Adobe Developers Association

ADA news

The PageMaker Class Library

I love paper towels. Whether to dry your hands, mop up a spill, or wipe your windshield, they are ready at a moment's notice. Use once, then throw away; paper towels are a modern convenience most people use without thinking. While not free, the cost of an individual paper towel is so small we rarely think about that either.

Which brings us to the PageMaker® Class Library (PCL). In many ways, PCL is the software equivalent of paper towels.

PCL is a rich C++ framework to create plug-in modules (formerly known as Additions) for Adobe PageMaker 6.0. While they can't mop up spilled coffee, PCL objects are lightweight, easy to use, and inexpensive. Like paper towels, they are designed to do a single job, then be thrown away.

Design Goals

PCL was designed to meet four primary design goals:

- Define a simple, natural syntax to express PageMaker commands and queries in C++.
- Insulate programmers from the mechanics of issuing commands and queries, and manage the memory associated with them.
- Balance performance and memory considerations.
- Provide robust type checking and error handling.

In this article, we'll look at how PCL achieves each of these goals.

PageMaker Plug–ins

If you are unfamiliar with PageMaker plug-ins, this section provides a brief overview of the main concepts. It also introduces several of the low-level PCL classes.

Plug-ins are small programs that work with the PageMaker application. Most plug-ins appear as menu items under the "Utilities–PageMaker Plug-ins" menu. When a user invokes a plug-in, the application loads its code and calls it with a small parameter block. An opcode in the parameter block indicates the action PageMaker is requesting the plug-in to perform. These actions include Load, Invoke, Unload, Cleanup, and Shutdown. For most plug-ins, the Invoke request is where the bulk of its work is done. Further details of these action requests are available in the PageMaker SDK documentation.

When a plug-in has been called, it can interact with the PageMaker application through two types of callbacks: *commands* and *queries*. The mechanics of both callbacks are the same, but their functions are different.

Commands cause actions in the PageMaker program. Most actions available to the user through the application's menus, floating palettes, and dialogs can be done programmatically through plug-in commands. For example, you can create a new document, add text and graphics to it, and save it automatically without user intervention.

continued on page 2

How To Reach Us

DEVELOPERS ASSOCIATION HOTLINE:

U.S. and Canada:
(415) 961-4111
M–F, 8 a.m.–5 p.m., PDT.
If all engineers are unavailable, please leave a detailed message with your developer number, name, and telephone number, and we will get back to you within 24 hours.

Europe:
+31-20-6511-355

FAX:

U.S. and Canada:
(415) 967-9231
Attention:
Adobe Developers Association

Europe:
+31-20-6511-313
Attention:
Adobe Developers Association

EMAIL:

U.S.
devsup-person@mv.us.adobe.com

Europe:
eurosupport@adobe.com

MAIL:

U.S. and Canada:
Adobe Developers Association
Adobe Systems Incorporated
1585 Charleston Road
P.O. Box 7900
Mt. View, CA 94039-7900

Europe:
Adobe Developers Association
Europlaza
Hoogoorddreef 54a
1101 BE Amsterdam Z.O.
The Netherlands

Send all inquiries, letters and address changes to the appropriate address above.

The PageMaker Class Library

Queries extract information from the PageMaker application. Nearly any information that the user can see either directly in a publication or displayed in dialog boxes can be retrieved programmatically by plug-in queries. Information may be textual (character strings), numeric, or boolean. Some queries return lists of these types.

Given the broad range of information that can be retrieved or changed (the PageMaker plug-in API defines more than 200 commands and queries), it is no surprise that commands and queries take many forms. Some require parameters, others do not. Some deal with simple data, like a single number or character string. Others deal with complex data structures, including variable length lists of information.

In all cases, the underlying action of a command or query is a callback into the PageMaker application. Thus the starting point for PCL is the PCallback, PCommand and PQuery classes (see figure 1). These low-level classes manage the process of making commands and queries, and are the building blocks for the high-level classes described in the next section.

The only public interface to PCommand and PQuery classes is their constructors; the only thing you can do to a PCommand or PQuery object is create it. Each class has multiple constructors that handle the variety of cases each must handle.

In most cases, you won't create PCommand and PQuery objects directly, but will create high-level objects to perform specific commands and queries.

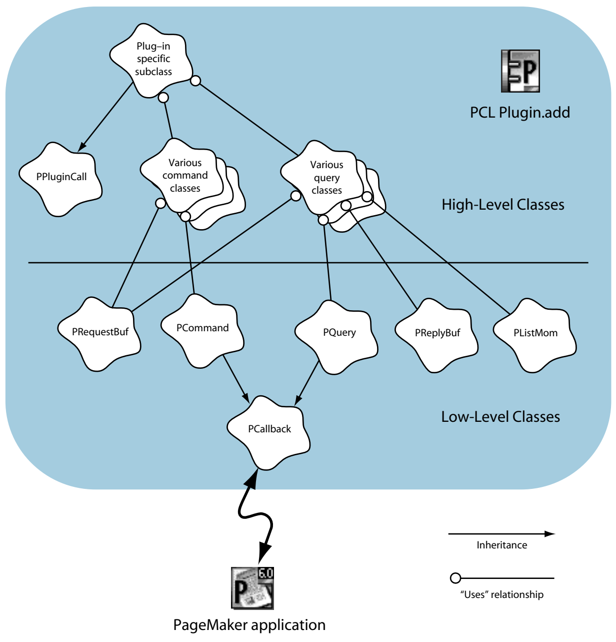

Figure 1. *The PCL Hierarchy*

The PageMaker Class Library

The High-Level Classes

The low-level classes PCommand and PQuery provide direct access to the callback mechanism described in the previous section. However, these classes are not ideal for writing plug-ins directly because of the wide variety of command and query parameters and returned values. For this reason, high-level command and query classes provide a more natural and consistent syntax for writing plug-ins. Moreover, they ensure that commands and queries are properly formed, and that the memory associated with callbacks is correctly managed to avoid potential memory leaks.

The concept behind the high-level classes is very simple. For each PageMaker command or query, there is a corresponding high-level PCL class. When you create an instance of one of these classes, the constructor for the class issues the appropriate command or query to the application.

Here's an example:

```
PGetPageNumber curPage;       // Get the current page number
if ((short) curPage == 27) {  // Check the page number value
    PPage newPage(33);        // Change to page 33
    ...
}
PPage oldPage(curPage);       // Change back to current page
```

In this example, we have created a query object, curPage, and two command objects, newPage and oldPage. These objects are created by declaring them with any appropriate parameters. (The PGetPageNumber query object has no parameters, the PPage command objects each require a single numeric parameter.)

This simple syntax closely mirrors the PageMaker scripting language. Here's the command to set the page number options of a publication:

```
pagenumbers 1, -2, true, arabic, "I-"   // PageMaker scripting
PPageNumbers pn(1, -2, true, kNumArabic, "I-"); // C++ & PCL
```

The C++ code is more efficient than the scripting code, however, because no interpreter step is needed; all PCL commands are directly processed by the application. Moreover, the C++ programmer has access to all the language's features and other capabilities that are unavailable in the scripting language.

Overloaded Operators

Notice that the PGetPageNumber class overloads operator short( ), permitting the natural syntax in the "`if (curPage == 27)`" expression shown in our first example. Many PCL classes use operator overloading in this fashion to achieve the first design goal.

continued on page 4

The PageMaker Class Library

Let’s look at a slightly more complicated example:

```
PDeselect deselect;  // deselect anything that is selected

// loop through each chosen page
for (short i = firstPage; i <= lastPage; i++)
{
  PPage thePage(i);              // change to page i
  PGetObjectList objectList;     // get list of objects on page i
  short n = objectList.Count();  // n is number of objects
  for (short j = 0; j < n; j++)  // examine each object
  {
    if ( objectList.cTypeOfObject == 4  // if circle or rect
      || objectList.cTypeOfObject == 5 )
    {
       PSelect select(objectList.nDrawingNumber); // select obj
       PColor color("mauve");  //set color of selected object
       PDeselect deselect;
    }
    objectList++;  // increment list
  }
}
```

In this example, we have created a number of PCL objects. The objectList object contains a variable amount of information about the objects on the current page. (In this case, we’re referring to PageMaker objects like text blocks, placed graphics, etc., rather than C++ objects.) We can iterate over this list using postfix increment notation (`operator++`) to examine each object in the list.

Memory Management

All the PCL objects created in this code example will be automatically destroyed when they go out of scope; this is guaranteed by the C++ language.

The (C++) object, objectList, may contain information about one, a few, or many dozens of (PageMaker) objects. The PGetObjectList destructor frees the memory block associated with this information, which was allocated by the application. Moreover, if an error occurs, any dangling memory returned by the PageMaker application will be automatically cleaned up by the exception mechanism used by the low-level classes.

These PCL memory management features address the second primary design goal: insulate the programmer from the mechanics of issuing commands and queries, and manage the memory associated with them.

Performance and Memory

Command and query objects are very lightweight. By design, most classes have no virtual functions, which eliminates the need for vtables for those objects and makes the construction and destruction of those objects very efficient. The member data portion of each class is small, ranging from zero to about 40–50 bytes for complex queries.

Command and query objects are designed to be created on the stack, as inline objects in a C++ program. In this way, destruction of these objects (and any associated memory blocks) is handled automatically by the C++ language.

Type Checking

The design of the constructors for high-level classes enforces proper type checking of parameters. For example, the PSelect class has two constructors that implement the two methods of selecting an object: by x–y position or by drawing order number.

```
class PSelect
{
public:
    PSelect(long x, long y);
    PSelect(long drawOrder);
private:
    void DoSelect(short nSelect, long x, long y);
    PSelect();
};
```

Notice that the default constructor, PSelect( ), is declared private so that the programmer cannot accidently create a PSelect object without specifying either position or drawing order.

Error Handling

PCL uses the C++ exception handling mechanism for handling error conditions. Errors in a plug-in can take many forms, the most common one is an non-zero result code returned from PageMaker by a callback.

If a callback into the application returns a non-zero result, the PCallback object will throw an exception. By default, all exceptions will be caught at the highest level in the main( ) routine for the plug-in. The error code will then be returned to the application, which will display an error alert window to the user.

You should always program with the assumption that *any command or query object may throw an exception when it is created*. If you don't want control to immediately return to the application, you must handle the error in your code:

continued on page 6

The PageMaker Class Library

```
try
{
   PGetPrivateData privData(…);
}
catch (PMErr err)
{
   if (err == CQ_NOPDATA)             // there was no private data.
     this->BuildErrorMessages("You don't have any private data",
                              "You idiot …");
    // now rethrow error, or continue elsewhere
}
```

As this example shows, you can use the PPluginCall::BuildErrorMessages( ) function to build custom error messages. For errors that are returned from the PageMaker application, the standard error messages will often be adequate, but you may want to provide more information to the user so they can take corrective action, or assist you in troubleshooting their problem.

Other PCL classes

The remaining low-level classes shown in figure 1 (PRequestBuf, PReplyBuf, and PListMom) support the high-level command and query classes. They manage the process of dealing with query data or packing parameters for commands and queries.

The high-level PPluginCall class provides a framework for the overall plug-in. By defining a subclass of PPluginCall, you can quickly build a PageMaker plug-in. To implement your plug-in's functionality, you simply override one or more PPluginCall functions. The example code and project stationery files included with the SDK illustrate this.

Summary

The PageMaker Class Library makes creating PageMaker plug-ins a snap. To PageMaker plug-in developers, it represents a paradigm shift—the work of issuing commands and queries shifts from the developer to the class library. The high-level syntax makes issuing commands and queries to PageMaker as easy as using the built-in scripting language. PCL bypasses the scripting language's text interpreter, however, resulting in much greater performance. Moreover, PCL affords complete access to the C++ language, allowing you to create very powerful plug-ins easily and quickly.

The PageMaker SDK is available on our web site at www.adobe.com in the Support and Training section. §

Colophon

This newsletter was produced entirely with Adobe PostScript® software on Macintosh® and IBM® PC compatible computers. Typefaces used are from the Minion® and Myriad® families from the Adobe Type Library.

Managing Editor:
Jennifer Cohan

Technical Editor:
Nicole Frees

Art Director/Designer:
Karla Wong

Contributors:
Tim Bienz, Paul Ferguson

Part Number ADA0063 2/96

Adobe PostScript

ACROBAT column™

This month we'll discuss how to create and use custom search fields in PDF files. Such fields are useful in situations where it's important to be able to search PDF files for information other than the document's text, title, subject, author, keywords, creation date, or other built-in fields. For example, an insurance company may wish to locate a document by its policy number. Although the ability to add and use custom search fields is not available through the Acrobat® products' user interfaces, it is available to developers via the products' APIs. There are three steps involved in the process—adding custom search fields to PDF files, adding the custom fields to Catalog's list of known fields and then indexing the files using Catalog, and searching them using the Acrobat Search™ plug-in for Acrobat Exchange™. The following paragraphs describe each step in more detail.

Custom search fields can be added to PDF files in three ways. First, they can be added when the file is converted from a PostScript language file to a PDF file using the Acrobat Distiller® application. This is accomplished using the **pdfmark** operator to place values into the PDF file's Info dictionary. For example, to add a field named ADBE_PolicyNum to a PDF file and set its value to 4321, add the following line to the PostScript language file before distilling it:

```
[ /ADBE_PolicyNum (4321) /DOCINFO pdfmark
```

Note that the key name includes Adobe's prefix, ADBE. All private data added to PDF files must use the developer's prefix. Note also that the policy number is a string—the values of all custom search fields must be strings, even if the field contains numeric data. Technical note #5150, "pdfmark Reference Manual," contains a complete description of the **pdfmark** operator. This technical note ships with the Distiller application and is also included in both the Acrobat SDK and the Acrobat Plug-ins SDK. The second way to add custom search fields to a PDF file is via the interapplication communication support in Exchange, using the set info Apple event on the Apple Macintosh or the PDDoc.SetInfo OLE automation method under Microsoft Windows®. The third way is by writing a plug-in for Exchange, using the plug-in API's method to set values in the Info dictionary.

After getting the custom search field information into the PDF files, you need to ensure that Catalog includes the custom fields in the search indices it builds. This is accomplished by editing the [Fields] section of *acrocat.ini* file under Windows or via the Catalog Preferences menu item on the Macintosh. Details are provided in the Catalog help file; see p. 61 of the help file for version 2.1. To add ADBE_PolicyNum under Windows, place the following line in the [Fields] section of *acrocat.ini:*

```
Field0=ADBE_PolicyNum,str
```

Finally, there are two ways to use the Search plug-in to locate custom fields—manually typing a query or writing a custom search dialog. Typing a query manually is straightforward. For example, to search for documents that have an ADBE_PolicyNum of 4321, type:

```
(ADBE_PolicyNum = 4321)
```

into the Search plug-in's "Find Results Containing Text" box. The other alternative is to create a custom search dialog containing fields into which users can enter search queries. After the user fills in the search dialog, you format the search query and pass it to the Acrobat Search plug-in, which performs the search and displays the results. You can create a custom search dialog either as a plug-in for Acrobat Exchange or as part of another application. If you choose to write a plug-in, the information and sample code you'll need is in the Acrobat Plug-ins SDK (see the HFTQuery code sample). If you choose to work from another application, using DDE under Windows or Apple events on the Macintosh, the information and sample code you'll need is in both the Acrobat SDK and the Acrobat Plug-ins SDK (see the VBSRCH

continued on page 8

and AEView code samples). Implementing a custom search dialog is an attractive solution if your users commonly want to search using custom search fields or are not proficient at formatting and typing complex queries. Be aware that even when you implement a custom search dialog, the results of the search appear in the Acrobat Search plug-in's built-in Search Results dialog. There is currently no way to modify the appearance of the results dialog, replace it with your own, or have the Acrobat Search plug-in programmatically return a list of hits to you.

The ability to add and use custom search fields with files opens a wide variety of new options for you and your customers. The Acrobat SDK and the Acrobat Plug-ins SDK contain all the documentation and sample code you'll need to take advantage of this powerful feature of the Acrobat products. Contact the Adobe Developers Association for further information on either SDK. §