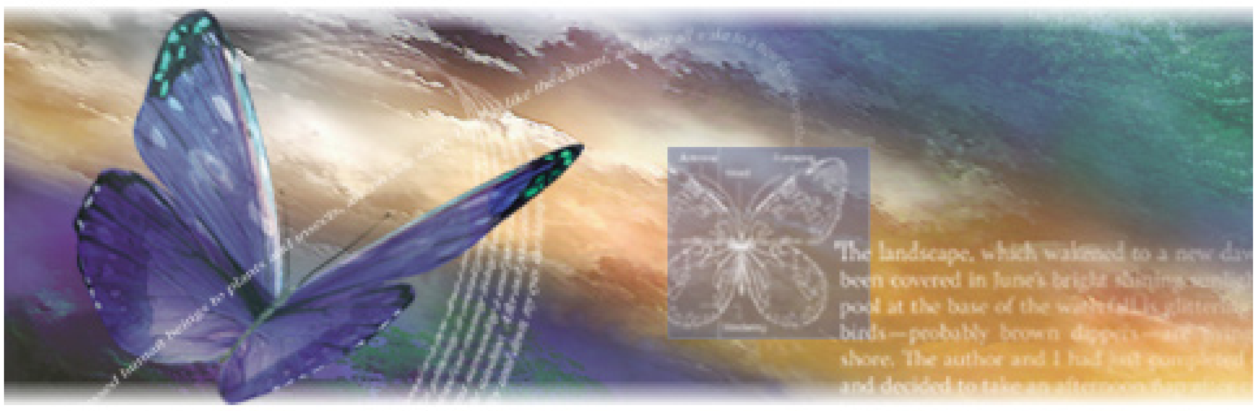
InDesign® Plug-in CookBook 1:

Making Your First InDesign Plug-in

Technical Note # 10045-English

Version InDesign 2.0

25 Mar 2002


The landscape, which wakened to a new day
been covered in June's bright shining sunlight
pool at the base of the waterfall is glittering
birds—probably brown dippers—are busy
shore. The author and I had just completed
and decided to take an afternoon

Adobe Systems Incorporated
Corporate Headquarters
345 Park Avenue
San Jose, CA 95110-2704
(408) 536-6000

Rev #	**Date**	**Author**	**Comments**
Draft 1	07-Nov-2001	Mitsutoshi Kikuta	First Draft
Draft 1.1	09-Nov-2001	Mitsutoshi Kikuta	
Draft 1.2	11-Nov-2001	Mitsutoshi Kikuta	
Draft 1.3	07-Feb-2002	Ken Sadahiro	Translated to enUS
Draft 1.4	22-Feb-2002	Ken Sadahiro	Fixed translations
1.0	01-Mar-2002	Ken Sadahiro	English version completed, with extra columns, formatting fixes.
1.1	04-Mar-2002	Ken Sadahiro	Fixed screen shot showing usage from menu
1.2	08-Mar-2002	Ken Sadahiro	Added substeps, removed extra column boxes, and changed code in Step 7/9 to use do-while.
1.3	11-Mar-2002	Ken Sadahiro	Added note about InD3 filetype, fixed StaticTextWidget's associated widget, and reimported Mac screenshots (CW menu/proj)
1.4	25-Mar-2002	Ken Sadahiro	Resynched with Japanese version: Added a column about plug-in types, split step 1.5 into 1.5/1.6 (bumping subsequent steps up) and changed step 1.9 to it's own chapter. Replaced DollyWizard installation instructions with a note about the Dolly User Manual tech note.

Table of Contents

Table of Contents

About this Cookbook

This Cookbook will guide you through the process of creating a basic InDesign plug-in step by step, while helping you become more familiar with the process.

The source code for InDesign plug-ins are almost entirely cross-platform compatible. By using the InDesign SDK, you can develop plug-ins that can be used with InDesign for Windows and Macintosh OS, as well as Mac OS X (starting with InDesign 2.0).

Unlike more other plug-in development processes, the InDesign plug-in development process is unique in that it has its own user interface (UI) framework as well as an object-oriented application programming interface (API). This cookbook will focus on these two unique aspects.

Welcome to our first cookbook!

This is the first in our series of InDesign Plug-In Development Cookbooks. We developed the first one (in Japanese, actually) for the November 2002 InDesign Developer BaseCamp in Tokyo. We would like to hear your feedback about this cookbook: what you liked/disliked, what could be improved, and other types of cookbooks you would like to see in the future. Please submit your feedback to us at `http://partners.adobe.com/asn/developer/feedback.html`.

The Goal of this CookBook

Let's start our development by painting a picture in our minds about the finished product: our first plug-in.

Since this is a Cookbook, we shall incorporate several fundamental and commonly used elements of InDesign plug-ins: UI elements (menu, dialog, pull-down menu, text edit box, static text field, button) and text.

The Story of Our Plug-in

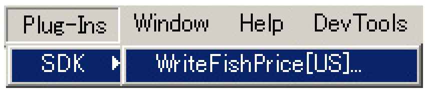
Now we will describe a scenario on how we want to use our very first InDesign plug-in.

First, we start InDesign.

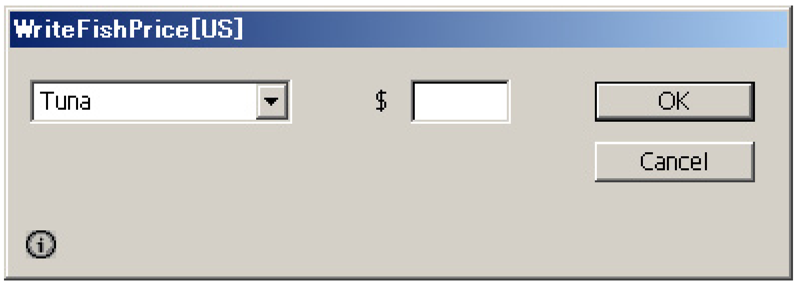
Then, we create a new document.

In the new document, we will create a text frame, and place the text cursor in the text frame.

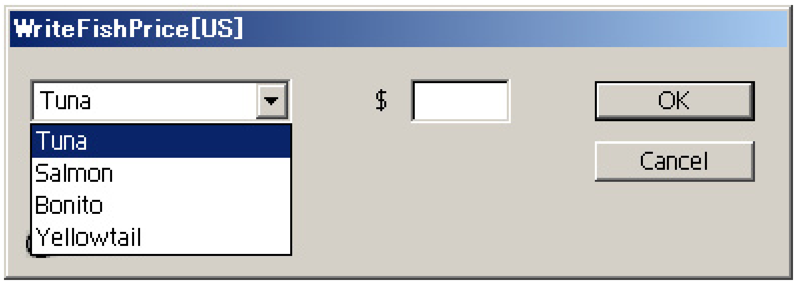
From the menu, we will select **Plug-Ins**.



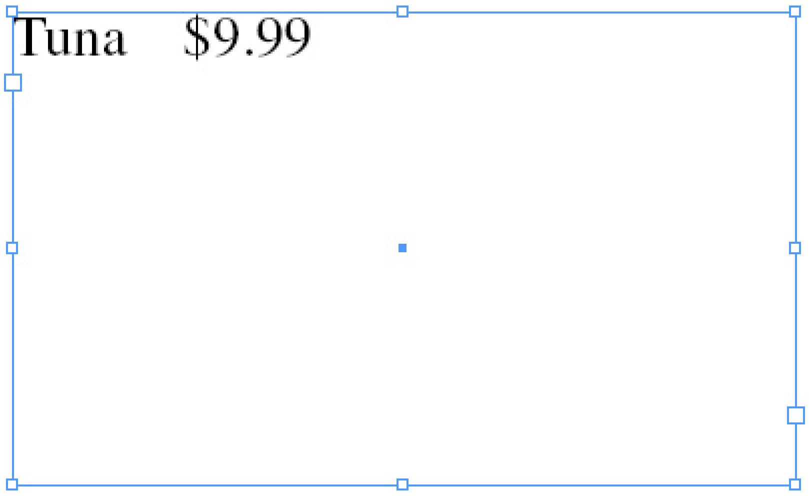
Doing that will open up a dialog like this:



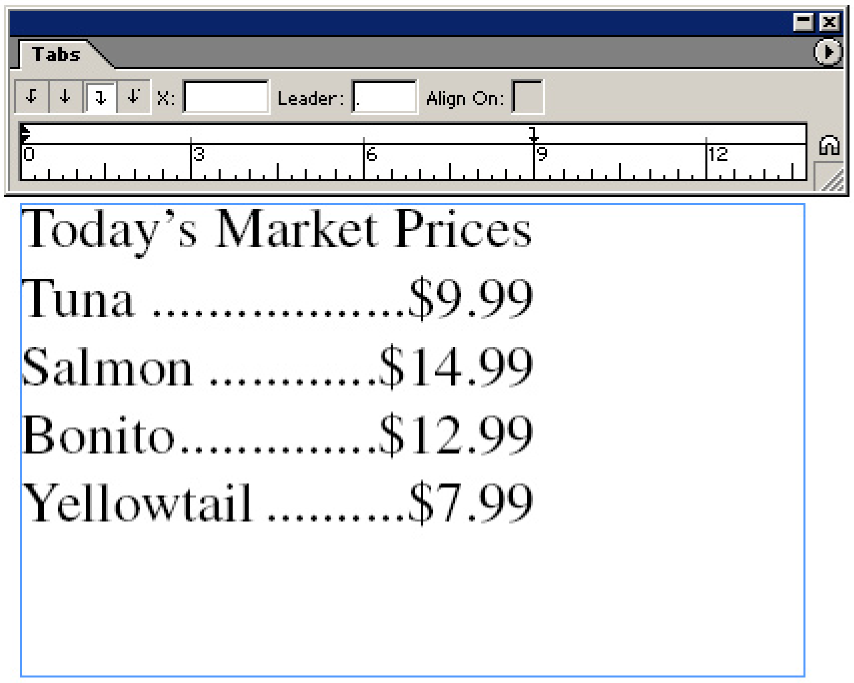
From the pull-down menu on this dialog, we select the name of a fish, and in the text edit box, we enter its price.



When we click on the OK button, the name of the fish we selected from the pull-down menu, and the price we entered, will appear in the text frame at the insertion point where we had our text cursor.


Tuna	$9.99

If we have already set our tab stop settings in the text frame, the text should appear like this:



Today's Market Prices
Tuna $9.99
Salmon $14.99
Bonito..............$12.99
Yellowtail $7.99

It's a simple plug-in, but it also should serve as a useful starting point.

About DollyWizard

Newly introduced in the InDesign 2.0 SDK is a plug-in development tool called DollyWizard. This application, written entirely in Java, allows you to generate fundamental plug-in projects for Microsoft Visual C++ and Metrowerks CodeWarrior (XML, to be imported), by cloning plug-in templates. By using DollyWizard, you can instantaneously create a starting point for your plug-in development.

DollyWizard is located in the Tools folder in the InDesign SDK. There are installers for both Windows and Macintosh, so please run the appropriate installer to install DollyWizard on your system.

For details on installing and running **Dolly Wizard**, please refer to the InDesign SDK TechNote titled **Dolly User Manual**.

Where can I obtain the Java 2 Standard Edition Runtime Environment (JRE) ?

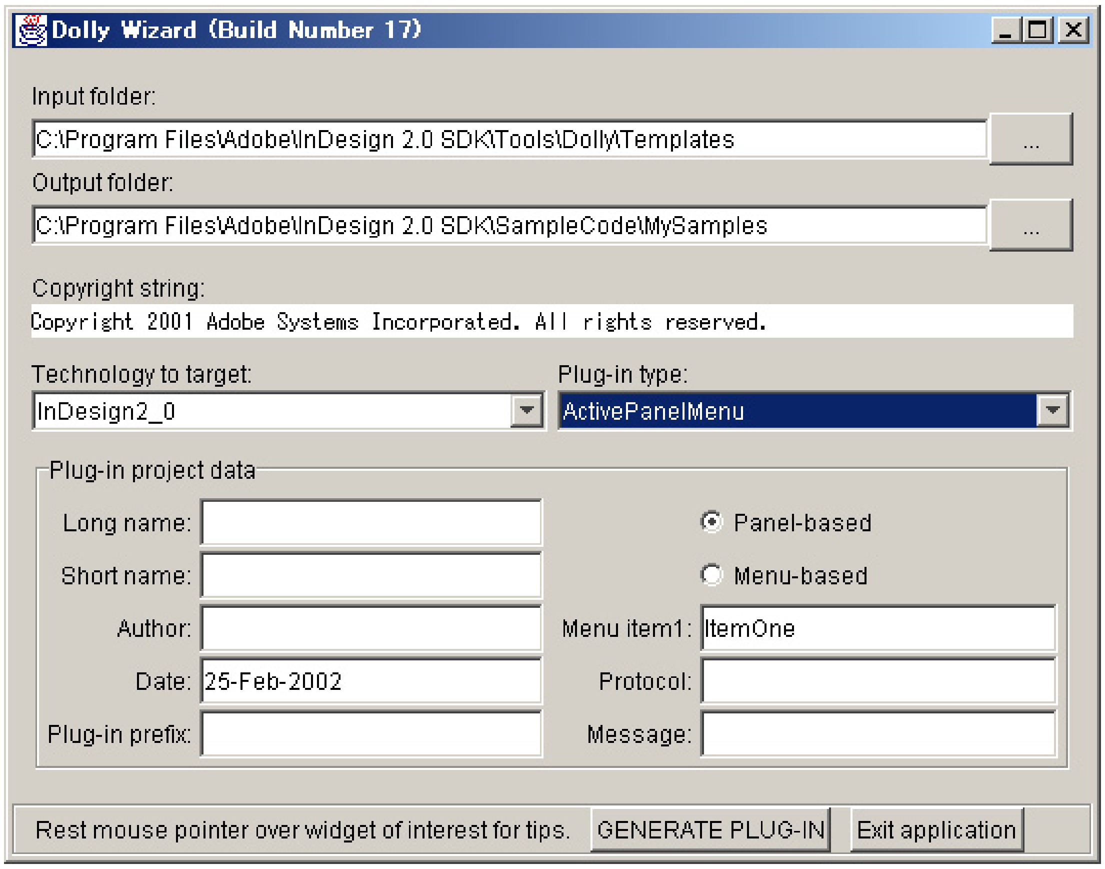
If you don't already have JRE installed, you can find out more information about it and download it from `http://java.sun.com/j2se/`.

Step 1: Using DollyWizard to generate a Dialog-based plug-in project

By using the templates provided with DollyWizard, you will generate a plug-in project.

Step 1.1: Launch DollyWizard

First, launch DollyWizard. When you launch DollyWizard, you will see a dialog like this:



Input folder: C:\Program Files\Adobe\InDesign 2.0 SDK\Tools\Dolly\Templates

Output folder: C:\Program Files\Adobe\InDesign 2.0 SDK\SampleCode\MySamples

Copyright string: Copyright 2001 Adobe Systems Incorporated. All rights reserved.

Technology to target: InDesign2_0

Plug-in type: ActivePanelMenu

Plug-in project data

Long name:

Short name:

Author:

Date: 25-Feb-2002

Plug-in prefix:

[✓ Panel-based] Menu-based

Menu item1: ItemOne

Protocol:

Message:

Rest mouse pointer over widget of interest for tips.

On this dialog, we will specify all necessary information for DollyWizard to generate the plug-in code. It would be convenient to generate your own project folder under the **{SDK}\SampleCode** folder. For the purpose of this exercise, we will work in a sub folder called **MySamples**. Make sure this folder exists on your system.

The **Input folder** is where you specify the location of the DollyWizard templates. The **Output Folder** is where you specify the folder in which DollyWizard generates the plug-in project. (Macintosh: Make sure you use '/' (forward slash) as a folder delimiter.)

You can specify your own copyright statement as the **Copyright string**. The **Technology to target** pull-down menu is where you specify the target InDesign version. The **Plug-in type** pull-down menu is where you select the template that DollyWizard will use.

Step 1.2: Specify Necessary Information

Let's specify the necessary information in each field on the DollyWizard dialog.

For **Input Folder**, specify the complete path to the **SDK/Tools/Dolly/Templates** folder. (e.g. **C:\Program Files\Adobe\Adobe InDesign 2.0 SDK\Tools\Dolly\Templates**, or /**Macintosh HD/Adobe/InDesign 2.0 SDK/Tools/Dolly/Templates**).

For **Output folder**, specify the complete path to the **SDK/SampleCode/MySamples** folder. (e.g. **C:\Program Files\Adobe\Adobe InDesign 2.0 SDK\SampleCode\MySamples**, or **Macintosh HD:Adobe:InDesign 2.0 SDK:SampleCode:MySamples**)

For the time being, leave the **Copyright string** alone.

For **Plug-in type**, select **Dialog**.

Now comes the most important step: specifying our plug-in names. There are two names to specify in DollyWizard: **Long name** and **Short name**.

The **Long name** specifies the name of the plug-in itself. This name is used as a string under the **Plug-Ins** menu as well as the **About this Plug-In** dialog. The **Short Name** is very important: this name is used as part of the source code files and class names that are generated. If you make this **Short Name** too long, not only will the class names become very long, but some filenames may end up being longer than 31 characters, resulting in an unsuccessful generation. It is recommended that **Short name** be no longer than 5-6 characters.

For this exercise, specify the **Long Name** as **WriteFishPrice**, and **Short Name** as **WFP**.

For **Author**, enter your own name. This string is used in the About this Plug-in dialog as well as comments in source code.

For **Date**, enter today's date, for example: **20-Mar-2002**.

Next comes the **Plug-in Prefix** field. A **Plug-in Prefix** is a unique ID assigned by Adobe Systems for use of your plug-in development. For production plug-ins that you will release outside your organization, please make sure that you use an ID prefix assigned to you by Adobe Systems. This is very important, as the ID prefix is used to define the plug-in's IDs and resources. Therefore, there must be no overlap in the ID prefixes used by plug-ins. For experimentation purposes, You may reuse the ID prefixes used in sample plug-ins in the SDK, but be careful not to release plug-ins that use randomly selected prefix IDs. Any plug-in that uses the same ID cannot be used simultaneously.

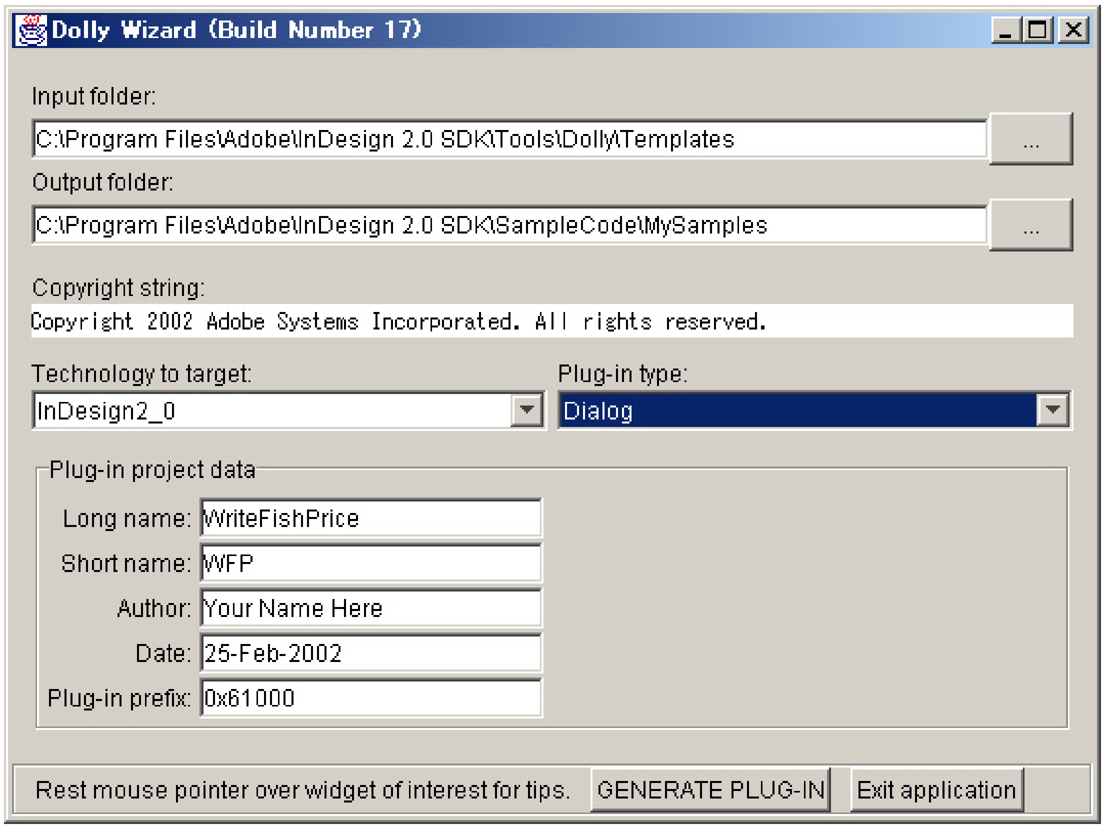
For this exercise, specify **0x61000**. This is an ID prefix that was specially allocated for the purpose of this exercise.

How do I obtain a Plug-in Prefix?

Please refer to the InDesign SDK Knowledgebase Article #50093.

Step 1.3: Verify Entered Information and Generate Plug-in Project

Have you entered all the fields? Please verify against the screen shot shown here.


Dolly Wizard (Build Number 17)
...
...
Plug-in project data
Rest mouse pointer over widget of interest for tips.
GENERATE PLUG-IN
Exit application

Input folder: C:\Program Files\Adobe\InDesign 2.0 SDK\Tools\Dolly\Templates
Output folder: C:\Program Files\Adobe\InDesign 2.0 SDK\SampleCode\MySamples
Copyright string: Copyright 2002 Adobe Systems Incorporated. All rights reserved.
Technology to target: InDesign2_0
Plug-in type: Dialog
Long name: WriteFishPrice
Short name: WFP
Author: Your Name Here
Date: 25-Feb-2002
Plug-in prefix: 0x61000

Once you have verified your settings, click on the **GENERATE PLUG-IN** button.

Check to see if DollyWizard generated files in the folder you specified in **Output folder**. Do you see any files there?

Step 1.4: Change Creator and Type of DollyWizard-generated Text Files,(Macintosh only)

If you are using a Macintosh: you have to go through a few extra yet important steps.

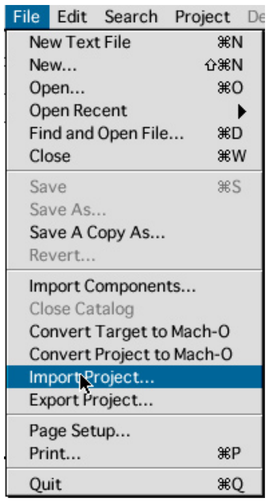
The source files generated by DollyWizard don't have their file type and creator set correctly. By using resource editors, such as ResEdit, edit the creator and filetype of the generated files (***.fr**, ***.cpp**, ***.h**, ***.xml**) as such: Creator: '**CWIE**' File type: '**TEXT**'.

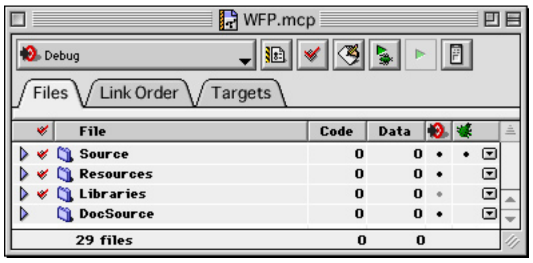
Step 1.5: Convert the XML File to a CodeWarrior Project (Macintosh only)

Also, to develop on the Macintosh, you must create a CodeWarrior project. DollyWizard generates an XML file that you can import into CodeWarrior. To create a CodeWarrior project from the XML file, start your CodeWarrior Pro 7 IDE, and select the **File >> Import Project...** menu.

Then, select the **WFP.mcp.xml** file that DollyWizard generated in the **WriteFishPrice:Project** folder. If the filetype of this XML file is not "TEXT", the XML file will not appear in this dialog box. When the XML file is imported, CodeWarrior will ask you for the path of the project to create, so instruct Code-

Warrior to create **WFP.mcp** in your **WriteFishPrice:Project** folder. When the project is generated, you will see a panel that looks like the screen shot here.





Finally, select the **Edit >> Debug Settings...** or **Edit >> Release Settings...** menu and select **PPC Target**. Make sure that the **Type** is set to **InD3**, not **InDa**. Do this under both the **Debug** and **Release** modes.

Step 1.6: Build the Plug-in

All right, let's build our plug-in!

If you are using Windows, open the **WFP.dsp** in your **WriteFishPrice\Project** folder with Microsoft Visual C++ 6.0. (Please be sure to use Service Pack 5 with the InDesign 2.0 SDK.) Once the project is open, make sure the **WFP - Win32 Debug** as the active configuration (under **Build >> Set Active Configuration...** menu), then select the **Build >> Build WFP.pln** menu to build the plug-in.

In CodeWarrior, open the **WFP.mcp** you just converted from the XML file, and select **Debug** from the **Set Default Target** menu or the **Target** popup menu. Then, select the **Project >> Make** menu, or click on the **Make** button on the **Project Window**.

This completes the first step of developing a plug-in with DollyWizard. Were you able to build your plug-in without compiler errors?

Three types of InDesign Plug-Ins

InDesign has three distinct kinds of plug-ins. On the Mac, these are identified by filetype, and on Windows, they are identified by file extension.
Required Plug-ins: These are plug-ins that must be installed, and are placed in the Required folder. In the Plug-in Settings dialog, these are listed as "Required", as they are indicated with a pad lock and an Adobe logo. On the Mac, the filetype is 'InDr', and on Windows, the file extension is ".rpln".
Adobe Plug-ins: These are plug-ins that are provided by Adobe, and are placed in functionality-based subfolders under the Plug-Ins folder. In the Plug-in Settings dialog, these are listed as "Adobe", as they are indicated with an Adobe logo. On the Mac, the filetype is 'InDa', and on Windows, the file extension is ".apln".
Third-party Plug-ins: These are plug-ins that are developed by third-party developers, and are placed in the Plug-Ins folder. In the Plug-in Settings dialog, these are listed as "Third Party". The sample plug-ins in the SDK are placed under this category. The plug-in for this cookbook is also of this type. On the Mac, the filetype is 'InD3', and on Windows, the file extension is ".pln".

Step 1.7: Prepare to Load the Plug-In

The debug build plug-in you just built has the filename **WFP.pln**, and can be found in the **Sample Code\BuiltPlugIns\Debug** folder under your SDK folder. If you build the release build, it will have the same filename but will be built in the **SampleCode\BuiltPlugins\Release** folder under your SDK folder. In either case, the plug-in that you just built needs to be in placed in the **Plug-Ins** folder under your **InDesign** folder for InDesign to load it at start up time. There are three ways to make this happen:

(1) You can copy the **WFP.pln** to the Plug-Ins folder under your InDesign folder. NOTE: This is the recommended way of loading the distribution version of your plug-in.

(2) You can modify the **PluginConfig.txt** file (Windows: In the **"%USERPROFILE%\Local Settings\Application Data\Adobe\InDesign\Version 2.0"** or **"Version 2.0J"** folder; Mac OS 9: **"{System Folder}:Preferences:Adobe InDesign:Version 2.0"** or **"Version 2.0J"**; Mac OS X: **"{SystemDrive}:Users:{UserName}:Library:Preferences:Adobe InDesign:Version 2.0"** or **"Version 2.0J"**) by adding the following (replace with your own paths)

```
=Path
"C:\Program Files\Adobe\Adobe InDesign 2.0 SDK\SampleCode\BuiltPlugIns\Debug"
```

or

```
=Path
"Macintosh HD:Adobe InDesign 2.0 SDK:SampleCode:BuiltPlugIns:Debug"
```

If you are using the release build of InDesign, change the Debug to Release in the statements above.

(3) (Macintosh only) You can create an alias to your **{SDK}:SampleCode:BuiltPlugIns:Debug** (or **Release**) folder, and place the alias in the **Plug-Ins** folder under your **InDesign** folder.

NOTE: Don't install debug-build plug-ins in the Plug-Ins folder for the Release build version of InDesign, and vice versa. Your plug-in will fail to load.

You can tell when this plug-in is loaded: You will see a **Plug-Ins** menu item after starting InDesign (in the next step).

Step 1.8: Start InDesign through Your IDE

If you have successfully built your plug-in, and setup a way for the plug-in to load, then you can start the InDesign application through your IDE.

In Visual C++: Select the **Project >> Settings...** menu, and click on the **Debug** tab. Under the **General** category, specify the full path to your **InDesign.exe** application in the

How do I know which service pack of Visual C++ 6.0 I am using?

If you have installed Visual C++ 6.0 Service Pack 5, you will see the following entry in the Windows Registry (**regedit.exe**):

```
\\HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\VisualStudio\6.0\ServicePacks\sp5
```

Also, you can also see if the following key contains the DWORD value **0x00000005**:

```
\\HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\VisualStudio\6.0\ServicePacks\latest
```

If neither of the above apply to you, chances are you don't have Service Pack 5 installed.

Without Service Pack 5 installed, you may encounter some compilation errors around an InDesign basetype called **PMPoint**.

Executable for the debug session text box. You can also browse for the **InDesign.exe** application by clicking on the right arrow button to the right of the same text box. Once you have selected the **InDesign.exe** application you want to use for your debug session, click **OK**. You may optionally save your workspace (**File >> Save Workspace**), so you don't have to specify the path the next time you use this project. To start debugging, select the **Build >> Start Debug >> Go** menu, or hit the **F5** key.

In CodeWarrior: Select the **Edit >> Debug Settings...** menu, and select the **Runtime Settings** under the **Target** tree item in the **Debug Settings** dialog. Specify the path (relative to the project or absolute) to your **InDesign 2.0** (or **InDesign 2.0J**) application. You can also **Browse** for this path. Then click Save on the **Debug Settings** dialog to save this path into your project file, and close the dialog. To start debugging, select the **Project >> Debug** menu, or hit the **Command-R** key sequence.

Step 1.9: Try Debugging

Now let's try to put some break points in the code so we can experiment with the debugger.

Put a breakpoint somewhere in one of the .cpp files, say **WFPActionComponent.cpp**, in the **DoAction()** method. Then select the **Plug-Ins >> WriteFishPrice** menu item. You may also try examining values of variables.

Once you are done, you can quit InDesign to finish your debugging session. Remember to do this before editing your code in each of the subsequent major steps in this cookbook.

What are the advantages of using a Debug build of InDesign for developing plug-ins?

The debug build of InDesign enables several key features that aid debugging, such as various asserts, checking for "boss leaks"(more later on bosses and boss leaks), and being able to get names of various symbols at runtime. Also, by building your plug-ins in the debug mode, you can debug through your own code with the use of breakpoints, variable checking, and call stack traversal.
The main difference between the debug and release builds of your plug-in projects is that they are linked to a different set of libraries: one set with extra exports for the debug build (**{SDK}\API\LibD**), one with only the set of exports for the release build (**{SDK}\API\LibR**).
Once you have fully tested your plug-in under the debug build of InDesign, you can then build your plug-in in the reelase mode, and test it against the release build of InDesign.

Examine the Files Included in the Project Generated by DollyWizard

Let's take a look at the files that DollyWizard generated for us. Go back to the project in your IDE, and let's examine each file, one by one.

WFPID.h

This header file is a central repository for the plug-in centric IDs, where an ID can be a numeric or a string that is unique across the application or within the plug-in. This file plays a critical role in the plug-in, as it is included by all plug-in project files.

WFPFactoryList.h

This header file contains macros that allow the core InDesign object model to create and destroy instances of the implementations through factory classes.

WFPNoStrip.cpp

This file prevents the C++ compiler optimizations from "dead stripping", which eliminates what appears to the compiler as unreferenced code. The most of the code in the plug-in is not used directly from within the plug-in itself. This file contains a function, void **DontDeadStrip()**, which includes **WFPFactoryList.h**.

WFPID.cpp

This files allows the IDs defined in **WFPID.h** to be included as strings in the debug build symbols.

SDKPlugInEntrypoint.cpp

This file is located in the **Utilities** folder under your SDK folder, and specifies the entry point of the plug-in. This file is not generated, but rather simply included in the project.

SDKUtilities.cpp

This file defines the **SDKUtilities** class. Just like **SDKPlugInEntryPoint.cpp**, this file is not generated, but simply included into the project.

TriggerResourceDeps.cpp

This file ensures that the ODFRC resource is re-linked when the **.fr** file is compiled under Windows.

WFPDialogController.cpp

This source file contains a class that is used for initializing, validating, and responding to dialog widgets. By writing the **WFPDialogController** class, you can specify what happens during when the dialog is initialized, and when you click on the OK button.

WFPDialogObserver.cpp

The **WFPDialogObserver** class in this file dynamically process changes to the widgets on the dialog. In our case, this class observes events pertaining to the information button placed on the dialog. When the information button is clicked, the About this Plug-In dialog is displayed.

WFPActionComponent.cpp

The **WFPActionComponent** class in this source file defines what happens when the plug-in's menu item is selected. In this case, the Plug-Ins menu item and About this Plug-In menu items are handled.

The About this Plug-In menu is displayed under the Apple Menu (Macintosh) or the Help >> About Plug-Ins >> SDK (Windows) menu items.

This class also opens the About this Plug-In dialog.

WFP.fr

This file defines resources that are unique to InDesign. These resource definitions are cross-platform compatible between the Macintosh and Windows platforms. This particular file contains resources other than strings.

WFP_enUS.fr

This file contains string resources in a string table resource, that is used for the US English locale. These resources are used when using this plug-in on InDesign US English locale. Also, this file can contain UI specifications, especially when they differ by locale.

WFP_jaJP.fr

This file contains string resources in a string table resource, that is used for the Japanese locale. These resources are used when using this plug-in on the InDesign Japanese locale. Also, this file can contain UI specifications, especially when they differ by locale.

WFP.rc

This file defines Windows-specific resources. In particular, the plug-in file version is defined in this file.

SDKResources.r

This file defines Macintosh-specific resources. In particular, the plug-in file version is defined in this file. This file is not generated, but simply included in the project from the **Utility** folder under your SDK folder.

What is Doc++, and where can I get it?

Doc++ is a tool that generated hyperlinked documentation from annotated C/C++ source files. It is similar to JavaDoc. The InDesign SDK team uses it to generate the API reference.

Doc++ can be downloaded from `http://sourceforge.net/projects/docpp/`. Currently, the Doc++ compiler does not exist for the Macintosh platform, however, you can download the UNIX source code from `http://sourceforge.net/projects/docpp/` and compile the Doc++ tool on Mac OS X.

SDKInfoButton.r

This file contains the Macintosh icon resource for the SDK information button. This file is not generated, but simply included in the project from the **Utility** folder under your SDK folder.

SDKInfoButton.ico

This file contains the Windows icon resource for the SDK information button. This file is not generated, but simply included in the project from the **Utility** folder under your SDK folder.

WFP.dsp

This is a Visual C++ project file for Windows.

WFP.mcp.xml

This is the XML version of the CodeWarrior project file for Macintosh. You use this by importing it and creating a CodeWarrior project.

Files included in the DocSource folder

PluginDesc.txt, **WFPBossClasses.txt**, **WFPDelta.txt**, and **WFPDesign.txt** are files annotated with Doc++ style comments, and are used for generating online documentation for your plug-in project.

WriteFishPrice.html

This HTML file is used as an entry point to the plug-in project's online documentation. The documentation is generated by **Doc++** in the **Documentation\WebDocs** folder under your SDK folder.

BuildDoc.bat

This batch file can be used to generate Doc++ documentation.

This concludes a brief description of the files included in the plug-in project generated by DollyWizard.

Detailed Descriptions of the Code Generated by DollyWizard

Let's go into more detail for the source files that we will be modifying throughout this exercise.

WFPActionComponent.cpp

WFPActionComponent Class

The WFPActionComponent class inherits the **CActionComponent** class, which implements the **IActionComponent** interface. **WFPActionComponent** class responds to menu selections in the **DoAction()** method, and distinguishes the actual selected menu item by means of the corresponding **ActionID**.

WFPActionComponent::DoAction()

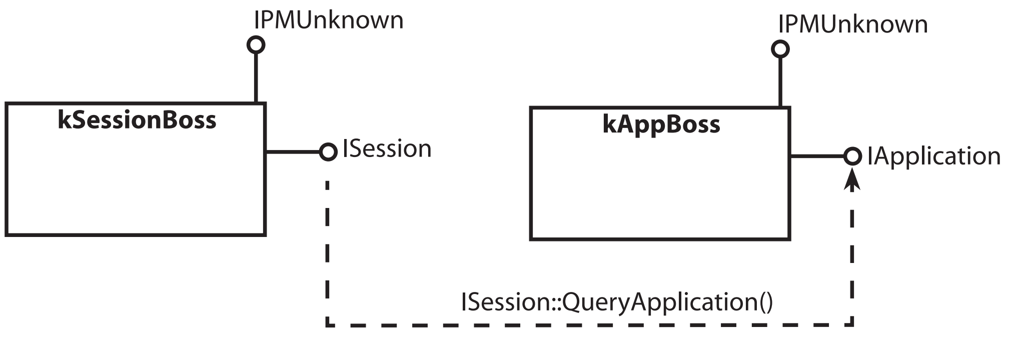
WFPActionComponent::DoAction() method, which overrides the **DoAction()** method in its parent class, (**CActionComponent**), receives the selected **ActionID** as a parameter and compares it with **kWFPAboutActionID** and **kWFPDialogActionID** that are defined in **WFPID.h**. If there is a match, it calls the **DoAbout()** and **DoDialog()** methods (the one we want), respectively.

WFPActionComponent::DoAbout()

WFPActionComponent::DoAbout() method is called from **WFPActionComponent::DoAction()**, and displays the About this Plug-In dialog (a simple modal alert box) provided by the **SDKUtilities::InvokePlugInAboutBox()** (in **{SDK}\Utilities\SDKUtilities.cpp**).

WFPActionComponent::DoDialog()

WFPActionComponent::DoDialog() method first obtains the **IApplication** interface by means of **gSession**. **gSession** is a global pointer to **ISession** interface aggregated in the **kSessionBoss**, which is a boss class object that describes the current InDesign application session. **IApplication** is an interface aggregated on the **kAppBoss**, which is a boss class object that describes the InDesign application itself. (More on boss classes later.)



From the **IApplication** interface, it obtains the **IDialogMgr** interface. This enables us to get to the InDesign dialog manager's boss class. Next, the **DoDialog()** method eagerly loads the dialog resources that corresponds to the current UI locale during the first instantiation, and saves to the InDesign database so that it could be loaded efficiently during subsequent instantiations. The current UI locale is obtained by instantiating a RsrcSpec object called dialogSpec by means of calling **LocaleSetting::GetLocale()**. There are several different kinds of constructors for the RsrcSpec object, but this method uses the following construct.

```
// Load the plug-in's resource.
PMLocaleId nLocale = LocaleSetting::GetLocale();
RsrcSpec dialogSpec
(
   nLocale,                    // Locale index from PMLocaleIDs.h.
   kWFPPluginID,               // Our Plug-in ID from WFPID.h.
   kViewRsrcType,              // This is the kViewRsrcType.
   kSDKDefDialogResourceID,           // Resource ID for our dialog.
   kTrue                       // Initially visible.
);
```

Once you instantiate a **RsrcSpec** object, the **DoDialog()** method now calls the **CreateNewDialog()** method on the **IDialogMgr** interface, as such:

```
IDialog* dialog = dialogMgr->CreateNewDialog(dialogSpec,IDialog::kMovableModal);
```

For the parameter list contains the **dialogSpec** we just instantiated, and a constant that specifies the modality of the dialog (**kMovableModal** constant defined in the **IDialog** interface). The **CreateNewDialog()** method then creates a movable dialog based on the **dialogSpec** and returns a pointer to a dialog window (**IDialog** on **kDialogWindowBoss**, or a derived boss class).

Finally, the **Open()** method on the **IDialog** interface is called, and the dialog is opened.

WFPDialogObserver.cpp

WFPDialogObserver Class

The **WFPDialogObserver** class inherits the **CDialogObserver** class, which implements the **IObserver** interface. Through the **WFPDialogObserver** class, you can register to listen to, or "observe", dynamic changes to the widgets on the dialog. Our example handles the information button on the dialog. When you click this button, the About this Plug-In dialog is displayed.

The "observer", which extends the **IObserver** interface, provides a mechanism to listen to changes to specific objects, known as "subjects". By attaching to a subject, observers can be notified when a change occurs, without having to poll for changes.

WFPDialogObserver::AutoAttach()

The **WFPDialogObserver::AutoAttach()** method, which is called by the InDesign application, enables subjects to attach themselves to an observer. In our example, the subject is the information button widget. If you need to observe other widgets on this dialog, you can add them here. Alternatively, you can observe each widget in separate observers, however, to keep the code simple, we are collectively observing all widgets on this dialog. The **OK** and **Cancel** buttons (with widget IDs of **kOKButtonWidgetID** and **kCancelButton_WidgetID**, respectively) are observed by the parent class, **CDialogObserver**, by default.

Let's examine how this works. First, the **WFPDialogObserver::AutoAttach()** method calls the **CDialogObserver::AutoAttach()** method in the parent class. This is so that the **OK** and **Cancel** buttons can be handled. Afterwards, the **IPanelControlData** interface (from the same boss object that hosts the current implementation, **WFPDialogObserver**) is obtained, and by using the parent class' **CDialogObserver::AttachToWidget()** method, it attaches the information button widget.

WFPDialogObserver::AutoDetach()

The **WFPDialogObserver::AutoDetach()** method, which is called by the InDesign application, allows subjects to be detached. Again, the **OK** and **Cancel** buttons are handled by default in the parent class, **CDialogObserver**. Like the **AutoAttach()** method, the **CDialogObserver::AutoDeta ch()** method is called to handled the **OK** and **Cancel** buttons. Afterwards, the **IPanelControlData** interface (from the same boss class that the current class resides in) is obtained, and by using the parent class' **CDialogObserver::DetachFromWidget()** method, it detaches the information button widget.

WFPDialogObserver::Update()

The **WFPDialogObserver::Update()** method is called by the host when a change occurs to the observed object. In our example, this is when the information button is clicked.

First, we call the **CDialogObserver::Update()** method to handle the **OK** and **Cancel** buttons up front, then we obtain the **IControlView** interface of the widget that caused the change, from the

theSubject parameter. To determine which widget ID actually caused the change, we call the **GetWidgetID()** method on the **IControlView** interface. If this widget ID corresponds to the ID for our information button (**kWFPIconSuiteWidgetID**) and if the message ID from the **theSubject** parameter is **kTrueStateMessage** (indicating that the button is pressed), we display the About this Plug-In dialog.

WFPDialogController.cpp

The WFPDialogController Class

The **WFPDialogController** class inherits the **CDialogController** class, which implements the **IDialogController** interface. This class handles the dialog initialization, as well data validation and the OK button click.

WFPDialogController::InitializeFields()

The **WFPDialogController::InitializeFields()** method initializes the widgets on the dialog. This method is called by the parent class when the dialog is opened, and when the dialog's **Reset** button (**Cancel** changes to **Reset** when you hold the Alt or Option key) is clicked, if you have not overridden the **CDialogController::ResetFields()** method. This method first needs to call the **CDialogController::InitializeFields()** method in the parent class.

WFPDialogController::ValidateFields()

The **WFPDialogController::ValidateFields()** method validates the fields on the dialog box. When the **OK** button is clicked, this method is called before the **ApplyFields()** method is called. Again, we first call the **CDialogController::ValidateFields()** method in the parent class. If there is even one field that has an invalid value, you can return the **WidgetID** to be selected. If all fields have valid values, then you can return **kDefaultWidgetId**, which will allow the parent class to call our **ApplyFields()** method.

WFPDialogController::ApplyFields()

The **WFPDialogController::ApplyFields()** method retrieves the values from the dialog fields and acts on them. In our example, we obtain the values from the widgets on the dialog, and do appropriate processing. The widgetId from the parameter list contains the widget ID that caused this method to be called. By default, this parameter contains **kOKButtonWidgetID**.

Where can I learn more about the available objects in the InDesign Object Model?

The InDesign object model is highly extensible, and quite large. There are various references available in the InDesign SDK:
(1) **{SDK}\Documentation\InterfaceList.txt**: We will show you how to use this file when we discuss resources in the **About Resources** section later.
(2) **{SDK}\Documentation\IObjectModel_*.txt and .xls**: These files make up a set of object model dumps obtained at InDesign runtime. The XLS version is an Excel spreadsheet with autofiltering, which you can use to help search for specific interfaces or implementations.
(3) **{SDK}:Tools:InterfaceListViewer** (Mac only): This is a useful tool that allows you to search the InDesign object model based on the name of a boss class or interface. For details, refer to **{SDK}:Tools:InterfaceListViewer:index.html** or **index-j.html**.

Step 2: Add a DropDownListWidget

Let's add a DropDownListWidget onto our dialog.

Step 2.1: Add a Widget ID for Our DropDownListWidget

First, to add a widget ID for our DropDownListWidget, open **WFPID.h** in your IDE.

Around line 63, you will find widget definitions for the dialog and information icon button.

```
// WidgetIDs:
DECLARE_PMID(kWidgetIDSpace, kWFPDialogWidgetID, kWFPPrefix + 0)
DECLARE_PMID(kWidgetIDSpace, kWFPIconSuiteWidgetID, kWFPPrefix + 1)
```

This is where we declare the widget ID for our DropDownListWidget.

On the following line, add:

```
// DropDownList widget ID
DECLARE_PMID(kWidgetIDSpace, kWFPDropDownListWidgetID, kWFPPrefix + 2)
```

Step 2.2: Define String Keys for List Items on Our DropDownListWidget

Next, let's define string keys for the items in the DropDownList. InDesign has a base type object called **PMString**. This string is used extensively for UI strings, and has a locale-based string lookup mechanism behind the scenes for automatic string translation. The translated strings are defined in a string table resource, and by specifying a string by its key, InDesign will automatically replace it with the corresponding localized string. If you look at **WFPID.h** after line 67, you can see several string keys defined there.

Now, let's define a string key for the DropDownListWidget items. Find the following in WFPID.h:

```
// Other StringKeys:
#define kWFPDialogTitleKey        kWFPStringPrefix "kWFPDialogTitleKey"
#define kWFPAboutBoxStringKey     kWFPStringPrefix "kWFPAboutBoxStringKey"
```

Right after that, add these lines:

```
#define kWFPDropDownItem_1Key kWFPStringPrefix "kWFPDropDownItem_1Key" // listitem
#define kWFPDropDownItem_2Key kWFPStringPrefix "kWFPDropDownItem_2Key" // listitem
#define kWFPDropDownItem_3Key kWFPStringPrefix "kWFPDropDownItem_3Key" // listitem
#define kWFPDropDownItem_4Key kWFPStringPrefix "kWFPDropDownItem_4Key" // listitem
```

If you know what your string key will be (e.g. **kWFP_TunaKey**), it might easier to define the key using the string, so that you can find it easier at a later time.

Step 2.3: Define Locale-specific Strings for List Items on Our DropDownListWidget

Next, we will define the string table resource entries that correspond to these string keys we just defined. These string tables are defined in **WFP_enUS.fr** and **WFP_jaJP.fr**, for use under the US English and Japanese locales, respectively. These two resource files are included by **WFP.fr**.

Ok, let's start by defining the US English string table entries. If you open **WFP_enUS.fr**, you will see that the string keys and the English strings are paired up, as shown below. Let's add strings for the 4 string keys we just added. Look for the following in **WFP_enUS.fr**:

```
resource StringTable (kSDKDefStringsResourceID + index_enUS)
{
   k_enUS,                                           // Locale Id
   kEuropeanMacToWinEncodingConverter,          // Character encoding converter
   {
...omitted
// ----- Panel/dialog strings
          kWFPDialogTitleKey,     kWFPPluginName "[US]",
```

Right after that, add the following:

```
   // Drop down list item strings
          kWFPDropDownItem_1Key,                     "Tuna",
          kWFPDropDownItem_2Key,                     "Salmon",
          kWFPDropDownItem_3Key,                     "Bonito",
          kWFPDropDownItem_4Key,                     "Yellowtail",
```

Similarly, we can add strings to the Japanese string table. Open **WFP_jaJP.fr**, and look for the following:

```
resource StringTable (kSDKDefStringsResourceID + index_jaJP)
{
   k_jaJP,        // Locale Id
   0,             // Character encoding converter
   {
...omitted
          // ----- Panel/dialog strings
          kWFPDialogTitleKey,     kWFPPluginName "[JP]",
```

Right after that, add the following:

```
          // Drop down list item strings
          kWFPDropDownItem_1Key,                     "まぐろ", // Tuna
          kWFPDropDownItem_2Key,                     "さけ", // Salmon
          kWFPDropDownItem_3Key,                     "かつお", // Bonito
          kWFPDropDownItem_4Key,                     "ぶり", // Yellowtail
```

Step 2.4: Add a DropDownListWidget to Our Dialog Resource

Now, we will add the DropDownListWidget on our dialog resource. The dialog resource is defined in **WFP.fr** around line 289. This resource already contains 3 widgets: the default OK button, the cancel button and the information icon button.

This is where we will add the DropDownListWidget. Look for this in **WFP.fr**, around line 289:

What if I can't enter these Japanese characters into the Japanese string table?

For the purpose of these exercises, it is not necessary to enter Japanese characters into the resource string tables. Our SDK sample plug-ins often put a locale-specific suffix on these strings, such as "MyString [JP]", so that you know the appropriate string table is being used. You can choose to do the same.

However, if you are actually going to release your plug-ins commercially, there is a chance that some InDesign Japanese version use will be using your plug-in. It would be a good idea to but in case someone will be using your plug-in in InDesign 2.0's Japanese version, it's a good idea to define strings for the Japanese string table. Better yet, you may even want to obtain assistance in translating the strings into Japanese, or other locales supported by InDesign.

```
resource WFPDialogWidget (kSDKDefDialogResourceID + index_enUS)
{
    __FILE__, __LINE__,
    kWFPDialogWidgetID,     // WidgetID
    kPMRsrcID_None,         // RsrcID
    kBindNone,              // Binding
    0, 0, 388,112,          // Frame (l,t,r,b)
    kTrue, kTrue,           // Visible, Enabled
    kWFPDialogTitleKey,     // Dialog name
    {
             ...omitted
             ADBEIconSuiteButtonWidget
             (
                      ...omitted
             ),
```

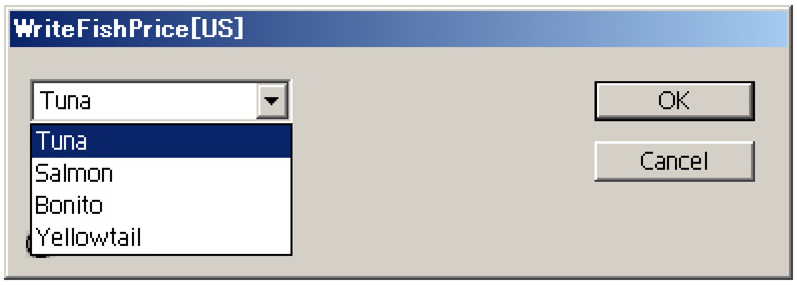
Right after that, add the following:

```
             // DropDownList Widget resource
             DropDownListWidget
             (
                      kWFPDropDownListWidgetID, kSysDropDownPMRsrcId, // WidgetId, RsrcId
                      kBindNone,                      // Frame binding
                      Frame( 10, 16, 140, 36 ),       // Frame (l,t,r,b)
                      kTrue, kTrue,                   // Visible, Enabled
                      {{                              // List Items
                               kWFPDropDownItem_1Key,
                               kWFPDropDownItem_2Key,
                               kWFPDropDownItem_3Key,
                               kWFPDropDownItem_4Key,
                      }}
             ),
```

This concludes the step of adding a DropDownListWidget to our dialog.

Step 2.5: Save, Build and Test

Now, let's save all of the source files you edited, build the plug-in, load it in InDesign, and try using our newly added DropDownListWidget.



Does your pull down widget show the 4 strings you just added?

If the 4 strings are showing in your DropDownListWidget, let's move on to the next step.

Step 3: Add a TextEditBoxWidget

Step 3.1: Add a Widget ID for Our TextEditBoxWidget

In preparation for adding a TextEditBoxWidget on our dialog, let's open **WFPID.h** again so we can define some widget IDs. Right after the place where you added some widget IDs in step 2, we shall add a widget ID for our TextEditBox. Look for the following:

```
// WidgetIDs:
DECLARE_PMID(kWidgetIDSpace, kWFPDialogWidgetID, kWFPPrefix + 0)
DECLARE_PMID(kWidgetIDSpace, kWFPIconSuiteWidgetID, kWFPPrefix + 1)
DECLARE_PMID(kWidgetIDSpace, kWFPDropDownListWidgetID, kWFPPrefix + 2) //DropDownList
```

Add the following on the next line:

```
DECLARE_PMID(kWidgetIDSpace, kWFPTextEditBoxWidgetID, kWFPPrefix + 3) //TextEditBox
```

Step 3.2: Add a TextEditBoxWidget Resource to Our Dialog Resource

Next, we shall add the TextEditBoxWidget resource into our dialog resource definition. Just like we did in step 2, we shall define this right after the place where we added our DropDownListWidget. Look for the following:

```
// DropDownList Widget resource
           DropDownListWidget
           (
           //      ...omitted
           ),
```

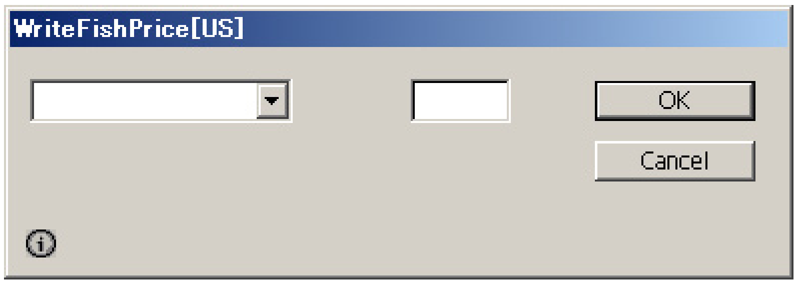
Add the following on the next line:

```
           // TextEditBox Widget resource
           TextEditBoxWidget
           (
                  kWFPTextEditBoxWidgetID,             // WidgetId
                  kSysEditBoxPMRsrcId,                 // RsrcId
                  kBindNone,                           // Frame binding
                  Frame(200, 16, 250, 36),             // Frame (l,t,r,b)
                  kTrue, kTrue                         // Visible, Enabled
                  0,             // Widget id of nudge button (0 so we dont get one)
                  0, 0,          // small,large nudge amount
                  0,             // max num chars( 0 = no limit)
                  kFalse,        // is read only
                  kFalse,        //**new in 2.0: should notify each key stroke
                  kFalse,        // range checking enabled
                  kFalse,        // blank entry allowed
                  0,             // Upper bounds
                  0,             // Lower bounds
                  "",            // Initial text
           ),
```

We are done adding a TextEditBoxWidget onto our dialog. This is all we need to be able to put a TextEditBoxWidget on a dialog.

Step 3.3: Save, Build and Test

Now, let's save all of the source files you edited, build the plug-in, load it in InDesign, and try using our newly added TextEditBoxWidget.



Do you see your TextEditBoxWidget as shown in the screen shot above?

Try entering some characters, or even copying some text from another application and pasting it into the TextEditBoxWidget using shortcut keys. InDesign handles the shortcut keys for us.

If you can do some simple operations with this TextEditBoxWidget, let's move on.

Step 4: Add a StaticTextWidget

So far, we have added onto our dialog a DropDownListWidget to select our product and a TextEditBoxWidget to enter the unit price. However, it seems like something is missing, doesn't it? We should have a currency symbol right next to our TextEditBox. InDesign can be used under various locales, so let's utilize the power of InDesign's international capabilities and display a $ (dollar symbol) or a ¥, based on the locale.

Step 4.1: Add a Widget ID for Our StaticTextWidget

To add an ID for our StaticTextWidget, open **WFPID.h**. Right after where we added the TextEditBoxWidget in the step 3, let's define the ID for our StaticTextWidget. Look for the following:

```
// WidgetIDs:
DECLARE_PMID(kWidgetIDSpace, kWFPDialogWidgetID, kWFPPrefix + 0)
DECLARE_PMID(kWidgetIDSpace, kWFPIconSuiteWidgetID, kWFPPrefix + 1)
DECLARE_PMID(kWidgetIDSpace, kWFPDropDownListWidgetID, kWFPPrefix + 2) //DropDownList
DECLARE_PMID(kWidgetIDSpace, kWFPTextEditBoxWidgetID, kWFPPrefix + 3) //TextEditBox
```

Add the following on the next line:

```
DECLARE_PMID(kWidgetIDSpace, kWFPStaticTextWidgetID, kWFPPrefix + 4) //StaticText
```

Step 4.2: Define a String Key for Our StaticTextWidget

Next, let's define a string key for each of the currency symbols that will be displayed in the StaticTextWidget.

```
// StaticText string key
#define kWFPStaticTextKey        kWFPStringPrefix "kWFPStaticTextKey"
                                 // StaticText string key. (yen or dollar character)
```

Step 4.3: Define Locale-specific Strings for Out StaticTextWidget

Then, let's define the localized strings that correspond to this string key in the string table resources in **WFP_enUS.fr** and **WFP_jaJP.fr**.

Look for the "// Panel/dialog strings" comment in **WFP_enUS.fr**, and add the following:

```
// StaticText string key. (yen or dollar character)
kWFPStaticTextKey,                                      "$",
```

Look for the "// Panel/dialog strings" comment in **WFP_jaJP.fr**, and add the following:

```
// StaticText string key. (yen or dollar character)
kWFPStaticTextKey,                                      "￥",
```

The yen symbol specified here is a dual-byte character (specifically, ShiftJIS code 0x8F81). If you can't enter Japanese characters, you can simply specify this string as a "Y" (upper case).

Step 4.4: Add a StaticTextWidget Resource to Our Dialog Resource

Next, let's add the StaticTextWidget resource right after where we added the TextEditBoxWidget in **WFP.fr**. Look for the following:

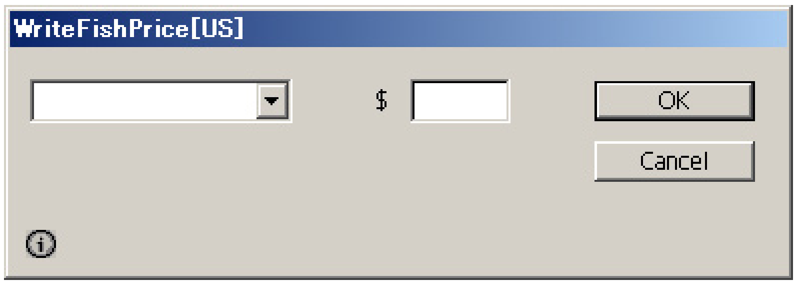
```
// TextEditBox Widget resource
TextEditBoxWidget
(
        //      ...omitted
),
```

Add the following on the following line:

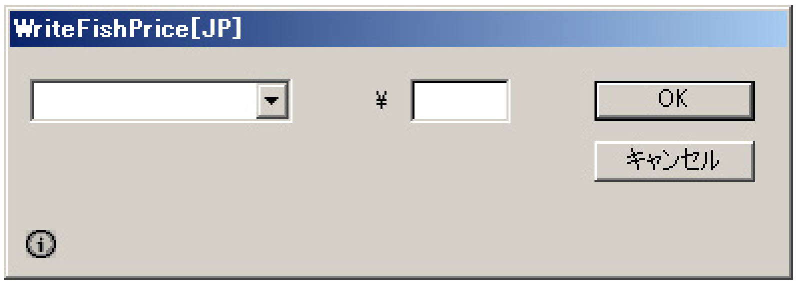
```
// StaticText widget resource
StaticTextWidget
(
        kWFPStaticTextWidgetID,        // WidgetId
        kSysStaticTextPMRsrcId,        // RsrcId
        kBindNone,                     // Frame binding
        Frame( 150, 16, 190, 36  ),    // Frame (l,t,r,b)
        kTrue, kTrue, kAlignRight,     // Visible, Enabled, Alignment
        kDontEllipsize, //**new element in InDesign 2.0: don't add any ellipses
        kWFPStaticTextKey,             // Text
        kWFPTextEditBoxWidgetID // WidgetId for associated cntrl for shortcut focus
),
```

Step 4.5: Save, Build and Test

Now, let's save all of the source files you edited, build the plug-in, load it in InDesign, and try using our newly added TextEditBox. When used under the US English locale, you'll see this:



Under the Japanese locale, you'll see this.



As you can see, different strings can be displayed based on the locale. You can do the same with other UI widgets in InDesign.

In this case, the meaning of the data you enter in the TextEditBoxWidget will change, so we may have to do some extra processing later.

This concludes our design of the dialog. As we have just demonstrated, you can design your plug-ins starting with the user interface, before implementating what it does. In otherwords, we can develop the "controller" in a manner that is decoupled from the rest of the plug-in.

The Story About Resources

Let's talk about resources very briefly. This plug-in's resources are all defined in **WFP.fr** (and the **_enUS.fr** and **_jaJP.fr** that are included). InDesign uses resource definitions for strings, dialog, panels, menus, and boss classes, that are cross-platform between the Macintosh and Windows development environments. (There are a few resource types that are not included in these resource definitions. They are limited to a few types that are not compatible across platforms: file version, icon, and picture resources.) In order to share resource definitions across platforms, InDesign uses the OpenDoc Framework Resource Compiler (ODFRC).

Let's take a closer look at the various resources that are defined in **WFP.fr**.

The PluginVersion Resource

```
/*
// Plugin version definition.
*/
resource PluginVersion (kSDKDefPluginVersionResourceID)
{
   kTargetVersion,
   kWFPPluginID,
   kSDKDefPlugInMajorVersionNumber, kSDKDefPlugInMinorVersionNumber,
   kSDKDefHostMajorVersionNumber, kSDKDefHostMinorVersionNumber,
   kSDKDefPersistMajorVersionNumber, kSDKDefPersistMinorVersionNumber,
   { kWildFS }
};
```

This is the resource that describes the plug-in version. **kTargetVersion** is the build number of InDesign that is the target for the plug-in. You normally specify this using the **kTargetVersion** macro, which expands depending on whether the plug-in is being built for the release or debug build. **kWFPPluginID** is an ID unique to this plug-in, and is defined in **WFPID.h**. Following that are the major and minor versions of the plug-in, the major and minor versions of the host application, and the major and minor version of the persistent data format used by this plug-in. These version numbers are defined in **SDKDef.h** for the InDesign SDK you are using. Unless you have persistent data that you read and write from your plug-in, you won't have to change these version numbers. Since our example doesn't persist any data into InDesign, we will leave these as is.

The last entry, {**kWildFS**}, is an array that specifies which "feature set" this plug-in works under. A "feature set" is an abstraction of a set of InDesign features, and is different from the concept of a UI Locale. The feature set setting allows you to customize not only the user interface but also the behavior of your plug-in. You can choose from three possible settings: **kWildFS** (any feature set), **kInDesignRomanFS** (Roman feature set), and **kInDesignJapaneseFS** (Japanese feature set). Our plug-in is specified to work under any feature set, so it specifies **kWildFS**. The **PluginVersion** re-

Where can I find out more about other resources in the InDesign SDK?

While it pertains to the InDesign 1.x API, the **UIProgrammingGuide** (**{SDK}\Documentation\UIProgrammingGuidepdf**) offers a further detailed view of all of these resources that we reviewed in this section.

source is defined in **{SDK}\API\Includes\objectmodeltypes.fh**.

The Boss Class Definition Resource

```
/*
// Boss class definition.
*/
resource ClassDescriptionTable(kSDKDefClassDescriptionTableResourceID)
{{{
   /**
...omitted
   */
   Class
   {
           kWFPDialogBoss,
           kDialogBoss,
           {
                   /** Provides management and control over the dialog.
                   */
                   IID_IDIALOGCONTROLLER, kWFPDialogControllerImpl,
                   /** Allows dynamic processing of dialog changes.
                   */
                   IID_IOBSERVER, kWFPDialogObserverImpl,
           }
   },
...omitted
}}};
```

These boss class definitions specify the InDesign object model. You can say that they are somewhat analogous to C++ class definitions. Boss classes are discussed in further detail in a column later in this cookbook, so we will only highlight the resource part of the boss class definition in this section.

Each boss class definition shown in this file begins with the resource keyword **Class**. This keyword **Class** is used to define a new boss class in the InDesign object model.

Also, while this plug-in doesn't define one, you can also start a boss class with the keyword **AddIn**. This allows you to add interfaces to existing boss class definitions. (You can refer to the SDK sample plug-in **{SDK}\SampleCodeGraphics\FrameLabel** to see an example of an **AddIn** resource.)

Let's come back to our plug-in. The next line, **kWFPDialogBoss**, is the ID for this boss class. The next line, **kDialogBoss**, indicates the ID of the parent boss class. This specifies our **kWFPDialogBoss** will inherit from **kDialogBoss**. All the functionality provided by **kDialogBoss** (all implementations backing the interfaces aggregated on the boss class) is provided for **kWFPDialogBoss**. **kWFPDialogBoss** can extend this functionality (by adding other interfaces), or adapt it (by overriding existing interfaces, mapping them onto its own implementation). If you don't want to specify a parent boss class, you specify **kInvalidClass** in its place.

Next is the interface-to-implementation mapping list for the boss class. In this dialog boss class, we are overriding the **IID_IDIALOGCONTROLLER** and **IID_IOBSERVER** interfaces from **kDialogBoss**. The actual C++ implementations are referred to indirectly by their implementation IDs, namely **kWFPDialogControllerImpl** and **kWFPDialogObserverImpl**, respectively.

In InDesign plug-ins, you use the **CREATE_PMINTERFACE( )** macro to bind a specific implementation ID onto a specific C++ class. This allows InDesign to call the C++ implementation by its implementation ID.

Let's open a text file called **InterfaceList.txt**, located in your **{SDK}\Documentation** folder, and search for "**kDialogBoss**". You can see that **kDialogBoss** inherits another boss class called **kPrimaryResourcePanelWidgetBoss**, and overrides 8 different interfaces. As you can see, the entire InDesign object model is built by a collection of these "Boss" classes, and we can build our plug-ins by overriding and/or extending existing boss classes, like you saw in **kWFPDialogBoss**.

```
kDialogBoss
kPrimaryResourcePanelWidgetBoss
 IID_ICONTROLVIEW, kPanelViewImpl, //"../API/WidgetIncludes/PanelView.h"
 IID_IOBSERVER, kCDialogObserverImpl, //"../API/WidgetIncludes/CDialogObserver.h"
 IID_IDIALOGCONTROLLER, kCDialogControllerImpl, //"../API/WidgetIncludes/CDialog..."
 IID_IPREVIEWDIALOGERRORHANDLER, kCPreviewDialogErrorHandlerImpl, //"../API/..."
 IID_ISYSFILEDATA, kPlugInFileDataImpl,
 // inherited from kPrimaryResourcePanelWidgetBoss
 IID_IWIDGETPARENT, kWidgetParentImpl, //"../API/Interfaces/UI/IWidgetParent.h"
 IID_IRESOURCESRCFILEINFO, kResourceSrcFileInfoImpl, //"../API/Interfaces/UI/IRes..."
 IID_ISYSFILEDATA, kPlugInFileDataImpl,
 // inherited from kGenericPanelWidgetBoss
 IID_IEVENTHANDLER, kPanelEventHandlerImpl, //"../API/Includes/CEventHandler.h"
 // rest omitted...
```

Implementation Definition Resource

```
/*
// Implementation definition.
*/
resource FactoryList (kSDKDefFactoryListResourceID)
{
   kImplementationIDSpace,
   {
#include "WFPFactoryList.h"
   }
};
```

This resource allows you to register the implementation IDs for your C++ implementations into the InDesign object model. The include file, **WFPFactoryList.h**, registers the implementation IDs with the use of the **REGISTER_PMINTERFACE()** macro, and is also included in **WFPNoStrip.cpp** to prevent deadstripping. By sharing this piece of code, we prevent a situation where we forget to specify the implementation ID in one place or the other.

The **REGISTER_PMINTERFACE()** macro in **WFPFactoryList.h**, defines the implementation ID when used in a resource definition, and prevents deadstripping when used in a **.cpp** file.

Menu Definition Resource

```
/*
// Menu definition.
*/
resource MenuDef (kSDKDefMenuResourceID)
{
   {
          // The About Plug-ins sub-menu item for this plug-in.
          kWFPAboutActionID, // ActionID (kInvalidActionID for positional entries)
          kWFPAboutMenuPath,             // Menu Path.
          kSDKDefAlphabeticPosition,     // Menu Position.
          kSDKDefIsNotDynamicMenuFlag,   // kSDKDefIsNotDynamicMenuFlag or ...

          // The Plug-ins menu sub-menu items for this plug-in.
          kWFPDialogActionID,
          kWFPPluginsMenuPath,
          kWFPDialogMenuItemPosition,
          kSDKDefIsNotDynamicMenuFlag,

   }
};
```

This defines the menu. The 1st block defines the menu that is used to show the About this Plug-In dialog, and the second block defines the menu that is used to show the dialog that we just designed in our plug-in.

The first line in each block is the action ID that is issued when of the menu is selected.

The second line specifies the menu path that corresponds to the action ID specified right above it. In our example, **kWFPAboutMenuPath** is a preprocessor #define that expands to **"Main:AppleMenu:AboutPlugins:SDK"** on the Macintosh, and **"Main:&Help:AboutPlugins:SDK"** on Windows.

The third line specifies the position of the menu item, relative to the other menu items under the same menu path. **kSDKDefAlphabeticPosition** is defined in **SDKDef.h**. If you use this constant (defined as 1.0), InDesign will build the menu after sorting individual menu items under the same path. In most cases, you can just use this constant.

The fourth line defines the behavior of the menu item. If you want to change the menu each time it is displayed, set this to **kSDKDefIsDynamicMenuFlag**, otherwise, set it to **kSDKDefIsNotDynamicMenuFlag**. Normally, you specify **kSDKDefIsNotDynamicMenuFlag**.

Action Definition Resource

```
/*
// Action definition.
*/
resource ActionDef (kSDKDefActionResourceID)
{
   {
          kWFPActionComponentBoss, // ClassID of bossclass that implements ActionID.
          kWFPAboutActionID,     // ActionID.
          kWFPAboutMenuKey,      // Sub-menu string.
          kOtherActionArea,      // Area name (see ActionDefs.h).
          kNormalAction,         // Type of action (see ActionDefs.h).
          kDisableIfLowMem,      // Enabling type (see ActionDefs.h).
          kInvalidInterfaceID,   // Selection InterfaceID this action cares about...
          kSDKDefVisibleInKBSCEditorFlag,      // kSDKDefVisibleInKBSCEditorFlag...
   ...omitted
   }
};
```

This defines the **action** that will be invoked from the menu. (An **action** is an abstraction for what happens when a menu item is selected or a shortcut key is pressed.) **kWFPActionComponentBoss**, in the first line, is a boss class that handles the action IDs. **kWFPAboutActionID**, in the second line, is an action ID that is to be handled by the boss class specified in the first line. **kWFPAboutMenuKey**, in the third line, is the string key that corresponds to the action ID listed in the second line. **kOtherActionArea**, in the fourth line, specifies the keyboard shortcut editor (KBSCE) area , and in our example, we are using "other". You can find KBSCE areas defined in **{SDK}\API\Includes\ActionDefs.h**.

The fifth line specifies the action type. You generally specify **kNormalAction** here. The sixth line specifies how the menu is enabled or disabled. Again, these enabling types are defined in **ActionDefs.h**. The seventh line specifies the interfaceID for the selection that is required for the action to be active. If you do not require any selections for your action to be active, you can specify **kInvalidInterfaceID**. The eighth line specifies whether the shortcut key entry is visible in the KBSCE or not.

Currently, all plug-in code that DollyWizard generates, makes even the About this Plug-In action visible in the KBSCE, however, it is probably better that you change it to **kSDKDefInvisibleInKBSCE ditorFlag**.

LocaleIndex Resource Definitinon for String Tables

```
/*
// LocaleIndex Definition.
// The LocaleIndex should have indices that point at your
// localizations for each language system that you are
// localized for.
*/
/*
// String LocaleIndex.
*/
resource LocaleIndex (kSDKDefStringsResourceID)
{
   kStringTableRsrcType,
   {
          kWildFS, k_enUS, kSDKDefStringsResourceID + index_enUS
          kInDesignJapaneseFS, k_jaJP, kSDKDefStringsResourceID + index_jaJP
   }
};
```

This resource cross-references the string tables with the InDesign Feature Set and locale information.

kStringTableRsrcType specifies the type of resource we are cross-referencing. In this case, this resource is used as a locale index resource to switch the string tables. **kWildFS** means that this entry applies to all Feature Sets (defined in **FeatureSets.h**). **k_enUS** specifies that the corresponding locale is US English. So the first line in the curly brackets after **kStringTableRsrcType** means: "For all feature sets and in the US English locale, use the string table that is referenced by the resource ID **kSDKDefStringsResourceID + index_enUS**".

The next line specifies that when the feature set is kInDesignJapaneseFS and the locale is Japanese (**k_jaJP**), use the string table that is referenced by the resource ID **kSDKDefStringsResourceID + index_jaJP**. In our example, we have omitted resource definitions for other locales (e.g.: French, German, UK English, etc.). It is extra work to define resources for other languages from the beginning of development, so if you change the **k_enUS** to **kWild** in the first line, your plug-in will use the US English string resources for locales other than Japanese. This is probably a practical change to make, until you can actually get time to define your resources for other locales.

As a side note, InDesign currently supports 13 different locales: US English, UK English, German, French, Japanese, Spanish, Portuguese, Swedish, Danish, Dutch, Italian, Norwegian, and Finnish.

```
resource LocaleIndex (kSDKDefStringsNoTransResourceID)
{
   kStringTableRsrcType,
   {
          kWildFS, k_Wild, kSDKDefStringsNoTransResourceID + index_enUS
   }
};

resource StringTable (kSDKDefStringsNoTransResourceID + index_enUS)
{
   k_enUS,                                      // Locale Id
   kEuropeanMacToWinEncodingConverter,   // Character encoding converter
   {
          // No-Translate strings go here:
   }
};
```

The next **LocaleIndex** resource defines a no-translation string table, as there may be strings that we don't want translated automatically.

LocaleIndex Resource Definition for Dialogs

```
/*
// Dialog LocaleIndex.
*/
resource LocaleIndex (kSDKDefDialogResourceID)
{
   kViewRsrcType,
   {
           kWildFS,       k_Wild, kSDKDefDialogResourceID + index_enUS
   }
};
```

This is similar to the **LocaleIndex** resource for the string tables, however, dialogs are defined as **kViewRsrcType** resources, instead of **kStringTableRsrcType**. In our example, all feature sets and locales use the same US English dialog resource. Despite the fact that the dialog resource comes strictly from the US English locale index, don't worry: the strings on the dialog are indeed localized, as we have defined above with **kStringTableRsrcType**.

Custom Type Definitions

```
/*
// Type definition.
*/
type WFPDialogWidget(kViewRsrcType) : DialogBoss(ClassID = kWFPDialogBoss)
{
};
```

This resource defines a widget type. In our example, **WFPDialogWidget** belongs to the **kViewRsrcType** resource type, and inherits the **DialogBoss** widget. (NOTE: This really should be **DialogWidget**, however for historical reasons, the name of the widget is kept as **DialogBoss**). This statement defines the boss class that backs the UI of this type.

The DialogBoss widget type inherits from another widget called **PrimaryResourcePanelWidget**, and are both defined in **{SDK}\API\WidgetIncludes\Widgets.fh**.

Dialog (View) Resource

```
resource WFPDialogWidget (kSDKDefDialogResourceID + index_enUS)
{
   __FILE__, __LINE__,
   kWFPDialogWidgetID,    // WidgetID
   kPMRsrcID_None,        // RsrcID
   kBindNone,             // Binding
   0, 0, 388,112,         // Frame (l,t,r,b)
   kTrue, kTrue,          // Visible, Enabled
   kWFPDialogTitleKey,    // Dialog name
   {
          DefaultButtonWidget
          (
                 kOKButtonWidgetID,            // WidgetID
                 kSysButtonPMRsrcId,           // RsrcID
                 kBindNone,                    // Binding
                 292, 16, 372, 36,             // Frame (l,t,r,b)
                 kTrue, kTrue,                 // Visible, Enabled
                 kSDKDefOKButtonApplicationKey,       // Button text
          ),
   ...omitted
   },
};
```

This resource defines our plug-in's dialog. This dialog is specified for the US English locale, however, as we specified in the LocaleIndex resource above, it will be used for all feature sets and locales. Since our plug-in does not require a different dialog definition per locale (that is, the widget arrangement the same no matter what locale is used), we have consolidate the definitions into one **.fr** file.

Since this resource is complex, here's a good way to navigate through the resource definitions. First, let's look at the definition of the parent widget type, DialogBoss. Open **{SDK}\API\WidgetIncludes\Widgets.fh**.

```
type DialogBoss (kViewRsrcType) : PrimaryResourcePanelWidget (ClassID = kDialogBoss)
{
};
```

You can see that the parent of **DialogBoss** is **PrimaryResourcePanelWidget**. Now, if we examine the definition of **PrimaryResourcePanelWidget**, you discover that its parent is the root **Widget**, so we've reached the top of the hierarchy. Let's take a look at **PrimaryResourcePanelWidget**.

```
type PrimaryResourcePanelWidget (kViewRsrcType) : Widget (ClassID = kPrimaryResourceP
anelWidgetBoss)
{
   ResourceSrcFileInfo;
   CControlView;
   CTextControlData;
   CPanelControlData;
};
```

Now, let's drill down further and examine **CControlView**. This is also defined in the same file, **Widgets.fh**.

```
type CControlView : Interface (IID = IID_ICONTROLVIEW)
{
   longint;              // fWidgetId
   PMRsrcID;             // fRsrcId, fRsrcPlugin
   integer;              // fFrameBinding
   Frame;                // fFrame
   integer;              // fVisible
   integer;              // fEnabled
};
```

Take a look at the first line in the type definition here. It's a bit different than other widget type definitions we have seen so far, as it specifies an **IID** instead of **ClassID**. This is known as an "interface type". This indicates that **CControlView** is a persistent interface in **kPrimaryResourcePanelWidgetBoss**.

Try searching for **kPrimaryResourcePanelWidgetBoss** in **InterfaceList.txt**. You will see its aggregated interfaces.

String Table Resource

Next is the String Table Resource. **WFP.fr** includes two other **.fr** files, **WFP_enUS.fr** and **WFP_jaJP.fr**. They specify US English and Japanese string table resources, respectively. By separating the string table resources by locale, they become easier to manage. Let's take a closer look.

```
resource StringTable (kSDKDefStringsResourceID + index_enUS)
{
   k_enUS,                                     // Locale Id
   kEuropeanMacToWinEncodingConverter,   // Character encoding converter
   {
          ...omitted
          kWFPDropDownItem_1Key,                             "Tuna",
          ...omitted
   }
};
```

This is the US English string table resource definition. The first line specifies the locale ID, **k_enUS**. The next line specifies the character encoding converter, which absorbs the differences between the hi-ASCII characters on the Macintosh and Windows platforms. The next line is where the actual string table is defined. The string key and the corresponding localized strings are paired up, separated by a comma.

Next, let's take a look at the Japanese locale string table. The first line specifies the locale ID, **k_jaJP**. Since we don't need to use a character encoding converter for Japanese, the next line contains a zero. The actual string table is defined after that, like in the US English string table, where the string key and the localized strings are paired up and separated by a comma.

```
resource StringTable (kSDKDefStringsResourceID + index_jaJP)
{
   k_jaJP,        // Locale Id
   0,             // Character encoding converter
   {
          ...omitted
          kWFPDropDownItem_1Key,                "まぐろ",      // Tuna
          ...omitted
   }
};
```

That was a brief tour of the string table resource definition.

Step 5: Obtain the value from the DropDownListWidget

In this step, we will obtain the string value from a dialog widget, and create a string that we could insert into an InDesign document.

Step 5.1: Get String Value of Selected Item from DropDownListWidget

First, we shall add code to get the fish name from the DropDownListWidget. In this plug-in, when the user clicks on the OK button, we want to insert text into the InDesign document. As you may recall, the method that gets called when the **OK** button is clicked is **WFPDialogController::Apply Fields()**. Since we had delegated the actual handling of the button click to **CDialogController**, the parent class of **WFPDialogController**, we had some basic dialog functionality in our plug-in.

Let's add some more functionality to the **WFPDialogController::ApplyFields()** method. Currently, your **WFPDialogController::ApplyFields()** method should contain the following code:

```
/* ApplyFields
*/
void WFPDialogController::ApplyFields(const WidgetID& widgetId)
{
   // Replace with code that gathers widget values and applies them.
   SystemBeep();
}
```

This code was generated by DollyWizard from the Dialog template. Delete the line with **SystemBeep()**.

In its place, we will call **CDialogController::GetTextControlData()** to obtain the text on the widget. This method requires a widget ID as a parameter and returns a **PMString** object. We will use this method (from the **ITextControlData** interface in the dialog boss class) to obtain the text data on the widget. The string that we get back is not the string that we see on our dialog, but actually the string key that we defined in the string table resource.

On the next line, by using the lookup feature of the **PMString** object, we will translate the **PMString** to a string in our current locale. Add the following code:

```
   //Get Selected text of DropDownList.
   PMString resultString;

   resultString = this->GetTextControlData( kWFPDropDownListWidgetID );
   resultString.Translate(); // Look up our string and replace.
```

Step 5.2: Save, Build and Test

Let's build our plug-in, move the plug-in to InDesign's Plug-Ins folder if needed, put a break point in **WFPDialogController::ApplyFields()** on the line with **resultString.Translate()**, and start InDesign.

Then, select our plug-in from the **Plug-Ins** menu, which will open the dialog box, and select **Bonito** from the DropDownListWidget, and click the **OK** button. The execution should break at the break point. Now step over that one line. Did the **fABuffer** (platform specific character encoding) field in the **resultString** change to the same value as the currently selected item on your DropDownList-Widget?

If you have verified that you are indeed getting the same string as selected, let's move on.

Step 6: Get the text in the TextEditBoxWidget

Step 6.1: Get String Value of the Text in TextEditBoxWidget

Just as we did in the previous step, we will use the **CDialogController::GetTextControlData()** method. Add this code right after the line after the line with **resultString.Translate()**.

```
// Get the editbox list widget string.
PMString editBoxString = this->GetTextControlData( kWFPTextEditBoxWidgetID );
```

Step 6.2: Form String to Insert into Text Frame

Let's move on. Let's concatenate a string to insert into the InDesign document. Again, we will utilize the **PMString** class.

We will concatenate the string in the following order: product name, tab character, currency symbol, price, new line.

```
PMString moneySign( kWFPStaticTextKey );
moneySign.Translate(); // Look up our string and replace.

resultString.Append( '\t' ); // Append tab code.
resultString.Append( moneySign );
resultString.Append(editBoxString);
resultString.Append( '\r' ); // Append return code.
```

In the code above, we created a **PMString** object called **moneySign**, which holds the string key for our currency symbol. Then we translate it based on the current locale.

Then, we concatenated the tab character, the actual currency symbol, the TextEditBoxWidget string that represents the price the user will enter, and a new line character, using the **PMString::Append()** method.

The **PMString** class has a wide variety of methods and is quite useful. It is frequently used in InDesign, so we recommend that you browse through the **PMString.h** file in the **API\Includes** folder in your InDesign SDK folder.

The Story About Boss Classes

What are "Boss Classes"?

Boss Classes refer to a class of objects in the InDesign object model. It is similar to a C++ class, however, boss classes are declared differently. You declare boss classes in a unique way. InDesign consists of boss classes that represent document objects, such as images, text, and layers, as well as widgets, such as dialog buttons and input fields. For example, InDesign pages are represented in this object hierarchy: document -> spread -> spread layer -> page. This hierarchy is represented by a boss class architecture.

Third party developers can access these boss class objects when developing InDesign plug-ins. However, in order to use the appropriate boss class for the desired task at hand, you must understand the InDesign object model and its architecture. Also, you need to be aware of which boss class provides what kind of functionality. The boss class objects, similar to C++ objects, can be invoked by calling methods (or member functions), but the way you call boss class objects differs from how you call methods in C++. Details are on the following section about "Interfaces".

Also, just like in C++ classes, boss classes can inherit other boss classes. For example, **kSplineItemBoss** inherits from **kDrawablePageItemBoss**, and furthermore, **kDrawablePageItemBoss** inherits from **kPageItemBoss**. Child boss classes can call methods in the parent boss classes, making for a truly abstract object-oriented programming model.

Also, boss classes in plug-ins developed by third-party developers are recognized immediately by InDesign, and are used just like boss classes that are part of the core InDesign application. For example, you can make a new boss class (in this case, this would be a custom page item) that inherits **kDrawablePageItemBoss**, instantiate this boss class, and put it on an InDesign document.

Where can I find out more about objects in an InDesign document?

Refer to the **Document Structure** chapcter of the InDesign Programming Guide (**{SDK}\Documentation\ProgrammingGuide.pdf**).

The Story About Interfaces

What are "Interfaces"?

When you call methods in a boss class, you call them in a style that is different from how you normally call a method on a C++ class. First, you obtain what's known as an "interface" from a boss class, and then call a method on that interface. We use the term "interface", referring to something unique in the InDesign object model. If you have experience with the Microsoft Component Object Model (COM), you might see the resemblance between InDesign "interfaces" and COM "interfaces", as they are analogous.

Normally, you would group together related methods into a set. Interfaces in the InDesign object model comprise a set of such grouped methods, and are denoted as pure abstract C++ classes. By denoting them as pure abstract C++ classes, you can call all methods within a particular interface in a boss class, even from outside the interface itself.

For example, kPageItemBoss is a boss class that represents the base class for all page items that can be placed on a document. This boss class aggregates (contains) an interface called **IHierarchy**. By obtaining this interface and calling its methods, you can obtain information about the object hierarchy of image and text frame items in an InDesign document.

You may have noticed by now that InDesign uses a naming convention such that all interface names begin with a capital "I", so that you can quickly determine interfaces at a glance.

IPMUnknown, the Parent of (almost) All InDesign Interfaces

The base class of almost all InDesign interfaces is **IPMUnknown**. In order for InDesign's object model to function correctly, interfaces must inherit from **IPMUnknown**, and support the **QueryInterface()**, **AddRef()**, and **Release()** methods. You can query a boss for an interface pointer of type **IPMUnknown**, and get back a valid interface pointer.

Querying for Interfaces, and Refcounts

QueryInterface() is used to query for an interface on a boss. This function returns a pointer to an interface, or nil if an instance of the interface is not available. **QueryInterface()** automatically performs an **AddRef()**, which increments the **refcount** on the interface. The object model keeps track of the **refcount** (reference count) for interfaces on bosses. If all of the interfaces on a boss have a refcount of zero, the boss can be marked for deletion. If you have used **QueryInterface()** to obtain an interface pointer, it is necessary to call the **Release()** method when you are through with the interface, so that the refcount for the interface is decremented correctly. Forgetting to call **Release()** will result in an interface with a positive refcount. This condition is known as a "boss leak", which is a memory leak in the InDesign object model.

What is InterfacePtr?

InterfacePtr (**{SDK}\API\Includes\InterfacePtr.h**) is a wrapper class that wraps **QueryInterface()**. In addition to the **AddRef()** that is automatically performed by **QueryInterface()**, this template-based wrapper class also performs a **Release()** when the pointer goes out of scope. This ensures that **Release()** is called on an interface, and prevents boss leaks.

Here is a sample of how you would instantiate an interface pointer to an **ISpreadList** from **IDocument**, using **InterfacePtr**:

```
InterfacePtr<ISpreadList> iSpreadList(iDocument, UseDefaultIID());
```

Which Variety of InterfacePtr Constructor Should I Use for What Situation?

There are as many as 10 different **InterfacePtr** constructors, so it may seem a bit overwhelming, however, there are 3 major types of constructors that are commonly used.

Type 1a: When you want to obtain an interface in the same boss class (using default PMIID)

```
InterfacePtr::InterfacePtr(const IPMUnknown* p, const UseDefaultIID&)
```

This assumes that you already have an **InterfacePtr** of some kind, or a pointer to an object derived from the **IPMUnknown**. You use this when you want to obtain an interface aggregated on the same boss class. If the interface declaration defines an enum **kDefaultIID**, the **UseDefaultIID()** construct will automatically grab the default **PMIID** (interface ID). In this case, the newly created **InterfacePtr** has its reference count incremented by means of the **IPMUnknown::AddRef()** method.

Example: See **IDocument**, **ISpreadList**

```
IDocument* doc = ::GetFrontDocument();
InterfacePtr<ISpreadList> iSpreadList(doc, UseDefaultIID());
```

Type 1b: When you want to obtain an interface in the same boss class (specifying PMIID)

```
InterfacePtr::InterfacePtr(const IPMUnknown* p, PMIID iid);
```

This assumes that you already have an **InterfacePtr** of some kind, or a pointer to an object derived from **IPMUnknown**. You use this when you want to obtain an interface aggregated on the same boss class, but the interface declaration does not define an enum **kDefaultIID**. You also use this when there are multiple implementations of the same interface aggregated on the same boss class. In this case, the newly created **InterfacePtr** has its reference count incremented by means of the **IPMUnknown::AddRef()** method. There are situations where you must specify a **PMIID**, such as when you want to obtain an **IStyleNameTable** on **kWorkspaceBoss**/**kDocWorkspaceBoss**. But you can also regard this as a trick to aggregate multiple implementations of the same interface into your boss class.

Example: See **IStyleNameTable**, **TextID.h**.

```
// docWorkspace is an IWorkspace aggregated on kDocBoss.
InterfacePtr<IStyleNameTable> iParaStyleTable(docWorkspace, IID_IPARASTYLENAMETABLE);
InterfacePtr<IStyleNameTable> iCharStyleTable(docWorkspace, IID_ICHARSTYLENAMETABLE);
```

Type 2: When you get a specific interface, not IPMUnknown*, from a bridge method

```
explicit InterfacePtr::InterfacePtr(IFace* p);
```

Generally, **Query***()** methods (commonly known as "bridge methods", see column below), return a pointer to an interface derived from **IPMUnknown**, and also increments the reference count. However, you would still like to take advantage of **InterfacePtr**'s automated cleanup. To prevent reference counts from incrementing, like in Type 1a/b above, you can use this constructor that does not call **IPMUnknown::AddRef()**.

Example: See ISpread, IGeometry.

```
InterfacePtr<IGeometry> iPageGeometry(iSpread->QueryNthPage(0));
```

NOTE: This looks like perfectly innocent code, however, if you execute this and shut down InDesign, you will get a boss leak.

```
InterfacePtr<IGeometry> iPageGeometry(iSpread->QueryNthPage(0), UseDefaultIID());
```

If you look carefully, **ISpread->QueryNthPage(0)** increments the reference count, and also, by means of **InterfacePtr** constructor Type 1a, the reference count increments again.

The easiest remedy is to remove **UseDefaultIID()**, however, if you leave it as is, and fail to notice that a call to **iPageGeometry->Release()** is necessary, you will get a boss leak.

Type 3a: When you want to obtain a persistent object on a database using a UIDRef

```
InterfacePtr::InterfacePtr(const UIDRef& ref, PMIID iid);
// Usable when kDefaultIID is defined
InterfacePtr::InterfacePtr(const UIDRef& ref, const UseDefaultIID&);
```

In this case, you will use a preexisting **UIDRef** on a boss class. A **UIDRef** is a combination of the database that is the target of persistence, and a unique ID (**UID**) of a boss class object. This constructor is useful after obtaining a **UIDList** from a command or a selection target.

Example 1: After processing **NewFrameCmd**, obtain the frame's **IHierarchy** . See **IHierarchy, UIDList**.

```
InterfacePtr<IHierarchy> newPageItemHierarchy((newFrameCmd->GetItemListReference()).G
etRef(0), UseDefaultIID());
```

Example 2: Code to obtain the first layer that actually contains page items. See **IDocument**, **IHierarchy**, **ISpreadLayer**.

```
IDocument* iDocument = ::GetFrontDocument();
UIDRef layerRef(::GetDataBase(iDocument), iSpreadHier->GetChildUID(2));
InterfacePtr<ISpreadLayer> spreadLayer(layerRef, UseDefaultIID());
```

Type 3b: When you want to obtain a persistent object on a database using a UID

```
InterfacePtr::InterfacePtr(IDataBase *db, UID uid, PMIID iid);
// Usable when kDefaultIID is defined
InterfacePtr::InterfacePtr(IDataBase *db, UID uid, const UseDefaulIID&);
```

This is similar to Type 3a, but is useful when you don't need to create another **UIDRef**, specifically when you are getting interfaces on the same database. This is commonly used when you navigate the page item parent/child relationship.

Example 1: Code to obtain the first layer (spread layer index 2) that actually contains page items. See **IDataBase**, **IDocument**, **IHierarchy**, **ISpreadLayer**.

About Databases and Objects

In InDesign, document files are called **databases**. (Let's leave aside the reason why they are called databases.) You can persist boss class objects into databases. In a C++ programming model, C++ classes are generally declared with a "class" keyword, and are instantiated in memory (e.g. heap). By "serializing" the data in the instantiated object, we can store the data in a complex class structure onto a file, and retrieve the same data from the file.

In order to store boss class objects into a database, a unique identifier known as a **UID** is assigned to each boss class object. **UID**s, which are type-defined as 32-bit integers, are treated somewhat like a pointer in the InDesign object model. For example, a document boss (**kDocBoss**) aggregates **ISpreadList**, an interface that owns the **UID**s of all spreads within a document, and within each spread, you can obtain page item

```
IDocument* iDocument = ::GetFrontDocument();
IDataBase* iDataBase = ::GetDataBase(iDocument);
InterfacePtr<ISpreadLayer> spreadLayer(iDataBase, iSpreadHier->GetChildUID(2),
UseDefaultIID());
```

Example 2: Code to navigate up from **kFrameItemBoss**(**ITextFrame**), **kMultiColumnItemBoss**, and to **kSplineItem**. See **IDataBase**, **IHierarchy**.

```
InterfacePtr<IHierarchy> frameItemHierarchy(iTextFrame, UseDefaultIID());
InterfacePtr<IHierarchy> mcitemHierarchy
    (iDataBase, frameItemHierarchy->GetParentUID(), UseDefaultIID());
InterfacePtr<IHierarchy> splineItemHierarchy
    (iDataBase, mcitemHierarchy->GetParentUID(), UseDefaultIID());
```

objects and pages by means of the **UID**s obtained from **IHierarchy** (a bridge interface for the object tree in a document). Furthermore, these **UID**s are persistent across InDesign application sessions, so even after quitting and restarting InDesign, the UIDs stored in your documents continue to be valid.
However, in order to call methods on interfaces aggregated on these boss class objects in a C++ program, the actual objects must be instantiated in memory as C++ objects. You can obtain pointers to these interfaces using the various **Get...()** and **Query...()** methods. These are called **bridge methods**. The general rule of thumb with bridge methods is that that **Get...()** methods do not increment **refcount**, while **Query...()** methods do increment **refcount**.

Step 7: Insert the string into a text frame

This is where we get into the main part of our plug-in. We will insert the string that we created in the previous step into a text frame in InDesign.

Step 7.1: Check if there is a TextFocus

The first thing we have to do is to check if the focus is on a text frame and whether we could insert text into it. Add these include statements at the top of **WFPDialogController.cpp**:

```
#include "ITextFocus.h"
#include "SelectUtils.h"
```

Then, add this code right after the last **resultString.Append(...)** statement in the **WFPDialogController::ApplyFields()** method:

```
do {
   // Insert resultString to TextFrame.
   // Check Text focus.
   InterfacePtr<ITextFocus> pFocus( QueryTextFocus() );
   if( pFocus == nil )
   {
          ASSERT_FAIL("WFPDialogController::ApplyFields: ITextFocus is nil!");
          break;
   }
```

We are using a selection utility method called **QueryTextFocus()**, defined in **{SDK}\API\Includes\SelectUtils.h**. As long as there is a text focus, this method will return a valid pointer to a **ITextFocus** interface. If there is no text focus, this returns nil.

Also, we are using a **do-while** loop is to avoid deep nesting of **if (...)** statements for checking the validity of the **InterfacePtr**'s. If an **InterfacePtr** turns out to be nil, we want to abort our operation by showing an assertion failure message (appears only in debug build) and breaking out of the **do-while** loop. (If you have nested loops, remember to use extra precaution.)

Step 7.2: Query the TextModel from TextFocus

Now, if the text focus is valid, we will use the **ITextFocus::QueryModel()** method to obtain a pointer to the **ITextModel** interface, which describes the "model" of the text frame. Add the following include statement at the top of **WFPDialogController.cpp**:

```
#include "ITextModel.h"
```

Then, add this code right after the line with **if (pFocus)** in the **WFPDialogController::ApplyFields( )** method:

```
   // Obtain TextModel from TextFocus.
   InterfacePtr<ITextModel> pTextModel( pFocus->QueryModel() );
   if( pTextModel == nil )
   {
          ASSERT_FAIL("WFPDialogController::ApplyFields: ITextModel is nil!");
          break;
   }
```

Step 7.3: Create a WideString Object from Our PMString

Before we insert some text, there is one more important thing to do. The text insert command (see The Story of Commands for more info about commands in general) in InDesign takes a **WideString** class. This class is a wrapper for UNICODE strings, the internal text storage format in InDesign.

By specifying a **PMString** object in the constructor of the **WideString** class, we can instantiate a **WideString** object that contains a converted UNICODE string. Insert this code on the next line:

```
// Create WideString from PMString
WideString* resultWideString = new WideString(resultString);
```

Now, we are ready to insert the actual string.

Step 7.4: Process the Insert Text Command

Next, we will process an InDesign command . Add the following include statements at the top of **WFPDialogController.cpp**:

```
#include "ICommand.h"
#include "CmdUtils.h"
```

Then, add this code right after the line where we instantiated the **WideString** object in the **WFPDialogController::ApplyFields()** method:

```
    // Process InsertTextCommand
    InterfacePtr<ICommand> pInsertTextCommand
            (pTextModel->InsertCmd( pFocus->GetStart(nil), resultWideString, kFalse ));
    if ( pInsertTextCommand == nil )
    {
            ASSERT_FAIL("WFPDialogController::ApplyFields: InsertTextCommand is nil!");
            break;
    }
    if ( CmdUtils::ProcessCommand(pInsertTextCommand) != kSuccess )
    {
            ASSERT_FAIL("WFPDialogController::ApplyFields: can't process InsertCmd");
    }
} while (kFalse);
```

In the above code, we created an interface pointer to the text insert command by using the **ITextModel::InsertCmd()** method. This method takes the insertion point, the UNICODE string to insert, and a boolean indicating whether we want to have it create an internal deep copy of the UNICODE string. We set this boolean to kFalse, since we have instantiated a new **WideString** earlier. (Note the lack of the delete operator.) To obtain the insertion point, we call the **ITextFocus::GetStart()** method. If there is a range of text that is selected in the text frame, this method will return the offset to the first character in the selection relative to the text in the model, and if the text cursor is shown, it will return the offset of the text cursor.

If the **ICommand** interface pointer is valid, we then call the **CmdUtils::ProcessCommand()** method to instruct InDesign to process the text insert command.

Bonus topic: Inserting Text Without Using Commands (New in InDesign 2.0)

In InDesign 2.0, you can use the newly introduced **ISelectionUtils** interface (aggregated on **kUtilsBoss**) and **ITextEditSuite** interface (aggregated on **kIntegratorISuiteBoss**) and insert text into a text focus with less lines of code, as shown here. (This would replace the above code starting from **InterfacePtr<ITextFocus> pFocus(QueryTextFocus())** , Steps 7.1-7.4**)**:

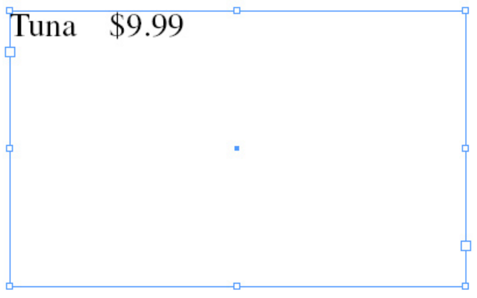
```
InterfacePtr<ITextEditSuite> textEditSuite
   ((ITextEditSuite*)Utils<ISelectionUtils>()->QuerySuite
      (ITextEditSuite::kDefaultIID));
if (textEditSuite->CanEditText())
{
    ErrorCode status = textEditSuite->InsertText(WideString(resultString));
    ASSERT_MSG(status == kSuccess,
       "WFPDialogController::ApplyFields: can't insert text");
}
```

The code we added in our exercise utilizes existing API calls from InDesign 1.x. Inside the **ITextEditSuite** interface, text is inserted in a very similar fashion by processing a command.

We wanted to introduce you to the concept of processing commands to InDesign, so that is why we opted for the classical method of inserting text. There are situations where you cannot use these selection-based suites: namely, when there is no active selection. If you want to be able to add text into the text model of an existing document outside of the context of a current user selection, you must use the text insert command, as we did in our example.

Step 7.5: Save, Build and Test

Once you have made the changes, save all of your edited source files, build the plug-in, move the plug-in to the Plug-Ins folder if needed, and run it. First, create a text frame on a new document, and make sure the cursor is blinking. Then select the **Plug-in** menu so that your dialog shows up, select a fish type and enter its price. Then click **OK**.



Does your text appear in the text frame? If your plug-in is working, let's move on.

Step 8: Enabling the menu only when there is a text focus or selection

Since we don't want this dialog to be opened when there is no text selection or text focus, we will make some changes so that the menu is disabled under those circumstances.

Step 8.1: Modify ActionDef to Disable Menu if Required Selection is not Available

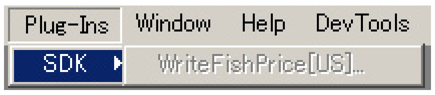
Open up the resource file, **WFP.fr**, and go down to the **ActionDef** resource, and down to the second block that which begins with **kWFPActionComponentBoss**. Change **kDisableIfLowMem** to **kDisableIfSelectionDoesNotSupportIID**, and on the next line, change **kInvalidInterfaceID** to **IID_NEED_TEXTSELECTION**. These constants are defined in **ActionDefs.h** and **ActionID.h**. There are plenty of other related constants, so take a look at **ActionDefs.h** in **{SDK}\API\Includes**.

```
resource ActionDef (kSDKDefActionResourceID)
{
   {
          kWFPActionComponentBoss, // ClassID of boss class that implements
ActionID.
          kWFPAboutActionID,     // ActionID.
          kWFPAboutMenuKey,      // Sub-menu string.
          kOtherActionArea,      // Area name (see ActionDefs.h).
          kNormalAction,         // Type of action (see ActionDefs.h).
          kDisableIfLowMem,      // Enabling type (see ActionDefs.h).
          kInvalidInterfaceID,   // Selection InterfaceID this action cares about...
          kSDKDefVisibleInKBSCEditorFlag,       // kSDKDefVisibleInKBSCEditorFlag...

          kWFPActionComponentBoss,
          kWFPDialogActionID,
          kWFPDialogMenuItemKey,
          kOtherActionArea,
          kNormalAction,
          kDisableIfSelectionDoesNotSupportIID,        // change this!
          IID_NEED_TEXTSELECTION,                      // change this!
          kSDKDefVisibleInKBSCEditorFlag,
   }
};
```

Step 8.2: Save, Build and Test

Once you have made the changes, save all of your edited source files, build the plug-in, move the plug-in to the **Plug-Ins** folder if needed, and run it. Is the menu enabled when there is no text selection? Now create a new document, put a new text frame on it, and see if the menu is enabled.



If the menu is working as specified, let's move on.

The Story About Commands

InDesign uses a construct called a Command to modify internal data. By using commands, we gain the following 3 benefits.

(1) We don't have to modify the internal data directly, or the internal data can stay encapsulated. This internal data is referred to as the "model".

(2) Commands facilitate actions such as Undo and Redo.

(3) Commands allow you to be notified about details about changes to the model.

Besides, you could create your custom commands. If you create your custom commands, you can separate the user interface and core feature implementation components, making for a more extensible design.

To fine out more about processing commands or creating custom commands, you can refer to the sample plug-ins in the **{SDK}\SampleCode\Commands** folder. Please have a look.

A Small Story About MVC

By now you must have noticed that it is possible to de-couple the user interface components and the core components that process user input in your plug-ins. In the object-oriented world, we call the user interaction component a "**controller**".

In this exercise, we were able to insert text into an InDesign text frame by processing a text insert command from the dialog controller.

When text is inserted, or in other words, when InDesign's internal data (the **model**) is changed, In-Design draws the text inside the text frame.

In the object-oriented world, the space that InDesign uses to draw the visible regions of the docu-ment including the text frame, is called a "**view**".

Collectively, such a programming model comprises what is known as the **Model-View-Controller** set of patterns, or more commonly known by the acronym, by taking the initials, the "**MVC**" pattern.

There is an excellent book we generally recommend to all developers of InDesign plug-ins. This book, ***Design Patterns***, describes in detail what an MVC pattern is and provides some concrete ex-amples to deepen you understanding.

Design Patterns: Elements of Reusable Object-Oriented Software

(Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John)

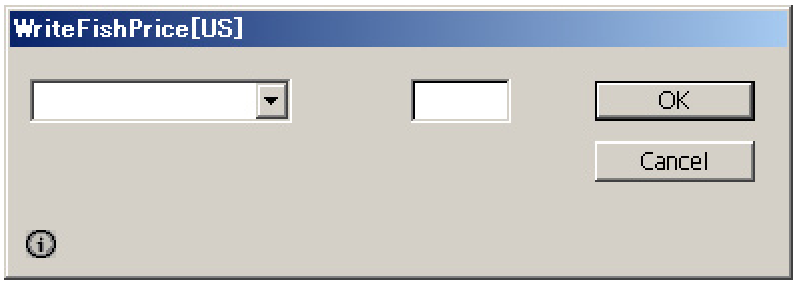
ISBN:0-201-63361-2 Addison Wesley

There are other numerous patterns included in this book that are incorporated into the InDesign architecture.

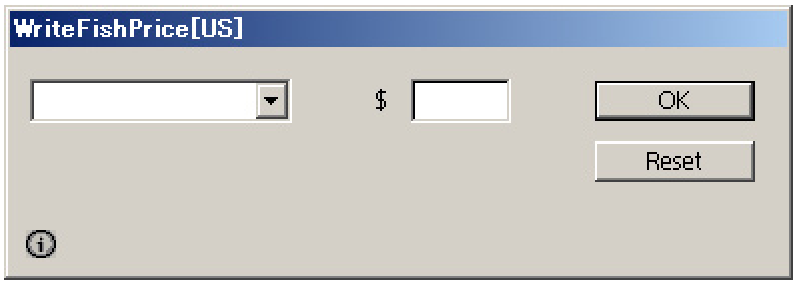
Step 9: Initialize the dialog widgets

We are close to completing our first plug-in that inserts the name of the selected fish and the price entered on the TextEditBox.

However, as shown in the dialog below, the DropDownListWidget shows nothing and the TextEditBoxWidget is blank when we open the dialog. Not quite the user-friendly UI that we want.



Also, there's a hidden feature in this dialog. While the dialog is displayed, try holding down the Option key (Macintosh) or the Alt key (Windows). Did you notice a change in the dialog? Pay close attention to the buttons: The string on the **Cancel** button changed to **Reset**. Dialogs in InDesign have the capability to reset fields to an initial state.



By adding functionality to the **WFPDialogController::InitializeFields()** method, you can handle the initialization and resetting of dialog fields.

So let's add some functionality to the **InitializeFields()** method.

Step 9.1: Add Code to Initialize the DropDownListWidget

Open **WFPDialogController.cpp**.

We will add some code to the **WFPDialogController::InitializeFields()** method. This method first delegates to the same method in the parent class, **CDialogController::InitializeFields()**. The parent class method sets a flag that keeps track of whether this **InitializeFields()** method was called, so make sure you call **CDialogController::InitializeFields()**.

```
/* InitializeFields
*/
void WFPDialogController::InitializeFields()
{
   CDialogController::InitializeFields();

   // Put code to initialize widget values here.
}
```

We will add some code to initialize the DropDownListWidget.

By calling the **CDialogController::QueryIfNilElseAddRef()** method, we will obtain a pointer to the **IPanelControlData** interface. This method takes an **IPanelControlData** interface pointer as a parameter, but if that pointer is **nil**, it returns the **IPanelControlData** interface of the same boss class, and if the pointer is not **nil**, it increments the **refCount** to that pointer and returns it. Add the following include file at the top of **WFPDialogController.cpp**:

```
#include "IPanelControlData.h"
```

Next, insert the following code right after where **CDialogController::InitializeFields()** is called.

```
do {
   // Get current panel control data.
   InterfacePtr<IPanelControlData> pPanelData( QueryIfNilElseAddRef(nil) );
   if( pPanelData == nil )
   {
      ASSERT_FAIL("WFPDialogController::InitializeFields: PanelControlData is nil!");
      break;
   }
```

If the **pPanelData** interface pointer is valid, we will call the **IPanelControlData::FindWidget()** in the **IPanelControlData** interface to obtain an **IControlView** interface pointer. This method takes a widget ID as a parameter, and returns the corresponding **IControlView** interface pointer. (Note: **IControlView** is the interface for **CControlView**. Do you remember seeing **CControlView** somewhere before? That's right, we saw it while climbing up the hierarchy of widget type definitions!) Insert the following code:

```
if( pPanelData != nil){
   // Find dropdown list menu control view from panel data.
   IControlView* pDropDownListControlView =
          pPanelData->FindWidget( kWFPDropDownListWidgetID );
   if( pDropDownListControlView == nil )
   {
      // Is the widget on the dialog?
      ASSERT_FAIL("WFPDialogController::InitializeFields: DDListCtrlView is nil!");
      break;
   }
```

Using the obtained **IControlView** interface pointer, we will obtain the **IDropDownListController** interface pointer, which exists in the same boss class, **kDropDownListWidgetBoss**. The **kDropDownListWidgetBoss** is responsible for controlling the DropDownListWidget (as defined in the resource definitions). However, to use the **IDropDownListController** interface, we must add the following include statement at the top of **WFPDialogController.cpp**:

```
#include "IDropDownListController.h"
```

Then add the following code right after the call to **pPanelData->FindWidget(...)**:

```
   // Get IDropDownListController interface pointer.
   InterfacePtr<IDropDownListController> pDropDownListController
           ( pDropDownListControlView, UseDefaultIID() );
   if( pDropDownListController == nil )
   {
      // Is the controller available?
      ASSERT_FAIL("WFPDialogController::InitializeFields: DDListCtrler is nil!");
      break;
   }
```

If the **pDropDownListController** interface pointer is valid, then we call the **IDropDownListController::Select()** method to set the initial state of the DropDownListWidget to show the first element. If nothing is selected in the DropDownListWidget, the **IDropDownListController::GetSelected()** method returns -1, which is an invalid index. We must keep in mind that the top of the list is index '0'. Add the following code right where we left off:

```
   // Select the element at the given position in the list.
   pDropDownListController->Select( 0 );
```

Step 9.2: Add Code to Initialize the TextEditBoxWidget

Next, we will initialize the TextEditBox. We will create an initial string using a **PMString** initialized with a **NULL** string. Then, we will set the value of the TextEditBoxWidget to the initial string by calling the **SetTextControlData()** method. Add the following code:

```
   // Initialize TextEditBox.
   PMString InitialString( "" );
   SetTextControlData( kWFPTextEditBoxWidgetID, InitialString );
} while (kFalse);
```

Step 9.3: Save, Build and Test

Ok. Let's build our plug-in, move it to the **Plug-Ins** folder if needed, and start InDesign so we can try out our plug-in. When the dialog is opened, do you see the first list element automatically shown in the DropDownListWidget? Try selecting something else in the DropDownListWidget.

Then hold down the **Option** or **Alt** key and click the **Reset** button on the dialog. Does the DropDownListWidget reset to its initial state?

If every thing is working, give yourself a pat on the back, because our plug-in is done! Congratulations!

In Conclusion

In this cookbook, we aimed to help you become familiar with developing plug-in based solutions for InDesign. This only amounts to having taken the first step in the vast world of InDesign, however, we went over many of the important fundamental aspects. If you study the code and header files used in our completed plug-in line by line, you will perhaps find more hidden functionality. The Adobe InDesign SDK contains an enormous amount of information to help you develop plug-in based solutions for InDesign. At first, you may be overwhelmed by the amount. If you try to attack of that information right from the start, you may be discouraged. Instead, we recommend that you start by building some of the provided sample plug-in projects and using them with the debug build of InDesign. We hope you will find the sample plug-ins useful.

Well then, we wish you lots of luck and enjoyment in your InDesign plug-in development efforts, as we conclude the first cookbook. We hope to see you in the next edition!