in this example, there is a tabbed palette containing a single panel.



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 withthe 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
- an instance of a PanelList resource is prefaced by resource.

Non-translated StringTable resource

There is a StringTable resource in which the plug-in developer can locate strings that should not be translated. Careful attention to localisation should be paid to minimise the amount of unnecessary duplication of strings across the locale-specific string tables. Inspect any of the SDK plug-ins to see an instance of this; in particular see BasicLocalizationDesign for an indication of how localisation can be performed cleanly and simply.

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.

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:

```
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 and use SDKUtilities::DumpBoss to find out what boss class the interface is aggregated on. For instance if you have an IControlView interface, see the bottom of the page in the on-line documentation 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 2, 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. There is an example in the SDK, which was inappropriately named HelloWord (as it isn't strictly speaking an introductory sample), described at HelloWorldDesign; this illustrates precisely this kind of factoring, where one plug-in (the user-interface plug-in) is a client of code delivered in a companion plug-in, which provides the core functionality.

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, Dolly 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. It is also relatively straightforward to create and change both the widget hierarchy and the geometry of the plug-in user interface with Freddy, and you should read the manuals for both these tools in some detail to understand how to take away some of the pain of creating plug-in user interfaces.

To write a user interface for an InDesign 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. 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,
- creating ODFRez data statements to specify the initial states of the user interface elements and localised string-data; tools such as Freddy can make this a lot easier, as it can generate the ODFRez data statements and the initial geometry and hierarchical relationships between widgets,
- 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 Technology is to take away some of the pain from this process, by creating developer tools such as Freddy, Dolly and the DevStudio macros 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.

It is very important if you are using Freddy to create initial layouts and modify the geometry of the layouts that you observe the practice of including the name of the parent ODFRez type in the new name. This is discussed in detail in the Freddy user manual (see tech-note #10026).

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 ae 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?

The relationship between the components in a tabbed palette is dealt with in a separate tech-note (#100xx). PalettePanelUtils 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 numerous user-interface samples that show how to navigate from the menu boss object to the panel boss object (look for PalettePanelUtils).

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.

```
// 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.

How do I create menu-entries?

The easiest way to do this is using Dolly, 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 Dolly, 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 Dolly template Dialog to get a basic dialog; this will give you code very similar to that in the SDK sample named BasicDialog.

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. 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. The sample described in SelectableDialogDesign shows how to work with selectable dialogs.

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.

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. There is an SDK sample named PreviewableDialog showing how to work with this in detail; consult PreviewableDialogDesign.

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.

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.

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 for example TableSorterDesign.

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.

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

The sample ScrollPanel indicates the bare bones of adding a multi-line static text widget to a panel and associating it with a scroll bar. The process 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.

How do I add a text edit box?

Freddy is capable of adding the vast majority of text edit box widgets in the API. 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; be aware that you should follow the naming principles for Freddy described in the tech-note (#10026); if you subclass, say, TextEditBoxWidget, the name MyTextEdtiBoxWidget would be acceptable to Freddy, which could deduce the type from the name. The name MyWidget loses the type information, and Freddy does not parse the ODFRez type declarations.

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, or the code generated by Freddy for an example.

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.

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. Note that you can add this type with Freddy, as long as you obey the naming rules for Freddy (#10026). 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.

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. Note that you can use Freddy to create the vast majority of specialised combo-boxes in the API; however, 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, as Freddy does not at the time of writing parse the type definitions and relies on a predictable naming scheme.

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 sample described in PicturIconDesign for more information on this issue.

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.

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 kInfoStaticTextWidget, which is bound to the ODFRez type named InfoStaticTextWidget. Initial font commitments can be defined in ODFRez data statements and the font displayed varied through the IUIFontSpec interface on the widget boss object.

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 CustomCntlViewDesign for an example of owner-draw panels.

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. Look at the sample described in ResizablePanelDesign for an example of how this can be implemented in practice.

How do I group widgets?

A widget is contained within another widget when it is present in the CPanelControlData list of another widget. Freddy makes the grouping of widgets very straightforward, since it has a built-in knowledge of which widget types can be container classes, and will cause container widgets to adopt new children if potential dependents are dropped within their frames. 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.

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 plug-in described in WListBoxBasicDesign for a simple example of using a list-box on a panel.

How do I vary the set of widgets displayed?

The sample described in DetailControlSetDesign 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.

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.

Summary

This section has introduced some of the terms, key concepts and design patterns required to understand the user-interface architecture. In terms of the plug-ins that you should look at first, see the following; BasicMenuDesign, BasicDialogDesign, BasicPanelDesign and BasicLocalizationDesign.

This document also described some of the core aspects of the user-interface architecture. It attempted to explain type bindings between boss classes (or API interfaces) and ODFRez types, and provide enough context to understand the representation of widget boss objects in ODFRez. It also introduced key elements in the localisation architecture.

Practical exercises

It is recommended to spend some time generating plug-ins with Dolly and studying the code; use the templates like IfPanelMenu and Dialog- see resource #10025 for more detail. You should consult the section “References: finding out more” and work through some of the material there, particularly to identify the complete widget boss class set available.

To help understand how to use the on-line documentation to full effect, visit pages for instance about IControlView and look down at the bottom of the page for a list of boss classes that can draw (and methods that can return an IControlView interface, which are bridging methods within the user interface API). See the page about IPanelControlData to view container widgets. Look at kBaseWidgetBoss to see the widgets that are immediate descendants of the base widget class and for instance, see kIntEditBoxWidgetBoss to see the edit box types that descend from this class.

Review questions

This section presents a list of questions that the reader can use to check their understanding of the content presented in this document.

1. Do all widget boss classes encapsulate a platform control? If not, give examples of ones which do and do not.
2. In what types of files are the user interface elements defined? What is the language that is used to specify the initial state and properties of user interfaces?
3. What is the principal difference between a boss class and a widget boss class?
4. What is the responsibility of a control-view? What interface represents this aspect?
5. What are observers attached to? What changes are they notified about? What method would be called on your observer when a change happened?
6. Name two things that can result in IActionComponent::DoAction being called.

7. What is the difference between an interface and a partial implementation class? Give an example of each within the user interface API.
8. What is the difference between a panel and a palette? Which contains the other, for instance?
9. Is is possible to re-use all the implementations of interfaces on existing boss widget classes? If not, why not?
10. What are the main factors (aspects) in the user interface programming model?
11. What method determines how a widget draws?
12. What triggers change in a control-data model?
13. What is another name for the Publish-Subscribe pattern?
14. Give two examples of a Controller pattern in the UI model.
15. What is meant by a binding between a boss class and an ODFRez type? Why is this important?
16. What is the boss class behind the behaviour of nudge control widgets?
17. Does an edit-box widget boss class encapsulate a platform control? If yes, give an example of additional capability the boss class might provide.
18. What type do all widget boss classes have as an ancestor?
19. What is the name of the ODFRez type bound to the boss class named kIntEditBoxWidgetBoss?
20. Given the interface ITextDataValidation, what is the name of the ODFRez type bound to this interface?
21. Why do you think not all the interfaces in a widget boss class are bound to fields in the associated ODFRez type?
22. What is the only interface that the kSeparatorWidgetBoss class adds to those in its superclass? Why does it add this interface?
23. Where do widgets read their initial state from, the very first time? Thereafter where do they write/read their state?
24. Why is the message ID space not a real ID space? What ID space do notification message really get defined in?
25. What is the principal difference between an IWidgetParent interface and an IPanelControlData interface?
26. What interface is concerned with making a widget observable?
27. What is the purpose of the class named kDialogBoss and why is the name misleading?

Answers to review questions

This section presents answers to the review questions in the preceding section.

1. Not all widgets have a platform control encapsulated. Ones which do are e.g. kIntEditBoxWidgetBoss, and ones which don't are e.g. kStaticTextWidgetBoss.
2. FR files, ODFRez.
3. The latter must have kBaseWidgetBoss as an ancestor. The former is less constrained in its ancestry.
4. Draws the widget, IControlView.
5. A subject (ISubject). For widgets- changes in the data model. IObserver::Update
6. Keyboard shortcut and menu item being executed.
7. Interface extends IPMUnknown, a utility class probably doesn't, a partial implementation class and a manager class? IObserver, PalettePanelUtils, CObserver, IPaletteMgr.
8. You can put widgets like kButtonWidgetBoss/ButtonWidget on panels but not palettes. Palettes contain panels.
9. No. May have some dependency on other interfaces being aggregated on the same boss class. Re-use is typically done via extending boss classes rather than re-using a single implementation.
10. Control-view, data model, attributes, event-handler.
11. IControlView::Draw
12. Event handler.
13. Observer.
14. Event handler, and dialog controller.
15. It is an association of an ODFRez custom resource type with a widget boss class. It tells the application core what type of widget to create and how to read the data to initialise it.
16. kNudgeControlWidgetBoss.
17. Yes. Validation and reading and writing values as integers.
18. What type do all widget boss classes have as an ancestor?
19. IntEditBoxWidget
20. TextDataValidation
21. They don't all need to initialise their state.
22. IControlView. To draw the line that forms the separator.
23. Plug-in resource. Saved-data database.
24. Just the same as the kClassIDSpace.
25. IWidgetParent lets you traverse to the root of the widget tree, IPanelControlData traverse to the leaves.
26. ISubject.
27. It's a panel boss class.

References: finding out more

This section identifies resources that can be used to help develop an understanding of the user interface programming model.

- See a list of the widget boss classes http://partners.adobe.com/asn/developer/indesign/explodedSDK/mac/Documentation/WebDocs/a/apibossclassindex.html#WIDGETS
- Open the spreadsheet IObjectModel_<whatever>.xls and select “Plug-in >> Custom >> contains” entering WIDGETS as the key in the drop-down. This yields the boss classes that are in the Widgets required plug-in and the interfaces they aggregated, harvested on the dynamic object model.
- Look at the page about IWidgetParent in the on-line documentation; the bottom of the page shows the boss classes that aggregate this interface in the core boss classes, which are ancestors of kBaseWidgetBoss.
- Look at the page about kNudgeEditBoxWidgetBoss in the on-line documentation; this shows boss classes that extend this class in the core set.
- Widgets.fh; this contains the ODFRez custom resource types that bind to the widget boss classes in the API. Look carefully at the boss class (or interface) associated with the ODFRez types. If a type is bound to a boss class (e.g. k<whatever>Boss) then it is possible that the widget type could be used on a user-interface. Types bound to interfaces are merely components of a compound type and can’t be used individually.