Andy Novobilski

PENPOINT™ PROGRAMMING

PenPoint™ Programming

PenPoint™ Programming

ANDY NOVOBILSKI

Addison-Wesley Publishing Company
Reading, Massachusetts Menlo Park, California New York
Don Mills, Ontario Wokingham, England Amsterdam Bonn
Sydney Singapore Tokyo Madrid San Juan Paris Seoul
Milan Mexico City Taipei

Library of Congress Cataloging-in-Publication Data

Novobilski, Andrew J.
PenPoint programming / Andy Novobilski.
p. cm.
Includes index.
ISBN 0-201-60833-2
1. Operating systems (Computers) 2. Penpoint (Computer file)
I. Title.
QA76.76.O63N72 1992
005.4'469—dc20 92-13076
CIP

Sponsoring Editor: Julie Stillman
Project Editors: Elizabeth Rogalin, Claire Horne
Production Coordinator: Vicki Hochstedler
Cover design by Ned Williams
Set in 10-pt Palatino by Electric Ink, Ltd.

1 2 3 4 5 6 7 8 9-MW-965949392
First printing, August 1992

To Mary Ellen and Claire,
who kept me smiling through my most pensive moments.

Lord, help me write software like Paul sewed tents (Acts 18).

Acknowledgments

In April of 1991, I read a small article on the back page of a magazine about a product called PenPoint. The more I read, the more excited I became, and the more I wanted to capture some of that excitement and pass it along to you. The process of sharing my excitement about PenPoint resulted in the book you are now reading. There are many people who helped me in my quest, and I am thankful to all of them. There are several that I would like to acknowledge in a special way.

First, the book is here because two individuals took the time to listen to my ideas and help me formulate them into a book. Thom Hogan from GO Corporation has helped me in just about every way possible, from introducing me to the appropriate people at GO to getting me early releases of software. Then Keith Wollman, Editor-in-Chief at Addison-Wesley, listened patiently to my idea and helped me to refine it.

In addition, I wish to thank Claire Horne, Joan Fitzgerald, Vicki Hochstedler, Ann Lane, Elizabeth Rogalin, Julie Stillman, and all the other people at Addison-Wesley for the time, effort, and experience they brought to the project. They took away every concern I had until I was free to write the best book I possibly could. I also wish to thank Gary Downing of GO Corporation for coordinating the technical edit to ensure that the information being presented to you is timely and up to date.

Finally, I wish to thank my wife Mary Ellen for all her help in completing this project. Her constant encouragement and willingness to help whenever possible proved invaluable.

Contents

Preface xiii

1 What's with the Pen? 1
A History Lesson 1
What is PenPoint? 4
PenPoint Internals 8
Wrap-up 10

2 The Class Manager 13
Why Objects? 14
PenPoint's Definition of OOP 20
The PenPoint Class Hierarchy 21
PenPoint's Class Manager 23
Building PenPoint Classes 31
Wrap-up 41

3 Application Building 43
PenPoint in Action 43
Taxonomy of a PenPoint Application 46
Compile-time Debugging Support 54
Tools for Debugging Applications 60
Wrap-up 64

4 The Application Framework 65
The Pre-framework Era 66
The Document Life Cycle 67
CoinApp 78
Wrap-up 88

5 The Calculator Example 89
Object-oriented Design and PenPoint 90
The Calculator Example 95
Implementing the Calculator Application 99
Wrap-up 112

6 Constructing a User Interface 115
PenPoint Windows 116
The Calculator Button View 124
The Implementation of clsCalcBtVw 128
Wrap-up 151

7 Using the Pen 153
Library Support for Handwriting Recognition 154
clsBoxCalcApp: A Box-based Calculator 159
clsHWXCalc: A Scratch-paper-based Calculator 173
Wrap-up 190

8 A Crossword Puzzle 191
The Crossword Puzzle User's Guide 191
Implementing the Crossword Application 198
PenPoint Classes Used in clsXWordApp and clsXWordData 199
clsXWordApp: The Crossword Application Class 203
clsXWordData: The Crossword Puzzle Model Class 216
Wrap-up 229

9 Coordinating Views 233
User Preference Support 234
Packaging Components for Reuse: DLLs 236
clsXWordView: The Crossword Puzzle View Class 237
clsXWordClueList: The Clue List View Class 256
Wrap-up 270

10 WYSIWYG GUIs 271
The ImagePoint Imaging Model 271
clsXWordGrid: A Direct Manipulation Crossword Grid 273
The Complete method.tbl File 295
Wrap-up 299

Appendix A: Background Reading 301

Appendix B: Source Code for Crossword Application 303
makefile 303
xwordapp.dlc 305
xwordapp.h 306
xwordapp.c 307
xwrddata.h 316
xwrddata.c 318
xwrdview.h 327
xwrdview.c 329
xwrdgrid.h 342
xwrdgrid.c 344
method.tbl 360
xwrdclue.h 363
xwrdclue.c 364
dll.lbc 373
xclu_mth.tbl 374

Index 375

Preface

On April 16, 1992, I sat in the Sheraton Palace Hotel in San Francisco and witnessed the official PenPoint product launch sponsored by GO Corporation. There were over 600 people in the auditorium, and the place was brimming with excitement. A new age was being ushered in with an interaction metaphor that would make computing more accessible to everyone—even those who are keyboard-phobic.

What was unique to the PenPoint launch was the number of software applications available at the same time the operating system was introduced. Usually, there is a time lag between the unveiling of a new programming environment and introduction of applications, due to programmers learning a new way of working. What shortened this cycle for PenPoint was the richness of its object-based development environment. Programmers using PenPoint were able to leverage the existing components provided by GO to perform the mundane tasks of organizing the application, and therefore spend more time on the application-specific parts of the software they were creating.

My goal in writing *PenPoint Programming* is to introduce you to the programmer's perspective of application writing using PenPoint objects, and to help you realize the productivity gains made possible through PenPoint's object model and application framework. I will consider myself successful if you, after reading this book, gain an understanding of why it's important to adhere to GO's guidelines for building PenPoint programs, especially with respect to the application framework.

I leave you with a thought. As a new dad, I often wonder how technology will affect my little girl as she grows up. I hope when she is old enough to read and comprehend what's in this book, her comment will be, "But Dad, isn't all this stuff old hat? Why did everyone make such a fuss over writing on a computer screen? I do it everyday. Besides, it's even easier to work with computers these days because ..."

Good luck, and happy coding.

A.N.
Bethany, CT

1

What's with the Pen?

Since you're reading this, I'm going to be presumptuous and assume you're interested in exploring PenPoint from the perspective of a developer. But before I start talking about programming, I'd like to take a minute and share my thoughts with you on pen-based computing's importance to users, and why PenPoint is important to pen-based computing.

A History Lesson

Back in the early '80s, I remember seeing a wonderful new product from Xerox called the Star. Although it was slow, bulky, and a resource hog, it was the end user's dream come true. After all, this new computer provided the user a mouse to interact with a simulated desktop environment, complete with tear sheets that mimicked ruled note paper. What's more, when the user was stuck trying to figure out how something worked, there was always a handy icon around to click on for help.

It took a few years, but mainstream acceptance of icon-based interfaces eventually came to be. For instance, in 1985 Drexel University mandated that every student entering the university must purchase a personal computer. The computer selected to adorn the dorm of every student was none other than the Macintosh, which had a display metaphor based on the mouse "point and click" Graphical User Interface technology from

Xerox. Icons had finally found their rightful place in the computing universe. It just goes to show that a picture really is worth a thousand words.

Or is it?

Mouse Trap

The GUIs of mouse fame suffer from two problems you'll recognize immediately. First, when's the last time you've seen a legal contract in pictures? Although a picture might be worth a thousand words, exactly which thousand words does it mean? This is a historical lesson, after all; otherwise you would be reading my words in hieroglyphics.

The second problem with mouse-based interfaces is the mouse itself. As you sit at a keyboard typing, you must stop what you're doing, look around to locate the mouse, pick up the mouse, coordinate its on-screen position with its physical location, and finally use it to point and click. If you're like me, just finding the dumb rodent is a royal pain. Of course, there are touch screens, but the resolution a finger provides is not quite fine enough to be useful on a small amount of screen real estate.

And Now, the Pen Makes Its Grand Entrance

The concept of a pen-based user interface is not exactly new, but that doesn't make its encore appearance less than grand. Early researchers in computer graphics at MIT long ago developed, used, and retired one version, the light pen. In those days, the location of the pen was determined by a light burst sent from character to character until the pen finally responded. Again, this method was not refined enough for the task at hand, so the light pen was retired in favor of your friend and mine, the mouse.

What makes today's pen-based systems so different *is* the fine level of granularity available in coordinating the pen's movements with the on-screen display. The current crop of pen-based machines (often referred to as tablets) is powered by 386 chips, with a sufficient amount of RAM to make life without a hard disk palatable. Also, the pen is just that, a pen. It looks, weighs, tastes (ok, so I chew my pens), and feels like many writing implements we use daily. As always, our peers in the hardware side of the house have done their job well.

Choices

When one is designing an operating system for a new class of hardware, there is always a choice. The hardware vendor can port an existing operating system by extending the metaphor as needed to fit the new hardware or build a new operating system that fits the new hardware from the beginning. Currently, the two major providers of operating systems for pen-based machines have chosen opposite sides. On one side, Microsoft has implemented a compatibility layer for the pen and integrated it into their Windows-NT product. On the other side, GO Corporation has chosen the route of building an operating system from the ground up, one tailored to the requirements of small, mobile, pen-based computers. It is GO's operating system, PenPoint, that I'm going to describe for you in this book.

Where the Pens Are

Ok, you've got the new tablet hardware and PenPoint to run applications on it. What applications are you going to develop? In my opinion, the hardware and software involved in pen-based computing are going to excel in three areas: mobility, unobtrusiveness, and ease of use.

Pen-based machines are finding their way into applications such as inventory control and tracking due to several factors. These machines tend to be physically small and easy to handle. They can be carried around and operated while being held in all sorts of positions. With the addition of innovative networking technologies such as infrared and radio nets, they will become very useful in factory settings.

A second class of applications well suited to pen-based computing is that in which a computer is extremely useful, as long as it's unobtrusive. Suppose you are a real estate agent who wants to help a client evaluate a loan. You can communicate with your client, plus enter the information into a computer, without a screen and keyboard between you and your client.

Finally, the replacement of the keyboard by the pen is going to open personal computing to many keyboard-phobic people. The pen's simple user interaction metaphor doesn't require the user to master the art of chicken pecking at typewriter keys. Its simplicity encourages people to try it, even those who have been intimidated by keyboards in the past.

What is PenPoint?

According to Robert Carr, a founding officer of GO, and Dan Shafer in their book *The Power of Penpoint*, Penpoint is

> a new operating system designed and built from the ground up by GO Corporation for the unique requirements of mobile, pen-based computers. It is a 32-bit, object-oriented, multi-tasking operating system that packs the power of workstation-class operating systems into a compact implementation that does not require a disk.

As important as what it is, however, is what it is not. It isn't a product based on someone's gut instincts, without any regard to work already in existence. Someone thought this product out and decided to adopt concepts that truly set PenPoint apart. First, PenPoint provides the user with a consistent metaphor for interacting with applications. Second, PenPoint isolates the system's components (display, handwriting translation, file system, and so on) from each other so they can be swapped out individually without affecting the entire system. Third, PenPoint is totally committed to an object-based development schema for writing applications.

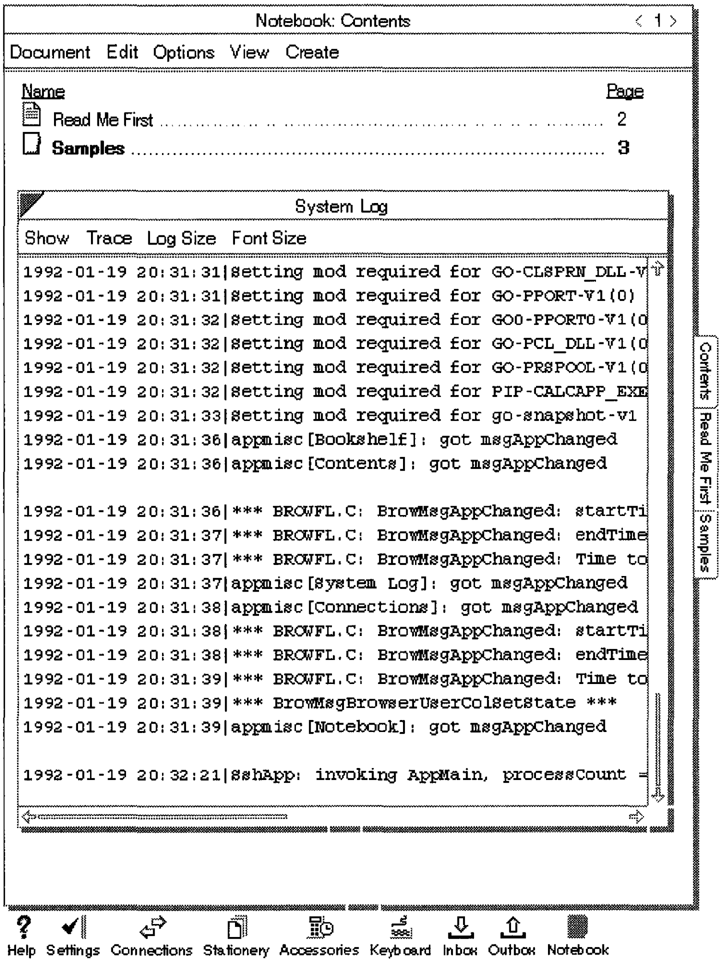
To describe these concepts, GO provides close to 3000 pages of documentation for programmers writing applications for PenPoint. The next several pages are the condensed version of the background information you should keep in mind when considering application development for PenPoint.

The User Interface

When you activate a mobile computer running PenPoint, you are presented with the table of contents from the Notebook. Figure 1.1 shows a sample view of the contents of a pen-based tablet, including the clock, stationary notebook, and the table of contents for the primary notebook. As you can see from the figure, the PenPoint user interface metaphor was implemented using the concepts traditionally associated with a notebook and, therefore, carries the name, **Notebook User Interface**, or **NUI**.

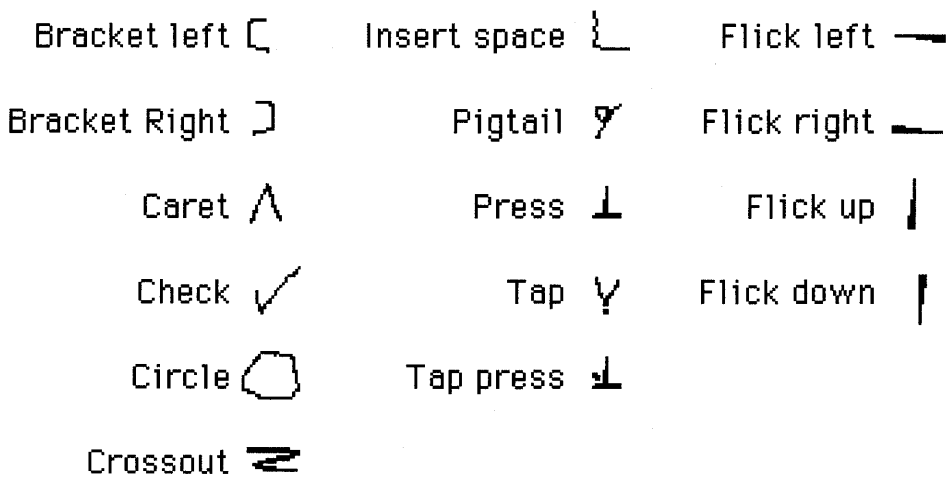
You work with your tablet by using a stylus to interact with the NUI. There are three very important forms of stylus interaction: selections, used to move between documents contained in the NUI; gestures, used to interact with a particular document; and handwriting, used to enter new information into your applications.

Figure 1.1 The PenPoint Notebook User Interface



For example, suppose you are writing a memo to be faxed at a later time. First, you might select an appropriate piece of faxable stationary to work with from the Create menu and add it to your table of contents. Next, you select the document which would cause PenPoint to "turn" to that page, making the new document available for editing. Then, you could use the gestures outlined in Figure 1.2 to indicate where you wish to start editing the document. Finally, you would enter new text by writing (actually printing) the text in the document using the pen.

Figure 1.2 The Core Gestures Available in PenPoint



The System Interface

Behind the scenes of the Notebook User Interface is a set of components that provides the basic PenPoint functionality. These components extend from the basic, such as user interface support, to the innovative, such as support for the recursive live embedding of applications within each other. To provide all this functionality without wasting space, PenPoint relies on object-oriented techniques, such as inheritance, to support code sharing among components.

Basic Components The basic functionality of PenPoint includes components for managing services such as file system access, handwriting recognition, and the high resolution bit-mapped displays that you would expect PenPoint to have. Through the use of an object-based architecture, each of these system-level components can be interchanged with other components that implement the same functionality, without rebuilding the operating system.

For example, consider the handwriting recognition system. The one currently supplied with PenPoint is based on stroke analysis to determine printed characters. It would be possible to exchange the default system for one that recognizes cursive or one that was built using neural net technology.

Recursive Live Embedding **Recursive live embedding** is one feature that reflects PenPoint's commitment to application building reusing code whenever possible. Simply put, recursive live embedding is the ability to embed live applications within other live applications. At first, it appears to be a simple extension of cutting and pasting data across different applications, but in reality it is much more.

Consider what's involved in adding a figure to a document using traditional word processors. Then consider the effort required to change the picture in the document at a later time. In PenPoint, it is not simply the picture's data that's embedded in the text document, but the entire picture application itself! As a programmer, you can imagine how much this frees you to concentrate on what your application does best and lets the user use other applications to enhance yours.

Inheritance and PenPoint PenPoint relies on the object-oriented technique of inheritance to reduce code bulk further by providing a formal method for code sharing. The increased reliance on code sharing results in a scalable operating system designed to make the most efficient use of the limited resources available on the new class of pen-based tablets. If you as the application designer do your job correctly, your PenPoint-based application will run correctly on everything from machines that exist in RAM only, to those with auxiliary storage (disk drive, RAM disk, and so on).

The Programmer's Interface

My first reaction when reading about the functionality required for a PenPoint application was to recall the Windows and Macintosh application development I've done in the past and reach for the aspirin bottle. Further reading, backed up with actual application development experience, has proven my first reaction to be a bit premature. Yes, PenPoint applications require a lot of functionality, but PenPoint already provides much of it in the form of a rich library of reusable components.

Consider the Application Programmer's Interface (API) to the various components of the PenPoint operating system. The APIs consist of a set of messages (requests for an object to do something) that can be sent to various objects contained in the operating system. The entire package is wrapped up in the concept of an **Application Framework** that supports a predefined set of messages sent to an application class at well-known points in the application's lifetime.

What is of immediate interest to people planning to develop applications for PenPoint is that the default behavior for each of the messages already exists in the application class. You only have to augment the default behavior when your application does something special. Through the use of the Application Framework PenPoint applications become consistent, not only in the way you write the software necessary to build them, but in how the user interacts with different applications.

PenPoint Internals

Several areas of PenPoint's implementation as an operating system are important to the concepts discussed in this book. As a programmer, you need to be aware of portability issues involved with running your application on different hardware platforms. This includes understanding the characteristics of the display and how the display and pen tracking systems work together in implementing the Notebook User Interface. Also, it's helpful to have some understanding of the connectivity features of PenPoint.

The Kernel

PenPoint 1.0 is a preemptive multi-tasking operating system designed to run in the native 32-bit, flat memory mode of the Intel 80386 microprocessor. It uses many of the same features present in O/S 2, including lightweight threads and Dynamic Link Libraries. PenPoint is isolated from a particular vendor's hardware through the **Machine Interface Layer (MIL)** which implements a virtual machine for PenPoint to interact with.

The kernel takes advantage of the memory protection available within the microprocessor to provide protection against data corruption, including an efficient warm-reboot mechanism. This is important for applications designed to run in the field collecting information that will be transferred to another machine at a later time. If you as an application programmer follow the published guidelines for initializing and manipulating data in PenPoint, then PenPoint will help your application recover gracefully from a system fault.

Display

PenPoint provides several layers of abstraction for building a user interface. At the highest level are stand-alone components, such as the Default Application menu. Next are reusable components, such as buttons and text fields, that can be combined to build a user interface. Finally, PenPoint allows you to access the drawing context for a particular window on the display device on which you can render a set of primitives, such as rectangles and text.

Although you can reference absolute pixel coordinates, it is generally a bad idea to do so for a couple of reasons. First, PenPoint can run on many different size screen devices, so if you accessed pixels directly, you would have to rewrite existing code. Second, in addition to size, the color capabilities of devices can vary.

PenPoint provides several mechanisms for helping application writers cope with writing generic applications that work on a multitude of different displays. For example, PenPoint supports the concept of relative layout; you can tell a window to position its top at the same point as the bottom of another window. When the user interface must redraw itself, PenPoint looks at the windows involved and then manages the layout for you.

PenPoint also manages a generic color model for you. This allows you to select colors and have PenPoint pick the closest match. You can also choose from a set of standard colors for the best possible look for your user interface regardless of the actual hardware you're running on.

Interacting with the Pen

The pen plays a dual role in pen-based systems. It is the user's main pointing device for quick interaction with the Notebook User Interface. In addition, it replaces the keyboard as the user's primary data entry tool by using character-recognition software when pointing isn't good enough.

The pen's intuitiveness tends to mask some of the complexities in using it with applications. For example, consider using the pen to draw two lines rotated 45 degrees crossing at right angles. Is this a rather complex description of the letter 'X'? Or is it a gesture that means "cross this out" or delete it? Or is it a method for showing the extent of a rectangle?

Quite possibly, it could be all three. The exact meaning would be determined at the time it was drawn, depending on the context in which its drawing occurred. PenPoint provides applications with the ability to "help" the handwriting recognition system by building in additional contextual information.

In addition to the various gestures and character recognition, the pen provides a richer set of locator type information. With the mouse, you get moves and buttons. With the pen, you get moves, strokes, and positions on the pen contacting the surface, plus proximity data—events generated when the pen comes within a certain distance from the display, but hasn't necessarily touched the screen. Finally, unlike the mouse, where visual feedback is required, visual feedback about the location of the pen is more nuisance than necessity. There is no need to show a user directly interacting with the display device the pen's current location, since the user makes selections by touching the appropriate place on the display with the pen.

Connectivity

Pen-based machines are meant to be mobile. Many current applications depend on the ability to work on a document with the understanding that some I/O operations might need to be deferred until later. Connectivity is handled at various levels, both from the application's and the system's points of view.

For instance, consider the letter faxing example described earlier. Once you complete the act of creating the document to be faxed, you would like to queue it up to be faxed and then forget about it. PenPoint provides any application a standard mechanism for doing this called the **In and Out Boxes.**

The In and Out Boxes are specialized floating notebooks that provide common organization (interface and architecture) for the mechanisms used to transfer information to and from other devices when the devices are in physical contact. This allows applications to transfer data logically when it makes sense for the application, and not when two pieces of hardware happen to be connected.

In addition to the In and Out Boxes, PenPoint provides for connectivity at both the network and file system level. PenPoint has been designed to work with any file system that can handle the API. It also has defined protocols for dealing with the various layers of the OSI networking standard. This allows PenPoint-based hardware to interact easily with both MS-DOS and Macintosh file systems.

Wrap-up

Operating systems built for small systems are amazing not only for what they do in a small space, but for how they do it. PenPoint is able to make available a large amount of reusable functionality because of the decision to embrace the tenets of software reuse whenever possible, including its object-based implementation.

In order to make the most of the resources available to you in PenPoint, you will find yourself adapting to the way PenPoint works more often than adapting the components of PenPoint to work the way you do. My best piece of advice, gained from several years of writing object-based code, is to let go and do it GO's way from the start.

Nowhere will this be more true than with respect to the PenPoint Application Framework. We all have our own style of organizing the various components of an application to work in a certain order. Now, that freedom has been replaced by a standard framework that also implements the order in which things happen. This standardization benefits you in the long way in two areas, despite the restrictions it places on your coding style. First, a standard behavior for all applications will lead to greater user acceptance of the environment and a willingness on the user's part to try new applications. Second, if new components you produce conform to PenPoint standards, they will be reusable in future development efforts you might undertake.

In the end, it's your commitment to building solid applications that fit the PenPoint pen-based metaphor that will determine the scope of your success.

2

The Class Manager

Over the past several years, the programming community has seen a rising interest in a new form of computing called **object-oriented programming**. Object-oriented programming takes its name from the fact that it provides a technique for segregating pieces of functionality into well-defined units called objects. Because the segregation process for creating objects is formalized, other implementation techniques such as inheritance and deferred binding can be used to increase the quality of the software being produced.

PenPoint capitalizes on the use of an object-oriented model to aid in reaching its goal of an operating system built of small interchangeable components that are also made available to other applications. It packages both low level items, such as buttons, and higher level items, such as the Application Framework, into classes of objects to be used.

This chapter discusses the various aspects of object-oriented programming in general and how PenPoint uses them. A general discussion of vocabulary is followed by a look at the various classes available in PenPoint. The last part of this chapter concentrates on the various functions and macro definitions that PenPoint provides for using objects.

Why Objects?

The two most important benefits of using objects in PenPoint are increased productivity and improved software quality. For example, whenever you reuse an object generated by a class in PenPoint, not only are you reusing predefined code, thereby saving development time and storage space, but you are reaping the benefits of GO Corporation's quality assurance program and its commitment to having their objects work well with each other—all for free.

In general, object-based environments are classified based on their implementation of various forms of abstractions. For example, you have probably heard of various programming languages such as Smalltalk, Objective-C, and C++ that support objects, and you might even have used one or more of them. You might also have heard the terms "encapsulation," "inheritance," and dynamic or static "binding." Because there are many different uses of these terms in the field right now, I think it's prudent to spend a few moments outlining the basic concepts so we can work with a consistent and common vocabulary.

Encapsulation

The term **encapsulation** describes the way in which the behavior and persistent data needed to implement a model of a real world abstraction is organized. By definition, only the behavior defined for the particular abstraction may access that entity's data. This produces a form of "firewall" protection by not allowing outsiders to interfere with the entity's internal workings. In essence, you use encapsulation to provide a data abstraction of the real world entity you wish to model.

For example, suppose you want to describe a light fixture that can be either on or off. You might say that in order for an object to be a light fixture, it must respond to being turned on or off, and it must remember whether or not it is on or off. The persistent data would be a variable that tracked whether or not a light fixture is on. The behavior would include ways to turn the fixture on and off and possibly a way to check the state that the light fixture is in.

The information used to keep track of an object is often organized as a set of variables called **instance variables**. Behaviors that manipulate this data are then called **instance methods**. Because the use of objects means data abstraction, only the instance methods of an object can directly access the instance variables of that object. Objects that have the same kind of private data and share the same instance methods belong to the same class of objects. Here's a more formal definition of what the light fixture object might be:

Class

LightFixture

Instance Variables

BOOL onOffFlag

Instance Methods

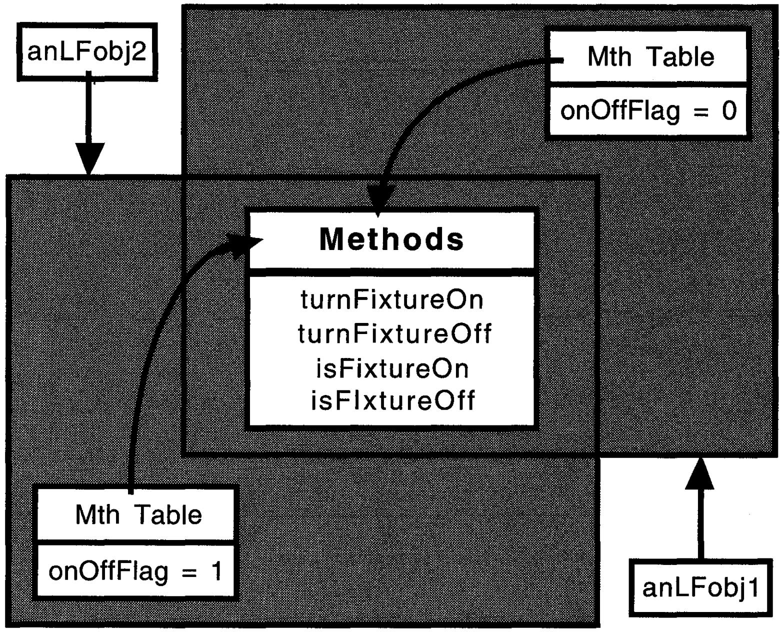
turnFixtureOn
turnFixtureOff
isFixtureOn
isFixtureOFF

What's nice about this formal definition is how easy it is to see that all light fixtures share the same behavior and same type of stateful information. What differs is that the information for each individual light fixture object is unique. This observation leads to the concept of each object having a unique part and a shared part. In general, the unique part of an object contains the instance data and a pointer to the shared information (generally the instance methods) for the kind of object it is.

Figure 2.1 shows the conceptual layout of several light fixture objects. Notice that although there are two different objects, both share a single copy of the instance methods.

One last point—it's important to realize that there are two fundamentally different views of the same object, based on whether you're the provider (sometimes called the producer) or the user (sometimes called the consumer) of the object. The producer of an object is the person who designs, develops, and produces the object that the consumer uses. The producer has access to the instance data of the object, while the consumer can only access the abstraction through the methods the producer provides. The consumer, on the other hand, should be able to reuse a producer's objects free of worry that the underlying abstraction is faulty. This means that the most productive programmers in an object-based environment are those who spend the bulk of their time as consumers of preexisting code.

FIGURE 2.1 How Light Fixture Objects Are Laid Out



Inheritance

There are times when you, as a consumer of objects, can find existing objects that almost match most of your specifications. At that point, you would like to be able to extend the existing object by adding the behavior and data needed to accomplish your task. In object-oriented programming, you accomplish this using a technique called **inheritance**.

Inheritance is a formal methodology for reusing large pieces of code by adding a new piece of code that defines only the differences between old and new. For example, you could produce a new type of light fixture, say a timed light fixture, by adding timing capabilities to an existing light fixture. The terminology used to represent this relationship is that the new class (TimedLightFixture) is a subclass of the pre-existing (LightFixture) class. Conversely, you would say that the pre-existing class (LightFixture) is the superclass of the new (TimedLightFixture) class. Here's a list that shows the extensions necessary to build a new class of timed light fixture objects.

Class

TimedLightFixture

SuperClass

LightFixture

Instance Variables

BOOL timerOnOff
TIME turnOnAt, turnOffAt.

Instance Methods

turnFixtureOn
turnFixtureOff
setOnTime (TIME newOnTime)
setOffTime (TIME newOffTime)

In addition to adding new instance data and methods, the new TimedLightFixture class redefines, or overrides, two methods (turnFixtureOn, turnFixtureOff) defined in its superclass. If you send the turnFixtureOn message to a light fixture object, that request will be handled by the method in the LightFixture class. On the other hand, the same request to a timed light fixture object will be handled in the TimedLightFixture class because that class has its own turnFixtureOn method defined. As in most object-oriented programming environments, PenPoint supports several ways for the subclass to access the behavior of a superclass's method that it overrides.

As I mentioned earlier, there are two views of every object: the producer's and the consumer's. The previous list defined the producer's view of the new TimedLightFixture class. It simply shows the differences between a timed light fixture and a basic light fixture. The next list defines the consumer's view of a timed light fixture, including the data and methods inherited from a light fixture.

Class

TimedLightFixture

Instance Variables

BOOL onOffFlag
BOOL timerOnOff
TIME turnOnAt, turnOffAt.

Instance Methods

isFixtureOn

isFixtureOFF
turnFixtureOn
turnFixtureOff
setOnTime (TIME newOnTime)
setOffTime (TIME newOffTime)

The inheritance hierarchy for this example is very simple and therefore fairly easy to explain. As you can imagine, deep inheritance hierarchies can quickly lead to very complicated objects and problems in trying to figure out exactly where something is happening. Inheritance is not a panacea and should be used as an implementation tool only. When designing objects, you should start with their requirements and then try to find an existing object that does the trick. Then if you can't find a direct match, you should consider using inheritance.

Binding Options

You work with objects by sending them messages. These messages are then translated to an appropriate method in a particular class, based on the type of object you sent the message to. For example, if you send the turnFixtureOn message to a timed light fixture object, it uses the method found in the TimedLightFixture class to carry out your request. On the other hand, if you sent the same message to a light fixture object, it uses the method found in the LightFixture class to carry out your request. The process of determining which method in which class should respond to which message sent to a certain object is known as **binding**.

Binding a requested behavior (a message send) to the actual implementation (the method) can take place at multiple points in the life cycle of an application. For example, you can define a macro that provides a set of functionality that is expanded during preprocessing. The macro is said to be bound at preprocessing time, because once the preprocessor has run, the behavior of that macro can't be changed. Another example of a binding tool is a linker, which resolves references to functions that might be located in different modules. This form of binding is exploited for reuse by defining libraries of functions that multiple applications can share.

The binding of messages sent to objects can occur at various times. For example, you can ask the linker to bind a message sent to a particular class of object to the actual method that will handle the request, but only if you know that information ahead of time. Although this produces efficient code, it severely limits reuse of objects, because the compiler must know every possible use of the object when it's being built. At the other end of the spectrum, you can defer binding until the message is actually sent to the object in question. At that time, a lookup is performed to see which method to call to access the required functionality. This allows you to build generic objects that will work with any other object, as long as that other object responds to the messages your original object expects to send.

Reliability, Reusability, and Cost

During any design process, there is always the possibility that a new requirement will arise that has no existing solution. When this situation occurs in object-based design, it is usually solved by implementing a new class. Unfortunately, most of the time the solution doesn't go far enough.

A reusable component will go through three distinct stages. The first stage is when the component is good enough for me, its designer, and developer. Because I'm the only one using it, I can make assumptions about its internal representation and limit the amount of destructive testing (exceeding boundary limits, for example) performed. Also, because of the limited requirements on its robustness, I can quickly get a class to this stage. The downside of stopping at this stage is that the cost to develop the component can be spread over one person only, since no one else would want to reuse such a component.

The second stage occurs when you convince me to share my custom component with you. In the process of using the component within a second application, I will spend additional time uncovering and fixing problems that didn't arise in my application, but manifested themselves in yours. Also, since we are sharing the same object, I'll have the added benefit of increased generality in my component. The actual effort in moving from stage one to stage two is usually a fraction of the time taken to get to stage one. However, now I can divide the cost of developing and maintaining the component by two.

Finally, there is stage three, when a component has been designed and tested to be as generic as possible to work in applications as yet unimagined. This type of reusability is available in PenPoint as a direct result of PenPoint's use of deferred binding in sending messages between objects. Getting a class to stage three tends to be fairly expensive, because now I must incur the cost of documenting the component for a much wider audience. However, the cost of developing the component can now be spread over a much larger number of users, both inside and possibly outside of my development organization.

PenPoint's Definition of OOP

PenPoint has adopted the best features of object-oriented programming (OOP) and has applied them to reusing code efficiently. For example, consider the idea of shared libraries. PenPoint uses Dynamic Link Libraries (DLLs) to allow multiple applications to share the same copy of the library by deferring linking until the application starts to run. This allows greater code reuse since multiple users of the same object work from a common piece of code, instead of individual copies. A very important but often overlooked bonus is that it is necessary to change an object in one place only to have the benefits of its updates extended to all its subclasses.

Encapsulation in PenPoint

PenPoint implements two types of objects: **instances** and **classes**. Instances are objects that inherit from clsObject and have the given instance data described by the class object. Class objects (sometimes known as meta-classes) inherit from clsClass and have the responsibility for defining what goes into their instance objects and which methods should respond to certain messages received by an instance of their class.

Because PenPoint objects tend to be heavyweights (that is, they routinely send messages to each other across process boundaries), PenPoint extends the concept of encapsulation to include allocating an object's instance data from protected memory, affording your application an extra bit of data security. Placing instance data in protected memory is useful but does necessitate caution when writing the data back into protected memory.

For example, suppose you de-reference the instance data to access one field and then call another method. The second method also accesses the same instance data, changes a field, and then writes the information back into protected memory. When it returns to your original code, you continue to modify the instance data, and then you too rewrite it. But, you're working from an original copy of the instance data that was invalidated by the message you sent, thereby corrupting the contents of the instance variables.

Inheritance in PenPoint

PenPoint implements a single inheritance model for its classes. A class inherits both instance data and methods from its superclass (sometimes referred to as its ancestor). Even though a class inherits its superclass's instance variables, it can only directly access those defined for it. It must send messages to its ancestors to access instance data that they define. Instance methods overridden by a subclass can be accessed one of three ways. First, you can specify that the inherited behavior be called before the new behavior. Second, you can specify that the inherited behavior be called immediately after the new behavior completes. Third, you can send a message to the superclass within the method asking for the inherited behavior to be executed.

Message Binding in PenPoint

PenPoint incorporates several mechanisms for sending messages synchronously and asynchronously, not only within a single process but across process boundaries as well. The user of the remote object can choose how a message is sent, a feature that gives a great degree of flexibility in configuring how things work together.

The PenPoint Class Hierarchy

Application building in object-based systems should be an assembly process based upon the availability of already built (designed, developed, tested, and packaged) components. Ideally, you would build your application by selecting what you need from the components list, providing a small amount of glue code, and linking everything together into your finished application. Unfortunately, most of the time you need to write more than "just a little glue code" to have everything necessary for your application; you end up needing additional classes.

Even if you find yourself having to design and develop new classes, the assembly metaphor is still a reasonable way to proceed. Any additional classes deemed necessary to develop should be as generic as possible so the next time someone is looking for that type of behavior, they can reuse what you have just created. Once you have manufactured the new class, you can slip back into assembly mentality and actually write that little piece of glue code necessary to bind the components together.

The success of any assembly effort relies on the designer having a strong understanding of the components available. In PenPoint, this translates into the application programmer having a feel for the contents of the class library and the steps involved with reusing them, both by assembling complex objects from existing ones (construction) and by customizing existing classes for a specific need (subclassing).

What's Available

PenPoint's class hierarchy consists of approximately 180 classes divided over six major areas of functionality. Although it's technically one big hierarchy because all objects inherit from clsObject, my experience has shown that a hierarchy this large is easier to comprehend if taken in smaller pieces. For discussion purposes I treat the smaller pieces as separate hierarchies. This is reasonable because the interaction between classes of different hierarchies tends to be across a small, rigidly defined, message-based interface with few assumptions made about how the internals of the interacting objects work. In general, you can apply the rules of software engineering with respect to coupling and cohesion to help you understand and delimit the boundaries of these subhierarchies.

The six subhierarchies of interest in PenPoint are the application classes, installation classes, windows and UI toolkit control classes, remote interfaces and file system classes, text and handwriting classes, and a set of miscellaneous classes. Classes that are members of the same small hierarchy tend to rely more upon in-depth knowledge of each other's internal implementation. Sometimes, this information is encoded in the form of shared superclasses that localize the need for cross class-specific information.

Application Classes PenPoint provides a methodology for building applications that insures all applications work in a similar manner. Known as the Application Framework, this methodology is implemented using the application class hierarchy, which includes the superclass of all application classes, clsApp. In addition to clsApp, the application class hierarchy also contains clsClass, the Class Manager itself.

Installation Classes The installation classes are used to implement behavior for managing the installation of system resources. These resources include fonts, handwriting prototypes, applications, services, and user preferences.

Windows and UI Toolkit Control Classes The windows and UI toolkit control classes are the largest of the PenPoint hierarchies, with over 60 classes dedicated to the implementation and control of the Notebook User Interface. In addition to being the largest, they also boast the deepest hierarchy leafs (clsPopUpChoice, for example, with nine superclasses). They include classes such as **clsWin** (the window class), which is superclass to all displayable items in the NUI, and the classes that actually manage the devices used to display the NUI.

The windows and UI toolkit control classes are probably the most mature set of classes in PenPoint from the viewpoint of historical precedence. This is due to the traditional strengths associated with object-oriented systems that have deferred binding (such as Smalltalk or PenPoint) for building user interface components. Therefore, the implementers of PenPoint were able to leverage a large amount of existing knowledge in building the NUI.

Remote Interfaces and File System Classes Included in the remote interfaces and file system class hierarchy is support for network-based computing, such as TOPS and SoftTalk. In addition to network support, there are also classes that provide for file management, hardcopy printing, and fax/modem support.

Text and Handwriting Classes The text and handwriting class hierarchy provides support for managing user input to applications. There are individual classes that support entities such as gestures, scribbles, keys, pen strokes, and so on. This class also contains a spelling manager for use within your PenPoint applications.

Miscellaneous Classes Finally, there is a hierarchy of classes that are supportive in nature, but not attachable to any subsystem in PenPoint. These classes include, for example, a string manager, a timer, and a battery monitor.

PenPoint's Class Manager

One of the most powerful PenPoint features is its total commitment to the use of objects. Unfortunately, GO chose not to support one of the many object-oriented programming languages available and instead implemented a set of function calls and macros for managing objects in the PenPoint environment based on ANSI-C. This makes managing the use of objects one of the most difficult and time-consuming requirements of writing application software for PenPoint, because many aspects of object-based programming managed by OOP languages must be done manually by the application programmer.

The functions, macros, and support classes used to implement the PenPoint object model are collectively known as the **Class Manager**. The next several sections outline the functional and macro-based interfaces to the Class Manager that are needed when using objects in PenPoint.

Identifiers

The term **UID**, or **Universal Identifier**, refers to a system-wide unique handle on an object. Actually, two types of Universal Identifiers are available in PenPoint: **Well known UIDs** which identify classes and **Dynamic UIDs** which identify instances created by your application.

Both types of UIDs are 32-bit numbers used by the Class Manager to identify an object. UIDs are not data pointers, rather they are a collection of information that includes an administered value from the GO Corporation. This administered value guarantees your new UID uniqueness from every other UID created, unless someone doesn't follow the rules. You will notice that the example classes contained in this book all use an administered value that GO provided for the purpose.

Class UIDs A Well Known UID for a new class is created using the `MakeWKN()` macro supplied in the go.h header file. For example, suppose GO gave you two administered values (1111 and 2222) for use in implementing the LightFixture and TimedLightFixture classes. You would use the `MakeWKN()` macro to generate Universal Identifiers for those classes by including

```
#define clsLightFixture MakeWKN(1111, 1, wknGlobal)
#define clsTimedLightFixture MakeWKN(2222, 1, wknGlobal)
```

where the first parameter (1111) is the administered value, the second (1) is the version number, and the third (wknGlobal) is the scope.

Alternately, you could use the `MakeGlobalWKN()` macro

```
#define clsLightFixture MakeGlobalWKN(1111, 1)
#define clsTimedLightFixture MakeGlobalWKN(2222, 1)
```

to automatically define the scope as global.

Other Well Known UIDs There are several other types of Well Known Universal Identifiers in addition to the ones used for objects. They include the management of unique values for status information, message identifiers, and tags.

For example, you can use the `MakeMsg()` macro to generate a unique identifier for the isFixtureOn method in class clsLightFixture by including

```
#define msgIsFixtureOn MakeMsg(clsLightFixture, 0)
```

where the first parameter is the name of the class the message is defined for and the numerical value indicates a unique message number for that class. Each message is truly unique because it makes use of the administered value found in the class's Well Known UID.

Similarly, error status values are created using the `MakeStatus()` macro:

```
#define stsFixtureShorted MakeStatus(clsLightFixture, 1)
```

The actual status value created is a signed value that indicates whether the status is an error (negative) or a non-error (positive). To create a non-error status code you could use the `MakeNonErr()` macro:

```
#define stsFixtureOn MakeNonErr(clsLightFixture, 1)
```

Finally, **tags** are 32-bit values that can be used to identify well-known constants within your application. Tags are commonly used to identify such things as option sheets, options cards, and Quick Help strings. Tags are generated using the `MakeTag()` macro which takes a class UID and a tag identifier as parameters and creates a unique tag.

Manipulating Objects

PenPoint defines several messages that are used to create, maintain, and free objects from your application. Most behavior for implementing this functionality is actually inherited by a new class from its root ancestor, clsObject, or clsObject's meta-class clsClass. (The use of meta-classes to build objects will be discussed in the next section.)

Creating Instances Creating an object (an instance of a class) in PenPoint is a two-step process involving the initialization of a default data structure and then the actual creation of the object itself. By convention, each class in PenPoint defines a *CLASSNAME*_NEW structure that contains the information necessary to initialize a new object.

For example, consider a block of code that creates a label object from clsLabel:

```
LABEL_NEW ln;
STATUS    s;
...
ObjCallRet( msgNewDefaults, clsLabel, &ln, s );
ln.label.pString            = "Default String Value";
ln.label.style.scaleUnits = bsUnitsFitWindowProper;
ln.label.style.xAlignment = lsAlignCenter;
ln.label.style.yAlignment = lsAlignCenter;
ObjCallRet( msgNew, clsLabel, &ln, s );
```

The first line defines a structure that holds the new information needed to initialize the label. This structure is defined in a file called label.h in the **Software Developer's Kit** or **SDK**. Next, the new structure is filled in by sending a message to clsLabel asking for its default values. Once you have the default values, you can modify them. In this case, the label will have the default string of `"Default String Value"`, will size itself so it takes up as much of the window as possible, and will center itself in its parent window. Finally, a msgNew message is sent to clsLabel with a pointer to the completed `LABEL_NEW` structure. When the message returns, the new structure's `ln.object.uid` field contains the UID of the label object that was just created.

If you want to use the default new structure returned by msgNewDefaults without modifications, you could replace the two calls in the code block with the single call sequence:

```
LABEL_NEW ln;
STATUS    s
...
ObjCallRet( msgNewWithDefaults, clsLabel, &ln, s );
```

The definition of `ln` is still needed because that is where the UID of the new object will be returned.

Removing Objects When you are done with an object, you must destroy it to free its resources for other processes to use. Keep in mind that memory is a valued resource in a tablet machine running PenPoint, and therefore you should not keep objects past their natural lifetime. You remove an object by sending it the msgDestroy message.

For example, to remove the label object just created, you would do the following:

```
ObjCallRet( msgDestroy, ln.object.uid, objNull, s );
```

The `objNull` parameter is a place keeper because the `ObjCallRet()` macro requires four parameters. `objNull` is a PenPoint-defined null object.

Inspecting Objects You can use several messages to inspect an object's contents. For example, to have a particular object send debugging information about itself to the debug output stream, you would send it the msgDump message:

```
ObjCallRet( msgDump, ln.object.uid, minS32, s );
```

The amount of information is controlled by the value passed in the argument parameter. By convention, the value `minS32` produces verbose output, including recursive verbose outputs for any embedded objects it contains. This is the maximum amount of information that you can request. You use values to specify how much information you want about an object.

0	Implementer chooses what information is most useful.
1	Terse, one line only.
-1	Terse, includes information about embedded objects. One line of information for the parent object, plus one line of information for each object embedded in it.
maxS32	Verbose, includes all possible information about the object.
minS32	Verbose, includes information about embedded objects. Provides maximum amount of information.
N	All other values are at the discretion of the implementer.

In addition to asking an object to dump its debugging information, you can also request other information using these messages.

msgIsA	Tests if the object is a member of the indicated class.
msgClass	Passes back the class used to create the object.
msgVersion	Passes back the version of the object.
msgTrace	Turns on message tracing for the messages sent to the object.

NOTE: This list is not complete.

Sending the msgTrace message to an object turns on message tracing. When this happens, every `ObjectCall()` to the object causes a three-line message to be printed:

```
C> Trace ObjectCall:@cls="ancestor name"    task="task"
C> object="object name"                     depth="D"
C> msg="message name", pArgs="address", pData="address"
```

where task is the task ID in hex, depth is the number of recursive dispatch loops, and pArgs/pData points to the method arguments and instance data, respectively.

When the trace is complete another message is printed:

```
C> Trace ObjectCall:return="status value"   task="task"
C> object="object name"                     depth="D/C"
```

Sending Messages

You send a message to an object in PenPoint using one of several predefined function calls. In general, when you send a message to an object, the Class Manager looks at the message sending type, the message ID, and the object the message is being sent to before determining exactly which method should be used to provide the requested behavior. PenPoint's very versatile messaging facility allows you to send messages not only in your process space, but also to objects located in other processes.

GO has defined a set of macros to help manage message sending and the status values returned. Before I explain these, however, let's look at the underlying functions that support the macros.

Message Functions In general, messages look like this:

```
s = ObjectCall(msgSomeMessage, anObject, pArgStructure )
```

where msgSomeMessage is a Well Known UID, anObject is a variable containing the Universal ID of the object that is to receive the msgSomeMessage message, and pArgStructure is a pointer to a structure containing the arguments required by the message being sent. The message function returns a status value indicating the success or failure of the message.

The actual function prototype of ObjectCall() is

```
STATUS EXPORTED ObjectCall(
   MESSAGE  msg,
   OBJECT   object,
   P_ARGS   pArgs
);
```

Use ObjectCall() to send messages to objects within your own process space. If you want to send a message across task boundaries, use the `ObjectSend()` function, defined

```
STATUS EXPORTED ObjectSend(
   MESSAGE  msg,
   OBJECT   object,
   P_ARGS   pArgs,
   SIZEOF   lenArgs
);
```

Because an ObjectSend() message crosses process boundaries, the pArgs block must be copied into the address space of the object that will receive the message. An additional parameter, `lenArgs`, facilitates this. If `lenArgs` equals zero, ObjectSend() interprets pArgs to point at a block of globally accessible memory. Once the data transfer is complete, ObjectSend() causes an ObjectCall() to execute in the process space of the target object.

Due to the crossing of process boundaries, the data pointed to by pArgs will not be updated and therefore cannot be used to retrieve information. If you need to retrieve information across boundaries, you use the `ObjectSendUpdate()` function which works like `ObjectSend()` but does copy the information back across boundaries.

`ObjectCall()`, `ObjectSend()`, and `ObjectSendUpdate()` are all synchronous functions. They block execution on the sending side until the receiving side has finished handling the message. Two functions, `ObjectPost()` and `ObjectPostAsync()`, remove the blocking requirement on the sender's side.

`ObjectPost()` is similar to `ObjectSend()`, except that it defers message sending until the receiving object's task returns to its top level dispatch loop. Because the receiving side waits until the object is at its top level dispatch loop, you can be assured that one message will be processed before another is sent.

If you aren't concerned with synchronizing the input loop with the message, you can use the `ObjectPostAsync()` function instead. However, you must be aware, if not concerned, about issues involving concurrency on the receiving object's side. Also, it isn't possible for `ObjectPostAsync()` or `ObjectPost()` to return a meaningful status value, because the sending task does not block while waiting for a reply.

Message Macros To facilitate exception handling, GO has defined a set of macros to be used when sending messages. The format for these macros is

<message-type><status-handler>
(message, object, pArgs, <opt-status>, <opt-label>)
where:

<message-type>	Is "ObjectCall," "ObjectSend," or a similar function.	
<status-handler>	Is one of	
	Ret	To return immediately if there is an error.
	Jmp	To jump to the <opt-label> if an error occurs.
	Ok	To test for an error and return the results of the test.

<opt-status>	Is the status value returned by the method handling the message.
<opt-label>	Is the point in the source code where control should be transferred in case of error.

Suppose, for example, that you are sending a message to create an object and an error occurs. If you want to exit the current function, the correct function to use is

```
ObjCallRet(msgNew, clsWhatever, &newStr, s);
```

where s had been previously defined as type STATUS.

On the other hand, if you need to perform some clean-up function, even if an error caused an abnormal status code to be returned, you can use `ObjCallJmp()`. For example, consider the code fragment:

```
    // allocate space for a local buffer
    s = stsOK;
    ObjCallJmp(msgNew, clsWhatever, &newStr, s, dealloc_mem);
    // do processing using the buffer
dealloc_mem:
    // free the temporary buffer
    return s;
```

This code uses a dynamically allocated buffer with an object that it creates. This function needs to free the buffer, even if it doesn't use it due to an error status being passed back from the msgNew message.

In this example, an error condition causes execution to be transferred directly to `dealloc_mem` without attempting to use the clsWhatever object. The value of status reflects the error value if `msgNew` was unsuccessful. If execution proceeds normally, then the status remains stsOK.

Utility Functions

One of the great difficulties with systems that support dynamic linking and loading of software concerns **versioning**. In PenPoint, one piece of information stored for each object is a current version number. This information can be used to determine whether it can be used as is with the currently loaded module that implements the object's behavior. It is possible to include routines with your custom objects that allow older versions of objects to be upgraded to the current version.

In addition to the version number, there is a set of functions that can be used to get ASCII information about a class. For example, you could use the `ClsMsgToString()` function to find the symbolic name of the message ID passed as an input parameter. This type of information is also helpful in producing debugging tools, or even the response to a msgDump message.

Building PenPoint Classes

In the perfect world, you could simply turn to a stash of built components and use them to construct your application, without having to add a single new component. This is very close to a reality in PenPoint, due to the rich set of components it provides. The single exception is that each application in PenPoint must have its own subclass of the application class to be able to do any useful work. But even then, only a small amount of new code needs to be written to implement a custom application class.

You must provide four pieces of information to PenPoint when adding a new class to those it makes available to your application. First, you must provide a set of functions that implement the behavior that sets your class apart from another. Second, you must provide a translation mechanism, called a **method table**, that translates a message into a Universal Identifier that can be used to bind the behavior request (message) with its implementer (the function). Third, you must provide a function that sends a message to the Class Manager to request your class be added. Consumers of your class use this function to register it with the Class Manager when they need it. Finally, you should provide an interface file containing message definitions, the NewDefaults structure, status values, and other information for consumers of your class.

The Implementation File

The implementation file for a class usually consists of the structure that defines the class's instance variables, plus the functions that implement the class's desired behavior. In addition, there is a function that can be called to register the class with the Class Manager.

Instance Variables Instance variables are defined using a C structure specified in the implementation file. When the class is registered, the Class Manager is passed the size of the instance variables structure so it knows how much memory to allocate for each instance from protected memory.

For example, the instance variable structure for clsLightFixture would look like this:

```
typedef struct INSTANCE_DATA {
   BOOL onOffFlag;
} INSTANCE_DATA, *P_INSTANCE_DATA;
```

There are several size limits for instance data. First, the instance data for any class is limited to 64K. Second, the instance data for an entire object (that is, the data my new class defines, plus all the data its ancestors define) is limited to 64K of protected data. Finally, unprotected instance data is limited to 64K per class, but there is no limit on unprotected instance data for the entire object.

Method Macros Methods are defined using macros that create functions with the names and parameter types the Class Manager expects. Each method receives five pieces of information from the Class Manager when it is invoked:

msg	The UID of the message used to invoke this method.
self	A pointer to the object that the message is being sent to.
pArgs	A pointer to a block of memory containing the argument data.
ctx	The context that contains the method being executed.
pData	A pointer to the instance data contained in protected memory.

The default definition for a method is handled using the `MsgHandlerPrimitive()` macro. This macro is defined

```
#define MsgHandlerPrimitive(fn, pArgsType, pInstData) fn(\
   const MESSAGE         msg,       \
   const OBJECT          self,      \
   const pArgsType       pArgs,     \
   const CONTEXT         ctx,       \
   const pInstData       pData)
```

A second macro, `MsgHandlerParametersNoWarning` works in tandem with the `MsgHandlerPrimitive()` macro to instruct the compiler to ignore any unused members of the `MsgHandlerPrimitive()` macros input declaration. This macro is defined

```
#define MsgHandlerParametersNoWarning \
   Unused(msg); Unused(self); Unused(pArgs); \
```

```
    Unused(ctx); Unused(pData)
```

Several variations on `MsgHandlerPrimitive()` can be used based on the needs of a particular application. For example, suppose you are not going to access the instance data pointer passed to the method, and the method itself has no arguments. You could use

```
MsgHandler(MyMethod)
{
    // do something
    MsgHandlerParametersNoWarning;
}
```

If, on the other hand, you need to access the arguments passed to the method, but not the instance data, you could use the `MsgHandlerArgType()` macro:

```
typedef struct MYARGS {
    char *aString;
} MYARGS, *P_MYARGS;

MsgHandlerArgType(MyMethod, P_MYARGS)
{
    STATUS s;
    ObjCallRet(msgSetStringValue, self, pArgs->aString, s);
    return stsOK;
    MsgHandlerParametersNoWarning;
}
```

`MsgHandlerArgType()` uses its second parameter to specify the type to cast the pointer to the arguments passed by the Class Manager. Also notice that I made use of the **self** variable to send a message. Self is defined to point at the UID of the object originally sent the message. The value of self never changes, even if the code that handles the message is not located in the class that the object belongs to.

Finally, `MsgHandlerWithTypes()` can be used when it's necessary to access both the instance data and the arguments:

```
typedef struct INSTANCE_DATA {
  BOOL onOffFlag;
} INSTANCE_DATA, *P_INSTANCE_DATA;

typedef struct MYARGS {
  char *aString;
} MYARGS, *P_MYARGS;

MsgHandlerWithTypes(MyMethod, P_MYARGS, P_INSTANCE_DATA)
```

```
{
  STATUS s;

  if (pData->onOffFlag)
   ObjCallRet( msgSetStringValue, self, pArgs->aString, s);
  else
   ObjCallRet( msgSetStringValue, self, "", s);
  return stsOK;
  MsgHandlerParametersNoWarning
}
```

Method Table

Each class has a method table that is used to translate between messages and methods. In PenPoint, it is your responsibility as the application programmer to maintain this table for each new class you build. The method table is composed of a set of entries that have three fields: the message field; the method field; and the attribute field. The attribute field indicates whether the ancestor method (if present) should be called before the current method (`objCallAncestorBefore`), after the current method (`objCallAncestorAfter`), or not at all (`0`). There is one entry per message/method pair in the class.

For example, consider the method table entries for the LightFixture class:

```
MSG_INFO clsLightFixtureMethods[] = {
  msgTurnFixtureOn,    "LightFixtureTurnOn", 0,
  msgTurnFixtureOff,   "LightFixtureTurnOff",0,
  msgIsFixtureOn,      "LightFixtureIsOn",   0,
  msgIsFixtureOff,     "LightFixtureIsOff",  0,
  0
};
```

This information is processed by a special tool, called mt.exe, that creates a table used specifically by the Class Manager. In this example, the four messages have been defined with the `MakeMsg()` macro in the header file used to publish the new class' interface. The listed methods correspond to the methods you created in the implementation file using the various `MsgHandle()` macros.

A second entry in the method table file organizes multiple classes' method tables in one place. It is called the `CLASS_INFO` structure and looks like this:

```
CLASS_INFO classInfo[] = {
   "clsLightFixture", clsLightFixtureMethods, 0,
   0
};
```

Each row in the table contains the class name in quotes, followed by the name of the MSG_INFO structure. A null terminates each row, and a null in the first column of the last row terminates the entire structure.

Registering a PenPoint Class at Runtime

Once you have built the appropriate structures, methods, and method table, you need to provide users of your new class a function they can call to register it with PenPoint. By convention you do this by defining the function:

```
STATUS ClsYourClassNameInit (void)
```

to register the new class with the Class Manager.

Continuing the clsLightFixture example, the class registration function would be

```
STATUS ClsLightFixtureInit (void)
{
   CLASS_NEW    new;
   STATUS       s;

   ObjCallJmp(msgNewDefaults, clsClass, &new, s, Error);

   new.object.uid       = clsLightFixture;
   new.cls.pMsg         = clsLightFixtureTable;
   new.cls.ancestor     = clsObject;
   new.cls.size         = SizeOf(INSTANCE_DATA);
   new.cls.newArgsSize  = SizeOf(LIGHTFIXTURE_NEW);

   ObjCallJmp(msgNew, clsClass, &new, s, Error);

   return stsOK;

Error:
   return s;

}
```

`ClsLightFixtureInit()` assumes that a global Well Known UID (clsLightFixtureTable) has been defined for the class, a method table (cls-

LightFixtureTable) exists for the class, and that a new data structure (LIGHTFIXTURE_NEW) and an instance variable data structure (INSTANCE_DATA) have been defined for the class. Finally, you need to specify the new class's superclass, or ancestor, as part of the CLASS_NEW data.

After modifying the CLASS_NEW structure returned by the msgNewDefaults message, msgNew is sent to clsClass to register the new class you've defined. In this example, only four of the available parameters are modified. The actual permutated list of available attributes you can change is

```
APP_MGR_NEW new;
new.object.newStructVersion;// Out: [msgNewDefaults] Vald
msgNew
                      // In:  [msgNew] Valid version
new.object.key;       // In:  [msgNew] Lock for the object
new.object.uid;       // In:  [msgNew] Well-known uid
                      // Out: [msgNew] Dynamic or Wkn uid
new.object.cap;       // In: [msgNew] Initial capabilities
new.object.objClass;  // Out: [msgNewDefaults] Cls called
                      // In:  [msgObjectNew] Class id
                      // In:  [msg*] Used by toolkit
new.object.heap;      // Out: [msgNewDefaults] Heap for
                      // additional storage.If capCall,pass
                      // OSProcessSharedHeap else pass
                      // OSProcessHeap
new.object.spare1;    // Unused (reserved)
new.object.spare2;    // Unused (reserved)
new.cls.pMsg;         // In: Can be pNull for abstract class
new.cls.ancestor;     // In: Ancestor to inherit from
new.cls.size;         // In: Size of instance data, can be 0
new.cls.newArgsSize;  // In: Size of XX_NEW struct, can be 0
new.cls.spare1;       // Unused (reserved)
new.appMgr.dir;                      // App monitor dir.
new.appMgr.appMonitor;               // App monitor object.
new.appMgr.resFile;                  // App res file.
new.appMgr.iconBitmap;               // Icon bitmap.
new.appMgr.smallIconBitmap;          // Small icon bitmap.
new.appMgr.appWinClass;              // App win class.
new.appMgr.defaultRect;              // Default rect (points).
new.appMgr.name[nameBufLength];      // Application name.
new.appMgr.version[nameBufLength];   // Version.
new.appMgr.company[nameBufLength];   // Company name.
new.appMgr.defaultDocName[""];       // Default doc name.
new.appMgr.copyright;                // Copyright.
new.appMgr.programHandle;            // Program handle.
new.appMgr.reserved[4];              // Reserved.
new.appMgr.flags;
```

This list of attributes should not be seen as a challenge—How many values can I change? Rather, you should view it as an indication of the type of benefits the object-based framework provides for you. You must only point out the differences in your class to PenPoint, because much of the work your application needs to do has already been implemented.

A Class Interface File

Early on, I mentioned that there are two distinct views of an object: the consumer's and the producer's. The first three items you provide for the Class Manager constitute the producer's view of the object: the methods, message-to-method translation table, and the actual mechanics of specifying the superclass of the new class.

There are times, however, when other consumers will want to use a class you produced for yourself. Although this doesn't happen with custom application classes very often, it does happen to classes such as clsLabel that are reused many times by many different consumers. The current convention is for you to produce an interface file for objects that might have consumers at a future time.

The interface file contains the definition of the class's `_NEW` structure and any new messages that the class defines. It also includes definitions for status values that might be passed as return codes, and data structure definitions for sets of information that might be passed as arguments to one of the methods during a message send.

The interface file for clsLightFixture might look something like this:

```
#ifndef LGTFXTR_INCLUDED
#define LGTFXTR_INCLUDED

#ifndef CLSMGR_INCLUDED
#include <clsmgr.h>
#endif

#define clsLightFixture MakeGlobalWKN( 1111, 1)

STATUS ClsLightFixtureInit (void);

#define stsFixtureShorted MakeStatus(clsLightFixture, 1)

#define stsFixtureArcing  MakeStatus(clsLightFixture, 2)
```

```
#define stsFixtureOn      MakeNonErr(clsLightFixture, 1)

#define stsFixtureOff     MakeNonErr(clsLightFixture, 2)


#define msgTurnFixtureOn  MakeMsg(clsLightFixture, 1)
#define msgTurnFixtureOff MakeMsg(clsLightFixture, 2)
#define msgIsFixtureOn    MakeMsg(clsLightFixture, 3)
#define msgIsFixtureOff   MakeMsg(clsLightFixture, 4)

typedef struct LIGHTFIXTURE_NEW_ONLY {
  BOOL initialState
} LIGHTFIXTURE_NEW_ONLY;

#define  lightFixtureNewFields \
         objectNewFields \
         LIGHTFIXTURE_NEW_ONLY lightFixture;

typedef struct LIGHTFIXTURE_NEW {
        lightFixtureNewFields
} LIGHTFIXTURE_NEW, *P_LIGHTFIXTURE_NEW
;
#endif // LGTFXTR_INCLUDED
```

The last part of the definition file specifies the structure used to allocate new instances of clsLightFixture. The `LIGHTFIXTURE_NEW_ONLY` structure defines the entries light fixture's initialize method needs. Those fields are added to the information clsLightFixture's ancestor, clsObject, needs. Finally, a `LIGHTFIXTURE_NEW` structure is defined to provide access to all the required initialization parameters.

Reusing Inherited Behavior

One way to reuse an existing class is to subclass it. You can then create a new object by indicating how it differs from the old object, its superclass. For example, consider how you would build the timed light fixture object from the existing light fixture object.

Defining the New Class This time, I'm going to start with the `ClsTimedLightFixtureInit()` function used to define the clsTimedLightFixture class:

```
STATUS ClsTimedLightFixtureInit (void)
{
   CLASS_NEW   new;
```

```
    STATUS      s;

    ObjCallJmp(msgNewDefaults, clsClass, &new, s, Error);

    new.object.uid          = clsTimedLightFixture;
    new.cls.pMsg            = clsTimedLightFixtureTable;
    new.cls.ancestor        = clsLightFixture;
    new.cls.size            = SizeOf(INSTANCE_DATA);
    new.cls.newArgsSize     = SizeOf(TMDLIGHTFIXTURE_NEW);

    ObjCallJmp(msgNew, clsClass, &new, s, Error);

    return stsOK;

Error:
    return s;

}
```

Notice that the only real difference between this function and `ClsLightFixtureInit()` are the values passed to the new structure used by clsClass. Like clsLightFixture, clsTimedLightFixture has its own method table and unique Well Known UID. However, instead of using clsObject as its immediate ancestor, clsTimedLightFixture uses clsLightFixture, and therefore also inherits from clsObject's functionality.

Overriding Methods in the Superclass In the class definition for timed light fixture two methods, turnFixtureOn and turnFixtureOff, have the same name as the methods in their ancestor's class. By definition, when you send a message to an object, the method table for the object is checked first, and if a match is found, it's executed. Otherwise, the ancestor's message table is searched, and so on, until a match is found. What happens when your new method needs to access the behavior for the same method name in its ancestor class?

One way you can do this in PenPoint involves specifying that the ancestor's behavior should be invoked either before or after the class's behavior based on an entry in the new class's method table. For example, the method table for clsTimedLightFixture is defined as

```
MSG_INFO clsTimedLightFixtureMethods[] = {
  msgTurnFixtureOn,       "TimedLgtFixTurnOn",
  objCallAncestorBefore,
  msgTurnFixtureOff,      "TimedLgtFixTurnOff",
  objCallAncestorAfter,
  msgSetOnTime,           "TimedLgtFixSetOnTime",0,
```

```
  msgSetOffTime,        "TimedLgtFixSetOffTime",0,
  0
};
```

The entries in the third field of msgTurnFixtureOn and msgTurnFixtureOff contain the predefined values `objCallAncestorBefore` and `objCallAncestorAfter`, respectively. Like the predefined names imply, `objCallAncestorBefore` causes the behavior inherited from the ancestor, in this case clsLightFixture, to be executed first followed by the method found in clsTimedLightFixture. In the same way, the method that corresponds to msgTurnOnFixture in clsLightFixture will be called after the method defined in clsTimedLightFixture completes executing. The entries for msgSetOnTime and msgSetOffTime appear with a null in the third field, indicating that the ancestor's behavior should not be invoked automatically.

Messages to the Superclass There are times when you might find it necessary to call an ancestor's behavior somewhere in the middle of a method that has been overridden. PenPoint provides two functions, `ObjectCallAncestor()` and `ObjectCallAncestorCtx()`, that allow a method to invoke its inherited behavior from anywhere inside itself.

`ObjectCallAncestorCtx()` is defined

```
STATUS EXPORTED ObjectCallAncestorCtx(
   CONTEXT ctx
);
```

It requires only the current context to be given as an input parameter. It automatically calls the ancestor function using the same set of arguments used to call the original method.

If for some reason you desire to change the information being passed back to the ancestor, PenPoint provides the `ObjectCallAncestor()` function, which is defined

```
STATUS EXPORTED ObjectCallAncestor(
  MESSAGE  msg;
  OBJECT   self;
  P_ARGS   pArgs;
  CONTEXT  ctx
);
```

You might consider the need to use this function as a red flag with respect to your object design. By definition, when you override a method, you should be augmenting and/or replacing its behavior in such a way that the external interface to the method doesn't change. For example, consumers should be able to send msgTurnFixtureOn to either clsLightFixture or clsTimedLightFixture objects without having to change the way the messages are sent. Therefore, the behavior inherited by clsTimedLightFixture from clsLightFixture should be accessed using the same set of values that were sent to the original method.

Wrap-up

This chapter covered a lot of ground in a very short time, especially with respect to object-oriented programming in general. It's a fact of life—if you're going to work with PenPoint, you're going to need to learn about object-oriented programming. I suggest that you supplement your reading of PenPoint material with some material on object-based programming in general. I've placed a reasonable reading list in the back of the book if you're looking for a place to start.

3

Application Building

Learning to build PenPoint applications can be a very time-consuming process. For example, users interact with PenPoint in an unusual way. In addition to changes in how the application interacts with the user, every operating system carries its own definition of what makes up the executable form of an application. For PenPoint, the problems associated with building applications are increased. For example, the current implementation of the Software Developer's Kit (SDK) for PenPoint requires a cross development environment, since there are no PenPoint-based tools for building applications. I hope that I can delete these words from a later edition of this book, but for now you must cope with building applications with MS-DOS and then testing them with PenPoint.

This chapter begins the process of answering the question, "What makes up a PenPoint application from the programmer's viewpoint?" I will start with a brief explanation of PenPoint's boot procedure. Next, I'll cover the minimum set of functionality an application must implement to run (notice I said "run," and not "do something spectacular") under PenPoint and some of the help available in debugging applications that don't work correctly the first time.

PenPoint in Action

With PenPoint, end users are given an environment in which the concept of "installing the operating system" makes no sense. Of course, someone had to pre-configure the operating environment at some point, but the user doesn't care about that. The user's only care is that the machine turns on and places the tablet in the same place and state that it was in when the user suspended it.

To make this a reality, PenPoint takes on some extra work in setting up and maintaining its operating environment. Some of this extra work is passed on to the application class and therefore is your responsibility as an application programmer to implement. However, there is a reasonable set of defaults defined in clsApp that handles most of the work for the application programmer, thereby reducing your load. This section discusses initializing the PenPoint environment for the first time and then discusses what's involved in adding and removing applications while PenPoint is active.

Starting PenPoint

The PenPoint boot sequence consists of five stages, starting with hardware initialization and ending with loading the default set of applications. In between, the user's environment is set up, the default DLLs (Dynamic Link Libraries) are loaded, and the system applications are installed.

mil.ini The first initialization file PenPoint accesses during its boot sequence is mil.ini. The MIL, or **Machine Interface Layer**, protects PenPoint from needing intimate knowledge of the exact hardware it is running on. The three most important pieces of hardware specified by mil.ini are the screen type, the stylus device, and the memory model. The boot program uses this information in mil.ini to configure the hardware, initialize the Machine Interface Layer, and then to actually start the PenPoint kernel.

environ.ini When the kernel starts, it uses the environ.ini file to get information about the environment that PenPoint will be operating in. For example, this file contains the location of the application that should be started for the user when PenPoint completes booting. It also specifies characteristics of the display such as pixel characteristics, whether the screen is in portrait or landscape mode, whether the volume is in RAM or on a permanent storage device such as a hard disk.

boot.dlc Once the environment is initialized, PenPoint reads the boot.dlc file for a list of all the DLLs that need to be loaded for the current configuration of PenPoint to work correctly. Most subsystems (UI toolkit, windowing system, and others) in PenPoint are implemented as DLLs and must be loaded before applications can be started.

PenPoint processes boot.dlc by looking for each specified DLL file in the `\\boot\penpoint\boot\dll` directory. Once the DLL is loaded, a call is made to the `DllMain( )` function which behaves much like the `main( )` function described later in this chapter. Once installation is complete, the components in the DLL can be shared by multiple applications running in the environment.

syscopy.ini Once the DLLs listed in boot.dlc are loaded, PenPoint copies all the files and directories listed in syscopy.ini into the selected volume. The first two entries in the list of files to be copied are sysapp.ini and app.ini.

sysapp.ini sysapp.ini specifies all the system applications needed for PenPoint to run correctly. By definition, applications loaded based on their presence in this file can not be removed by the user. Therefore, if you're providing pre-loaded pen-based machines for a vertical application that you want to keep intact, you should load all the applications from sysapp.ini.

app.ini Finally, app.ini contains a list of the applications that should be started when PenPoint is started. Unlike applications specified in sysapp.ini, applications present in app.ini can be removed by the user. This is helpful if you want a pen-based machine to be loaded with the software for three different types of jobs, and then let the user of the tablet delete what's not needed.

Installing an Application

PenPoint provides a consistent mechanism for installing applications. This mechanism is called the **Installer**, and it relies on application distribution to be in a predefined form. The actual installation of an application can occur either at boot time or runtime, whichever is most convenient for the end user. If you want to preload your application, you need to copy it onto the boot volume in the predefined format.

Installing at Boot: app.ini app.ini contains a list of applications that should be loaded prior to finishing the PenPoint boot process. Each line in the file is a separate entry and corresponds to a single application. For example, the entry for a calendar application might be

```
\\boot\penpoint\app\Personal Calendar
```

Notice that the PenPoint name is given, not the name of the MS-DOS executable. Also, \\boot maps to the hard drive with the volume name "boot." PenPoint requires its boot volume to be named "boot."

Runtime Installation PenPoint provides the Installer application to allow the user to bring new applications onto a pre-loaded machine after initialization. It does this by searching all known volumes for installable applications and then listing them for the user. It does a selective search: it looks in all volumes for the directory \penpoint\app and then presents the user with the applications named in the subdirectories it finds.

To make your application installable, you need to copy it to a subdirectory of \penpoint\app on a named volume. This volume could be a hard disk, a RAM disk, or even a floppy disk containing the release of commercial software. The Installer can then add your application to the list the user chooses from.

Application Modules and DLLs One way PenPoint reduces space overhead is by supporting Dynamic Link Libraries (DLLs). As a result, if you and I are sharing the functionality of the same class, we both don't need our own separate copy—we can just share one. Since a single application could depend on multiple DLLs being present for it to run correctly, a PenPoint methodology was devised to track the dependency requirements.

This is done by providing a file with the same name as the application except with the extension dlc. When you request that a certain application be installed, the Installer first checks for the application name with the dlc extension. If found, it checks for the required DLLs, loads those it hasn't loaded yet, and indicates whether the application can be installed or not.

If you have a DLL that needs to be pre-loaded, you can always place it in boot.dlc. But for the most part, you shouldn't preload DLLs until the application that needs them is about to be installed. This helps to cut down on memory usage for the pen-based systems and leaves more room for PenPoint to run.

Taxonomy of a PenPoint Application

The code for creating the simplest form of a PenPoint application is straightforward and consists of three source files: the application source file; the application header file; and the method tables for the classes the application defines. For the first example, it's necessary to define stub structures because much of the functionality is not needed.

The Application Source File

The application source file contains two C function definitions essential to a PenPoint application. The first function, `main()`, is used by the application monitor to start PenPoint applications the first time and then to indicate when the user starts new PenPoint documents (instances of an application). The second function, `Cls`*`ClassName`*`Init()`, is called by `main()` the first time it is executed and is used to register the application class with the Class Manager.

main() As with all C programs, PenPoint applications need a `main()` routine. PenPoint uses this as the entry point for installing the application and for creating instances of the application (user documents). The demo application `main()` source is

```
void CDECL
main( int argc, char *argv[], U16 processCount )
{
  STATUS s;
  if (processCount == 0 ) {
    ClsDemoAppInit();
    AppMonitorMain( clsDemoApp, objNull );
    }
  else
    AppMain();
}
```

Notice the definition of the main() routine:

```
void CDECL
main( int argc, char *argv[], U16 processCount )
```

It has an extra parameter, `processCount`, passed as input to it. The PenPoint kernel will call the `main()` routine at two different times in the application's lifetime: first, when the application is installed, and second, any time a document for that application is activated. The Penpoint kernel keeps a count of the number of processes running a particular program and passes that number to the main routine via the processCount variable. The application can then use this variable to decide whether the application is being installed (processCount == 0), or whether the user is attempting to activate an instance of the application (processCount > 0).

When the application is first installed, it must create two types of objects and register them with the Class Manager. The first is the application object itself, which the Class Manager uses to build documents for the user. The second type of objects are those required by the application itself but currently not available in PenPoint. By convention, objects are created and registered with the PenPoint operating environment by calling a function of the form `Cls`*`classname`*`Init( )`.

In DemoApp, when processCount is zero, then the conditional:

```
if (processCount == 0 ) {
  ClsDemoAppInit();
  AppMonitorMain( clsDemoApp, objNull );
  }
```

is true and the initialization function `ClsDemoAppInit()` is called.

After ClsDemoApp is initialized, the `AppMonitorMain()` routine is used to set up a dispatching loop for completing the application installation, and monitoring behavior for importing documents, copying stationary and resources, and similar activities.

When `main()` is invoked with processCount greater than zero, the PenPoint kernel is signalling that an instance of the application (a user document) is being activated. For DemoApp, this is handled by calling the PenPoint function `AppMain()`. `AppMain()` sets up the dispatching loop that handles messages sent to the application in the normal course of the document's lifetime.

ClsDemoAppInit() The DemoApp application object is created by calling the `ClsDemoAppInit()` function when the `main()` function is called with a processCount of zero. Two things are happening here. First, I'm going to be creating the structure necessary to register my application object with PenPoint. Second, I'm creating a new instance of the Application Manager to manage the interaction of my application with the user.

One of the data structures required to initialize a class is the table used to translate between messages and the methods that respond to those messages. For the demonstration application this table, clsDemoAppTable, is defined in the method.tbl file to be discussed soon. The actual object is created by running the PenPoint utility mt.exe on the method.tbl file which produces the method.h file as one of its outputs. The method.h file contains the declaration for the message translation table. The declaration for DemoApp is

```
extern const U32 clsDemoAppTable[];
```

The actual function is defined

```
STATUS ClsDemoAppInit(void)
{
  APP_MGR_NEW new;
  STATUS s;
```

```
  ObjCallJmp( msgNewDefaults, clsAppMgr, &new, s, Error);

  new.object.uid          = clsDemoApp;
  new.object.pMsg         = clsDemoAppTable;
  new.object.ancestor     = clsApp;
  new.object.size         = Nil(SIZEOF);
  new.object.newArgsSize  = SizeOf(APP_NEW);

  ObjCallJmp( msgNew, clsAppMgr, &new, s, Error );

  return stsOK;

Error:
   return s;
}
```

The application initialization:

```
STATUS ClsDemoAppInit(void)
{
  APP_MGR_NEW new;
  STATUS s;
```

is defined to return the status of the initialization. It also indicates that it has no input parameters. Finally, it defines two local variables, new and s, which are used during the initialization. `NEW` is a structure of type `APP_MGR_NEW` that is used by the Application Manager class clsAppMgr (defined in the SDK) to initialize itself when it is first created.

The `NEW` structure is used to create a new application object by first obtaining a reasonable set of defaults from clsAppMgr and then modifying only the parts that need to be changed. This structure will be filled in as a result of the first message sent in the function:

```
ObjCallJmp(msgNewDefaults, clsAppMgr, &new, s, Error)
```

`ObjCallJmp()`, a macro defined in the SDK, sends the message indicated in the first parameter (`msgNewDefaults`) to the object indicated in the second parameter (`clsAppMgr`), passing the third parameter (a pointer to the `NEW` structure) as the argument to the method being invoked. If the returned status value indicates that an error occurred, control transfers to the label (`Error:`) indicated in the fourth parameter. In the case of both message sends defined in this function, control transfers to the `Error:` label if the status indicates an error occurred. At that point, the function returns with the value of `s` that indicates the problem that caused the error.

Once the msgNewDefaults message is sent to clsAppMgr, the NEW structure contains the correct set of default parameters for creating an instance of clsAppMgr. The NEW structure is then modified so it contains the correct value for registering the DemoApp application. In essence, I am setting up a class registration object to be used as an input to the method that responds to msgNew in clsAppMgr. For this application, I am asking the Application Manager class to register my application class as

```
new.object.uid = clsDemoApp;
```

having the Well Known UID clsDemoApp, with the method dispatch table,

```
new.object.pMsg          = clsDemoAppTable;
```

and the superclass (or ancestor object);

```
new.object.ancestor      = clsApp;
```

This particular demonstration application doesn't have any instance data, so

```
new.object.size          = Nil(SIZEOF);
```

sets its size to zero. Finally I indicate the size of the structure that will be used to create a new instance of clsDemoApp:

```
new.object.newArgsSize   = SizeOf(APP_NEW);
```

Once the NEW structure has been modified for the application object, it is sent as a parameter for the msgNew message that is sent to class clsAppMgr via the message:

```
ObjCallJmp( msgNew, clsAppMgr, &new, s, Error );
```

At that time, clsAppMgr registers clsDemoApp for you and proceeds to make everything ready to finish the installation of the application.

Including External Definitions In most cases, you will use constructs (structures, #define macros, and so on) defined by interface files. At the beginning of the demonstration application's source file are the preprocessor directives:

```
#ifdef APP_INCLUDED
#include <app.h>
#endif
```

```
#ifdef APPMGR_INCLUDED
#include <appmgr.h>
#endif

#ifdef DEBUG_INCLUDED
#include <debug.h>
#endif

#include "method.h"
#include "demoapp.h"
```

Like most applications, the demonstration application includes the following definition files:

app.h	Defines the API for the Application Framework class clsApp.
appmgr.h	Defines the API for the Application Manager class clsAppMgr
debug.h	Defines a set of support routines that enable the debugging of PenPoint applications.

Notice that each of the PenPoint definition files are surrounded by the construct:

```
#ifdef FILENAME_INCLUDED
#include <filename.h>
#endif
```

This construct saves compiling time by preventing the inclusion of a definition file more than once per compile. This construct depends on the definition file defining the token *FILENAME*_INCLUDED the first time the file is read.

In addition to the generic definition files included by the application source file, there are several definition files specific to the application itself. For the demonstration application, they are

method.h	Contains the declaration of the method table data structure used by clsApp.
demoapp.h	Defines the Well Known UID of the application class.

The number of application-specific definition files changes based on the requirements for the application class. For example, if your application class uses a label object, then it must include label.h from the PenPoint Includes directory to work correctly.

The Application Interface File

The demo application interface file, demoapp.h, contains one piece of information, the global Well Known UID. It is defined

```
#ifndef DEMOAPP_INCLUDED
#define DEMOAPP_INCLUDED

#ifndef GO_INCLUDED
#include <go.h>
#endif

#define clsDemoApp MakeGlobalWKN( 4140, 1)

#endif // DEMOAPP_INCLUDED
```

In practice, this interface file could easily be done away with by including the `MakeGlobalWKN ( )` macro in the demoapp.c source file itself. However, non-trivial examples sometimes need to export information to other classes in the application, and this is the ideal place to do it.

Notice that the first two lines in the file backup the convention defined to stop multiple inclusions of definition files:

```
#ifdef FILENAME_INCLUDED
#include <filename.h>
#endif
```

The conditional check and definition insure that even if the definition file is accessed again, its contents won't be processed more than once:

```
#ifndef DEMOAPP_INCLUDED
#define DEMOAPP_INCLUDED
```

The Method Table

The third source code file you must provide to complete the demo application is the method table for the clsDemoApp application class. The method table definition discussed in Chapter 2 consists of two parts in a separate file named method.tbl. The first part is

```
MSG_INFO clsDemoAppMethods[] = {
  0
};
```

It's used to specify the name of a message and the actual location of the behavior to use when that message is sent to the object. In the case of clsDemoApp, no methods are defined; instances of clsDemoApp rely completely upon inherited behavior to implement their functionality.

The method table compiler, mt.exe, uses the second part of the file to construct the method table data structure that PenPoint expects to work with internally by mapping the table name to the structure containing the dispatch table:

```
CLASS_INFO classInfo[] = {
  "clsDemoAppTable",       clsDemoAppMethods, 0,
  0
};
```

There are no restrictions on the number of method/message translation tables that can be specified in a CLASS_INFO structure, and no limit to the number of classes that can be described in one method.tbl module.

The output from running mt.exe on a method table is a definition file that contains declarations used to reference each method translation table defined in the file, and method prototypes for each of the methods referenced in the method tables.

method.h for the demonstration application looks like this:

```
//  WARNING: DO NOT EDIT this file.
//  WARNING: File generated by Penpoint 386 Method
Compiler.

#include <clsmgr.h>

extern const U32 clsDemoAppTable[];
```

There are no method prototypes because the message translation table didn't define any. If you were to modify the demonstration application class to override the method that responds to the msgAppInit message so that the method table for the class now reads

```
MSG_INFO clsDemoAppMethods[] = {
  msgAppInit,"DemoAppAppInit",
    callAncestorBefore,
  0
};
```

then the output from running mt.exe on the new method.table would be

```
//  WARNING: DO NOT EDIT this file.
//  WARNING: File generated by Penpoint 386 Method
Compiler.

#include <clsmgr.h>

MsgHandler(DemoAppAppInit);

extern const U32 clsDemoAppTable[];
```

Application Identification Information

The last bit of information needed to make a fully functioning PenPoint application is provided at link time. This information links the name given to the executable created with MS-DOS (demoapp.exe) to an application name and a load module name used by PenPoint.

The application name is the name that PenPoint displays to the user during installation and de-installation of the application. This name is not bound by the eight- and three- character rule and can contain spaces. It is given to the application by the Stamp utility that comes with the PenPoint SDK.

PenPoint uses the load module name to associate a company and version number along with a unique module identifier to a particular executable. The format of this identifier is

```
companyName-moduleName-majorVersion(minorVersion)
```

Using this format, the load module name for the demonstration application would be

```
pip-demoapp-V1(0)
```

Version information is kept so that PenPoint can detect version mismatches between objects and the classes used to create them.

Compile-time Debugging Support

One of the most underrated areas in application development is support for debugging application code. Most programmers are happy enough when they hear that they can print messages to a console or log file from anywhere in their code. Fortunately, this is often enough to lead the programmer to a problem's source. There are times, however, when it becomes necessary to use dedicated software for aiding in application development—time to turn to a source level debugger.

In addition to msgDump discussed in the last chapter, PenPoint's SDK supports a functional interface to a set of debug flags with a dedicated output stream for debug messages. The functional approach to debugging mimics the way C programmers tend to use printf() statements to dump information, while the source level debugger allows you to test and modify your code without going through the arduous cross development cycle.

The interface for PenPoint's compile time debugging support functions can be found in the debug.h include file. This file supplies three types of coding support for application debugging. First, it supplies a set of macros that allow you to selectively include or remove debugging information from your code. Second, it provides a set of functions that enable you to send messages to a dedicated output stream. Third, it provides a functional interface to a globally available set of debug flags.

The Debug Macros

debug.h defines several macros that allow you to have debugging calls in your code, but to turn them off at compile time using the define preprocessor directive. The mechanism in place uses the token DEBUG to check if code should be included for compiling. It is your responsibility to define DEBUG as one of the parameters passed to the compiler.

Dbg() The first macro, `Dbg()`,is defined

```
#ifdef DEBUG
#define Dbg(x) x
#else
#define Dbg(x)
#endif
```

The macro takes a single parameter, a statement to be executed inside the parenthesis. If DEBUG is defined, its contents replace the macro; otherwise, it evaluates to the empty string.

For example, if you wanted to enable a msgDump of some object only during a debugging session, you would specify

```
Dbg( ObjectCallWarn( msgDump, someObject, someLevel ); )
```

ASSERT() ASSERT(), the second macro provided by debug.h, is defined

```
#ifdef DEBUG
#define ASSERT(cond, str)((void)(!(cond) ? \
    (Debugf("==> ERROR, File: %s, Line: %d ==> %s\n", \
        __FILE__, __LINE__, str)),1: 0))
#else
#define ASSERT(cond, str)
#endif
```

The ASSERT() macro checks the validity of an assertion (cond) at some point in your code and will print an error message composed of the specified string (str), file name (__FILE__), and line number (__LINE__) at which the error occurred if the assertion fails. It also returns a 1 if the assertion fails and a 0 otherwise, so you can take appropriate action by placing the macro inside an if statement. __LINE__ and __FILE__ are predefined, set by the preprocessor to apply the value of the current line number and file name.

The ASSERT() macro can be used to implement whitebox testing, that is, testing done to a component by itself to make sure its integrity stays intact. This can be very useful if you are using someone else's objects and expect them to behave one way, when in reality they were not designed to do so.

For example, suppose you're working with a string object that you think is of variable width, when in fact the string object uses a fixed size buffer. The implementer of the string object might have included an assertion at the start of the method that handles the msgSetNewString message that looks like this:

```
if (ASSERT((strlen(pData->pStr)<FBUFF_SIZE),
            "buffer overflow") )
  ; \\ do something that responds to the error condition.
```

If you send a message with a string parameter that exceeds the appropriate length, *and you have a version of the String class build with the DEBUG token defined,* the following message would be sent to the output debug stream:

```
==> ERROR, File: String.c, Line: 57 ==> buffer overflow
```

followed by the action specified in the if statement being executed.

DbgFlag() The last macro works in conjunction with a set of flags kept by PenPoint to help with the debugging task. This macro, DbgFlag(), is defined

```
#ifdef DEBUG
#define DbgFlag(f,v,e) if (DbgFlagGet(f, v)) e
```

```
#else
#define DbgFlag(f,v,e)
#endif
```

This macro uses the DbgFlagGet() function to check if the specified flag is set. If it is, the statement represented by e is executed; otherwise nothing happens.

The Debug Functions

PenPoint provides two types of debug functions. The first is a function that sends formatted information in ASCII form to the debug output stream. The second works in conjunction with a set of system flags to implement a global scheme for application monitoring and debugging.

Debugf() The `Debugf()` function is used to write information to the dedicated debug output stream in the same manner as `fprintf(stderr, ...)` would be used in other C-based programming environments. The definition of `Debugf()` is

```
void CDECL Debugf( char * str, ...);
```

where str specifies the output to the debug stream. str can contain formatting characters used to insert the value of the parameters following str in the function call. If you're not familiar with the C `fprintf()` function, its types of parameters include strings, signed and unsigned numbers in multiple (decimal, hexidecimal, and other) formats.

For instance, suppose you want to leave a trail of debug messages inside your application to display the various activities of an instance of your application. You could insert this statement at the beginning of each of your routines:

```
Dbg(Debugf("msgFoo, inside file %s", __FILE__);)
```

This causes the appropriate log message to be printed whenever you define DEBUG during the compile.

The Debug Flags PenPoint also provides a means to control debugging information and actions on the fly through the use of a global set of debug flags. The flags are organized into 256 sets with each set having access to 32 bits of flags. Inside debug.h is a complete description of the flag ranges reserved by GO and those available to other developers, plus the various flags that are valid in each of the sets.

For example, the flag set used to interact with the debug system is known as set 'D' (hex value 0x44) and contains the following flags:

D0001	Disables all DebugStr output.
D0002	Disables StringPrint output.
D0004	Disables System Log output.
D0008	Disables System Log Non Error output.
D0010	Disables System Log App Error output.
D0020	Disables System Log System Error output.
D8000	Writes output to the penpoint.log file, flushed every *n* chars based on the environment variable DebugLogFlushCount.
D10000	Disables mini-debugger in production version of Penpoint.
D20000	Disables memory statistics gestures (M,N,T) on Bookshelf.

You can alter and/or check these flags at any time during the course of PenPoint's uptime using the `DebugFlagSet()` and `DebugFlagGet()` functions. They are defined

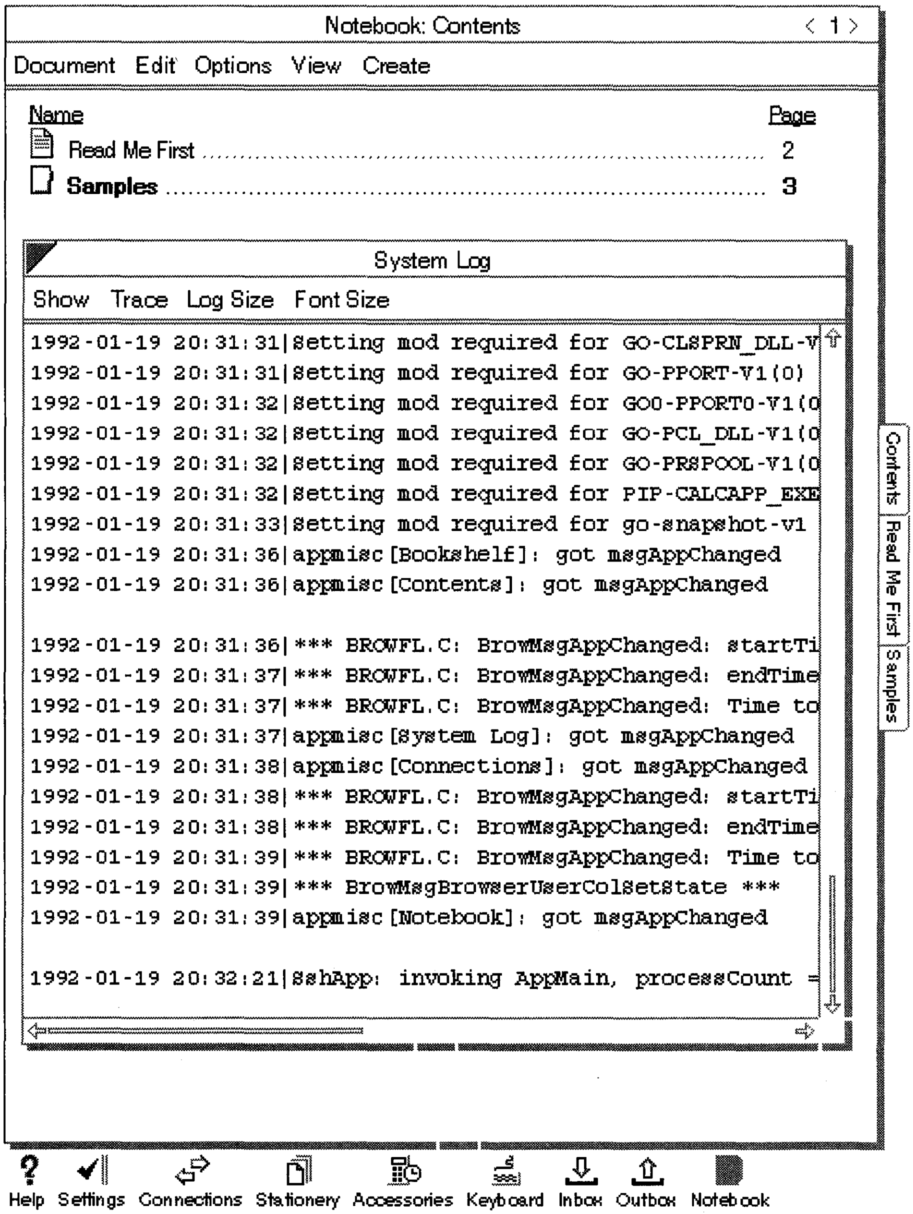
```
U32 EXPORTED DbgFlagGet(U16 set, U32 flags);
U32 EXPORTED DbgFlagSet(U16 set, U32 flags);
```

Notice that both functions use U16 to specify the flag set so that even though there are only 256 currently supported flag sets, the system can be expanded in the future.

The Messages Window

PenPoint provides a special application for viewing system information. The application System Messages shown in Figure 3.1 is located in the Accessory menu. This application can be used to display information such as memory usage, and error and non-error messages sent between applications. It also displays current lists of tasks in the system and devices that PenPoint knows about.

FIGURE 3.1 The System Log Window



This application watches the debug output stream and displays messages that pass its current set of filters. Since the update list can grow quite long, the application allows you to specify how much information is retained over time.

Tools for Debugging Applications

PenPoint supports two debuggers for use in application building. The first debugger is a mini-debugger that trades space and time requirements for reduced functionality. The second debugger, called DB, is a full symbolic debugger that you can use in application development. You tell PenPoint which debugger to use by placing the resources appropriate for the debugger you want in the boot.dlc file.

In general, you use the symbolic DB debugger during application development, when flexibility and ease of use (while working at your application code level) is of primary importance. Using DB incurs a performance penalty, however, because it requires space to manage the symbol tables and CPU cycles to work with symbolic (versus binary) information. You can reduce the performance penalty and still have the ability to collect information about an unexpected application or system failure by using the mini-debugger instead.

When an error occurs, PenPoint automatically invokes the appropriate debugger and causes the debugger to display information indicating what caused the exception. If an error occurs within PenPoint and no debugger is present, then the default action is to dump registers, print a stack backtrace, print task information on the debugging output stream, and then continue.

Mini-Debugger

The mini-debugger is a limited feature tool that enables you to collect information about a problem that might have occurred unexpectedly. Mini-db does allow you to collect information about tasks, heaps, segments, memory usage, and the file system. The biggest difference between the mini-debugger and DB is that you have to work at the assembly level when using mini-db. You can still set breakpoints, disassemble code, and modify registers, but you must do it at the assembler level; that is, you have to understand the underlying architecture of the platform to make it worthwhile.

However, you can always collect the pertinent information and then retire to another area to solve the problem.

You can also use the mini debugger to change the state of the system debug flags using the **fs** command. For instance, suppose you want to enable message logging. You enter mini-db by signalling the operating system (on a PC press the Pause key) and waiting for mini-db's prompt to appear. Next, to start the logging process, you enter:

```
fs /DD8000
```

When you finish logging information and want to turn off logging, you again enter mini-db and again type

```
fs /DD8000
```

Finally, the mini-debugger gives you a full list of its commands with cryptic explanations of what they do. This is useful, because mini-db seems to be entered when you least expect it. You access the help screen by typing "?" at the input prompt.

The DB Source Debugger

There is a certain stage in application development when problems occurring can only be solved by watching the state of the information change as each operation takes place. This can be accomplished by inserting debugf() statements after every line of code that dump the values of pertinent information. However, the problem with this approach is that in order to effect a change, you have to recompile, relink, re-install, and rerun the application. This is where a source level debugger is most useful.

In PenPoint, the ability to explore and modify an application is even more important because you are working in a cross development environment, where in addition to the standard development cycle, you might also have to add time to resume MS-DOS and reload and run PenPoint before you can try again. DB provides you with the ability to set breakpoints, set and get values, look at the task list, and so on, all at the symbolic level.

What follows is a brief synopsis of DB. If you're an experienced debugger user, I suggest you read through the DB manual to familiarize yourself with its full set of capabilities and features. If you've never used a symbolic debugger before, I recommend you take a few minutes to do the examples outlined in the first several chapters of the DB manual. They'll give you a feel for what a debugger can provide.

Getting Ready to Use DB You need to be aware of three basic steps when preparing to use DB. First, you need to compile and link your application so that the full set of symbols are maintained. Second, you need to add the Dynamic Link Libraries that implement DB to the files to be loaded in BOOT.DLC. Third, you need to specify that the debug versions of the PenPoint objects be loaded at runtime instead of the normally loaded production objects.

If you are going to work with the same application over an extended period of time, you can do several things to make your work easier. First, you can specify the location of a default file that DB should use when it's loaded to initialize its environment. This is done in the environ.ini file by adding the line

```
DBIni=\\dev_vol\apps\anapp\anappdb.ini
```

where dev_vol is the volume name of the device on which you are doing your application development.

Then, you can place a separate db.ini file in each application development directory that you might wish to debug. That way, you only have to change the pointer in environ.ini to customize DB for a particular application.

For example, to debug the demo application, you place the line

```
DBIni=\\dev_vol\apps\demoapp\demodb.ini
```

in your environ.ini file. Then the \\dev_vol\apps\demoapp directory would be the file demodb.ini containing the lines

```
sym "demoapp"
srcdir "demoapp" \\dev_vol\apps\demoapp
```

which load the appropriate set of symbols to debug DemoApp, plus would provide DB with a location to look for source files referenced in the symbolic information tables.

DB Contexts The basic operation mode for DB is based on the requirement that DB always debugs applications relative to a specific context. For DB, a context is composed of a current task, a current call, and the current scope. The current task is represented by an ID task and contains the address of the code currently executing. The current call references the stack frame of the currently active function. Finally, the current scope represents the name scope in which to look up identifiers used in commands.

You set the current context by giving the ID task followed by the **ctx** command. DB acknowledges that it is in a particular context by displaying the executable module name and the ID task as part of its prompt. To find out the ID task of the task you wish to debug, you can type "tl" (task list) while inside DB.

For example, if you are going to start debugging DemoApp, you first find DemoApp's task ID using the tl command. Suppose, for the sake of the example, that the task ID came back as 0x75 hex. You can then change the context by typing

```
> 075 ctx
```

DB would acknowledge the change by making its prompt read

```
"pip-demoapp-v1(0)"[0] 075>
```

The number in square brackets indicate the number of instances of that executable module that is currently registered with the system.

Examining the Current State Several sets of commands are available in DB for examining an application's current state. These commands provide functionality for evaluating the contents of a variable, looking at the contents of the stack, and looking at the source code that surround where the application is currently paused.

For example, the command for evaluating the contents of a data value is simply **?**. However, you can add modifiers to it that evaluate the data (either a physical address or a symbolic one) as a string pointer or a long int, for example. So, if pStr pointed to a block in memory that contained a null terminated string, you could display its contents by typing

```
> ? *pStr,s
```

The same flexibility lets you view the current line being executed either as source or as the disassembled instructions that the source line was compiled into.

Modifying Execution Behavior There are four basic ways to modify the execution behavior of an application. First, you can press the hard-wired interrupt key to pause the execution of the application and give control to the debugger. Second, using the **bp** command you can set a breakpoint so that when a particular function executes, the application pauses and the debugger takes control. Third, you can specify a watch point on memory, so that any time a particular location in memory is accessed, you are notified by the application stopping and control being given to the debugger. Finally, while in the debugger you can specify how execution is to proceed.

You can specify that execution continues in one of several ways. First, you can have the application continue to execute without stopping using the **g** command. Or, you can tell the debugger to use the **P**, **p**, **T**, or **t** to execute the next statement and come back when the statement is completed. Actually, you can even indicate to the debugger whether it should step into a function (that is, continue tracing the functionality, even through the subfunction) if the line to be executed contains a functional call (the **T** and **t** commands), or to step over the call and treat it as if it was a simple statement (the **P** and **p** commands). This level of control allows you to verify assumptions about how your application executes and whether or not the data value matches what you expected.

Wrap-up

When I sat down to write this book, I debated how much information to give on the mechanics of the compiler and linker. At first I thought "lots," but later I began to wonder. As you can tell, my final decision was to defer the discussion of the compiler to the compiler manual itself. GO has done a good job of predefining compiler/linker options through default rules passed to the Make utility, and I would suggest a copy and modify approach to building makefiles until you truly need something unique.

4

The Application Framework

Whenever a new technology appears, I categorize it based on the contributions it makes to various areas it interacts with. Once I establish categories, I rank the features contained in the categories according to their impact. There is no doubt that the pen-and-paper metaphor is the most striking contribution PenPoint makes in the user interaction category.

In the application developer's category, the choice for the number one spot is less clear. After all, PenPoint provides the application writer with a lot of help, including an object-based environment complete with several rich sets of predefined classes for tackling the various problems that come with writing an application. But for my money, the single most important feature of PenPoint from the application developer's point of view is its insistence that the application be written by extending a predefined Application Framework.

By insistence, I mean that GO has made it next to impossible for you to write a PenPoint application that doesn't work by extending the framework. Using the Application Framework forces you to trade a small amount of flexibility for the benefits of code reuse and consistency of operation. In my opinion that trade is worth it. First, code reuse means your applications don't take up extra space by re-implementing functionality already present in PenPoint's predefined classes. Second, consistency of operation means your applications behave in a well-defined manner so features such as embedded documents are available to all applications, even those developed by different vendors.

This chapter briefly touches on the history of application frameworks and how PenPoint fits in. It demonstrates the life cycle of a PenPoint application through the use of a tiny example. The example is then extended to illustrate the life cycle of a document (an instance of the application). Finally, a more complete application example illustrates how the application fits into the framework, and some of the functionality the framework provides to the application.

The Pre-framework Era

The first application I wrote for a "user-friendly" environment was for an early Apollo machine running Display Manager, and the process was a nightmare. On top of the operating system, I had to contend with a bit-map graphic display and a new set of user input events in which the location of the mouse was important. It took several pages of code just to create a window, display the text "Hello World," and allow the user to exit gracefully. But, the availability of a bit-map display and its benefits to the user interface made the effort worth it.

Two years later, I worked on a new product from Microsoft called Windows which promised to provide the PC with a better user interface. One thing was for sure: it added a lot more pages of code to the number needed to build a simple "Hello World" application. Two years later, I moved to the Apple Macintosh and, you guessed it, pages and pages of code to do the simplest tasks. What I found most frustrating in all three environments was that I was reusing the same code framework to build each application by applying the cookie cutter approach: copy framework, edit framework for new application, compile new application. I kept asking myself, "Why? If I'm using the same code, couldn't there be a way to reuse the bulk of it by sharing it among applications?"

As it turns out, I wasn't alone in my frustration. Several research groups, including one at Apple, also recognized the need for formalized reuse. The Apple group was able to use their knowledge of object-oriented programming to build Object-Pascal, a language which added the extensions necessary to support objects in Pascal. Once that tool was in place, they proceeded to build a set of classes for writing Macintosh applications that would do most of the work. Known as MacApp, this product provided a framework that programmers could customize to build their specific applications.

Based on reading GO's literature, it appears that their development staff was influenced by the work done at Apple and adopted many concepts present in MacApp. However, as I already mentioned, there is no alternative means of building a PenPoint application; you can't ignore the framework and interact with the user directly like you could in building

Macintosh applications. You must build PenPoint applications by extending the framework.

I don't know about you, but I'm always a little wary when someone sticks me with something "for my own good." It's not enough for the operating system vendor to say "trust me" and I'll immediately accept. I need reasons. Good ones. In the case of PenPoint, the Application Framework provides several, not just in what the framework does, but more importantly, in how the framework does them.

For example, the framework imposes consistent and *well-known* application and document life cycles within the Penpoint operating environment. Application writers know in advance when and how an application will be installed and removed. They also know in advance how an application will be told to save its state, to start itself, to gracefully exit itself, and to embed an instance of itself inside another running application.

Another significant difference between GO's framework and MacApp's is that GO chose not to embrace an object-oriented language such as Object-Pascal, Objective-C, or C++ for implementing their concept of objects for PenPoint. Instead they took the approach of having the programmer manage the object model for the application by hand coding the method tables. This means that PenPoint application code can look messy compared to its MacApp counterpart.

If this is your first exposure to object-based application frameworks, two other examples might be of interest to you in exploring this form of application development. First, you might want to read about ET++ which was written in C++ and based on MacApp. Second, you might look at NeXTSTEP which was written using Objective-C. Both these frameworks use C-based object technology to allow greater code reuse.

The Document Life Cycle

From the programmer's point of view, a PenPoint application exists in three forms. First, the application exists as source code that is compiled and linked. Once the application is successfully built, it enters its second form, that of installed application. Once installed, the application has a factory that can build the third form of existence, PenPoint documents which are instances of the installed application. The Application Framework is used by the programmer in its source form to build PenPoint applications that work consistently with other PenPoint applications.

The Application Framework supports application development by providing a well-defined protocol for informing PenPoint what an application needs on installation and how the instances of that application (the documents) should perform. In Chapter 3, the first Penpoint application defined only the behavior necessary to install the application and relied entirely on inherited behavior from clsApp to manage instances of the application. Once installed, the inherited behavior was used to create and manage user documents as they went through their individual life cycle.

The third and final component of the application framework concerns the life cycle of a PenPoint document.

States in the Document Life Cycle

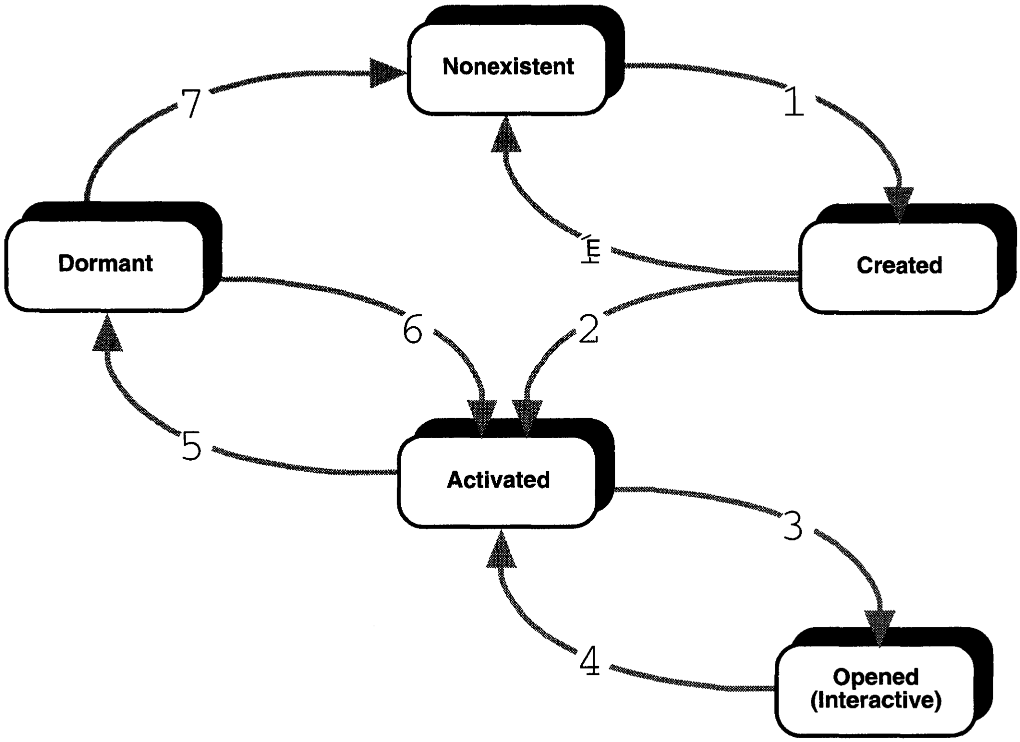
Each document is a separate instance of the application class and can be thought of as existing in one of five well defined states: nonexistent, created, activated, opened, and dormant. The document is cycled through these five states in response to actions by the user. Figure 4.1 shows the path of the document through each state, with the order of flow represented by the paths labeled with circled numbers. The next sections describe these states.

Created The first step in the document life cycle shows the framework creating an instance of the application class (the user's document). The framework uses the behavior in clsAppMgr to create a directory for the document and to save its stateful information (instance data) there. The application must insure that a document is created so that an error in this state allows the framework to remove the instance in a well-behaved manner. The document can move from this state to either the nonexistent state due to an error or to the activated state because of a user request.

Activated A document in the activated state is a completely functioning instance of an application, except it has no interactive interface with the user. A process exists for the document that contains the document object with that object's application data in a valid state. The application data was made valid either through initialization (path 2 from the created state) or by restoring it from previously saved data (path 6 from the dormant state).

When a document transits to the activated state from dormant or created, its next normal transition is to move to the opened state (path 3) so the user can interact with it. When a document transits back to the activated state from the opened state (path 4), its next logical transition is to dormant (path 5). One exception to this is when the document is kept in "hot" mode either by a user's request for efficiency reasons or an application's request because an operation must be completed before the document can release its thread and become dormant by not freeing the document's thread.

FIGURE 4.1 The States of a Document Life Cycle



Opened A document in the opened state is equivalent to one in the activated state with the addition that the document's process is given access to the display and the user is able to interact with it. A document is always placed in the opened state by a transition from the activated state (path 3) and always transits back to the activated state (path 4).

Dormant A document in the dormant state, like one in the created state, has a directory. In addition to the directory, a document in the dormant state also has a resource file. The document's application data (or object's state) is stored in this resource file in the PenPoint file system. Finally, no active process is associated with the application when it is in the dormant state.

A document in the dormant state can either be reactivated (path 6) when the user wishes to interact with the document or can transit to the nonexistent state (path 7) when the user removes the document from the PenPoint file system. A document will not transit from the dormant state to the nonexistent state when the user de-installs the document's application from the PenPoint file system. Instead, the document remains dormant and PenPoint prompts the user appropriately if the user attempts to turn to the document.

Nonexistent A document in the Nonexistent state has no representation within the PenPoint file system. A document can be transit to the nonexistent state from the dormant state (path 7) when the user removes the document from the Notebook's table of contents. A document also transits to the nonexistent state when an error occurs during the document's creation (path E).

DemoApp Updated

From your perspective as the programmer, each stage in the document life cycle is defined by a set of messages sent by the Application Manager to the PenPoint application. Most applications provide methods for handling one of more of the following seven messages: msgInit, msgSave, msgRestore, msgFree, msgAppInit, msgAppOpen, and msgAppClose. By overriding these messages, your new application class can insert the behavior it requires to work correctly before calling the inherited behavior. You can also rely on the default behavior inherited from the Application Manager for each of these messages to handle the framework's demands in a reasonable and consistent manner. You should note that the Application Manager sends the messages to the application object, so in addition to providing some default behavior, the Application Manager also provides some forms of housekeeping when errors occur.

I have extended the demonstration application so it correctly overrides each of the seven messages. Currently, each method prints a message to the debug log and then allows its ancestor's behavior to be executed in the correct sequence. The debug messages will appear on the debug log. The following sections describe each of the methods located in demoapp.c application source file.

msgInit Your application receives the msgInit message as a result of the user turning the page to a document from your application, which causes PenPoint to create a new instance of the application. This causes the document to pass into the the activated state. The code for this method is

```
MsgHandler(DemoAppInit)
{
  Debugf("DemoApp:DemoAppInit");
  return stsOK;
```

```
    MsgHandlerParametersNoWarning;
}
```

The method that responds to msgInit is responsible for creating and initializing the instance data for the application. This method receives no extra parameters in order to perform its function. It is important that you specify in the method dispatch table that your application's ancestor's msgInit method be called first. Let it do such housekeeping as opening the application's directory in the file system and updating its own instance data.

msgAppInit Your application is sent a special message, msgAppInit, the first time a user turns to the document. The code for this method is

```
MsgHandler(DemoAppAppInit)
{
    Debugf("DemoApp:DemoAppAppInit");
    return stsOK;
    MsgHandlerParametersNoWarning;
}
```

This is the time at which clsApp actually creates the resource file for the document. The application's method table should specify that its ancestor is called first in order for clsApp to also create the main window. This is where you set up the resources necessary to file and manage any stateful objects your application uses.

msgFree When the user turns away from the document supported by your application, the Notebook terminates the document by sending it the msgFree message (unless the document is in hot mode, in which it appears to go away but the process is still intact). This is the part of the framework that is responsible for moving the document from its dormant state to its nonexistent state. The code for this method is

```
MsgHandlerWithTypes(DemoAppFree, P_ARGS, P_INSTANCE_DATA)
{
    Debugf("DemoApp:DemoAppFree");
    return stsOK;
    MsgHandlerParametersNoWarning;
}
```

This method is responsible for freeing the instance data used by the application object, including freeing objects created to support the document. In the case of this method, its ancestor should be called after this method completes. Also, this method should always return `stsOK` to insure appropriate default behavior if a problem should occur during the freeing of the object. In the case of this method, the instance data is passed as a pointer to the method handling the message.

msgAppOpen PenPoint sends msgAppOpen to your application to signal the moving of the document from the activated to the opened state. Your application uses this method to create the windows and control objects needed to interact with the user. The sample code for this method is

```
MsgHandlerWithTypes(DemoAppOpen, P_ARGS, P_INSTANCE_DATA)
{
  Debugf("DemoApp:DemoAppOpen");
  return stsOK;
  MsgHandlerParametersNoWarning;
}
```

This method is the one that builds the user interface to your application for the document by creating and attaching objects to the main window. It is important that the method table specifies that the ancestor is called after your application's method completes so that the environment can be set up correctly.

msgAppClose Your application receives msgAppClose when the user turns away from the document (providing the document is not in hot mode). The code for this method is

```
MsgHandler(DemoAppClose)
{
  Debugf("DemoApp:DemoAppClose");
  return stsOK;
  MsgHandlerParametersNoWarning;
}
```

The method table for your application should specify that the ancestor's method is called first, followed by your application's processing. This method is used to destroy all stateless objects used by your application.

msgSave Your application object is sent msgSave to indicate that you should file your instance data, including other objects used by the document, in the file indicated. The code used to implement this method is

```
MsgHandlerWithTypes(DemoAppSave, P_OBJ_SAVE, P_INSTANCE_DATA)
{
  Debugf("DemoApp:DemoAppSave");
  return stsOK;
```

```
  MsgHandlerParametersNoWarning;
}
```

A parameter passed to the method indicates the file in which to save your data. You should set the method table so that your ancestor is called before your method. The framework uses this opportunity to automatically file for you information stored in the objects you reused from the PenPoint class library.

msgRestore The framework sends your application msgRestore when it wants you to recreate the object from saved data. The code for this method is

```
MsgHandlerWithTypes(DemoAppRestore, P_OBJ_RESTORE,
P_INSTANCE_DATA)
{
  Debugf("DemoApp:DemoAppRestore");
  return stsOK;
  MsgHandlerParametersNoWarning;
}
```

This method has the file used to restore the objects used by the application to support the document passed to it as an argument. The method table should be set up so that the ancestor is called before the new method is invoked.

Header Files Included from the SDK In order to work correctly, the demo application must access some of the information contained in the header files included in the Penpoint SDK. They are

```
#ifndef APP_INCLUDED
#include <app.h>
#endif

#ifndef APPMGR_INCLUDED
#include <appmgr.h>
#endif

#ifndef DEBUG_INCLUDED
#include <debug.h>
#endif

#include "DemoApp.h"
#include "method.h"
#include <string.h>
```

For the sake of the demonstration, I have defined an instance data structure, even though it isn't used, allowing you to see its placement relative to the rest of the source file. In addition, several method definitions require access to instance data for the job they normally do. In this example, no instance data is needed, but the definition is included to placate the C compiler. The stub definition used is

```
typedef struct INSTANCE_DATA {
  U32 dummy;
} INSTANCE_DATA, *P_INSTANCE_DATA;
```

main() Revisited I have changed the `main()` and the `ClsDemoAppInit()` functions so they output debug messages to show how the flow of execution passes through them. The changes are noted in **bold** text.

```
STATUS ClsDemoAppInit(void)
{
  APP_MGR_NEW new;
  STATUS      s;

  Debugf("DemoApp:ClsDemoAppInit");

  ObjCallJmp(msgNewDefaults, clsAppMgr, &new, s, Error);

  new.object.uid      = clsDemoApp;
  new.cls.pMsg        = clsDemoAppTable;
  new.cls.ancestor    = clsApp;
  new.cls.size        = SizeOf(INSTANCE_DATA);
  new.cls.newArgsSize = SizeOf(APP_NEW);
  strcpy( new.appMgr.company, "PenPoint Programming" );
  strcpy( new.appMgr.defaultDocName, "Framework Demo App" );

  ObjCallJmp(msgNew, clsAppMgr, &new, s, Error);

  return stsOK;

Error:
  return s;
}


void CDECL
main(
  int   argc,
  char *argv[],
  U16   processCount)
```

```
{
  STATUS s;

  Debugf("DemoApp:main");

  if (processCount == 0) {
      Debugf("DemoApp:main-processCount=0");
      ClsDemoAppInit();
      AppMonitorMain(clsDemoApp, objNull);
      }
  else {
      Debugf("DemoApp:main-processCount>0");
      AppMain();
      }

  Unused(argc); Unused(argv);
} /* main */
```

Updated Method Table It is necessary to update the method dispatch table to reflect the changes made in the application object definition. These changes add the preprocessor directives that include the definition files necessary for the table to build correctly. The source code for the method table is

```
#ifndef CLSMGR_INCLUDED
#include <clsmgr.h>
#endif

#ifndef APP_INCLUDED
#include <app.h>
#endif

#ifndef FWAPP_INCLUDED
#include <demoapp.h>
#endif

MSG_INFO clsDemoAppMethods[] = {
  msgInit,                "DemoAppInit",
objCallAncestorBefore,
  msgSave,                "DemoAppSave",
objCallAncestorBefore,
  msgRestore,             "DemoAppRestore",
objCallAncestorBefore,
  msgFree,                "DemoAppFree",
objCallAncestorAfter,
  msgAppInit,             "DemoAppAppInit",
objCallAncestorBefore,
  msgAppOpen,             "DemoAppOpen",
```

```
objCallAncestorAfter,
  msgAppClose,          "DemoAppClose",
objCallAncestorBefore,
  0
};
CLASS_INFO classInfo[] = {
  "clsDemoAppTable",    clsDemoAppMethods,0,
  0
};
```

If you recall from the previous paragraphs, each method that is overriden requires that its ancestors be called before or after. This information is specified in the third parameter of the method table entries.

The additional definition files specified at the beginning of the method table file are needed because they contain definitions for some of the messages that Demo App is overriding.

Running DemoApp

The following paragraphs outline the results of installing the correctly built DemoApp application in a running PenPoint environment. The output shown was gathered by turning on the debug logging functionality and saving the resulting messages in a file.

The first step is for the user to install the application, resulting in the following debug output from the DemoApp application:

```
Loader: Loading pip-demoapp-v1(0)
DemoApp: main
DemoApp: main-processCount=0
DemoApp:ClsDemoAppInit
```

As expected, the `main()` function is called with processCount = 0, indicating that the application should initialize any classes needing to be registered with the Class Manager. In this case, it only initializes the application object.

The user's next action is to create an empty document for the installed application and add it to the Notebook. The result is—Nothing! It is not necessary for the application to handle any messages when the user creates the document.

Next, the user turns to the document that was just created. The resulting debug information is

```
DemoApp:main
DemoApp:main:processCount>0
```

```
DemoApp:DemoAppInit
DemoApp:DemoAppAppInit
DemoApp:DemoAppOpen
DemoApp:DemoAppSave
```

The application's `main()` routine is called with the value of processCount greater than 0, indicating that the user is turning to a new document that has been created, but is currently not activated. The application receives the msgInit message followed by msgAppInit message, since it's the first time the document is being opened. Next, the framework sends the messages to the application necessary to create the objects that will be used to display the document on the screen for the user to interact with. Finally, the application is told to save the document's data to the PenPoint file system for the first time.

At this point, the user begins to interact with the application. Since DemoApp is a no-functionality type application, there is nothing interesting for the user to do. So the user turns back to the table of contents. At that point, the following debug information appears.

```
DemoApp:DemoAppClose
DemoApp:DemoAppSave
DemoApp:Free
```

PenPoint notifies your application that the user is turning away from the document by sending the msgAppClose message to it. It then sends the msgSave message so that the application object can save the appropriate state data (instance variables, other objects, etc.) in the resource file. Finally, the framework frees the data associated with displaying the document to the user. This set of messages would be different if the user was keeping the document in hot mode.

The next time the user turns to the document, the application's execution path is

```
DemoApp:main
DemoApp:main-processCount>0
DemoApp:DemoAppInit
DemoApp:DemoAppRestore
DemoApp:DemoAppOpen
```

This set of trace information is the same as the first time the user turns to the document, except msgAppInit does not need to be received a second time. Again, the sequence of events would have been different if the user was opening a document that was being kept in hot mode.

The last thing the user does in the life cycle of the document is to remove it from the table of contents. In the case of our demonstration application, all this processing is done through behavior inherited from the Application Framework, so there are no logged events.

CoinApp

Now that the various functions of the application life cycle have been discussed, I would like to close this chapter by implementing a simple application, CoinApp, that changes the state of the display whenever the user turns from the document. The application shows its state by displaying the words "Heads" or "Tails" when the user turns to the page of the document. This example gives you a chance to understand what is involved with saving the state of the application even as it goes through the application and document life cycles.

method.tbl

Again, the application consists of three files: the source and definition files for the coin application's class contained in CoinApp.c and CoinApp.h respectively and the method table definitions contained in method.tbl. You will notice that this application overrides all the messages, except msgFree described in the last section, so it can save and restore its stateful data correctly. The method.tbl method table definitions for the CoinApp class are

```
#ifndef CLSMGR_INCLUDED
#include <clsmgr.h>
#endif

#ifndef APP_INCLUDED
#include <app.h>
#endif

MSG_INFO clsCoinAppMethods[] = {
  msgInit,               "CoinAppInit",
  objCallAncestorBefore,
  msgSave,               "CoinAppSave",
  objCallAncestorBefore,
  msgRestore,            "CoinAppRestore",
  objCallAncestorBefore,
  msgAppInit,            "CoinAppAppInit",
  objCallAncestorBefore,
```

```
    msgAppOpen,           "CoinAppOpen",
    objCallAncestorAfter,
    msgAppClose,          "CoinAppClose",
    objCallAncestorBefore,
    0
};

CLASS_INFO classInfo[] = {
    "clsCoinAppTable",clsCoinAppMethods,0,
    0
};
```

coinapp.c

coinapp.c contains the method definitions for the clsCoinApp class and the `main()` entry point for the coin application. I have listed the routines in the order that the framework sends messages to them. Again I remind you that no method is defined for responding to msgFree. Instead, CoinApp relies on the behavior inherited from clsApp.

The beginning of the file contains the necessary include directives to provide the compiler with the definitions needed to compile the module. They are

```
#ifndef APP_INCLUDED
#include <app.h>
#endif

#ifndef APPMGR_INCLUDED
#include <appmgr.h>
#endif

#ifndef OS_INCLUDED
#include <os.h>
#endif

#ifndef RESFILE_INCLUDED
#include <resfile.h>
#endif

#ifndef FRAME_INCLUDED
#include <frame.h>
#endif

#ifndef DEBUG_INCLUDED
#include <debug.h>
#endif
```

```
#include "coinapp.h"
#include "method.h"
#include <string.h>
```

Instance Variables Following the include directives, come the instance data declarations:

```
typedef enum COIN_STATUS {
  heads, tails
} COIN_STATUS;

typedef struct INSTANCE_DATA {
  COIN_STATUS coin;
} INSTANCE_DATA, *P_INSTANCE_DATA;
```

In the case of CoinApp, class clsCoinApp has one instance variable, coin, which contains the state the coin should be "displayed" in (heads or tails) when the user turns to the document.

CoinAppInit The first method defined in the module is CoinAppInit, which responds to msgInit, after its ancestor is called. This method is defined as

```
MsgHandler(CoinAppInit)
{
  INSTANCE_DATA inst;

  inst.coin = heads;
  ObjectWrite(self, ctx, &inst);

  return stsOK;
  MsgHandlerParametersNoWarning;
}
```

This method is used to initialize the values of the instance data so that if a problem in the document set up occurs, PenPoint can remove its contents and not worry about an undefined state. Since the coin instance variable is an enumerated type, it is ok to arbitrarily assign it to one of the two valid states. Next, `ObjectWrite()` is used to update the instance data. This function must be used because the instance data is kept in an area of memory protected by the operating system. `ObjectWrite()` takes three parameters: self, a pointer to the object responding to the message; ctx, the context of the object receiving the message; and a pointer to the instance data that has been modified in the local memory.

CoinAppSave The CoinAppSave method responds to msgSave, after its ancestor is called. This method is defined as

```
MsgHandlerArgType(CoinAppSave, P_OBJ_SAVE)
{
  STREAM_READ_WRITE     fsWrite;
  STATUS    s;

  fsWrite.numBytes = SizeOf(INSTANCE_DATA);
  fsWrite.pBuf = pData;
  ObjCallRet(msgStreamWrite, pArgs->file, &fsWrite, s);

  return stsOK;
  MsgHandlerParametersNoWarning;
}
```

The CoinAppSave method requires one parameter, a pointer to the information that specifies the object to be used in saving the application's instance data. This method sets up the members of a `STREAM_READ_WRITE` structure to indicate where the instance data begins and how many bytes it contains. Although `pData` is not visibly defined in the source code, the `MsgHandlerArgType()` macro automatically does it for you. After the structure is filled in, it is sent as a parameter to the file object specified in the input using the msgStreamWrite message.

CoinAppRestore The CoinAppRestore method responds to msgSave, after its ancestor is called. This method is defined

```
MsgHandlerArgType(CoinAppRestore, P_OBJ_RESTORE)
{
  INSTANCE_DATA         inst;
  STREAM_READ_WRITE     fsRead;
  STATUS                s;

  fsRead.numBytes = SizeOf(INSTANCE_DATA);
  fsRead.pBuf = &inst;
  ObjCallRet(msgStreamRead, pArgs->file, &fsRead, s );
  ObjectWrite(self, ctx, &inst);

  return stsOK;
  MsgHandlerParametersNoWarning;
}
```

This method is used to read the value of the document's application instance data from the data stored before the document was made dormant. The `STREAM_READ_WRITE` structure is initialized to contain the size of the instance data and a pointer to the space to copy the specified bytes into. Next, the instance data is written to the official copy kept in protected memory.

CoinAppAppInit The CoinAppAppInit method responds to msgAppInit, after its ancestor is called. Recall that msgAppInit is only called the first time the user turns to the document. This method is defined as

```
MsgHandler(CoinAppAppInit)
{
  INSTANCE_DATA          inst;
  STATUS                 s;

  inst = IDataDeref(pData, INSTANCE_DATA);
  inst.coin = heads;
  ObjectWrite(self, ctx, &inst);

  return stsOK;
  MsgHandlerParametersNoWarning;
}
```

This method sets the initial state of the coin to heads the first time the user turns to the document. It uses the macro `IDataDeref()` to place a copy of the instance data from the protected area of memory into the local memory area where it can be modified. Next, the instance variable coin is set to the heads value. Finally, the instance data is written back to the protected area via a call to `ObjectWrite();`.

CoinAppOpen The CoinAppOpen method responds to msgAppOpen. Its ancestor is called after it completes its processing. This method is defined

```
MsgHandlerWithTypes(CoinAppOpen, P_ARGS, P_INSTANCE_DATA)
{
  APP_METRICS          am;
  LABEL_NEW            ln;
  STATUS               s;

  ObjCallRet(msgNewDefaults, clsLabel, &ln, s);
  ln.label.style.scaleUnits = lsScaleFitWindowProper;
  ln.label.style.xAlignment = lsAlignCenter;
  ln.label.style.yAlignment = lsAlignCenter;
  ln.label.pString =
  (pData->coin == heads) ? "Heads" : "Tails";
  ObjCallRet(msgNew, clsLabel, &ln, s);
```

```
    ObjCallJmp(msgAppGetMetrics, self, &am, s, Error);

    ObjCallJmp( msgFrameSetClientWin, am.mainWin,
                         ln.object.uid, s, Error);

    return stsOK;
    MsgHandlerParametersNoWarning;

  Error:
    return s;
  }
```

Notice that this method uses a different method definition macro, `MsgHandlerWithTypes()` that lists `P_INSTANCE_DATA` as one of its types. Unlike the other methods that cast `pData` inside a macro or function used by the method, the CoinAppOpen method does so by including it in the list of types in the method header. This casts `pData` to be a pointer to the protected copy of the instance data allowing it to be used directly by the method in a read-only capacity. The only problem was that in order to use `MsgHandlerWithTypes()`, I had to specify a type for the middle parameter. So I used the default P_ARGS provided by the SDK.

CoinAppOpen is responsible for creating the objects that the document needs to interact with the user. In the case of CoinApp, this is simply a label that displays the coin's status. In order to create an instance of clsLabel, a message is first sent to it asking to fill in a `LABEL_NEW` structure with a reasonable set of default values. CoinAppOpen then modifies the scale and alignment of the label object so it is centered and takes up the entire window. Next, it sets the value of pString, the pointer used to initialize the value of the label, to one of two strings (`"Heads"` or `"Tails"`) based on the current state of the coin instance variable. Once the structure is complete, it is sent as a parameter to the Class Manager to create a new instance of the clsLabel class.

The final two messages are used by the application to set the label object as the client of the application's main window. First, a message is sent to self that invokes behavior necessary to fill in the `APP_METRICS` structure passed to it as a parameter. Next, the label object is passed as a parameter in the msgFrameSetClientWin message, which is sent to the main window object returned by the msgAppGetMetrics message. These last two messages are sent via the `ObjCallJmp()` macro which causes the flow of control to go immediately to the code following the label "`Error`" if `stsOK` is not returned.

CoinAppClose The CoinAppClose method responds to msgAppClose, after its ancestor is called. This method is defined

```
MsgHandler(CoinAppClose)
{
  INSTANCE_DATA inst;

  inst = IDataDeref(pData, INSTANCE_DATA);
  inst.coin = (inst.coin == heads) ? tails : heads;
  ObjectWrite( self, ctx, &inst );

  return stsOK;
  MsgHandlerParametersNoWarning;
}
```

The first action this method takes is to create a local copy of the instance data that can be modified. It then modifies the information and writes it back into the protected area of memory. This method is made necessary by the requirement that the status of the coin (heads or tails) changes when the user *closes* the document. It would be possible to avoid this method entirely by updating the value of coin in CoinAppOpen after the label object is set, but it would be incorrect since the specification says it should be done when the user closes the document.

ClsCoinAppInit The `ClsCoinAppInit()` function is very similar to those used in previous examples. It's defined

```
STATUS ClsCoinAppInit(void)
{
  APP_MGR_NEW      new;
  STATUS           s;

  ObjCallJmp(msgNewDefaults, clsAppMgr, &new, s, Error);

  new.object.uid      = clsCoinApp;
  new.cls.pMsg        = clsCoinAppTable;
  new.cls.ancestor    = clsApp;
  new.cls.size        = SizeOf(INSTANCE_DATA);
  new.cls.newArgsSize = SizeOf(APP_NEW);

  strcpy(new.appMgr.name, "Coin Application");
  strcpy(new.appMgr.company, "PenPoint Programming");

  ObjCallJmp(msgNew, clsAppMgr, &new, s, Error);

  return stsOK;
```

```
Error:
  return s;
}
```

Two new things are shown in this example. First, since the CoinApp class has instance data, the value of the initialization structure is changed to reflect the size of the instance data being used.

```
new.class.size = SizeOf(INSTANCE_DATA);
```

Second, in addition to the company name being specified, the application name itself is being specified using the statement:

```
strcpy(new.appMgr.name, "Coin Application");
```

main() The `main()` function for the coin application is a copy-and-paste version of the same `main()` function used in previous examples. It's defined

```
void CDECL
main( int argc, char *argv[], U16 processCount)
{
  STATUS s;

  if (processCount == 0) {
    ClsCoinAppInit();
    AppMonitorMain(clsCoinApp, objNull);
    }
  else
    AppMain();

  Unused(argc); Unused(argv);
}
```

coinapp.h

In addition to the class and method definitions contained in coinapp.c, there are the definitions included in coinapp.h. In this example, coinapp.h is used to define a Well Known UID for the coin application. It's defined

```
#ifndef COINAPP_INCLUDED
#define COINAPP_INCLUDED

#ifndef CLSMGR_INCLUDED
#include <clsmgr.h>
#endif
```

```
#ifndef UID_INCLUDED
#include <uid.h>
#endif

#define clsCoinApp MakeGlobalWKN( 4141, 1 );
#endif
```

Installing and Using CoinApp

After CoinApp is built and installed, the user can create a document based on the application. The first time the user turns to the document, it displays the text "Heads" so that the text is centered and takes up the entire document window. The next time the user turns to this document, it will display the text "Tails" in the same fashion. The document's display toggles between these two values until the user decides to remove the document from the machine. If you're interested in watching the flow of execution, you can add debugging statements to the beginning of each method definition so you will know when the method is being invoked.

Debugging CoinApp

Now would be an interesting time to step back and take a second look at the symbolic debugger. I would like to demonstrate a brief debugging session using the techniques discussed in the last chapter for preparing an application for debugging, and then actually use the debugger to explore the application.

The first step in using DB with the coin application is to insure that coinapp.exe has been built to contain all the necessary symbolic information. Next, check the boot.dlc file to insure that the proper DLLs have been or will be loaded for DB to work. Finally, once PenPoint is running, install CoinApp and then press the interrupt key (Pause key on PCs). You should now be inside the debugger.

Choosing a Context It is necessary to specify the context of the coin application before you can begin debugging it using symbolic information. To get the ID task , you type

```
> tl'
```

and watch for the task ID associated with pip-coinapp-v1(0).

For the sake of the example, assume that the task ID for the coin application is 07. You would type

```
> 07 ctx'
```

and DB would respond

```
"pip-coinapp-v1(0)"[0] 07>
```

indicating it is ready to work with the symbols from the specified task.

Next, you need to tell DB where to find the symbolic and source information for the coin application. This is done by entering the commands

```
"pip-coinapp-v1(0)"[0] 07> sym "coinapp"
"pip-coinapp-v1(0)"[0] 07> srcdir "coinapp"
\\dev_vol\penpoint\coinapp
```

Now you are ready to begin debugging.

Setting a Breakpoint Setting a breakpoint for the CoinAppAppInit method and then starting the application is as simple as typing

```
"pip-coinapp-v1(0)"[0] 07> bp CoinAppAppInit
```

followed by

```
"pip-coinapp-v1(0)"[0] 07> g
```

You are notified when you create a document by the debugger showing you a status update that a breakpoint has been encountered. You can then use the various commands to access information about the application in general.

A Quick Look at CoinAppAppInit Once you stop at the breakpoint, you can type v to see an abbreviated listing of the code that surrounds the breakpoint. For example, after typing v, you would see

```
STATUS ClsCoinAppInit(void)
{
  APP_MGR_NEW   new;
  STATUS        s;

  ObjCallJmp(msgNewDefaults, clsAppMgr, &new, s, Error);
```

You can then choose to explore the contents of various data values or continue execution.

When you're done debugging CoinAppAppInit, you can remove the breakpoint by typing bl for a list of current breakpoints and then type bc followed by a number to remove the one that represents CoinAppAppInit.

You might consider taking a few moments to use DB with CoinApp to get a feel for how the symbolic debugger performs.

Wrap-up

When you think about the application classes role in the management of documents, it might be helpful to view it as the factory that creates documents. It assembles the components into the document itself, but the individual components do the work. Working with this mind-set, you can create objects your application needs that work within the framework of PenPoint and therefore can be reused by others.

This chapter provided a large amount of information fundamental to understanding application writing in PenPoint. It started by discussing the life cycle of a document, or instance of an application, that the user interacts with. It discussed the messages sent as the document moves through its life cycle states of nonexistent, created, activated, opened, and dormant by creating a simple application that overrode the messages sent to the application and printed out debugging information that showed the flow of execution based on user actions.

Finally, it provided an example of an application that managed a single instance variable through all phases of the document life cycle. This application, called CoinApp, managed the display of the state of a hypothetical coin to the user each time the user turned to the document. Although contrived, CoinApp gives a more realistic feel for working with the full set of components in a real application.

As I mentioned in the beginning, of all that PenPoint provides the programmer, the use of an Application Framework is most significant in encouraging productivity and quality gains. I hope by now that even if you don't agree with me, you have been able to grasp the potential available when all applications behave in a consistent manner by responding to a set of well-defined protocols.

5

The Calculator Example

Ever been to a coffee pot round table? It's the type of discussion that happens when people are at the coffee machine taking a break and someone puts forth an idea that just begs to be challenged. I recently was part of such a round table during a class on object-oriented design and analysis that ended up as a debate on the issue of how much time should be spent on pre-coding, versus actual coding. The collective wisdom present pronounced that 75 percent of the effort should go into the requirements, analysis, and design phases before any code is written, with the rest of the effort devoted to coding and testing. The catch with this statement is that it assumes you know enough about the components of your design to accurately schedule time for their implementation. If not, you could be building an application based on so many unknowns that the project might never be complete.

Of course, in the ideal world where schedules aren't an issue, you would do the perfect design, complete your code, and then look back at the project to compute the design-to-coding ratio. Unfortunately, most of us live in a world where schedules are very much a reality and often find ourselves cutting design time to insure that code will be completed on schedule. My experience with object-based software indicates that this problem can be greatly reduced by providing standards for coding and reuse whenever possible.

PenPoint offers the application designer and developer a lot of help in this area by providing a consistent Application Framework plus a large number of reusable classes to draw on when building new applications.

For example, when building a user interface, you could always use a generic component to get your application up and running and then customize it later. PenPoint supports you in these activities by providing a wide choice of customizable wigits that fit almost any need, plus the inheritance facility for easily extending the behavior of a particular type of object you might want.

The first four chapters of this book deal with the basics of creating generic PenPoint applications. In this chapter, I'm going to begin introducing the concepts needed to design and build real applications that have data and do work. I will begin with a small section on object-oriented design for PenPoint, followed by a section on using an implementation strategy geared towards reuse. Once these topics are on the table, I'll go back and apply them to produce a design and implementation strategy for a simple calculator example. I'll end the chapter by explaining the application and model classes for the calculator.

Object-oriented Design and PenPoint

PenPoint's use of objects carries with it a new set of responsibilities in terms of application design and implementation. One example of the new way of doing things can be found in the analysis and design of the objects involved in implementing an application. My experience has shown that it is generally not hard to identify candidates for objecthood. What's hard is classifying the objects into groupings and then documenting their interrelationships.

The View/Data Approach

One of the most important features of objects in general is their ability to separate responsibilities into well-defined units known as classes. Early on, Smalltalk programmers recognized that most classes of objects used in building applications would fall into one of three categories: model (or data) objects for representing the underlying application domain; view objects for presenting the information contained in the model to the user; and controller objects for managing the interaction between the user and the application.

Consider a PenPoint application that mimics a simple noteboard used to exchange messages on the refrigerator. The model for this application might be a text data buffer that can hold a predefined number of characters. The data contained in the buffer would then be preserved across page turns by the Application Framework. What the model doesn't contain is the functionality for interacting with the user. That is the responsibility of the view and controller objects.

The view object for the noteboard application would display the contents of the model in a way that is comprehensible to the user. There can be multiple views for a single model with each view presenting the model's information to the user in a different manner. The application then uses controller objects to manage the user's interaction with the model. You can quickly imagine applying various PenPoint gestures to the view to implement actions such as adding, erasing, and changing the displayed text.

The appearance of the view and the interaction with the controller are influenced heavily by the user interface (presentation) tools available to the programmer. In the case of PenPoint, the use of the pen is so well integrated with the presentation of information to the user that the view and controller objects are referred to by the name view alone.

Notification of Observers

The techniques of associating data and views have been refined in PenPoint to include a mechanism for having the views register with the model as an observer of the model's data. The model then issues a message to all its observers telling them to update themselves because the data contained in the model has changed. The ability of one object to register as an observer is not limited to user interface objects only, rather the mechanism has been generalized so any object can be an observer of any other object.

The observer/notification mechanism consists of a set of eight messages used to add, remove, and update observers on an object's notification list. This behavior is inherited by all objects in the PenPoint system.

Adding and Removing Observers The observer/notification mechanism uses three messages for adding and removing observers from an object's notification list. They are

msgAddObserver and msgAddObserverAt	Add an object to another object's observer list.
msgRemoveObserver	Remove an object from an object's notification list.

An object registers as an observer of another object by sending a message to the object it wishes to observe with its own UID as the argument. For example, if objectA wishes to become an observer of objectB, then objectA sends objectB the message:

```
ObjCallWarn(msgAddObserver, objectB, objectA, s);
```

The object asking to be added or removed from an object notification list will be sent a msgAdded or msgRemoved message when the update to the contents of the list actually takes place.

Notifying Observers An object can send a message to its observers using one of two different messages, depending upon whether it wants to wait for each notification message to complete (msgNotifyObservers) or whether it wants to send and forget (msgPostObservers). Both messages are sent to self and take a pointer to an `OBJ_NOTIFY_OBSERVERS` structure as an argument.

The `OBJ_NOTIFY_OBSERVERS` structure defines the message to be sent and a pointer to the argument block to be sent with it. For example, if objectB wishes to notify the observers on its notification list of a change in its state, it would use

```
SOME_DATA              data;
OBJ_NOTIFY_OBSERVERS   nobs;
STATUS                 s;

setSomeData( &data );
nobs.msg = msgSomeDataHasChanged;
nobs.pArgs = &data;
nobs.lenSend = SizeOf(SOME_DATA);

ObjCallRet( msgNotifyObservers, self, &nobs, s );
// or for posting
// ObjCallRet( msgPostObservers, self, &nob, s);
```

It is the responsibility of the observer and observee to coordinate which messages would be ok for notification purposes. To avoid some problems associated with coordination between class from different sources, PenPoint also provides a generic msgUpdate message that every object is guaranteed to respond to.

Managing the List of Observers In addition to the messages already discussed, the observer/notification mechanism provides three messages for working with the contents of a list of observers.

msgEnumObservers	Get back the list of observers.
msgGetObserver	Getting the observer back to a particular position in the list.
msgNumObservers	Get the number of observers included in the list.

CRC Cards

Once you've decided on the problem you wish to tackle, it is important for you to identify and formally describe an initial set of objects. Kent Beck, Ward Cunningham, and several others have been vocal proponents of a simple approach for collecting the pertinent information, organized on 3 x 5 cards, they call the **Class-Responsibility-Collaborator (CRC)** Approach. The CRC Approach allows you to organize the important information about an object early on by facilitating a process of making many small decisions about the division of responsibilities between different objects in the system.

The word Class stands for class name and represents the process of creating a name space for the application at hand. Carefully chosen names allow you to construct a working vocabulary that encompasses both the user's domain knowledge and the designer's concept of what the system should do. As an example, look at the names of the user interface components from PenPoint's class library. In particular, consider clsButton objects that have well-defined and significant meaning to both the end user (push a button for some action) and the programmer (when the user pushes a button, perform an action).

In addition to naming a class, it is important that you are able to write a concise description of the responsibilities of the class. Each class is used to produce objects that have a specific responsibility for implementing part of the application's behavior. Responsibilities are identified with short phrases such as "executes an action in response to the user pressing a button." Although you can create classes without responsibilities as place holders, you should seriously consider removing objects without any responsibilities at the end of the design phase.

The third piece of information collected by the CRC Approach are the other objects the class collaborates with to fulfill its responsibilities. For example, the application object collaborates with objects like the window manager in order to fulfill its responsibilities for managing the application. Listing the collaboration between objects helps identify the interrelationships between components and which areas might be ripe for reuse. Classes with lots of collaborators indicate an object that is probably doing too many things and signal the need to go back and check its responsibilities.

The CRC Approach was originally designed to use 3x5 cards to collect the information. These cards had the advantages of being cheap and small. Cheap was an advantage in that you could afford as many as you needed and small was an advantage in that it made you think about the relative importance of information being added to the card. Using cards also has the advantage of allowing you to reorder the cards based on a particular design need.

For example, you might reorder all cards that collaborate with a particular item to verify that the object in question is not doing too much work. You can also spatially organize the cards to show various forms of interrelationships of the application as a whole. Finally, when coming to implementation, the CRC cards could be ordered based on a taxonomy that could be used to create an inheritance hierarchy for implementation.

For the purpose of the book, I will substitute tables for 3x5 card images. Each table will contain the following information:

Class Name	The name of the class represented by the table.
Description	A concise description of what the class will be used for.
Superclasses	The enumerated list of classes, in order of parentage, that the new class will inherit from. I use an expanded list, even though just the superclass name would be enough, because it provides a handy reminder of what types of behaviors might be found.
Responsibilities/ Collaborator Pairs	A list of responsibilities the new object is going to undertake, along with any objects it's going to collaborate with in carrying out the listed responsibility.

Identifying Reuse

Perhaps the largest and most important area of reuse is the framework described in Chapter 4. By relying on a common framework, you can produce applications that work consistently in a smaller amount of time. PenPoint also supplies a large library of components that can be reused when building custom applications that fit in nicely with an object design based on the CRC Approach.

For example, once the CRC cards for the noteboard application are complete, you can compare them to a list of known components already supplied by PenPoint. Suppose you have identified a clipboard buffer in a text editing application. It makes sense to consider using a prebuilt text class instead of building a new object from scratch. The thought process encoded in the CRC cards allows you to measure the effectiveness of the fit and weigh the cost of reusing generic versus building specific application classes.

The Calculator Example

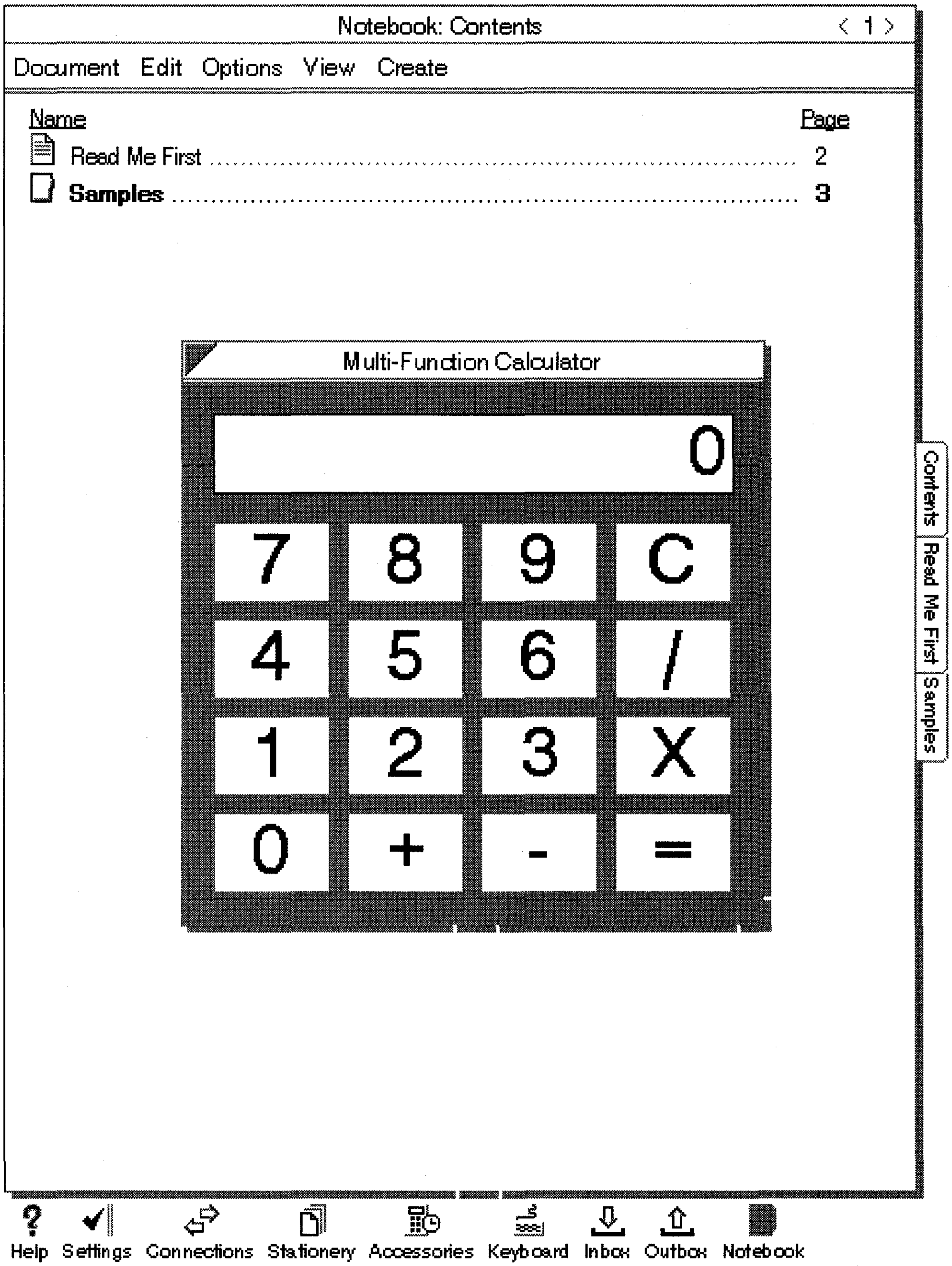
Armed with a stack of blank CRC cards, it's time to sit down and lay out the design of a simple calculator. By simple, I mean the button-based four-function (add, subtract, multiply, and divide), integer-based calculator shown in Figure 5.1. The user interacts with the calculator by tapping the buttons with the pen and viewing the results of the computations in the top window.

I find it useful to work with CRC cards in stages. First, for each real world entity that might become an object in my application I create a card with the class name and description filled in. Then, I go back over the entire deck, writing a list of responsibilities for each class. Finally, I look at each of the responsibilities and decide (a) if there is an appropriate existing class I might reuse as is, (b) if there is an existing class I might derive a new subclass from, or (c) to resign myself to the fact that I cannot reuse any existing code.

Calendar Application Classes

When the interactions are complete, the Calculator application will consist of the three classes: clsCalcApp, which is responsible for managing the calculator application's interaction with the PenPoint Application Framework; clsCalcEng, which provides the application's model of an accumulator-based calculation engine; and clsCalcBtVw, which is a button-based interface between the user and the calculation engine.

FIGURE 5.1 The Calculator Application



clsCalcApp Table 5.1 shows the CRC information for the calculator's application class. Notice that one of clsCalcApp's responsibilities is to register the application and associated classes with PenPoint. This responsibility refers specifically to the `main()` routine required for each PenPoint application. Although technically not part of the actual application class, this responsibility needs to be accounted for and belongs here. In addition to having the `main()` routine, clsCalcApp is also responsible for creating the clsCalcEng model and clsCalcBtVw view that make the application unique.

TABLE 5.1 CRC Table for the Calculator's Application Class

Class: clsCalcApp

Description: manages the Calculator application's interaction with the application framework.

Ancestors: clsApp, clsObject

Responsibilities/Collaborators:

- registers the application and associated classes with PenPoint.
- with clsCalcEng, creates the calculator engine used as the application's model object.
- with clsCalcBtVw, creates the button view used to interact with the model.

clsCalcEng Table 5.2 on the next page shows the CRC information for the Calculator Engine class. Objects created from this class are used as the model for the Calculator application. It manages a set of mathematical operations, including checking for error conditions, and notifies its observers when the value of its accumulator changes or when an error condition occurs. Notice that it relies on behavior inherited from its clsObject ancestor to manage the notification process.

clsCalcView Table 5.3 on the next page shows the CRC information for the Calculator Button View class. Objects created from this class provide the user with an interface to the underlying calculator engine model. A prominent feature of this class is that it serves as an example of design by construction. When you look at the responsibilities the class has, you quickly realize that they are mostly organizational in nature. For example, clsCalcBtVw will build its interface by reusing instances clsLabel, clsButton, and clsTkTable to construct the calculator keypad paradigm instead of building components from scratch.

TABLE 5.2 CRC Table for the Calculator Engine Class

Class: clsCalcEng

Description: computes the value of an accumulator based on a series of mathematical operations.

Ancestors: clsObject

clsObject provides the mechanism to notify observers that the accumulator has changed.

Responsibilities/Collaborators:

- adds, subtracts, multiplies, and divides values with the accumulator.
- notifies observers when the accumulator value changes.
- notifies observers when a divide by zero or accumulator overflow error occurs.

TABLE 5.3 CRC Table for the Calculator Button View Class

Class: clsCalcBtVw

Description: provides the user with a button-based view for using the calculator engine.

Ancestors: clsView, clsCustomLayout, clsBorder, clsEmbeddedWin, clsGWin, clsWin, clsObject

- clsView registers view as an observer of the model.
- clsView causes the model to be saved and restored.
- clsCustomLayout manages the resizing and layout of child windows.

Responsibilities/Collaborators:

- with clsTkTable, creates a table of clsButtons that represent a calculator keypad.
- with clsLabel, displays the current value of the accumulator.
- with clsResFile, manages the save and restore of stateful data.
- with clsButton, converts user input into values for the calculator engine.
- with clsButton, converts user input into commands for the calculator engine.
- with clsCalcEng, computes the user's requests.

Implementing the Calculator Application

Now that the design for the Calculator application has been laid out, it's time for you to begin the implementation phase. For this application, each CRC description represents a separate PenPoint class that needs to be written. The rest of this chapter concentrates on building the application class, clsCalcApp, and the model class, clsCalcEng. I've left the discussion of the user interface class, clsCalcBtVw, for the next chapter.

The clsCalcApp Application Class

The application class for the calculator is actually rather small, due to its ability to reuse much of the default behavior supplied by its ancestor, clsApp. It is only necessary for clsCalcApp to override one method, msgAppInit, to accomplish its responsibilities. Additionally, the file that contains clsCalcApp also contains the `main()` routine required to register the application with PenPoint.

calcapp.c The calcapp.c file contains the code for the method that overrides msgAppInit and the `main()` routine. It begins by including the files necessary for it to compile:

```
#ifndef APP_INCLUDED
#include <app.h>
#endif

#ifndef APPMGR_INCLUDED
#include <appmgr.h>
#endif

#ifndef OS_INCLUDED
#include <os.h>
#endif

#ifndef FRAME_INCLUDED
#include <frame.h>
#endif

#ifndef DEBUG_INCLUDED
#include <debug.h>
#endif

#ifndef CALCENG_INCLUDED
#include <calceng.h>
#endif
```

```
#ifndef CALCAPP_INCLUDED
#include <calcapp.h>
#endif

#ifndef CALCBTVW_INCLUDED
#include <calcbtvw.h>
#endif

#include "method.h">
#include <string.h>
```

CalcAppAppInit The CalcAppAppInit method responds to the msgAppInit message the first time a document for the application is created. The code for CalcAppAppInit is

```
MsgHandler(CalcAppAppInit)
{
  CALCBTVW_NEW         cbv;
  CALCENG_NEW          cn;
  APP_METRICS          am;
  STATUS               s;

  ObjCallWarn(msgNewDefaults, clsCalcEng, &cn, s);
  ObjCallRet(msgNew, clsCalcEng, &cn, s);

  ObjCallWarn(msgNewDefaults, clsCalcBtVw, &cbv, s);
  cbv.view.dataObject = cn.object.uid;
  ObjCallRet(msgNew, clsCalcBtVw, &cbv, s);

  ObjCallWarn(msgAppGetMetrics, self, &am, s, Error);
  ObjCallJmp(msgFrameSetClientWin, am.mainWin,
             cbv.object.uid, s, Error);

  return stsOK;
Error:
  return s;
  MsgHandlerParametersNoWarning;
}
```

This method creates the model and view objects for the document and installs the view object into the window hierarchy. The various forms of the document (initialized, stored, opened, etc.) will be managed for you by behavior inherited from clsApp. Notice that the UID of the calculator engine is passed as input to the button view object. This is part of the convention established by ancestors of the View class. Once a View class is created with an instance of the model class it is a view of, it manages that model object automatically. This includes saving and restoring the model's state when appropriate. In essence, you transferred ownership of the model from the application to the view.

ClsCalcAppInit The ClsCalcAppInit function is called by the `main()` routine when the Calculator application is installed. It is implemented as

```
STATUS ClsCalcAppInit (void)
{
  APP_MGR_NEW new;
  STATUS      s;

  ObjCallJmp(msgNewDefaults, clsAppMgr, &new, s, Error);

  new.object.uid          = clsCalcApp;
  new.cls.pMsg            = clsCalcAppTable;
  new.cls.ancestor        = clsApp;
  new.cls.size            = Nil(SIZEOF);
  new.cls.newArgsSize     = SizeOf(APP_NEW);

  new.appMgr.flags.accessory = TRUE;

  strcpy(new.appMgr.name, "Button Calculator");
  strcpy(new.appMgr.company, "Penpoint Programming");

  ObjCallJmp(msgNew, clsAppMgr, &new, s, Error);

  return stsOK;

Error:
  return s;
}
```

The function registers the Calculator Application class clsCalcApp with PenPoint so it can be used to create documents' forms. In addition to specifying the traditional information about ancestor and instance variables size, I have asked the application to place the installed Calculator application onto the Accessories menu using the statement:

```
new.appMgr.flags.accessory = TRUE;
```

You can also use the flags structure in the `APP_MGR_NEW` to indicate any of the following information to the application manager:

stationary	Put the application in the stationary Notebook.
accessory	Put the application in the Accessory menu.
hotMode	Create application documents in hot mode.
allowEmbedding	Allow child-embedded applications within this application's documents.
confirmDelete	Ask the user for confirmation before deleting a document.
de-installable	The user can de-install the application.
systemApp	The specified application is a system application.
appMonitor	Create an application monitor.

Finally, the Calculator class is given a Well Known UID, `clsCalcApp`, that was defined in the header file for the applications:

```
#define clsCalcApp MakeGlobalWKN(4142, 1)
```

main() `calcapp.c` also contains the `main()` routine used to install the application. It is defined

```
void CDECL
main(
  int                  argc,
  char *               argv[],
  U16                  processCount)
{
  STATUS s;

  if (processCount == 0) {
    ClsCalcAppInit();
    ClsCalcEngInit();
    ClsCalcBtVwInit();
    AppMonitorMain(clsCalcApp, objNull);
    }
  else
    AppMain();
  Unused(argc); Unused(argv);
}
```

As you probably expected, the `main()` routine is another cut-and-paste function with the additional behavior that the Calculator Engine model and Calculator Button views are created when the user turns to a document.

calcapp.h The header file for clsCalcApp contains the definition of the Well Known UID used to identify the calculator application class. In addition it also includes the macros necessary to produce the UID. Its complete definition is

```
#ifndef CALCAPP_INCLUDED
#define CALCAPP_INCLUDED

#ifndef CLSMGR_INCLUDED
#include <clsmgr.h>
#endif

#define clsCalcApp MakeGlobalWKN(4142, 1)

#endif // CALCAPP_INCLUDED
```

method.tbl and clsCalcApp method.tbl contains the following `MSG_INFO` structure for mapping messages to methods in clsCalcApp:

```
MSG_INFO clsCalcAppMethods[] = {
  msgAppInit, "CalcAppAppInit", objCallAncestorBefore,
  0
};
```

The clsCalcEng Model Class

The calculator engine works by handling messages that request a certain operation be performed on the data contained in the accumulator. Instead of passing the result back, the object simply notifies its observers that a change has occurred. This technique allows multiple views to watch the same model and receive an update message whenever any view action—whether self-originated or not—caused the model to change.

The model class for the calculator contains the methods necessary to implement a small but functional calculator. Like most model classes, clsCalcEng does not collaborate with many other classes, because the bulk of its responsibility is to define the intrinsic behavior of the application itself. clsCalcEng relies on clsObject for help in filing and managing the notification of an object's observers when changes happen and on clsResFile to provide a means to save its persistent data when the need arises.

In addition to the functions used to respond to the messages defined by the clsCalcEng class, I also use several functions to simplify the reading of the code. This is one advantage of a hybrid environment that allows you to mix both function calls and message sends to obtain the best possible source code. In general, when using the PenPoint message-sending macros, it is best to pass pointers to return values and define the actual function to return type `STATUS`. This eliminates the compiler generating a warning that the function is attempting to return a different type. The reason for the conflict is that the message-passing macros such as `ObjCallRet` produce code that returns the status value if an error is detected, which would not be of the same type if the function was declared to return something other than `STATUS`.

calceng.h The Calculator engine is defined using two files: calceng.h and calceng.c. calceng.h contains definitions of the classes, Well Known UID, messages, and error status values used by both clsCalcEng and its observers. calceng.h begins

```
#ifndef CALCENG_INCLUDED
#define CALCENG_INCLUDED

#ifndef CLSMGR_INCLUDED
#include <clsmgr.h>
#endif

#define clsCalcEng MakeGlobalWKN( 4143, 1)

STATUS ClsCalcEngInit (void);
```

These lines include the definition of the global Well Known UID used to identify the clsCalcEng class within the Calculator application.

Next, calceng.h defines the two status values that an instance of clsCalcEng might send its observers. They are

```
#define stsCalcEngOverflow MakeStatus(clsCalcEng, 1)
#define stsCalcEngZeroDiv MakeStatus(clsCalcEng, 2)
```

Notice that the `MakeStatus` macro generates a unique status value by combining a Well Known UID (clsCalcEng) with an ordinal (1 or 2).

Next, the following messages are defined for use with clsCalcEng objects.

```
#define msgCalcEngGetAccm      MakeMsg(clsCalcEng, 1)
#define msgCalcEngSetAccm      MakeMsg(clsCalcEng, 2)
#define msgCalcEngClr          MakeMsg(clsCalcEng, 3)
```

```
#define msgCalcEngAdd          MakeMsg(clsCalcEng, 4)
#define msgCalcEngSub          MakeMsg(clsCalcEng, 5)
#define msgCalcEngDiv          MakeMsg(clsCalcEng, 6)
#define msgCalcEngMul          MakeMsg(clsCalcEng, 7)
#define msgCalcEngAccmChanged  MakeMsg(clsCalcEng, 8)
#define msgCalcEngError        MakeMsg(clsCalcEng, 9)
```

The last two messages, `msgCalcEngError` and `msgClacAccmChanged`, have no corresponding methods in `calceng.c`. These messages are sent by a calculator engine object to its observers when the value of its accumulator changes or it encounters an error condition. The observing object is responsibile for defining a method for responding to these messages when they are sent.

Following the message definitions, calceng.h defines two data structures used for interacting with the calculator engine. The first

```
#define CalcEngNewFields ObjectNewFields

typedef struct CALCENG_NEW {
  CalcEngNewFields
} CALCENG_NEW, *P_CALCENG_NEW;
```

is defined so that objects creating instances of clsCalcEng can follow the normal convention of using msgNewDefaults followed by msgNew, even though clsCalcEng doesn't add any additional initialization values.

The second structure, used to pass values back and forth from the calculator engine, is defined

```
typedef struct CALCENG_VAL {
  S32 value;
} CALCENG_VAL, FAR *P_CALCENG_VAL;

#define CALCENG_MAX_DIGITS 8
```

Notice that the calculator engine was implemented to take and receive 32-bit signed integers. In turn, it will do all its computations in 32-bit signed integers and will signal overflow based on the result exceeding `CALCENG_MAX_DIGITS`. Placing the value inside a structure allows an upgrade of the calculator to floating point at a later time without requiring a massive edit of the code that uses the calculator engine.

Finally, the last line of calceng.h is

```
#endif // CALCENG_INCLUDED
```

calceng.c calceng.c contains the implementation of the Calculator Engine class. It begins by including the necessary definition files:

```
#ifndef CLSMGR_INCLUDED
#include <clsmgr.h>
#endif

#ifndef FS_INCLUDED
#include <fs.h>
#endif

#ifndef CALCENG_INCLUDED
#include <calceng.h>
#endif

#include "method.h"
```

During the initial development phase, you could also add

```
#ifndef DEBUG_INCLUDED
#include <debug.h>
#endif
```

This provides access to the debugging help provided by PenPoint. Next, the instance data and accumulator limits are defined:

```
typedef struct INSTANCE_DATA {
  S32 accm;
} INSTANCE_DATA, *P_INSTANCE_DATA;

#define POS_OVERFLOW 100000000
#define NEG_OVERFLOW -100000000
```

The rest of the data in calceng.c can be divided into these groups: support functions for notifying observers; methods for managing the clsCalcEng with respect to PenPoint; and methods for updating and monitoring the accumulator.

Notifier Support Functions calceng.c contains two functions used by the methods in clsCalcEng to notify observers about a change in the accumulator or a possible error. The first function, `tellObjsAccmChanged()`, is defined

```
STATUS tellObsAccmChanged( OBJECT obsrObj, S32 newVal )
{
  CALCENG_VAL    cv;
  OBJ_NOTIFY_OBSERVERS          nobs;
  STATUS                        s;

  cv.value = newVal;
```

```
    nobs.msg = msgCalcEngAccmChanged;
    nobs.pArgs = &cv;
    nobs.lenSend = SizeOf(CALCENG_VAL);
    ObjCallJmp(msgNotifyObservers, obsrObj, &nobs, s, Error );

    return stsOK;
Error:
    return s;
}
```

This function is used to notify observers of a calculator engine object that the value of the accumulator has changed. It does this by filling in an `OBJ_NOTIFY_OBSERVERS` structure with the msgCalcEngAccmChanged message and passing a pointer (&cv) to a structure that contains the new value of the accumulator. Finally, it issues the msgNotifyObservers message to the observed object.

In the same way, the `tellObsError()` function is used to notify observers of the obsrObj that an error has occurred. This is the only error notification an observer of an object will get. Since the calculator engine has been set up as a stateless entity, a user of a calculator engine object could continue to send requests for calculations and receive updated accumulator values, even though those values would be incorrect. The `tellObsError()` function is implemented as

```
STATUS tellObsError( OBJECT obsrObj, STATUS errval )
{
    OBJ_NOTIFY_OBSERVERS          nobs;
    STATUS                        s;

    nobs.msg        = msgCalcEngError;
    nobs.pArgs      = &errval;
    nobs.lenSend    = SizeOf(STATUS);
    ObjCallJmp(msgNotifyObservers, obsrObj, &nobs, s, Error );

    return stsOK;
Error:
    return s;
}
```

Accumulator Access Methods The calculator engine supports seven methods for accessing and/or updating the value contained in the accumulator. They are CalcEngReadAccm, CalcEngSetAccm, CalcEngClr, CalcEngAdd, CalcEngSub, CalcEngMul, and CalcEngDiv.

The first one, CalcEngClr, responds to the msgCalcClr message and resets the value of the accumulator to zero. It is defined

```
MsgHandler(CalcEngClr)
{
  INSTANCE_DATA inst;

  inst = IDataDeref(pData, INSTANCE_DATA);
  inst.accm = 0;
  ObjectWrite(self, ctx, &inst);

  return tellObsAccmChanged( self, inst.accm );
}
```

It is necessary to de-reference that instance data from the pointer passed into the method before changing its value, because the pointer to the instance data indicates a protected memory area. Once updated, the value of the accumulator is written back into the protected memory with a call to `ObjectWrite()`. Finally, the CalcEngClr method uses the `tellObsAccmChanged()` function to notify its observers that the accumulator has a new value.

clsCalcEng defines four methods for updating the accumulator through a requested mathematical operation. Each method also receives a pointer to a `P_CALCVAL` structure containing the value that is to be used to update the accumulator. The first operation method is

```
MsgHandlerArgType(CalcEngAdd, P_CALCENG_VAL )
{
  INSTANCE_DATA inst;

  inst = IDataDeref(pData, INSTANCE_DATA);
  inst.accm += pArgs->value;
  if ( inst.accm >= POS_OVERFLOW )
      return tellObsError( self, stsCalcEngOverflow );
  else {
      ObjectWrite(self, ctx, &inst);
      return tellObsAccmChanged( self, inst.accm );
      }
}
```

This method de-references the instance data, adds to it the values passed to the method, and then checks for overflow. If the addition is in error, the `tellObsError()` function is called. If the operation succeeds, the instance data is written back into protected memory and the `tellObsAccmChanged()` function is called.

The CalcEngSub method works in a similar manner and is defined

```
MsgHandlerArgType(CalcEngSub, P_CALCENG_VAL )
{
  INSTANCE_DATA inst;

  inst = IDataDeref(pData, INSTANCE_DATA);
  inst.accm -= pArgs->value;
  if ( inst.accm <= NEG_OVERFLOW )
    return tellObsError( self, stsCalcEngOverflow );
  else {
    ObjectWrite(self, ctx, &inst);
    return tellObsAccmChanged( self, inst.accm );
    }
}
```

The CalcEngMul method multiplies the given value by the accumulator and checks for both positive and negative overflow. It is defined

```
MsgHandlerArgType(CalcEngMul, P_CALCENG_VAL )
{
  INSTANCE_DATA inst;

  inst = IDataDeref(pData, INSTANCE_DATA);
  inst.accm *= pArgs->value;
  if ( (inst.accm >= POS_OVERFLOW) ||
       (inst.accm <= NEG_OVERFLOW) )
    return tellObsError( self, stsCalcEngOverflow );
  else {
    ObjectWrite(self, ctx, &inst);
    return tellObsAccmChanged( self, inst.accm );
    }
}
```

The last operation method, CalcEngDiv, divides the accumulator by the input value. It checks the input value. If the value is zero it generates an error condition before attempting the operation. It is defined

```
MsgHandlerArgType(CalcEngDiv, P_CALCENG_VAL )
{
  INSTANCE_DATA inst;

  if (pArgs->value == 0 )
      return tellObsError( self, stsCalcEngZeroDiv );

  inst = IDataDeref(pData, INSTANCE_DATA);
  inst.accm /= pArgs->value;
  ObjectWrite(self, ctx, &inst);
```

```
  return tellObsAccmChanged( self, inst.accm );
}
```

In addition to the operation methods, clsCalcEng also provides methods for setting and getting the value of the accumulator. They are defined

```
MsgHandlerWithTypes(CalcEngGetAccm, P_CALCENG_VAL,
                                    P_INSTANCE_DATA)
{
  pArgs->value = pData->accm;

  return stsOK;
  MsgHandlerParametersNoWarning;
}

MsgHandlerArgType(CalcEngSetAccm, P_CALCENG_VAL )
{
  INSTANCE_DATA inst;

  inst = IDataDeref(pData, INSTANCE_DATA);
  inst.accm = pArgs->value;
  ObjectWrite(self, ctx, &inst);

  return tellObsAccmChanged( self, inst.accm );
}
```

Object Maintenance Methods calceng.c defines several methods for maintaining clsCalcEng objects. The first of these methods responds to msgInit and is used to set the accumulator to zero. It is defined

```
MsgHandler(CalcEngInit)
{
  INSTANCE_DATA inst;

  inst.accm     = 0;

  ObjectWrite(self, ctx, &inst);

  return stsOK;
  MsgHandlerParametersNoWarning;
}
```

Methods are also provided to save and restore the state of the engine by saving the contents of the accumulator in a resource file. The method for saving data is

```
MsgHandlerArgType(CalcEngSave, P_OBJ_SAVE)
{
  STREAM_READ_WRITE fsWrite;
  STATUS            s;

  fsWrite.numBytes= SizeOf(INSTANCE_DATA);
  fsWrite.pBuf     = pData;
  ObjCallRet(msgStreamWrite, pArgs->file, &fsWrite, s);

  return stsOK;
  MsgHandlerParametersNoWarning;
}
```

The method for restoring the data is

```
MsgHandlerArgType(CalcEngRestore, P_OBJ_RESTORE)
{
  INSTANCE_DATA     inst;
  STREAM_READ_WRITE fsRead;
  STATUS            s;

  fsRead.numBytes = SizeOf(INSTANCE_DATA);
  fsRead.pBuf     = &inst;
  ObjCallRet(msgStreamRead, pArgs->file, &fsRead, s);

  ObjectWrite(self, ctx, &inst);

  return stsOK;
  MsgHandlerParametersNoWarning;
}
```

Finally, calceng.c contains the function `ClsCalcEng()` which is called by the `main()` routine in response to a new document being opened. The `ClsCalcEng()` function registers the clsCalcEng class with the Class Manager so calculator documents can use its objects. The function is defined

```
STATUS ClsCalcEngInit (void)
{
  CLASS_NEW   new;
  STATUS      s;

  ObjCallJmp(msgNewDefaults, clsClass, &new, s, Error);

  new.object.uid        = clsCalcEng;
  new.cls.pMsg          = clsCalcEngTable;
  new.cls.ancestor      = clsObject;
  new.cls.size          = SizeOf(INSTANCE_DATA);
```

```
    new.cls.newArgsSize    = SizeOf(CALCENG_NEW);

    ObjCallJmp(msgNew, clsClass, &new, s, Error);

    return stsOK;

  Error:
    return s;

  }
```

method.tbl and clsCalcEng method.tbl contains the following MSG_INFO structure for mapping messages to methods in clsCalcEng:

```
  MSG_INFO clsCalcEngMethods[] = {
    msgInit,              "CalcEngInit",
    objCallAncestorBefore,
    msgSave,              "CalcEngSave",
    objCallAncestorBefore,
    msgRestore,           "CalcEngRestore",
    objCallAncestorBefore,
    msgCalcEngGetAccm,    "CalcEngReadAccm",    0,
    msgCalcEngSetAccm,    "CalcEngSetAccm",     0,
    msgCalcEngClr,        "CalcEngClr",         0,
    msgCalcEngAdd,        "CalcEngAdd",         0,
    msgCalcEngSub,        "CalcEngSub",         0,
    msgCalcEngMul,        "CalcEngMul",         0,
    msgCalcEngDiv,        "CalcEngDiv",         0,
    0
  };
```

Wrap-up

This chapter has covered a lot of ground with respect to application development in general and building PenPoint applications in particular. Writing PenPoint applications requires the use of an object-based toolkit. In Chapter 4 you learned how the Application Framework was used by exploring all the different methods your application could override. Now, you get to the first real application and discover that you need only override one method for the application to exhibit the necessary behavior.

You've got to love it.

In the next chapter, I'm going to take a quick break from developing the calculator example to discuss the PenPoint windowing model and how you can use (or should I say reuse) the predefined objects contained in the environment. I'll finish up the calculator by implementing a button-based view for the calculator engine class.

6

Constructing a User Interface

One of the most rewarding aspects of building applications is the design and implementation of the user interface. While previous chapters covered important topics that are indispensable in PenPoint, who could say that watching the Application Framework go through its paces is fun? But user interfaces? Now there's an area that's fun because you can *see* the results of your work.

PenPoint takes full advantage of the object-oriented paradigm in implementing its Notebook User Interface. First, PenPoint uses inheritance to classify the behavior of a windowing system into a set of pre-built library components. At the base of this inheritance hierarchy are a series of generic window components, including clsBorder (the Border Layer class) that implements the concept of rectangular drawing regions. In the middle of the hierarchy are components which add layout and control behavior to the basic windowing behavior of their ancestors. Finally, the leaf classes of the hierarchy implement the components that the user touches directly, such as clsText, clsMenu, clsTextField, and many more.

In this chapter, I reuse several of PenPoint's predefined user interface components to build a front-end view for the calculator engine of the last chapter. My goal for this chapter is to provide you with an understanding of some of the major ways in which interfaces are constructed, including the use of custom layout and table objects. Armed with this information, you can weigh your implementation options when it comes time to design your application's user interface.

PenPoint Windows

The term **window** has come to mean many different things, depending on the context in which it is used. When I talk about a window in PenPoint, I am referring to a rectangular region, often thought of as a bitmap, that has a drawing context attached to it. This is not to be confused with a PenPoint **frame**, a complex composite of various window types, such as scrollbars, title bars, and menus that you interact with when running a program. In actuality, the title, menu, scrollbar, and other parts of a frame are actually windows! PenPoint uses the window class as a basic building block to construct more complex components for the user to interact with.

One of the best analogies for dealing with this style of window is to think of the sheets of acetate cartoonists use. The cartoonists draw on each sheet of acetate using different types of ink and writing implements, depending on their needs. After each individual sheet is drawn, it can be laid on top of others to create the desired effect. Also, because the sheets are organized in this layered fashion, careful planning allows the various sheets to be used at a later time.

Consider a two-person animation team constructing a cartoon. The person rendering the background might choose greens and browns to build a wooded scene. In the process of designing the scene, this animator might opt for reusable pine trees by drawing them on a separate layer. At the same time, the person creating the characters is busy drawing them in different poses on separate layers so they can be overlaid on the background scene and the animation sequence shot.

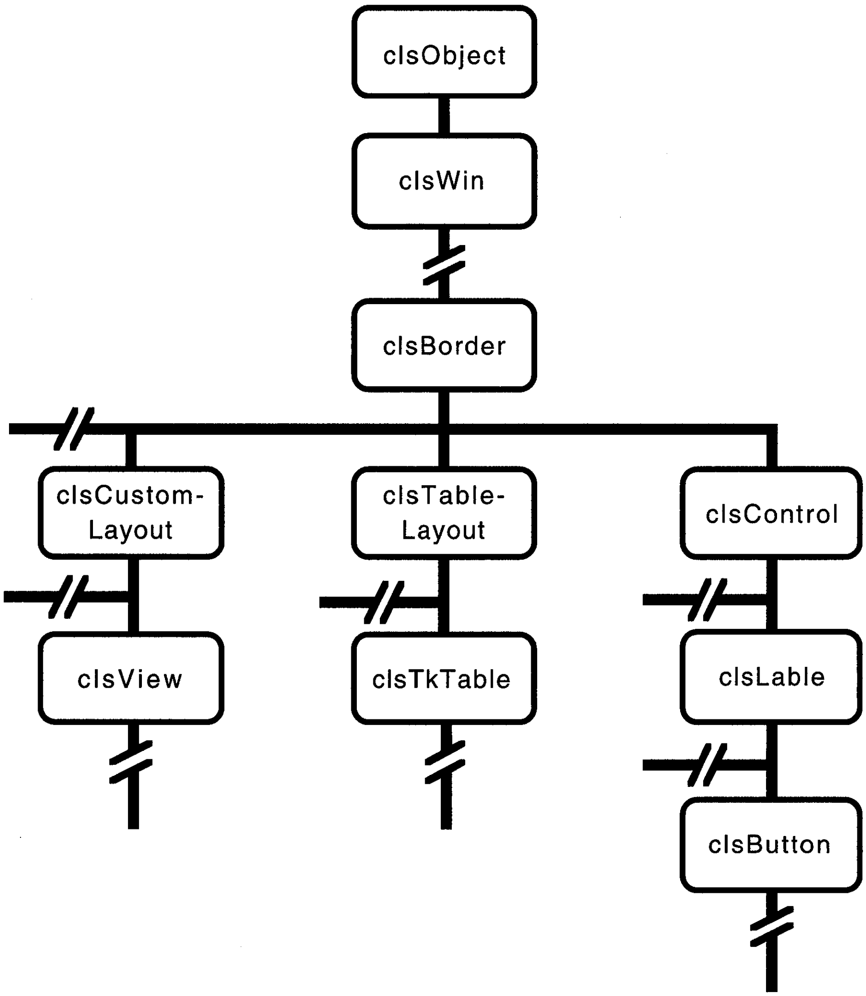
In PenPoint, the layers of acetate are called windows and are the basic building blocks for all the components in the Notebook User Interface (NUI). Windows are kept in a **visual hierarchy** (not to be confused with the inheritance hierarchy) that PenPoint manages for you. This hierarchy is responsible for managing which areas of the screen are visible to the user, which areas get repainted when "dirty," and many other functions. The **Window Manager**, in addition to its repainting responsibilities, also manages the distribution of user events to individual windows. It uses the window hierarchy to decide which window is allowed first response to a user event (pen tap, pen move, and so on).

A Sample of Component Classes

The component library used to build user interfaces in PenPoint can be briefly described using a core subset of the available classes. Figure 6.1 shows an edited version of the user interface management hierarchy that lists the core classes. The following sections provide a brief description of each core component.

clsWin The window class **clsWin** is the patriarch of the window management system for PenPoint's NUI. It provides the basic functionality for maintaining a window's place in the window hierarchy and managing the window's relationship with its parent and child windows, including resizing and layout. It also provides support for the more mechanical tasks of clipping (managing which part of the window needs to be redrawn) and saving the state of the window.

FIGURE 6.1 The Penpoint User Interface Component Hierarchy (edited)



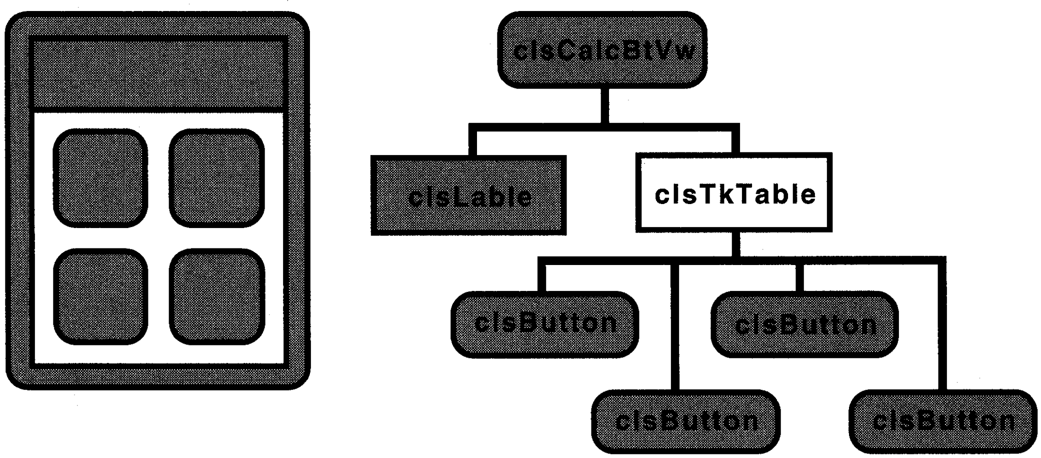
Window objects in PenPoint use the family metaphor to describe their relationship to one another. A window

- Will have a parent window that depends on the window hierarchy.
- Might have one or more child windows depend on it in the window hierarchy.
- Might have several sibling windows which share the same parent window with it.

If a parent window is visible and its child window wants to be visible, it will be. If the child window wants to be hidden, even if the parent is visible, it will be. However, if a child wants to be visible but its parent is hidden, the child also remains hidden until its parent becomes visible.

Figure 6.2 shows a wire frame view of a simple set of window objects that demonstrates the family relationship of a view hierarchy. The parent window for the sample drawing includes both a clsLabel child window and a clsTkTable child window, which are window siblings. The clsTkTable child window in turn contains a group of clsButton objects (which are also descendants of clsWin) organized in a table format. A window can be inserted as a child of, or sibling to, another window by sending a message to itself indicating the window it wants to be related to, and how it wants to be related.

FIGURE 6.2 A Sample View Hierarchy of Window Objects



In addition to visibility, parents also manage the clipping and layout of their children. For example, you size and locate a child window relative to its parent window, not the actual device. This philosophy extends to the restriction that child windows can't draw outside their parent's boundaries, but a parent can choose to have its drawing clipped away from the visible part of its child windows. There is also an option that allows a child window to make itself transparent, allowing the contents of its parent to show through.

The parent, in addition to its clipping responsibilities, also manages the layout and resizing of its children. A built-in protocol allows parent and children to work together to compute a reasonable screen layout for windows. This includes **shrink-wrapping,** which allows windows to change size based on their contents and to pass the size information back to their parent.

Finally, you use the behavior in the window class to control the user's input to the application. Windows are given access to pen and keyboard input from the user, based on their position in the window hierarchy. This includes behavior that allows events to be filtered: to be handled and/or distributed to other windows besides the one that got the event originally.

clsBorder clsBorder, which inherits from several abstract ancestors, is the backbone window object for the NUI. A border window is a simple rectangular region with a border that has a drawing context attached to it. clsBorder is special because it contains the majority of the behavior necessary to create controls that work with other PenPoint components such as style sheets and menus.

You can control how a border window looks by specifying its visual attributes either at creation or through messaging. For example, PenPoint provides a predefined set of default values for the visual attribute values, such as width, height, and border type. You can use the values supplied by PenPoint or you can specify custom values when you set up the attributes of a clsBorder object. The advantage of using predefined values is that they provide optimum performance in the PenPoint environment, as well as a higher level of consistency with other NUI components.

Your ability to customize components is not limited to changing just the visual attributes of clsBorder objects. You can also create custom components by inheriting from clsBorder and adding new behavior. This allows you to have the standard behaviors where appropriate, plus override the various drawing methods that are part of the window repainting protocol used to update the display screen. You can then concentrate on the custom portion of your component while inheriting the majority of behavior from the ancestor class.

Figure 6.3 shows the various parts of a border object. Notice that there are several ways to talk about the size of a border window. First, you can talk about the size of the border object that includes everything: the inside, the margin, the border, and the border's shadow. Then in progression, you can describe the size in terms of the bordered area (everything excluding the shadow), the margin area (the inside and margin only), and the inner area (only the inside). clsBorder contains messages for obtaining and manipulating the values of these geometries in a way that facilitates lay out of border objects relative to each other.

clsControl PenPoint uses **clsControl**, which inherits directly from clsBorder, as an abstract superclass that provides a consistent paradigm for writing the components that translate a user's action into a program request. Using this common control paradigm, you can guarantee that a certain amount of consistency exists between classes of control objects, such as text fields and buttons, even though these controls have a different appearance from each other.

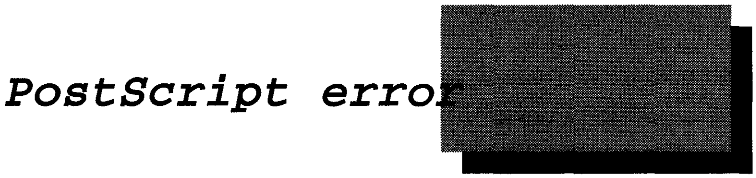
clsControl supports the concept of a generic control model by defining an instance variable used to hold the value of the control, such as the text in a label value or the location of the indicator for a scroll bar. Each control object also has a client object which is notified of the final outcome of a user's interaction with a control.

Controls are built by subclassing clsControl and overriding an appropriate subset of methods that is used to indicate the type of interaction the user requested. By definition, a clsControl object responds to certain types of user actions by sending itself messages. It is the subclass's responsibility to override the method corresponding to a desired user interaction so it can then send the appropriate message to the client object. There are four times that clsControl objects send messages to themselves:

- When the user first selects the control.
- When the control enters preview mode.
- When the user accepts the previewed condition or action.
- When the control notifies its client of the user's request.

The user begins interacting with the control by gesturing on it (Gesture response is managed by clsControl's clsGWin ancestor class), which in turn places the control in preview mode. For example, you would "press a button" by tapping on it with the pen, causing the button to enter preview mode. The button's response to entering preview mode is to invert itself to provide you with appropriate feedback. You would accept the action represented by the button by lifting the pen off the screen, causing the client of the button to receive a notification to go do something.

FIGURE 6.3 The Parts of a clsBorder Window Object



Although most controls reuse the standard behavior for handling preview mode, at times the availability of the preview protocol becomes useful. For example, you are building a telephone number entry control that only accepts valid area codes. You would be able to override the method that responds to the msgControlAcceptPreview message which clsControl sends to itself to check for a valid area code when the user indicates completion. You could then accept the value and notify the control's client of the new value, or you could indicate to the user that the value entered is incorrect and start over.

Finally, controls can be disabled and enabled programmatically, allowing you to present the user with different combinations of available controls based on the state of your application. For example, an empty document might have its printing controls disabled.

clsLabel clsLabel, the simplest of all controls, is used to display information, such as text strings, to the user without providing any interactive capabilities. It is an ancestor class for other classes, such as buttons, menu items, and title bars, that can reuse a standard set of text layout facilities, but require different forms of previewing and notification behaviors.

When creating a clsLabel object, you can specify attributes such as the label's string or child window; whether the text or child window is left justified, centered, or right justified; the rotation of text; whether the text is selectable or not; and the decoration used to set the label off from its surroundings. Although clsLabel objects don't receive input from the user, they are able to implement gesture recognition through control messages. This would allow, for example, a label object used to annotate another control object to respond to the "?" help gesture and provide information about the control it is being used to annotate.

clsButton clsButton adds preview and notification behavior to its clsLabel ancestor by implementing three different styles of buttons: momentary contact, toggle, and lock-on. Momentary contact buttons are used to change the value of a control to On and then to Off when the button is released. Toggle buttons change, or toggle, their value between "On" and "Off" (or any two values) each time the button is pushed. Finally, a lock-on button will set its value to "On," and then prohibit the user from turning the button off once it's on. Lock-on buttons (like their cousins, the radio buttons) are generally used to implement a selection of choices where one, and only one, must be selected.

Each button style differs in how it handles preview and notification messages, but shares the same layout capabilities. For example, momentary contact buttons provide the user with feedback in preview mode but don't send a notification message to the client unless the user raises the pen while the button is still highlighted. On the other hand, toggle and lock-on buttons don't provide many preview capabilities and are generally used in components that require the user to set options and then select a completed interaction.

Managing Collections of Controls

One very powerful feature of the PenPoint NUI class library is its addition of several classes for managing the layout of a collection of controls. These classes are re-used in many different ways, from creating complex style sheets composed of multiple types of control objects, to creating user menus from collections of standard buttons. These layout classes are not restricted to objects defined in the PenPoint class library, but work with any view that responds to the appropriate messages.

Layout classes use information you supply to determine its child windows' lay outs. In this manner, a parent knows how to lay out its child windows, and depends on those windows to know how to lay out their children, if there are any. A layout window positions its children by first asking them for size information and then specifying to each child how it should lay itself out. This happens recursively, until each child in the window hierarchy has had a chance to lay itself out. One interesting feature of table and custom layout windows is that they can "shrink wrap" around the windows of the controls they contain.

Two basic types of layout classes, tables (subclasses of clsTableLayouts) and custom (subclasses of clsCustomeLayouts), are included in the class library. **clsTableLayouts** objects manage collections of same-sized controls that easily fit in a grid format. **clsCustomeLayouts** objects manage collections of controls, possibly of different sizes, that are grouped together but can't or won't fit into a grid-based grouping.

clsTableLayouts clsTableLayouts with its subclass, clsTkTable, provides the basis of many PenPoint NUI staples. Table layouts allow all their components to be created at one time and to share the same client for notification purposes. Additionally, the table layout, as its name implies, manages the layout of its child windows according to information given to it during initialization. Subclasses of clsTkTable, and hence clsTableLayouts, include clsChoice, clsTabBar, clsMenu, clsIconWin, and other objects that provide table-based groupings of choices for the user to make.

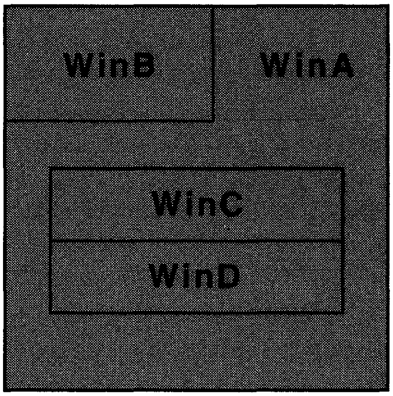
In addition to a client for each of the individual controls, clsTableLayout objects also include a **manager** for the table itself. This allows a controller object to receive notification messages from the individual components included in the table. This is very useful in building option sheets that might allow or disallow certain choices based on other components in the sheet. The manager would receive notification of controls being manipulated and would have the chance to then alter the overall state of the sheet appropriately.

Individual components contained in a table are created at the same time and are inserted as children of the table. Each component shares the same client, default msgNew structure, and pointer to the information that defines the individual components. As the table layout creates each component, its specific information is read from that definition table. The definition table in turn contains information about each individual component in a generic format that each individual type of component interprets according to the component's needs. For example, when creating button components, this information is used to give each button its label, notification message, and associated notification value.

clsCustomeLayout Custom layouts make it possible for you to create and position child windows according to a set of logical constraints. Unlike tables, no attempt is made to ease the burden of coordinating controls and control clients. Custom layout objects are generally used for complex views constructed from a small number of components.

Child windows contained in custom views can be laid out using a set of logical constraints to specify positioning. For example, instead of computing exact (x,y) coordinates of each child window, you can give relative positioning instructions such as "align top of window A with bottom of window B" or "center windows A and B" instead. This alignment is specified using a macro defined by PenPoint called **`CLAlign()`**. Figure 6.4 shows `CLAlign()` being used to specify the alignment constraints for several windows.

FIGURE 6.4 Various Uses of the CLAlign() Macro to Specify the Position of One Window Relative to Another.



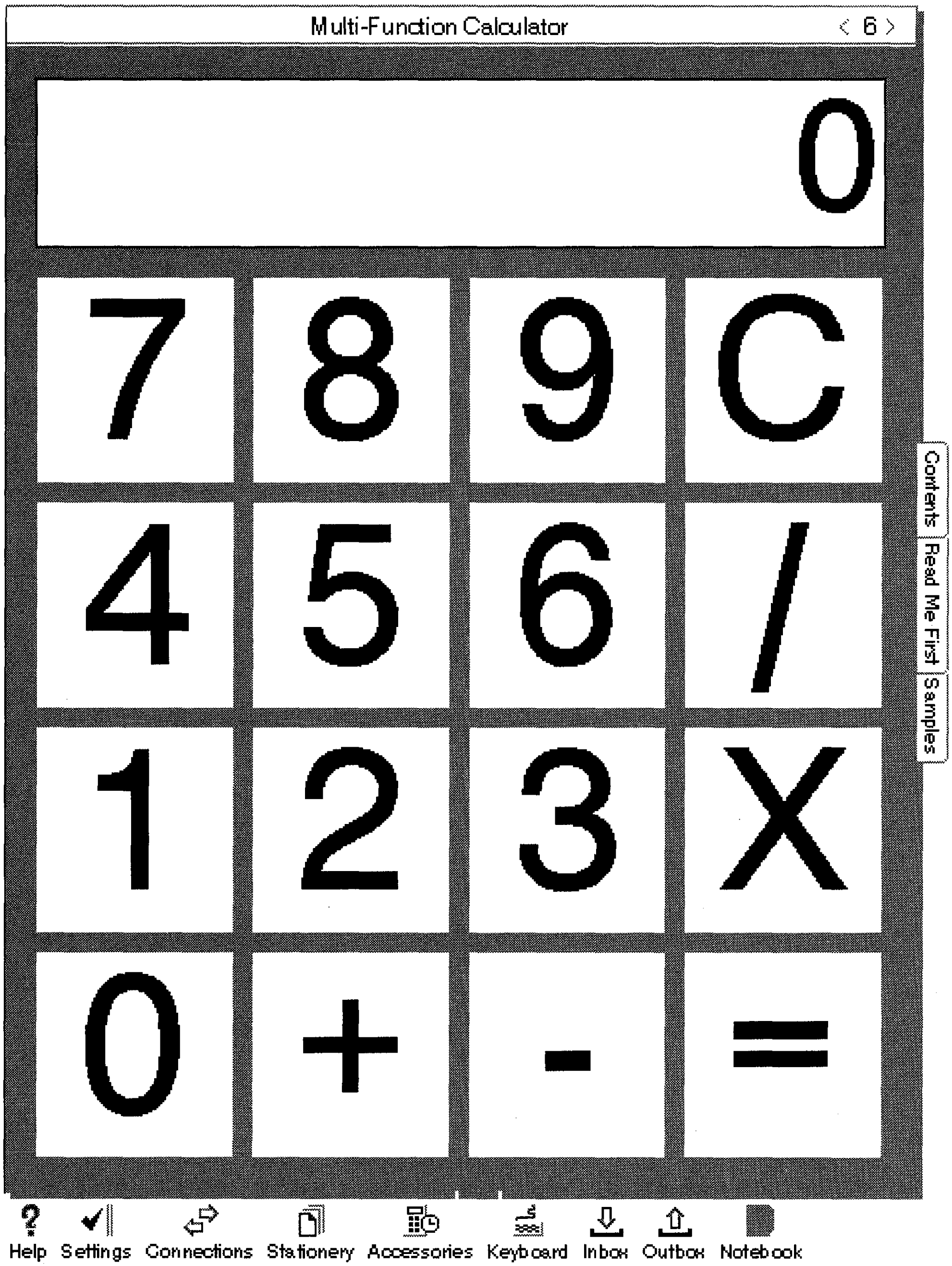
The relationships of the windows in Figure 6.4 to one another are

- WinB as it relates to WinA: x -> CLAlign(clMinEdge, clSameAs, clMinEdge), y -> CLAlign(clMaxEdge, clSameAs, clMaxEdge).
- WinC as it relates to WinA: x,y -> CLAlign(clCenterEdge, clSameAs, clCenterEdge).
- WinD as it relates to WinC: x -> CLAlign(clMinEdge, clSameAs, clMinEdge), y -> CLAlign(clMaxEdge, clSameAs, clMinEdge).

The Calculator Button View

Chapter 5 covered the data model for the calculator application. Now it's time to discuss the user interface to, or view of, the calculator's engine used in the application. Although Figure 6.5 shows the calculator application on the NUI in document form, the calculator actually makes itself an accessory and uses its application window to hold the button-based calculator view. You might have noticed that the calculator shown in Figure 6.5 is exactly proportional to the one shown in Chapter 5. This is a direct result of relying on PenPoint's auto-sizing and auto-layout capabilities to render the application's user interface to the screen.

FIGURE 6.5 The Calculator Application Running as a Document in the Notebook User Interface



The calculator view itself is comprised of two subviews that are kept as child windows. The first subview is the display window, which is a clsLabel object. The second subview is the keypad used for data entry, which is built using a table layout object. The class that implements this calculator button view is called clsCalcBtVw and has the responsibility for creating, laying out, and freeing its subviews during the normal course of a PenPoint document's life cycle. In addition, it also manages the user interaction needed to send updates to the application's model, the calculator engine.

clsCalcBtVw fulfills its responsibilities by relying heavily upon the inherited behavior of its ancestor, clsView, and the borrowed behavior of the clsTkTable object it uses to manage the keypad. It also defines the logic necessary to maintain the state of the user's actions, even if the user turns away from the document during the entry of a number. Although the example itself is somewhat contrived, the actions it goes through in supporting its role as view to the user are standard for any custom view you might create.

Subclassing clsView

clsView objects in PenPoint are windows used to display and possibly modify the contents of a data object. For instance, you use a text field view to display modifiable text to the user. When users modify the view, they are actually indicating to your application that the underlying data (the text) that the view is observing should be changed. It is also possible for the text field to change for a reason other than user interaction, and, therefore, the data model has the ability to tell the view to update itself with the new value.

It is common in object-based programming to have multiple views for a single model. Consider an application that shows the contents of an array of numbers in various forms. One view might be a pie chart, another a line graph, while a third might be a tabular display of the numbers that the user can modify. Since each view shares the same model, a change to the model results in each of the views updating themselves to reflect to the data's contents.

Managing the Data When a clsCalcBtVw object is created, part of the information passed into its msgNew message is the data object for the view. The clsView ancestor class manages this object by registering itself as an observer of it so it can be notified when the data changes. The view also takes care of saving the data object when it receives the msgSave request. Although this is convenient for data objects with just one view, you need to be careful when saving and restoring views that share a data object with other views.

The clsCalcBtVw object used for the interface is created in the clsCalcApp application class described in the last chapter. The actual method used to create the button view is `CalcAppAppInit`, which responds to the `msgAppInit` message and is located in the calcapp.c source file. It reads

```
MsgHandler(CalcAppAppInit)
{
  CALCBTVW_NEW     cbv;
  CALCENG_NEW      cn;
  APP_METRICS      am;
  STATUS           s;

  ObjCallWarn(msgNewDefaults, clsCalcEng, &cn, s);
  ObjCallRet(msgNew, clsCalcEng, &cn, s);

  ObjCallWarn(msgNewDefaults, clsCalcBtVw, &cbv, s);
  cbv.view.dataObject = cn.object.uid;
  ObjCallRet(msgNew, clsCalcBtVw, &cbv, s);

  ObjCallWarn(msgAppGetMetrics, self, &am, s, Error);
  ObjCallJmp(msgFrameSetClientWin, am.mainWin,
                      cbv.object.uid, s, Error);

  return stsOK;
Error:
  return s;
  MsgHandlerParametersNoWarning;
}
```

After the view object is created, it is inserted as the client window for the application. At that point, you can rely upon the Application Framework and inherited behavior to manage much of the mechanical work for the button view. For instance, when saving the place of a user who turns away from the document, the Application Framework starts the closing process which includes asking the window hierarchy to save itself. Behavior inherited by clsCalcBtVw from clsWin insures that all child windows, including the view, accumulator display, and keypad, are also given the opportunity to save their contents. Additionally, inherited behavior from clsView guarantees that the data object will be given a chance to write and store its stateful data. The process will then be reversed when the user reactivates the document by turning back to it.

Managing the Layout In addition to inheriting behavior to manage the data object from clsView, clsCalcBtVw also inherits layout abilities from another of its ancestor classes, clsCustomeLayout. clsCalcBtVw uses its layout capabilities to manage its two windows by setting up relative positioning instructions by responding to a message from its parent window requesting the size and constraints of its children.

This was the logical choice for managing the child windows, since clsCalcBtVw was a subclass of clsCustomLayout as a result of being a subclass of clsView. In general, the use of custom layouts is not restricted to creating new custom layouts by subclassing only. It is also possible to use custom layouts directly by creating one, adding child windows, and then specifying the constraints for each individual child window. However, this method has the disadvantage of requiring a large space overhead, since the information must be stored once for each child window.

Using clsTkTable

clsTkTable is a layout class that it reused by instancing it, as opposed to inheriting from it. clsCalcBtVw uses an instance of clsTkTable to manage the array of buttons that make up the keypad for the user to enter data. The keypad is used to construct numerical values and to indicate when the data object should be updated with the contents of the value. For this application, I've chosen to allow clsCalcBtVw to manage the interaction of the user and the buttons, but some of this responsibility might be handled by a manager constructed especially for this purpose and used to control the table layout.

The Implementation of clsCalcBtVw

The calculator button view works by collecting input from the user that makes up a number, followed by what the user wants to do with the number that follows it. For example, it keeps track of the user pressing the 1 and 2 keys and would present it to the calculator engine (its data object) as the value 12 if the next key pressed was an operator key (add or subtract, for example.). The operator entered is stored as the next operation to perform, and the calculator starts collecting a new value. The next time an operator key is pressed, the new value is sent with the old operator to the calculator engine and a new value is computed.

To provide this capability, clsCalcBtVw is implemented to maintain three pieces of information about the state of the user's interaction with it. First it tracks what the value would be if the next key the user presses is an operator key. Second, it tracks the number of digits the user types. This index is also used to maintain error states in which the user is not allowed to enter any additional numbers. Finally, since the engine it's working with takes a value and operation that is to be applied to the engine's current accumulator value, clsCalcBtVw keeps track of the next operation that should be executed when a user finishes entering a number.

What it doesn't do is process a return value from a calculator engine operation. Instead, it depends on being notified by the calculator engine that a new accumulator value exists or an error has occurred.

The source code that defines clsCalcBtVw is contained in three files: the header file, calcbtvw.h, which contains the external interface to the class; the source file, calcbtvw.c, which contains the support functions and method definitions for clsCalcBtVw; and the method table file, method.tbl, which contains the information needed to map messages sent to clsCalcBtVw objects to the methods that respond to them. The next sections explain each file in detail.

calcbtvw.h

calcbtvw.h contains the external interface for the calculator button view class, clsCalcBtVw. The beginning of the file includes other required definition files, then a macro for building the private Well Known UID that uniquely identifies the class to the Class Manager. The beginning of the file looks like

```
#ifndef CALCBTVW_INCLUDED
#define CALCBTVW_INCLUDED

#ifndef GO_INCLUDED
#include <go.h>
#endif

#ifndef CLSMGR_INCLUDED
#include <clsmgr.h>
#endif

#ifndef VIEW_INCLUDED
#include <view.h>
#endif

#ifndef CLAYOUT_INCLUDED
#include <clayout.h>
#endif

#define clsCalcBtVw MakeGlobalWKN(4144,1)
```

```
STATUS FAR PASCAL ClsCalcBtVwInit(void);
```

Following the definition of clsCalcBtVw is the declaration for the function ClsCalcBtVwInit() which is called by the calculator application to register clsCalcBtVw with the Class Manager.

calcbtvw.h also contains message definitions for the two messages used by the buttons in the keypad table to indicate the method that should be called when the user presses a certain type of button. Those messages are defined

```
#define msgCalcBtVwDigit  MakeMsg(clsCalcBtVw, 1)
#define msgCalcBtVwFnc    MakeMsg(clsCalcBtVw, 2)
```

Finally, calcbtvw.h contains the definitions for the CALCBTVW_NEW structure used to create new instances of clsCalcBtVw:

```
#define calcbtvwNewFields \
  viewNewFields

typedef struct CALCBTVW_NEW {
  calcbtvwNewFields
} CALCBTVW_NEW, *P_CALCBTVW_NEW;

#endif
```

Notice that the CALCBTVW_NEW structure is built using the calcbtvwNewFields macro which has been defined to point back to the viewNewFields macro. This is because clsCalcBtVw doesn't need anything extra during initialization beyond what is already provided to a subclass of clsView.

calcbtvw.c

calcbtvw.c contains the actual implementation of the Calculator Button View class, clsCalcBtVw. It begins by including the header files:

```
#ifndef WIN_INCLUDED
#include <win.h>
#endif

#ifndef FS_INCLUDED // defines for reading and writing to
files
#include <fs.h>
#endif
```

```
#ifndef LABEL_INCLUDED
#include <label.h>
#endif

#ifndef BUTTON_INCLUDED
#include <button.h>
#endif

#ifndef TK_INCLUDED
#include <tk.h>
#endif

#ifndef CALCBTVW_INCLUDED
#include <calcbtvw.h>
#endif

#ifndef CALCENG_INCLUDED
#include <calceng.h>
#endif

#ifndef DEBUG_INCLUDED
#include <debug.h>
#endif

#include <method.h>
#include <stdio.h>
#include <string.h>
```

Following the header file include statements are a series of macro definitions used by the TkTable object in clsCalcBtVw during initialization. They are

```
#define KB_ROW_SPACING 5
#define KB_COL_SPACING 5

#define W_MARGIN 10
#define H_MARGIN 10
```

The `spacing` macros are used to set the number of space units that should exist between individual items contained in the table. The `margin` macros are used to define the amount of space to be left between the outside of the table and the inside of the surrounding parent window.

A final set of macros are defined to be used in communicating the overall state of the calculator to the user with respect to possible errors. These macros are used in conjunction with the index instance variable described below. They are

```
#define ERROR_CONDITION(x) (x<0)
#define OVR_ERROR -1
#define OVR_ERROR_STR "-Overflow-"
#define DIV_ERROR -2
#define DIV_ERROR_STR "-Zero Div-"
```

Next comes the definition of the instance variables used by clsCalcBtVw:

```
typedef struct INSTANCE_DATA {
  S16       index;
  S32       value;
  U32       nextOp;
  OBJECT    model;
  OBJECT    accmDisplay;
  OBJECT    keypad;
} INSTANCE_DATA, *P_INSTANCE_DATA;
```

The first three values (index, value, and nextOp) constitute the stateful data of the clsCalcBtVw object. The **index** instance variable tracks how many digits the user has entered and whether or not the user should be allowed to continue, based on possible error conditions in the calculator engine data object. By definition, a negative index value indicates an error condition occurred. Macros defined at the head of the file manage this. I chose to implement error handling in this manner because the overhead of using status constructs isn't needed: the information is not shared outside the compilable module.

The **value** instance variable tracks the value that should be passed to the calculator engines when the user presses an operator key. The value is computed by multiplying the current value by 10 and adding the result to the value of the number key pressed.

Finally, the **nextOp** instance variable tracks the next operation to be used with the calculator engine. When the user presses an operator key, the message inside nextOp is sent with the contents of the value instance variable as its argument. When that completes, nextOp is set to the value of the operator key that began the operation and the user can again enter input.

The final three instance variables manage the view when it is active. The instance variable **model** is set to the data object being managed by clsCalcBtVw's clsView ancestor class. I have done this to avoid having to query self for the data object every time the user causes an interaction. The next two instance variables, **accmDisplay** and **keypad**, hold the UIDs of the label object and table object that make up the accumulator display and the keypad, respectively.

The Accumulator Child Window One problem with writing code for PenPoint is that the code becomes very verbose, even for the simplest things. I have adopted the style of placing logical units of functionality into functions for the purpose of making some of the code more readable. One such function is CVBuildAccmDisplay(), which constructs the label value used to display the current value of the accumulator. This function is defined

```
STATUS LOCAL CBVBuildAccmDisplay( OBJECT self,
                                  P_OBJECT pDisplay )
{
  LABEL_NEW         ln;
  STATUS            s;

  ObjCallWarn(msgNewDefaults, clsLabel, &ln);
  ln.label.style.scaleUnits    = bsUnitsFitWindowProper;
  ln.label.style.xAlignment    = lsAlignRight;
  ln.border.style.edge         = bsEdgeAll;
  ln.win.flags.style           &= ~wsSendFile;
  ObjCallRet(msgNew, clsLabel, &ln, s);

  *pDisplay = ln.object.uid;
  return stsOK;
}
```

This function takes the UID of the object that called it, along with a pointer to the UID that is to hold the created label object and return a status code. This function and any others that contain the message sending macros must define the function to return `STATUS` to avoid mismatched return type warnings. This requirement is a result of some macros being defined to return `STATUS` on detection of an error.

Inside the `CBVBuildAccmDisplay()` function, a `LABEL_NEW` structure is initialized by sending a msgNewDefaults message to clsLabel. The structure is then modified so that the label's font size is large enough to fit the window it's given during resizing operations (`bsUnitsFitWindowProper`), and that the label's text is right-aligned with an edge around the entire label. Finally, the wsSendFile attribute of the label is turned off, so that when the label's parent window is saved, the label will not be. Instead, clsCalcBtVw messages manage the accumulator (and keypad table) directly. This saves file system space by freeing windows when they are no longer needed and recreating them when the view becomes visible once more.

The Keypad Child Window The calculator button view's keypad is built using a clsTkTable object to manage the creation and layout of buttons that represent the number and operation keys. clsCalcBtVw builds the keypad by calling the `CBVBuildKeypad()` function with the UID of the clsCalcBtVw object and a pointer to the variable holding the UID of the newly created keypad. This function, like `CBVBuildAccm()`, also returns a `stsOK STATUS` if the table is created and an error status otherwise. The function is defined

```
STATUS LOCAL CBVBuildKeypad( OBJECT self , P_OBJECT pKeypad )
{
  TK_TABLE_NEW     tktNew;
  P_BUTTON_NEW     pBn;
  STATUS           s;

  static TK_TABLE_ENTRY keyEntries[] = {
    { "7" , msgCalcBtVwDigit, (U32)7 },
    { "8" , msgCalcBtVwDigit, (U32)8 },
    { "9" , msgCalcBtVwDigit, (U32)9 },
    { "C", msgCalcBtVwFnc, msgCalcEngClr },
    { "4" , msgCalcBtVwDigit, (U32)4 },
    { "5" , msgCalcBtVwDigit, (U32)5 },
    { "6" , msgCalcBtVwDigit, (U32)6 },
    { "/", msgCalcBtVwFnc, msgCalcEngDiv },
    { "1" , msgCalcBtVwDigit, (U32)1 },
    { "2" , msgCalcBtVwDigit, (U32)2 },
    { "3" , msgCalcBtVwDigit, (U32)3 },
    { "X", msgCalcBtVwFnc, msgCalcEngMul },
    { "0" , msgCalcBtVwDigit, (U32)0 },
    { "+", msgCalcBtVwFnc, msgCalcEngAdd },
    { "-", msgCalcBtVwFnc, msgCalcEngSub },
    { "=", msgCalcBtVwFnc, msgCalcEngSetAccm },
    {pNull}
  };

  ObjCallWarn( msgNewDefaults, clsTkTable, &tktNew);

  tktNew.win.flags.style              |= wsTransparent;
  tktNew.win.flags.style&                = ~wsSendFile;
  tktNew.border.style.backgroundInk      = bsInkGray33;
  tktNew.border.style.edge               = bsEdgeNone;
  tktNew.tableLayout.style.growChildWidth = true;
  tktNew.tableLayout.style.growChildHeight= true;
  tktNew.tableLayout.numRows.constraint   = tlAbsolute;
  tktNew.tableLayout.numRows.value        = 4;
  tktNew.tableLayout.numCols.constraint   = tlAbsolute;
  tktNew.tableLayout.numCols.value        = 4;
  tktNew.tableLayout.rowHeight.constraint = tlMaxFit;
  tktNew.tableLayout.rowHeight.gap        = KB_ROW_SPACING;
```

```
    tktNew.tableLayout.colWidth.constraint  = tlMaxFit;
    tktNew.tableLayout.colWidth.gap         = KB_COL_SPACING;
    tktNew.tkTable.client                   = self;
    tktNew.tkTable.pEntries = (P_TK_TABLE_ENTRY)keyEntries;

    pBn = tktNew.tkTable.pButtonNew;
    pBn->label.style.scaleUnits          = bsUnitsFitWindowProper;
    pBn->border.style.edge               = bsEdgeAll;
    pBn->border.style.join               = bsJoinSquare;
    pBn->border.style.shadow             = bsShadowNone;
    pBn->gWin.style.gestureEnable        = false;
    pBn->gWin.style.gestureForward       = false;

    ObjCallRet(msgNew, clsTkTable, &tktNew, s);

    *pKeypad = tktNew.object.uid;
    return stsOK;
}
```

Creating a table of buttons to be used as the calculator's keypad involves two steps. First, the buttons' layout must be specified. Second, the information that makes each button unique must be organized and presented to the table in an organized manner.

The first step in creating a new clsTkTable object is to send msgNewDefaults to clsTkTable and pass to it an address of the `TK_TABLE_NEW` structure to be filled in. For the calculator keypad, the following statements cause the window containing the individual buttons to be transparent.

```
tktNew.win.flags.style | = wsTransparent;
tktNew.win.flags.style & = ~wsSendFile;
```

As a result of these statements, the TkTable parent window becomes the background that surrounds the space not covered by the individual buttons. Then, by turning off the wsSendFile flag, you tell the TkTable object not to save its components when its parent is being saved.

The next two statements set the border style with a InkGray33 background and no edge:

```
tktNew.border.style.backgroundInk   = bsInkGray33;
tktNew.border.style.edge            = bsEdgeNone;
```

It is possible to specify exact shades for the background color, but in doing so, you take risks with the visual appearance of your object. Whenever possible, use the predefined default values because they have been developed to look good, regardless of the platform your application runs on.

The next set of statements specify the attributes of the table layout itself.

```
tktNew.tableLayout.style.growChildWidth    = true;
tktNew.tableLayout.style.growChildHeigh    = true;
tktNew.tableLayout.numRows.constraint      = tlAbsolute;
tktNew.tableLayout.numRows.value           = 4;
tktNew.tableLayout.numCols.constraint      = tlAbsolute;
tktNew.tableLayout.numCols.value           = 4;
tktNew.tableLayout.rowHeight.constraint    = tlMaxFit;
tktNew.tableLayout.rowHeight.gap= KB_ROW_SPACING;
tktNew.tableLayout.colWidth.constraint     = tlMaxFit;
tktNew.tableLayout.colWidth.gap= KB_COL_SPACING;
```

The table used to build the keypad allows its children to grow to the maximum size allocated to them in the tabular grid; the key pad will have exactly four rows and four columns. Finally, the table specifies the amount of space between rows and columns in the table.

Following the positioning information come the actual control data and control management information beginning with the statements:

```
tktNew.tkTable.client       = self;
tktNew.tkTable.pEntries     = (P_TK_TABLE_ENTRY)keyEntries;
```

These statements tell the TkTable object to make the client of each control it creates self. Within this application that means the clsCalcBtVw object. Next the individual button data is taken from the array keyEntries, which was defined

```
static TK_TABLE_ENTRY keyEntries[] = {
  { "7" , msgCalcBtVwDigit, (U32)7 },
  { "8" , msgCalcBtVwDigit, (U32)8 },
  { "9" , msgCalcBtVwDigit, (U32)9 },
  { "C", msgCalcBtVwFnc, msgCalcEngClr },
  { "4" , msgCalcBtVwDigit, (U32)4 },
  .......
  {pNull}
};
```

clsTkTable looks at the number of rows and columns and uses that information to place each control in its proper location. Nothing is specified for the keypad table because the default of row-by-column suffices. Each entry in the array specifies the label to be applied to the control, the message to be sent when the control is activated, and the value to be sent as data with the message being sent. The data is not restricted to simple numeric values—it can be any 32-bit piece of information. In the case of the keypad, number buttons send the value they represent, while operator buttons send the message that should be used to carry out that operation.

Finally, the last several lines of code are used to specify attributes common to all the buttons in the table. They are

```
pBn = tktNew.tkTable.pButtonNew;
pBn->label.style.scaleUnits        = bsUnitsFitWindowProper;
pBn->border.style.edge             = bsEdgeAll;
pBn->border.style.join             = bsJoinSquare;
pBn->border.style.shadow           = bsShadowNone;
pBn->gWin.style.gestureEnable      = false;
pBn->gWin.style.gestureForward     = false;
```

Although I have been using buttons in this example, a table can be used to hold any type of control. The major differences are in the control definition structure because the values in the array change based on the type of control being used.

Registering clsCalcBtVw For a clsCalcBtVw object to be created, it must first be registered with the Class Manager as a subclass of clsView with space set aside for the instance data defined in the `INSTANCE_DATA` structure. Registration is done using the function:

```
STATUS ClsCalcBtVwInit(void)
{
  CLASS_NEW       c;
  STATUS          s

  ObjCallWarn(msgNewDefaults, clsClass, &c);
  c.object.uid          = clsCalcBtVw;
  c.class.pMsg          = clsCalcBtVwTable;
  c.class.ancestor      = clsView;
  c.class.size          = SizeOf(INSTANCE_DATA);
  c.class.newArgsSize   = SizeOf(CALCBTVW_NEW);
  ObjectCallJmp(msgNew, clsClass, &c, s, Error);

  return stsOK;
Error:
  return s;
}
```

Initializing the View When the Calculator Application class creates an instance of clsCalcBtVw, it is initialized using the CalcBtVwInit method, which is defined

```
MsgHandlerArgType(CalcBtVwInit, P_CALCBTVW_NEW)
{
  INSTANCE_DATA        inst;
  WIN_METRICS          wm;
  BORDER_STYLE         bs;
  CALCENG_NEW          cn;
  STATUS               s;

  inst.value           = 0;
  inst.index           = 0;
  inst.nextOp          = msgCalcEngSetAccm;

  if (!(pArgs->view.dataObject) &&
        pArgs->view.createDataObject) {
  ObjCallWarn(msgNewDefaults, clsCalcEng, &cn);
  ObjCallRet(msgNew, clsCalcEng, &cn, s);
  ObjCallRet(msgViewSetDataObject, self, cn.object.uid,s);
  inst.model = cn.object.uid;
  }
  else
        inst.model = pArgs->view.dataObject;

  CBVBuildAccmDisplay( self, &inst.accmDisplay );
  CBVBuildKeypad( self, &inst.keypad );

  ObjectWrite(self, ctx, &inst);

  ObjCallWarn(msgBorderGetStyle, self, &bs);
  bs.backgroundInk = bsInkGray33;
  ObjCallRet(msgBorderSetStyle, self, &bs, s );

  ObjCallRet(msgLabelSetString, inst.accmDisplay, "0", s );

  wm.parent = self;
  wm.options = wsPosTop;
  ObjCallRet( msgWinInsert, inst.accmDisplay, &wm, s );
  wm.parent = self;
  wm.options = wsPosBottom;
  ObjCallRet( msgWinInsert, inst.keypad, &wm, s );

  return stsOK;
  MsgHandlerParametersNoWarning;
}
```

The first three lines of the method

```
inst.value      = 0;
inst.index      = 0;
inst.nextOp     = msgCalcEngSetAccm;
```

are responsible for initializing the persistent part of the instance data so that the current value is zero, with no errors detected, and the first operation to be done when the user indicates a value is ready for the calculator engine, is to set the calculator engine's accumulator to that value.

Next, the local copy of the data object is set if available, otherwise, the following code checks to see if it should be created.

```
if (!(pArgs->view.dataObject) &&
    pArgs->view.createDataObject) {
    ObjCallWarn(msgNewDefaults, clsCalcEng, &cn);
    ObjCallRet(msgNew, clsCalcEng, &cn, s);
    ObjCallRet(msgViewSetDataObject, self, cn.object.uid,s);
    inst.model = cn.object.uid;
    }
else
    inst.model = pArgs->view.dataObject;
```

This code is the responsibility of the subclass of clsView. It allows someone to use an instance of the view object without worrying about creating the appropriate data object.

Next

```
CBVBuildAccmDisplay( self, &inst.accmDisplay );
CBVBuildKeypad( self, &inst.keypad );

ObjectWrite(self, ctx, &inst);
```

creates the accumulator display and keypad child windows used to implement the calculator button view. Once completed, the instance data is written into protected memory.

The last piece of code in the initialization function

```
ObjCallWarn(msgBorderGetStyle, self, &bs);
bs.backgroundInk = bsInkGray33;
ObjCallRet(msgBorderSetStyle, self, &bs, s );

ObjCallRet(msgLabelSetString, inst.accmDisplay, "0", s );

wm.parent = self;
wm.options = wsPosTop;
```

```
ObjCallRet( msgWinInsert, inst.accmDisplay, &wm, s );
wm.parent = self;
wm.options = wsPosBottom;
ObjCallRet( msgWinInsert, inst.keypad, &wm, s );
```

accomplishes the initialization of both the clsCalcBtVw's background and the text in the accumulator display object, followed by the insertion of the accumulator display and keypad windows as children of the view itself.

Initializing and Freeing the View Because the accumulator display and keypad windows have been removed from the normal child window processing loop, they need to be freed. This is done by overriding the method for freeing a clsView by adding a new CalcBtVwFree method:

```
MsgHandlerWithTypes(CalcBtVwFree, P_ARGS, P_INSTANCE_DATA)
{
  STATUS s;

  ObjCallRet(msgDestroy, pData->accmDisplay, objNull, s );
  ObjCallRet(msgDestroy, pData->keypad, objNull, s );

  return stsOK;
  MsgHandlerParametersNoWarning;
}
```

Changing the View's Data Object Because clsCalcBtVw objects keep a shadow copy of the data object as one of their instance variables (model), it is necessary to know when that value changes. Currently, that can happen at two times. First, when the instance of clsCalcBtVw is initialized, and second, when the consumer of the object sends a msgViewSetDataObject message to the calculator button view. The initialization method CalcBtVwInit handles the first case, while the second case is handled by overriding the method used to respond to `msgViewSetDataObject` with

```
MsgHandlerArgType(CalcBtVwSetDataObject, OBJECT )
{
  INSTANCE_DATA   inst;

  inst = IDataDeref( pData, INSTANCE_DATA );
  inst.model = pArgs;
  ObjectWrite(self, ctx, &inst);

  return stsOK;
  MsgHandlerParametersNoWarning;
}
```

This method de-references the instance data set aside for the instance of clsCalcBtVw, sets the model instance variable to the new data object, and then rewrites the instance data back into the protected area of memory. Notice in the method table definition that this method allows its ancestor's method to be called prior to executing its own behavior.

Saving State clsCalcBtVw objects are sent msgSave as a result of their parent window being told to save its state. clsCalcBtVw objects respond by writing only their persistent data to a file, choosing to allow their child windows to be freed. It then rebuilds those windows as a result of receiving a msgRestore message.

The method used for saving the clsCalcBtVw's state is

```
MsgHandlerWithTypes(CalcBtVwSave, P_OBJ_SAVE,
                                  P_INSTANCE_DATA)
{
  STREAM_READ_WRITE      fsWrite;
  STATUS                 s;

  fsWrite.numBytes=SizeOf(S16);
  fsWrite.pBuf= &(pData->index);
  ObjCallRet(msgStreamWrite, pArgs->file, &fsWrite, s);
  fsWrite.numBytes= SizeOf(S32);
  fsWrite.pBuf= &(pData->value);
  ObjCallRet(msgStreamWrite, pArgs->file, &fsWrite, s);
  fsWrite.numBytes= SizeOf(U32);
  fsWrite.pBuf= &(pData->nextOp);
  ObjCallRet(msgStreamWrite, pArgs->file, &fsWrite, s);

  return stsOK;
  MsgHandlerParametersNoWarning;
}
```

For this example, I adopted the convention for saving instance data which requires that individual data items be saved in a file. An alternative would have been to define a substructure that contained only the persistent data and to replace the three msgStreamWrite messages with a single message.

Restoring State clsCaclBtVw objects receive a msgRestore message when it's their turn to be re-created from a file during a window hierarchy's process of rebuilding itself. The method that handles msgRestore for clsCalcBtVw is

```
MsgHandlerArgType(CalcBtVwRestore, P_OBJ_RESTORE)
{
```

```
INSTANCE_DATA             inst;
STREAM_READ_WRITE         fsRead;
WIN_METRICS               wm;
STATUS                    s;
char                      buff[CALCENG_MAX_DIGITS+1];

fsRead.numBytes =SizeOf(S16);
fsRead.pBuf     = &inst.index;
ObjCallRet(msgStreamRead, pArgs->file, &fsRead, s);
fsRead.numBytes = SizeOf(S32);
fsRead.pBuf     = &inst.value;
ObjCallRet(msgStreamRead, pArgs->file, &fsRead, s);
fsRead.numBytes = SizeOf(U32);
fsRead.pBuf     = &inst.nextOp;
ObjCallRet(msgStreamRead, pArgs->file, &fsRead, s);

ObjCallRet( msgViewGetDataObject, self, &inst.model, s);

CBVBuildAccmDisplay( self, &inst.accmDisplay );
CBVBuildKeypad( self, &inst.keypad );

ObjectWrite(self, ctx, &inst);

switch( inst.index ) {
      case OVR_ERROR:
           strcpy( buff, OVR_ERROR_STR );
           break;

      case DIV_ERROR:
           strcpy( buff, DIV_ERROR_STR );
           break;

      default:
           sprintf( buff, "%ld", inst.value );
           break;
}
ObjCallRet(msgLabelSetString, inst.accmDisplay, buff, s );

wm.parent        = self;
wm.options       = wsPosTop;
ObjCallRet( msgWinInsert, inst.accmDisplay, &wm, s );
wm.parent        = self;
wm.options       = wsPosBottom;
ObjCallRet( msgWinInsert, inst.keypad, &wm, s );
```

```
    return stsOK;
    MsgHandlerParametersNoWarning;
}
```

The CalcBtVwRestore method is similar to the initialization method in that it first restores the stateful data, then it gets the value of the data object that has been restored by its superclass. Next it re-creates the accumulator and keypad views used. After it re-creates itself, it writes its information back into protected memory. Next, it checks the contents of the index instance variable to determine if an error condition exists and sets the initial value of the accumulator label value accordingly. Finally, it reinstalls its child windows into the window hierarchy so they can be properly displayed.

Managing the Layout of the Child Windows It is important for subclasses of clsCustomLayout to provide a method for specifying the layout information for the child windows it creates. The clsCalcBtVw class provides this information by overriding the msgCstmLayoutGetChildSpec method defined in its ancestor with the following method:

```
MsgHandlerWithTypes(
    CalcBtVwCLGetChildSpec, P_CSTM_LAYOUT_CHILD_SPEC,
    P_INSTANCE_DATA)
{
    if ( pArgs->child == pData->accmDisplay ) {
        pArgs->metrics.h.constraint         = clPctOf;
        pArgs->metrics.h.value              = 15;
        pArgs->metrics.w.constraint         = clSameAs;
        pArgs->metrics.w.value              = -(2*W_MARGIN);
        pArgs->metrics.x.constraint=
              ClAlign(clCenterEdge, clSameAs, clCenterEdge);
        pArgs->metrics.y.constraint         =
              ClAlign(clMaxEdge, clSameAs, clMaxEdge);
        pArgs->metrics.y.value              = -H_MARGIN;
        }
    else if ( pArgs->child == pData->keypad ) {
        pArgs->metrics.h.relWin             = pData->accmDisplay;
        pArgs->metrics.h.constraint         =
              ClExtend(clSameAs, clMinEdge);
        pArgs->metrics.h.value              = -H_MARGIN;
        pArgs->metrics.w.relWin             = pData->AccmDisplay;
        pArgs->metrics.w.constraint         = clSameAs;
        pArgs->metrics.x.relWin             = pData->AccmDisplay;
        pArgs->metrics.x.constraint         = clSameAs;
```

```
        pArgs->metrics.y.constraint         =
              ClAlign(clMinEdge, clSameAs, clMinEdge);
        pArgs->metrics.y.value              = H_MARGIN;
        }
    return (stsOK);
    MsgHandlerParametersNoWarning;
}
```

This single method provides the information for each of its child windows. It uses a set of if-then-else constructs to check which object is being laid out and sets the CSTM_LAYOUT_CHILD_SPEC child specification structure accordingly. This structure contains a set of metrics for specifying information about four types of constraints: height, width, x, and y. These metrics include both relative positioning and absolute adjustments. For example, a window could be told to make its height the same as another window's except that it should be smaller by a certain number of units.

In the case of clsCalcBtVw's children, if the child being laid out is the accumulator display, then the structure is set with

```
pArgs->metrics.h.constraint             = clPctOf;
pArgs->metrics.h.value                  = 15;
pArgs->metrics.w.constraint             = clSameAs;
pArgs->metrics.w.value                  = -(2*W_MARGIN);
pArgs->metrics.x.constraint             =
      ClAlign(clCenterEdge, clSameAs, clCenterEdge);
pArgs->metrics.y.constraint             =
      ClAlign(clMaxEdge, clSameAs, clMaxEdge);
pArgs->metrics.y.value                  = -H_MARGIN;
```

This indicates that the accumulator's display height should be 15 percent of its parent's height and its width should be the same as its parent's width minus two times the value of the w_margin macro. It specifies that the center of its x edge should be the same as the center of its parent's x edge, relying on the values it specifies for its width constraint to make it fit properly on its parent. In the same way, it uses the CLAlign() macro to set its y constraint.

In a similar manner, the layout specification for the keypad table child window is

```
pArgs->metrics.h.relWin         = pData->accmDisplay;
pArgs->metrics.h.constraint     =
     ClExtend(clSameAs, clMinEdge);
pArgs->metrics.h.value          = -H_MARGIN;
pArgs->metrics.w.relWin         = pData->AccmDisplay;
```

```
pArgs->metrics.w.constraint     = clSameAs;
pArgs->metrics.x.relWin         = pData->AccmDisplay;
pArgs->metrics.x.constraint     = clSameAs;
pArgs->metrics.y.constraint     =
    ClAlign(clMinEdge, clSameAs, clMinEdge);
pArgs->metrics.y.value          = H_MARGIN;
```

As you can see, the specification for the keypad is very similar to that of the accumulator display. However, one interesting difference concerns specifying the height of the keypad. I have chosen to specify height as relative to another window other than the parent by setting

```
pArgs->metrics.h.relWin       = pData->accmDisplay;
pArgs->metrics.h.constraint   =
    ClExtend(clSameAs, clMinEdge);
pArgs->metrics.h.value        = -H_MARGIN;
```

and then using the `CLExtend()` macro to extend the height of the keypad so it reaches the minimum edge of the accmDisplay child window minus the value of `h_margin`.

What is often overlooked is the amount of work PenPoint is doing in computing what should go where when windows are given sizes in values relative to other windows. You have to be careful that *something* is given a set of constraints that can be figured out without dropping into a cyclical loop. In this example, I have set the accumulator display to be a certain height (a percentage of its parent window) and do not alter that in the layout specifications.

Managing User Interaction The user interacts with the calculator view by tapping the keypad buttons. These buttons have been defined in the `TK_TABLE_ENTRY` structure to send one of two messages, depending on the button pressed. The clsCalcBtVw object also receives a value indicating the exact button pushed.

The first message, msgCalcBtVwDigit, is sent when the user presses one of the digit (0-9) keys. The method that handles this message is

```
MsgHandlerWithTypes(CalcBtVwDigit, P_ARGS, P_INSTANCE_DATA)
{
  INSTANCE_DATA    inst;
  int              val;
  STATUS           s;
  char             buff[CALCENG_MAX_DIGITS+1];

  if ( ERROR_CONDITION(pData->index) )
    return stsOK;
```

```
    if ( pData->index >= CALCENG_MAX_DIGITS )
      return stsOK;

    inst = IDataDeref( pData, INSTANCE_DATA );

    val = (int)pArgs;
    if ( inst.index == 0 )
      inst.value = val;
    else
      inst.value = inst.value*10 + val;
    ObjectWrite( self, ctx, &inst );

    sprintf( buff, "%ld", inst.value );
    ObjCallRet(msgLabelSetString, inst.accmDisplay, buff, s);
    inst.index++;

    return stsOK;
    MsgHandlerParametersNoWarning;
}
```

The first thing this method does is to check if the user is still able to add to the value by checking whether an error condition exists or whether the maximum number of allowable digits has already been typed. If neither condition is true, it then de-references the instance data and proceeds to update the contents of the value instance variable. Once complete, it then updates the accumulator display, increments the index, and writes the instance data back into protected memory.

The user pressing an operator key (+,-,*,/,=,C) causes the msgCalc-BtVwFnc message to be sent with the message that should be used as the next operation. The method that responds to this message is

```
MsgHandlerWithTypes(CalcBtVwFnc, P_ARGS, P_INSTANCE_DATA)
{
  INSTANCE_DATA inst;
  CALCENG_VAL   cv;
  U32           nextSel;
  STATUS        s;

  if ( (U32)pArgs == msgCalcEngClr ) {
     ObjCallRet(msgCalcEngClr, pData->model, &cv, s );
     nextSel = msgCalcEngSetAccm;
     }
  else if ( ERROR_CONDITION(pData->index) ) {
     return stsOK;
     }
```

```
    else if ( pData->index != 0 ) {
       cv.value = pData->value;
       ObjCallRet(pData->nextOp, pData->model, &cv, s );
       nextSel = (U32)pArgs;
       }
    else
       nextSel = pArgs;

    inst = IDataDeref( pData, INSTANCE_DATA );
    inst.nextOp = nextSel;
    ObjectWrite( self, ctx, &inst );

    return stsOK;
    MsgHandlerParametersNoWarning;
}
```

If you are familiar with other object-based programming environments, then you recognize this as more of a controller type behavior than a view behavior. However, as is often the case, both types of behaviors get rolled into the same object. In essence, this method decides how the view should be allowed to update the data object that is the calculator engine.

First, it checks to see if the user pressed the Clear button. If so, the states of the engine and the view are reset and control is returned. Next, if the Clear key wasn't pressed, this method checks to see if an error condition exists. If so, it returns without allowing the user to change anything. This forces the user to press Clear to recover from an error state.

After checks for clear and errors, the next thing checked is whether the user has entered a value yet. If so, that value is passed to the calculator engine via the message contained in the nextOp instance variable, and nextOp is set to be the operation that was pushed, causing the method to be invoked. If the user has not entered a value, then the next operation is changed to the one just entered.

Notice that I'm not waiting for the calculator engine to return to me the results of the operation request. Instead, I'm assuming that the request makes it and I'm relying on the data object to send me an update message indicating whether the accumulator value has changed or an error has occurred.

Finally, I de-reference the instance data, update the value of nextOp and rewrite the data to protected memory. I needed to be very careful about de-referencing and updating instance data in an asynchronous environment because the opportunity exists for other methods to de-reference and rewrite instance data as a result of a message I send. A good rule of thumb is avoid de-referencing data when a message you send might cause another message to be sent back to you.

Handling Updates from the Calculator Engine That last two methods are used to respond to messages sent by the calculator engine to its view when the contents of its accumulator changes or it perceives an error has occurred. The first method is used to respond to the msgCalcEngAccm-Changed and is defined

```
MsgHandlerArgType(CalcBtVwAccmChanged, P_CALCENG_VAL)
{
    INSTANCE_DATA           inst;
    STATUS                  s;
    char                    buff[CALCENG_MAX_DIGITS+1];

    inst = IDataDeref( pData, INSTANCE_DATA );
    inst.value = pArgs->value;
    inst.index = 0;
    ObjectWrite( self, ctx, &inst );

    sprintf( buff, "%ld", inst.value );
    ObjCallRet(msgLabelSetString, inst.accmDisplay, buff, s );

    return stsOK;
    MsgHandlerParametersNoWarning;
}
```

This method simply updates the value and index instance variables to reflect the change in the calculator engine and then updates the accumulator display.

The second method in clsCalcBtVw is used to respond to a msgCalc-EngError sent by the calculator engine to its view. That method is defined

```
MsgHandlerArgType(CalcBtVwAccmError, P_STATUS)
{


    INSTANCE_DATA inst;
    STATUS        s;
    char          buff[256];

    inst = IDataDeref( pData, INSTANCE_DATA );

    inst.value = 0;
    switch( *pArgs ) {
        case stsCalcEngOverflow:
            inst.index = OVR_ERROR;
            strcpy( buff, OVR_ERROR_STR );
            break;
```

```
        case stsCalcEngZeroDiv:
            inst.index = DIV_ERROR;
            strcpy( buff, DIV_ERROR_STR );
            break;
        }
    ObjectWrite( self, ctx, &inst );
    ObjCallRet(msgLabelSetString, inst.accmDisplay, buff, s );

    return stsOK;
    MsgHandlerParametersNoWarning;
}
```

This method checks the error type and then sets the index instance variable appropriately. It also changes the accumulator display label value to indicate the type of error that occurred.

method.tbl

In this final section, I present the complete method table file method.tbl, which contains definitions for all three classes (clsCalcApp, clsCalcEng, and clsCalcBtVw). It is defined

```
#ifndef CLSMGR_INCLUDED
#include <clsmgr.h>
#endif

#ifndef APP_INCLUDED
#include <app.h>
#endif

#ifndef WIN_INCLUDED
#include <win.h>
#endif

#ifndef CLAYOUT_INCLUDED
#include <clayout.h>
#endif

#ifndef CALCENG_INCLUDED
#include <calceng.h>
#endif

#ifndef CALCBTVW_INCLUDED
#include <calcbtvw.h>
#endif
```

```
MSG_INFO clsCalcAppMethods[] = {
  msgAppInit,"CalcAppAppInit",                  objCallAncestorBefore,
  0
};

MSG_INFO clsCalcEngMethods[] = {
  msgInit,              "CalcEngInit",          objCallAncestorBefore,
  msgSave,              "CalcEngSave",          objCallAncestorBefore,
  msgRestore,           "CalcEngRestore",       objCallAncestorBefore,
  msgCalcEngGetAccm,    "CalcEngGetAccm",       0,
  msgCalcEngClr,        "CalcEngClr",           0,
  msgCalcEngAdd,        "CalcEngAdd",           0,
  msgCalcEngSub,        "CalcEngSub",           0,
  msgCalcEngMul,        "CalcEngMul",           0,
  msgCalcEngDiv,        "CalcEngDiv",           0,
  msgCalcEngSetAccm,    "CalcEngSetAccm",       0,
  0
};

MSG_INFO clsCalcBtVwMethods[] = {
  msgInit,              "CalcBtVwInit",         objCallAncestorBefore,
  msgFree,              "CalcBtVwFree",         objCallAncestorAfter,
  msgSave,              "CalcBtVwSave",         objCallAncestorBefore,
  msgRestore,           "CalcBtVwRestore",      objCallAncestorBefore,
  msgViewSetDataObject,  "CalcBtVwSetDataObject",
                                                objCallAncestorBefore,
msgCstmLayoutGetChildSpec, "CalcBtVwCLGetChildSpec",
                                                objCallAncestorBefore,
  msgCalcBtVwDigit,     "CalcBtVwDigit",        0,
  msgCalcBtVwFnc,       "CalcBtVwFnc",          0,
  msgCalcEngAccmChanged,"CalcBtVwAccmChanged", 0,
  msgCalcEngError,      "CalcBtVwAccmError",  0,
  0
};

CLASS_INFO classInfo[] = {
  "clsCalcAppTable",    clsCalcAppMethods,      0,
  "clsCalcEngTable",    clsCalcEngMethods,      0,
  "clsCalcBtVwTable",   clsCalcBtVwMethods,     0,
  0
};
```

Wrap-up

This chapter presented some of the most powerful parts of PenPoint's user interface toolkit. You have seen the relatively small amount of code needed to build some of the most common forms of user interface objects: those grouped into subviews and, more specifically, those whose subviews can be expressed as tables.

I invite you to take some time to consider the following object-driven enhancements to the clsCalcBtVw class. First, a rule of thumb in object-based programming is that the presence of a case statement indicates the need for a better class decomposition. This is very evident in the manner in which the accumulator display is updated. It should be a very simple exercise to subclass clsLabel to build a new control, clsAccmDisplay, that responds to messages indicating error status. In the same way, there is a general need for a keypad object. You might consider subclassing clsTkTable to develop an actual clsKeypad that would build a standard keypad, plus provide a standard controller for how the keys work together in sequence.

My motivation in asking you to look at these enhancements is simple: in the process of building a custom application, you are now in the position to add a little extra effort in exchange for increasing the number of potential reusers of your custom control objects. If nothing else, you can add these new objects to your bag of tricks for the next application you build.

7

Using the Pen

As an applications developer, I am always looking for ways to make software more accessible to the end-user. One way I accomplish this is to follow naturally occurring metaphors whenever possible. For example, what could be more natural to the user than the button-based calculator implemented in the past several chapters? It looks and works like a desk calculator. The user sees this electronic PenPoint version and immediately knows how to interact with it. It just couldn't get any better.

That's what I thought until the people at the first PenPoint seminar I attended pointed out that we actually do calculations on paper a little differently; we write numbers and operators on paper and then solve the problem. I realized that a lot of the metaphors I was thinking of using were based on the compromises necessary to have electronic help available for tasks I normally do on paper. For example, I mistakenly assumed that the hardware-constrained metaphor of a button-based calculator would be best on a pen-based system without fully thinking the problem through.

In this chapter, I discuss PenPoint handwriting recognition by presenting two slightly different examples of calculator applications. The first sample program replaces the keypad input with two numerical and one character input components. The user forms a calculation by handwriting two numbers and an operator into the input components; the result is computed whenever the pen moves out of proximity (away from the screen). The second application actually changes the metaphor of performing calculations to one of pen and paper using a scratch-pad approach.

Library Support for Handwriting Recognition

PenPoint provides several levels of support for translating handwritten input into information the application can use. Objects exist at the lowest level for collecting, storing, and translating raw pen input into character and gesture data. At the form building level are a set of predefined components for simple well-defined types of input, such as integers and small amounts of text. At the text entry level are components that can accept, translate, and render large amounts of information in several standard formats, including Microsoft's RTF (Rich Text Format).

In addition to components for capturing input, a set of translation utilities, both libraries and classes, allows you to capture application specific information, such as allowable characters, and make it available to the **handwriting recognition system (HWX)**. By combining various objects and library routines, you can quickly construct a user interface that handles validation of handwritten input at various times and gives the user consistent feedback on how to enter the intended data correctly.

The Handwriting Recognition Process

The process of going from the user moving the pen on the tablet to the application receiving interesting data involves several stages. First, PenPoint monitors and records the physical movement of the pen on the display screen. Second, PenPoint collects individual pen movements and organizes them in groups called **scribbles**. Third, PenPoint translates scribbles into usable data such as gestures and characters. Finally, PenPoint notifies interested objects when the translated data is available.

The object responsible for handwriting recognition for PenPoint applications is the **HWX Engine**. Currently the HWX Engine's functionality, implemented as a real-time background task, includes recognizing both uppercase and lowercase letters, numbers, symbols, and punctuation that the user presents to the system on standard ruled "paper." The HWX Engine's task of character recognition can be aided in several ways. First, the user can provide information to the HWX Engine by writing inside boxed input panels that assist the HWX Engine in character segmentation. Second, the programmer can provide information to the HWX Engine that indicates valid values for a particular input object.

Finally, since the HWX Engine is an object, another object that meets the specifications for handwriting recognition can replace it at any time. This allows advances in handwriting recognition to be introduced apart from updates to the operating system. It also has the potential to optimize the translation process for various languages by building HWX Engines for specific languages.

The Acetate Layer Grab a pen and a piece of clean paper and draw a box in the center of it. Now, draw a line from the upper left to the lower right corner of the box. Simple. Now, for the sake of argument, assume that you could interact in exactly the same way with an application on a pen-based machine. Unlike a mouse, whose visual feedback is an on-screen cursor, the pen gives you visual feedback when you write directly on the portion of the screen that interests you—you can watch "ink" dribble out of the pen as you move it across the screen.

PenPoint's **acetate layer** provides this form of visual feedback to the user. This layer coordinates the pen's movement anywhere on the screen by providing an optional "flow of ink" that trails the pen's point and provides visual feedback to the user. The acetate layer then collects the individual pen movements, turns them into input events, and places them in the input queue.

Input Events PenPoint generates input events in one of two ways: through device drivers that collect user interactions and format them into data records that represent events; or through software emulation of hardware events that allow an application to simulate what a user might do. These device drivers include both pen and non-pen (keyboard, for example) input devices. However, because the main focus of PenPoint's input system is the pen, I'll highlight pen events.

Table 7.1 lists the standard pen events. As you look over the list, you'll notice a reference to the pen moving in and out of proximity. **Proximity** refers to the time that the tablet can sense the location of the pen, but the user is not touching the screen with the tip of the pen. In essence, this provides another degree of freedom (in addition to x,y location) that the application can use while interacting with the user. For example, the most common use of out-of-proximity is when the user has written on the acetate and then moves the pen off the tablet, indicating that a translation from strokes to data should occur.

TABLE 7.1 Pen Events.

Event	*Meaning*
eventTipUp	Pen tip in proximity
eventTipDown	Pen tip touches the screen
eventMoveUp	Pen moved while in proximity
eventEnterUp	Pen entered a window in proximity
eventEnterDown	Pen entered a window touching the screen
eventExitUp	Pen exited a window in proximity
eventExitDown	Pen exited a window touching the screen
eventInProxUp	Pen entered proximity, but not touching
eventInProxDown	Pen entered proximity, touching the screen
eventOutProxUp	Pen exited proximity, but not touching
eventOutProxDown	Pen exited proximity, touching the screen
eventStroke	Pen made a stroke
eventTap	Pen made a tap
eventTimeout	Pen up and gesture timed out
eventHoldTimeout	Pen down and hold timed out
eventHWTimeout	Pen up and handwriting timed out
eventOther	Used for any other pen action

Scribbles Scribbles, the intermediate objects used to collect stroke input events that will be passed to the HWX Engine for translation, are supported by **clsScribble**, a subclass of clsObject. clsScribble manages both the collection of pen events and the interface used by the translators to access them. Because scribbles are used often, they have been highly optimized in the current implementation of PenPoint, and therefore, you should be careful to observe the API when using them.

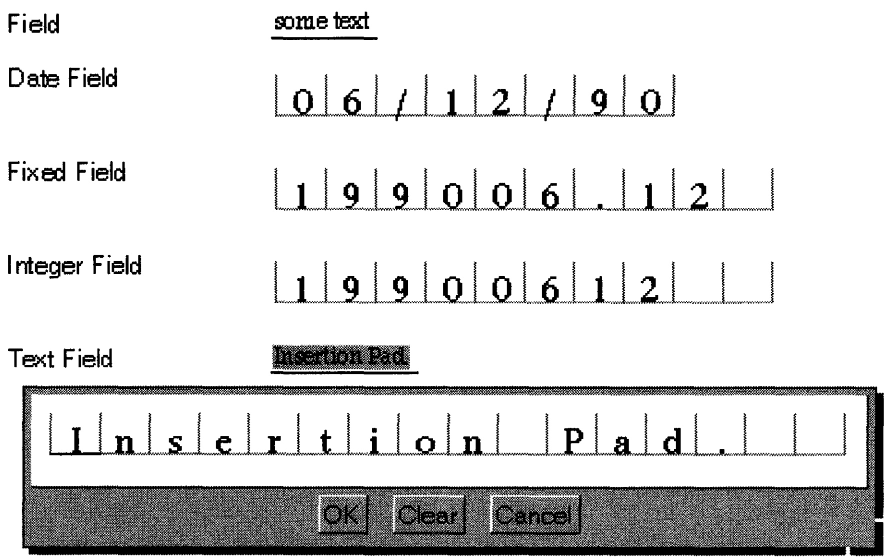
In addition to collecting and returning groups of strokes, scribbles can also selectively add and delete strokes from a self-contained strokelist. Finally, scribbles contain the behavior necessary to render the pen strokes onto a valid drawing context, allowing efficient repair of damaged windows containing scribbles.

Handwritten Input

One common method for a user to interact with an application is through a group of related data fields in a form format. These groups are known by several different titles, including dialog boxes, data entry panels, and option sheets. Usually, the information can be exchanged using discrete selection style controls such as menus, radio buttons, and toggles. But in some cases, the user must enter analog style data, such as numbers and small pieces of text. In PenPoint, analog data is entered by writing on the screen.

Fields PenPoint supports non-discrete data entry by providing the application builder with a subclass of clsLabel, **clsField**, that can collect handwritten data and translate it into the appropriate information. clsField and its subclasses **clsIntegerField**, **clsDateField**, **clsFixedField**, and **clsTextField** are shown in Figure 7.1. They provide you with several well-defined prebuilt components for managing data entry, including hooks for using your application to validate context specific information that might be contained in the entry field.

FIGURE 7.1 Predefined Components for Handwritten Input



clsField objects can interact with the user in different ways, based on the style information used to create them. For example, you can specify that a field be made in-line so that it provides full handwriting and gesture recognition directly on the field itself. At the other end of the spectrum, you can specify that the user is not able to modify the contents of a field directly. Instead, the user selects the field, causing an insertion pad to appear. Finally, the middle ground consists of overwrite fields that look like insertion pads (separate character boxes, each character can be overwritten) but allow editing directly, without creating a separate insertion pad.

Scratch Paper Sometimes it's necessary to get information from the user that exceeds the capabilities of a field. PenPoint provides an object called **clsSPaper** that manages re-display, simple editing, and translation of stroke data that doesn't fit a predefined field. Typically, you create a clsSPaper object, attach a translator to it, and then wait for it to notify your application that it has translated data available for you to process.

clsSPaper is a composite object that manages the interaction between three parties: the scribble, the translator, and the SPaper's client. For example, when the user places a stroke on the SPaper object, the SPaper object sends a msgScrAddStroke message to the scribble, which in turns sends the same message to the translator. This continues until SPaper decides that the user is done. Although this decision can be made several different ways, it usually occurs when the user moves the pen out of proximity after adding stroke data to the SPaper object.

When the SPaper object receives the msgSPaperComplete message it sends the msgScrComplete message to the scribble object, which in turn sends it to the translator. When the translator finishes translating the information, it sends a msgXlateComplete message to the SPaper object, which in turns notifies its client (your application) that translated data is available It is then up to your application to extract this information from the SPaper object and use it.

clsSPaper objects can be set up many different ways, depending on the style flags set during component creation. This, coupled with the ability to provide custom translators, make SPaper a very valuable object for collecting application-specific information.

Application-specific Input Support

Although this chapter has discussed the wonders of handwritten input, using a keyboard and mouse has some advantages. For example, a circular pen stroke can indicate various things—the number 0 (zero), the letters o or O, or a request by the user to edit selected text. This wouldn't be a problem for keyboard or mouse input because there are different keys for 0, o, and O, and editing selections come from the mouse alone.

PenPoint addresses this problem by allowing you to help the HWX Engine translate scribbles into gestures and characters. You provide this help by specifying custom translators for the generic input controls to use. For example, if you were collecting a series of numbers separated by the + and - operators, you could provide a clsSPaper object with a translator that recognized only 0123456789+-. The list of characters 0123456789+- is a **template** and is used as part of building a translator.

Translators, Templates, and XLists Currently, PenPoint discriminates the different types of translation objects based on the type of recognition they support. For example, **clsXGesture** supports the translations of scribbles into gesture commands, while **clsXText** supports the translation of scribbles into text characters. **clsXWord**, a subclass of clsXText, translates scribbles into words.

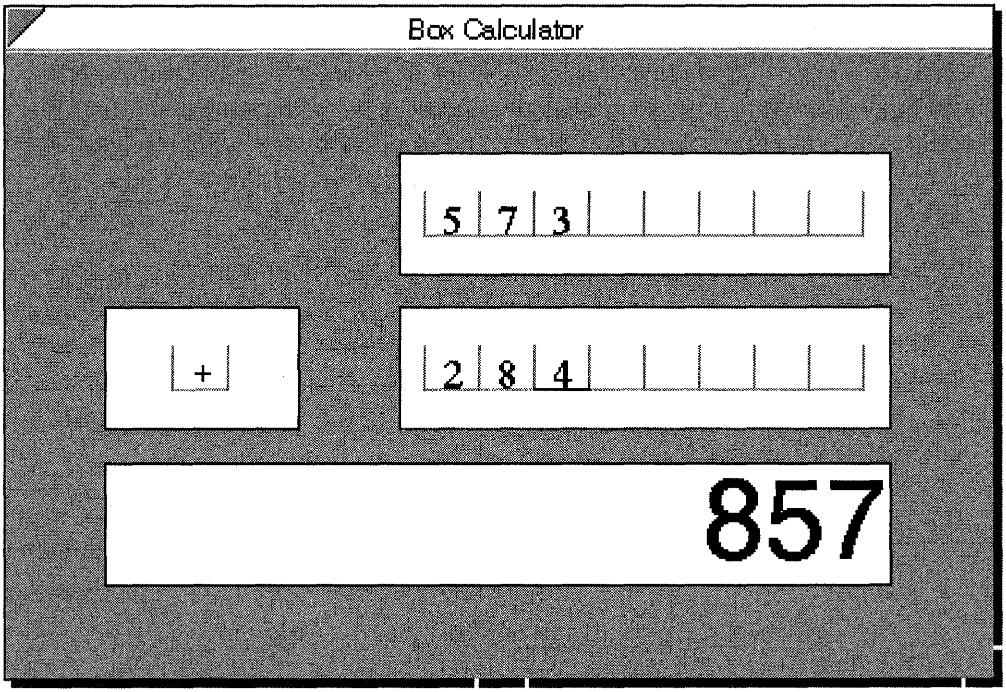
Each translator has different attributes that control how the translator notifies its clients about changes in the state of translation. These include attributes for signalling the addition of common character sets such as alphabetic, punctuation, and numeric to the list of recognized characters. Translators also use templates to more tightly constrain the types of characters recognized.

You build a template by specifying a list of valid characters and optional constraints to a utility function that compiles the template into a single, contiguous block of memory. This template is then passed as one of the parameters to the translator that you create. The translator uses this template to resolve ambiguities in handwriting while building the **XList** (translated list) it keeps as a result of the completed translation. For example, the calculation template 0123456789+- described earlier would help the translator resolve ambiguities by making scribbles match a 5 instead of an s, a 2 instead of a z, and so on.

clsBoxCalcApp: A Box-based Calculator

Now you have a rudimentary background in the components available for handwriting recognition. It's time to move on to several applications that actually use them. The first example is the two-integer, one-operator calculator shown in Figure 7.2. This application is constructed from two clsIntegerField objects that contains numbers and one clsField object that contains the operator.

FIGURE 7.2 The Box-based Calculator



The user interacts with this calculator by changing the value of any of the three fields and then moving the pen out of proximity. When the pen leaves proximity, translation takes place and the application is notified that new data exists. Once the application receives notification, it recovers the value from each component, constructs the equation, does the computation, and sets the display value accordingly.

The operator field in this example uses the validation messages to be given the opportunity to make sure the operator is valid. This is the same type of validation procedure you use if you want to check that a certain entered code is valid. In the next example, I avoid the validation messages by relying on a template to pre-screen the data. Although the template approach is also appropriate with the operator field in the box calculator, I think it's worthwhile to expose you to both types of validation.

This example also demonstrates the use of **tags** in identifying windows. A tag is a unique identifier based on an administered value that allows you to locate a window in a hierarchy without knowing its UID. This feature is very useful because UIDs change when objects are filed in and out, but tags do not. Therefore, if you know a window's tag, you can always obtain the window's current UID.

The BoxCalc Application

Instead of creating a subclass of clsCustomLayout, I have a single class, the application class, that uses an instance of clsCustomLayout as its main window. The following sections describe the functions and methods that make up the box-based calculator example.

Definitions The following header files contain definitions, message identifiers, and function prototypes for services required by the box calculator. This list includes one new definition file, tkfield.h, required by objects that need to use clsField or a subclass of clsField.

```
#ifndef APP_INCLUDED
#include <app.h>
#endif

#ifndef APPMGR_INCLUDED
#include <appmgr.h>
#endif

#ifndef FRAME_INCLUDED
#include <frame.h>
#endif

#ifndef WIN_INCLUDED
#include <win.h>
#endif

#ifndef CLAYOUT_INCLUDED
#include <clayout.h>
#endif

#ifndef TKFIELD_INCLUDED
#include <tkfield.h>
#endif

#ifndef LABEL_INCLUDED
#include <label.h>
#endif

#ifndef DEBUG_INCLUDED
#include <debug.h>
#endif
```

```
#include <method.h>
#include <stdio.h>
#include <string.h>
```

Administered Identifiers Following the statements for including definition files are a set of identifiers built using an administered value. They include the class ID for clsBoxCalcApp,

```
#define clsBoxCalcApp MakeGlobalWKN( 4146, 1 )
```

a status value required by the validation method used with the operator field,

```
#define stsNonValidOp   MakeWarning( clsBoxCalcApp, 1 )
```

and a set of tags:

```
#define firstNumTag     MakeTag( clsBoxCalcApp, 1 )
#define secondNumTag    MakeTag( clsBoxCalcApp, 2 )
#define operatorTag     MakeTag( clsBoxCalcApp, 3 )
#define resultsTag      MakeTag( clsBoxCalcApp, 4 )
```

The tags identify each child window after the application has been saved and then restored.

In addition to aiding in the restoration of the instance variable structure in this application, tags can also be used to access components of a composite object, such as style sheets.

Instance Variables The box calculator application uses four instance variables to maintain pointers to its child windows:

```
typedef struct INSTANCE_DATA {
  OBJECT firstNumWin;
  OBJECT secondNumWin;
  OBJECT operatorWin;
  OBJECT resultsWin;
  } INSTANCE_DATA, *P_INSTANCE_DATA;
```

The instance variables are not technically required because you could identify each window through its tag when needed. However, caching the window UIDs during restoration of the application increases efficiency by reducing messaging overhead.

main() and ClsBoxCalcAppInit() The box calculator uses a standard `main()` function that calls ClsBoxCalcInit() to register the box calculator application class with the Class Manager:

```
void CDECL
main(
  int  argc,
  char *argv[],
  U16  processCount)
{
  if (processCount == 0) {
    ClsBoxCalcAppInit();
    AppMonitorMain(clsBoxCalcApp, objNull);
    }
  else
    AppMain();

  Unused(argc); Unused(argv);
}
```

ClsBoxCalcInit() is defined

```
STATUS ClsBoxCalcAppInit (void)
{
  APP_MGR_NEW       new;
  STATUS            s;

  ObjCallJmp(msgNewDefaults, clsAppMgr, &new, s, Error);

  new.object.uid      = clsBoxCalcApp;
  new.cls.pMsg        = clsBoxCalcAppTable;
  new.cls.ancestor    = clsApp;
  new.cls.size        = SizeOf(INSTANCE_DATA);
  new.cls.newArgsSize = SizeOf(APP_NEW);

  new.appMgr.flags.accessory             = true;
  new.appMgr.flags.stationery            = false;
  new.appMgr.flags.allowEmbedding        = false;
  new.appMgr.defaultRect.size.w          = 450;
  new.appMgr.defaultRect.size.h          = 300;

  strcpy(new.appMgr.name, "Box Calculator");
  strcpy(new.appMgr.company, "Penpoint Programming");

  ObjCallJmp(msgNew, clsAppMgr, &new, s, Error);

  return stsOK;
```

```
Error:
  return s;
}
```

This function creates clsBoxCalcApp, a subclass of clsApp, with its own instance variables. The following code indicates that the box calculator is an accessory only, doesn't have stationery, and cannot be used with recursive live embedding. Also, this accessory should be started with a default width of 450 LSU (Logical Screen Units) and a height of 300 LSU.

```
  new.appMgr.flags.accessory      = true;
  new.appMgr.flags.stationery     = false;
  new.appMgr.flags.allowEmbedding = false;
  new.appMgr.defaultRect.size.w   = 450;
  new.appMgr.defaultRect.size.h   = 300;
```

Application Initialization clsBoxCalcApp objects respond to msgAppInit with the following method:

```
MsgHandler(BoxCalcAppAppInit)
{
  INSTANCE_DATA      inst;
  APP_METRICS        am;
  WIN_METRICS        wm;
  CSTM_LAYOUT_NEW       cln;
  STATUS             s;

  BuildNumberWin( self, firstNumTag, &inst.firstNumWin );
  BuildNumberWin( self, secondNumTag, &inst.secondNumWin );
  BuildOperatorWin( self, operatorTag, &inst.operatorWin );
  BuildResultsWin( resultsTag, &inst.resultsWin );

  ObjectWrite(self, ctx, &inst);

  ObjCallWarn(msgNewDefaults, clsCustomLayout, &cln );
  cln.border.style.backgroundInk = bsInkGray33;
  ObjCallWarn(msgNew, clsCustomLayout, &cln );

  wm.parent    = cln.object.uid;
  wm.options   = wsPosTop;

  ObjCallRet(msgWinInsert, inst.firstNumWin, &wm, s);
  ObjCallRet(msgWinInsert, inst.secondNumWin, &wm, s);
  ObjCallRet(msgWinInsert, inst.operatorWin, &wm, s);
  ObjCallRet(msgWinInsert, inst.resultsWin, &wm, s);

  AlignChildren( cln.object.uid, &inst );
```

```
  ObjCallWarn(msgAppGetMetrics, self, &am);
  ObjCallJmp(msgFrameSetClientWin,am.mainWin,cln.object.uid,
               s, Error);

  return stsOK;
Error:
  return s;
  MsgHandlerParametersNoWarning;
}
```

The first five lines initial the application's instance variables using function calls to construct the required components. Each function is passed: self receives notification messages when the individual components have new translated information to report; an individual tag identifies the window when the instance data is rebuilt during a restore; a pointer to the instance variable contains the identifier of the new object created by the function. Once each component has been created, and the values of the instance variables have been filled in, the instance data is written to protected memory.

Next, a clsCustomLayout object is created with a gray background, and all the components are inserted into the custom layout as the layout's children. After all four windows have been successfully inserted, the `AlignChildren()` function is called to provide the custom layout object with the specification for the location of each individual component.

Finally, the newly created custom layout object and its child windows are inserted as the application's main window.

Child Window Alignment Child window alignment inside the custom layout is accomplished by the function

```
STATUS LOCAL
AlignChildren(OBJECT cstmLayoutObj,P_INSTANCE_DATA pInst )
{
  CSTM_LAYOUT_CHILD_SPEC clcs;
  STATUS s;

  CstmLayoutSpecInit(&clcs.metrics);
  clcs.metrics.h.constraint = clPctOf;
  clcs.metrics.h.value      = 20;
  clcs.metrics.w.constraint = clPctOf;
  clcs.metrics.x.constraint =
            ClAlign(clMinEdge,clPctOf,clMaxEdge);
  clcs.metrics.y.constraint =
            ClAlign(clMinEdge,clPctOf,clMaxEdge);
```

```
    clcs.child                  = pInst->firstNumWin;
    clcs.metrics.w.value        = 50;
    clcs.metrics.x.value        = 40;
    clcs.metrics.y.value        = 65;
    ObjCallRet(msgCstmLayoutSetChildSpec, cstmLayoutObj,
               &clcs,s);

    clcs.child                  = pInst->secondNumWin;
    clcs.metrics.w.value        = 50;
    clcs.metrics.x.value        = 40;
    clcs.metrics.y.value        = 40;
    ObjCallRet(msgCstmLayoutSetChildSpec, cstmLayoutObj,
               &clcs, s);

    clcs.child                  = pInst->operatorWin;
    clcs.metrics.w.value        = 20;
    clcs.metrics.x.value        = 10;
    clcs.metrics.y.value        = 40;
    ObjCallRet(msgCstmLayoutSetChildSpec, cstmLayoutObj,
               &clcs, s);

    clcs.child                  = pInst->resultsWin;
    clcs.metrics.w.value        = 80;
    clcs.metrics.x.value        = 10;
    clcs.metrics.y.value        = 15;
    ObjCallRet(msgCstmLayoutSetChildSpec, cstmLayoutObj,
               &clcs, s);

    return stsOK;
}
```

The first set of statements initializes a child specification object that sets the height to always be 20% of the available window. It also specifies that the x, y, and w (width) values should be considered a percentage of the available window.

Next it sets the w, x, and y values for each individual component used to construct the box calculator. Then for each child window it sends a message to the custom layout object indicating that child's window specifications.

Restoring the Application As I mentioned earlier, I'm using tags to help restore the instance variables so they can be used to access the various components when their values change. Since saving and restoring individual components can conveniently be left to the underlying framework, my only required functionality for restoring the application is to query the custom layout for the new UIDs for each of its child windows. This is handled by the method that responds to the msgRestore message:

```
MsgHandler(BoxCalcRestore)
{
   INSTANCE_DATA   inst;
   APP_METRICS     am;
   OBJECT          frmWin;
   STATUS          s;

   ObjCallWarn(msgAppGetMetrics, self, &am);
   ObjCallJmp(msgFrameGetClientWin, am.mainWin, &frmWin,
                   s, Error);
   inst.firstNumWin=
      (WIN)ObjectCall(msgWinFindTag,frmWin,(P_ARGS)firstNumTag);
   inst.secondNumWin=
      (WIN)ObjectCall(msgWinFindTag,frmWin,(P_ARGS)secondNumTag);
   inst.operatorWin=
      (WIN)ObjectCall(msgWinFindTag,frmWin,(P_ARGS)operatorTag);
   inst.resultsWin=
      (WIN)ObjectCall(msgWinFindTag,frmWin,(P_ARGS)resultsTag);

   ObjectWrite(self, ctx, &inst);

   return stsOK;
Error:
   return s;
   MsgHandlerParametersNoWarning;
}
```

This method first queries the application for its frame—the custom layout window in the case of the box calculator. Next it sends the msgWinFindTag message to search the window hierarchy for each component window, filling in the values of the instance variables as it finds them. Finally, it writes the instance data containing the current UIDs back into protected memory.

Creating the Required Display Components

Earlier I mentioned that each component was created using one of several support functions. The use of functions to organize the creation of the display components was predominantly organizational: it improves the readability of the code. Each function, with the exception of `BuildResultsWin()`, takes as input the client object, a unique tag, and a pointer to a memory location that stores the UID of the object created.

BuildResultsWin() This function builds a standard clsLabel object that will be used to display the result of the computation. `BuildResultsWindow()` is defined

```
STATUS LOCAL
BuildResultsWin( TAG uTag, P_OBJECT pResWin )
{
  LABEL_NEW ln;
  STATUS    s;

  ObjCallRet(msgNewDefaults, clsLabel, &ln, s);
  ln.win.tag                  = uTag;
  ln.label.style.scaleUnits = bsUnitsFitWindowProper;
  ln.label.style.xAlignment = lsAlignRight;
  ln.border.style.edge      = bsEdgeAll;
  ln.label.pString          = "0";
  ObjCallRet(msgNew, clsLabel, &ln, s);

  *pResWin = ln.object.uid;
  return stsOK;
}
```

BuildNumberWindow() Each number window is an instance of clsIntegerField that has been initialized to contain a maximum of eight columns (which means eight digits) and to have a border around the editable field. The `BuildNumberWindow()` function is defined

```
STATUS LOCAL
BuildNumberWin(OBJECT clientObj,TAG uTag,P_OBJECT pNumWin )
{
  INTEGER_FIELD_NEW   ifn;
  STATUS              s;

  ObjCallWarn(msgNewDefaults, clsIntegerField, &ifn);
  ifn.win.tag                   = uTag;
  ifn.control.client            = clientObj;
  ifn.field.style.clientNotifyModified= true;
  ifn.label.style.numCols       = lsNumAbsolute;
  ifn.label.cols                = 8;
  ifn.border.style.edge         = bsEdgeAll;
  ifn.field.maxLen              = 8;
  ObjCallRet(msgNew, clsIntegerField, &ifn, s);
  ObjCallRet(msgControlSetValue, ifn.object.uid, 0, s );
  ObjCallWarn(msgControlSetDirty, ifn.object.uid,


(P_ARGS)(U32)false);
```

```
    *pNumWin = ifn.object.uid;
    return stsOK;
  }
```

In addition to defining this field's appearance characteristics, the function also defines two important pieces of control information. First, the statement

```
ifn.control.client = clientObj;
```

indicates that notification of the control's value changing should be sent to the object referenced by clientObj.

Second, the statement

```
ifn.field.style.clientNotifyModified  = true;
```

tells the field to notify the client object by sending it the msgFieldModified message when the value in the field changes.

Finally the integer field is given an initial value of zero and then reset using the msgControlSetDirty message so that any change in its value results in a msgFieldModified message being sent to the client object.

BuildOperatorWindow() The Operator window is an instance of clsField that is initialized to be one character in length and surrounded by a border. `BuildOperatorWindow()` is defined

```
STATUS LOCAL
BuildOperatorWin(OBJECT clientObj, TAG uTag,
                 P_OBJECT pOpWin)
{
 FIELD_NEW ifn;
 STATUS    s;

  ObjCallWarn(msgNewDefaults, clsField, &ifn);
  ifn.win.tag                           = uTag;
  ifn.control.client                    = clientObj;
  ifn.field.style.focusStyle            = fstNone;
  ifn.field.style.clientValidate        = true;
  ifn.field.style.clientNotifyModified  = true;
  ifn.label.style.numCols               = lsNumAbsolute;
  ifn.field.style.editType              = fstOverWrite;
  ifn.label.cols                        = 1;
  ifn.border.style.edge                 = bsEdgeAll;
  ifn.field.maxLen                      = 1;
  ObjCallRet(msgNew, clsField, &ifn, s);
  ObjCallRet(msgLabelSetString, ifn.object.uid, "+", s );
```

```
    ObjCallWarn(msgControlSetDirty, ifn.object.uid,
                                     (P_ARGS)(U32)false );

    *pOpWin = ifn.object.uid;
    return stsOK;
  }
```

Several additional specifications within this component's style field indicate the type of editing and validation it should support. The statements

```
ifn.field.style.focusStyle = fstNone;
ifn.field.style.editType   = fstOverWrite;
```

indicate that it will use a boxed format that allows the character already displayed in the box to be overwritten without grabbing the input focus.

Also, like the integer field created earlier, the statements

```
ifn.control.client             = clientObj;
ifn.field.style.clientValidate = true;
```

indicate it wishes to notify its client when the value in the field has been modified.

In addition to notifying the client when the field is modified, the statement

```
ifn.field.style.clientNotifyModified = true;
```

tells the field to issue a msgFieldValidateEdit message to the client object to allow it to validate the new value of the control before a msgFieldModified message is sent.

Finally the operator field is given an initial value of + and then reset using the msgControlSetDirty message so that any change in its value results in a msgFieldModified message being sent to the client object.

Validation and Computation

The last two methods in the box calculator example are responsible for validating the operator view in response to a msgFieldValidateEdit message and updating the value of the results window when it receives a msgFieldModified message.

Field Validation Validation of the operator field occurs in the box calculator application object at the request of the msgFieldValidateEdit message sent to it by the operator field object. The method that handles the validation is defined

```
    MsgHandlerArgType(BoxCalcOpValidate, P_FIELD_NOTIFY)
{
  STATUS         s;
  CONTROL_STRING cStr;
  char      buff[32];

  cStr.len = 2;
  cStr.pString = buff;
  ObjCallRet(msgLabelGetString, pArgs->field, &cStr, s);

  switch( buff[0] ) {
     case '+':
     case '-':
     case 'x':
     case '/':
       return stsOK;
       break;

     default:
       ObjCallRet(msgLabelSetString, pArgs->field, "?", s);
       ObjCallWarn(msgControlSetDirty, pArgs->field,
       (P_ARGS)(U32)false );
       pArgs->failureMessage = msgFieldNotifyInvalid;
       return stsNonValidOp;
       break;
     }
  MsgHandlerParametersNoWarning;
}
```

This method first accesses the control to find out what value the translator thinks the user entered using the msgLabelGetString message. It then checks to see if the character is one of four (+, -, x, and /) it considers valid. If the character is valid, it returns stsOK, which causes the field to immediately issue a msgFieldModified message to its client object. If the character is invalid, the field is reset with ? to provide the user with visual feedback, the control's dirty bit is reset, and a failure message is constructed. Finally, a *warning* (not error) status value is returned.

Computing the Result The method that computes the value of the operation specified by the two numerical fields and the operator field works on the brute force method of computing. When in doubt, reset everything and depend on error handling to avoid conflict.

Computation of the result is handled by the method

```
MsgHandlerWithTypes(BoxCalcCompute, P_ARGS,
                    P_INSTANCE_DATA)
{
  S32            num1, num2, res;
  STATUS         s;
  CONTROL_STRING cStr;
  char           buff[32];

  ObjCallWarn(msgControlSetDirty, pData->firstNumWin,
                                  (P_ARGS)(U32)false );
  ObjCallWarn(msgControlSetDirty, pData->secondNumWin,
                                  (P_ARGS)(U32)false );
  ObjCallWarn(msgControlSetDirty, pData->operatorWin,
                                  (P_ARGS)(U32)false );

  ObjCallJmp(msgFieldValidate, pData->operatorWin,
                                  (P_ARGS)0,s, Error);

  ObjCallJmp(msgControlGetValue, pData->firstNumWin,
                                  &num1,s, Error);
  ObjCallJmp(msgControlGetValue, pData->secondNumWin,
                                  &num2,s, Error);
  cStr.len     = 2;
  cStr.pString = buff;
  ObjCallJmp(msgLabelGetString, pData->operatorWin, &cStr,
             s, Error);

  switch( buff[0] ) {
     case '+': res = num1 + num2; break;
     case '-': res = num1 - num2; break;
     case 'x': res = num1 * num2; break;
     case '/': res = (num2?(num1/num2):num1); break;
     case '?': return stsOK;
     default: goto Error;
     }

  sprintf( buff, "%d", res );
  ObjCallRet(msgLabelSetString,pData->resultsWin,buff,s);

  return stsOK;
Error:
  return s;
  MsgHandlerParametersNoWarning;
}
```

This method first resets the dirty bit for each component. This is necessary because I'm not taking time to find out which component issued the update message. Next each component is queried for its value. Note that an integer field containing incorrect information (didn't pass validation) generates an error, thereby leaving the method early without disturbing the value in the results window.

If all three values are valid, a switch statement computes the result and displays it in the results window. Notice that if the operator is ?, the method returns without disturbing the value in the results window.

The Method Table

Finally, no PenPoint application is complete without a file containing the necessary method table(s). For the box calculator, this file is defined

```
#ifndef CLSMGR_INCLUDED
#include <clsmgr.h>
#endif

#ifndef APP_INCLUDED
#include <app.h>
#endif

#ifndef FIELD_INCLUDED
#include <field.h>
#endif

MSG_INFO clsBoxCalcAppMethods[] = {
  msgAppInit, "BoxCalcAppAppInit",
  objCallAncestorBefore,
  msgRestore, "BoxCalcRestore", objCallAncestorBefore,
  msgFieldValidateEdit, "BoxCalcOpValidate", 0,
  msgFieldModified, "BoxCalcCompute", 0,
  0
};

CLASS_INFO classInfo[] = {
  "clsBoxCalcAppTable",clsBoxCalcAppMethods,0,
  0
};
```

clsHWXCalc: A Scratch-paper-based Calculator

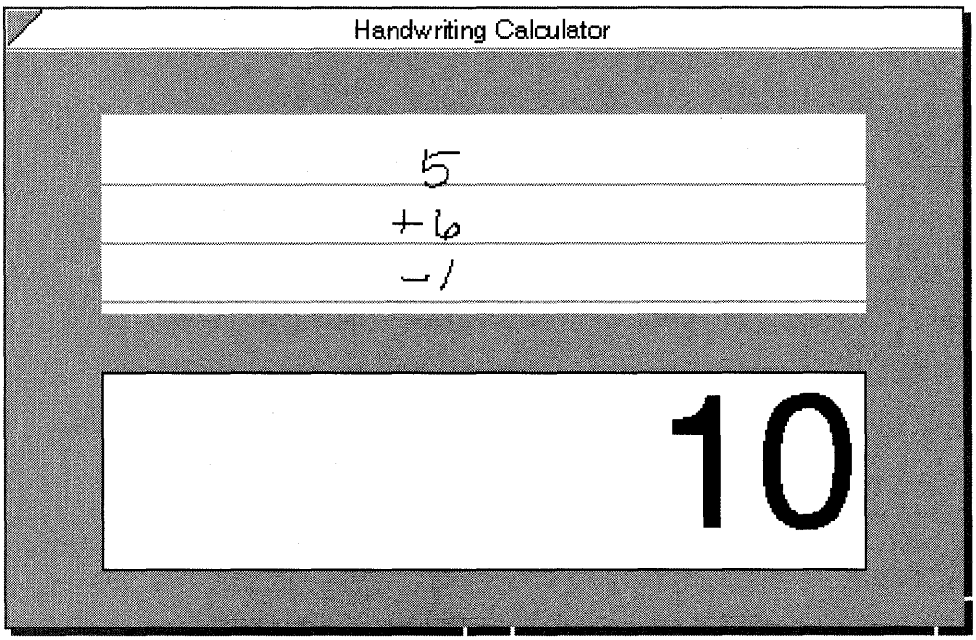
By now, you probably have a good idea of what I'm leading up to—a handwriting-based calculator not artificially constrained by boxed entry fields. Although not functionally a shining star, this calculator demonstrates a powerful use of the pen-and-paper metaphor.

Figure 7.3 shows the scratch-paper-based calculator (SPaper Calculator) in its full glory. The way the application works is that the handwritten data remains displayed on the screen, while its value is displayed in the results window. If there is an error in handwritten entry, the application displays what it thinks you wrote in the results window instead. The old handwritten calculation is removed with the first stroke of the pen when something new is placed in the results window.

Pretty simple, right? Well, not quite. I had to subclass clsSPaper (the scratch paper) to provide the stroke display and cleanup behavior I wanted so that I could keep the stroke data visible while the user looked at the result. I also implemented the subclass to automatically provide the SPaper object with a translator that only recognizes 0123456789+-. (As I said, the functionality isn't great.)

The following sections describe in greater detail the various objects that make up the application.

FIGURE 7.3 The Scratch-paper-based Calculator



clsHWXCalcApp

clsHWXCalcApp is the application class for the scratch-paper-based calculator. It contains methods and functions for managing the application, functions for creating and laying out components, and methods and functions for evaluating the results of the calculator's scratch pad translations.

Definitions The following header files contain definitions, message identifiers, and function prototypes for services required by the scratch pad calculator:

```
#ifndef APP_INCLUDED
#include <app.h>
#endif

#ifndef APPMGR_INCLUDED
#include <appmgr.h>
#endif

#ifndef FRAME_INCLUDED
#include <frame.h>
#endif

#ifndef WIN_INCLUDED
#include <win.h>
#endif

#ifndef CLAYOUT_INCLUDED
#include <clayout.h>
#endif

#ifndef LABEL_INCLUDED
#include <label.h>
#endif

#ifndef CLCSPAPR_INCLUDED
#include <clcspapr.h>
#endif

#ifndef XLATE_INCLUDED
#include <xlate.h>
#endif

#ifndef XLFILTER_INCLUDED
#include <xlfilter.h>
#endif
```

```
#ifndef DEBUG_INCLUDED
#include <debug.h>
#endif

#include <method.h>
#include <string.h>
#include <stdlib.h>
```

This code fragment introduces several new definition files: clcspapr.h is the definition file for the calculator scratch paper; xlate.h contains the interface for accessing the information resulting from handwriting translation; xlfilter.h defines the interfaces to various functions used to filter the translated data; stdlib.h contains several useful functions for manipulating and parsing information contained in ASCII strings.

In addition to the definition files' include statements is the definition of the application's Well Known UID:

```
#define clsHWXCalcApp MakeGlobalWKN( 4147, 1 )
```

the definition of several window tags:

```
#define entryTag    MakeTag( clsHWXCalcApp, 1 )
#define resultsTag  MakeTag( clsHWXCalcApp, 2 )
```

and the definition of the instance data structure:

```
typedef struct INSTANCE_DATA {
  OBJECT entryWin;
  OBJECT resultsWin;
  } INSTANCE_DATA, *P_INSTANCE_DATA;
```

main() and ClsBoxCalcAppInit() The scratch paper calculator uses a standard `main()` function that calls ClsHWXCalcInit() to register the Scratch Pad Calculator application class with the Class Manager. It also calls ClsCalcSPaperInit() to register my subclass of clsSPaper.

```
void CDECL
main(
  int  argc,
  char *  argv[],
  U16  processCount)
{
  if (processCount == 0) {
     ClsHWXCalcAppInit();
     ClsCalcSPaperInit();
     AppMonitorMain(clsHWXCalcApp, objNull);
     }
```

```
  else
     AppMain();

  Unused(argc); Unused(argv);
}
```

ClsHWXCalcInit() is defined

```
STATUS ClsHWXCalcAppInit (void)
{
  APP_MGR_NEW new;
  STATUS      s;

  ObjCallJmp(msgNewDefaults, clsAppMgr, &new, s, Error);

  new.object.uid       = clsHWXCalcApp;
  new.cls.pMsg         = clsHWXCalcAppTable;
  new.cls.ancestor     = clsApp;
  new.cls.size         = SizeOf(INSTANCE_DATA);
  new.cls.newArgsSize = SizeOf(APP_NEW);

  new.appMgr.flags.accessory        = true;
  new.appMgr.flags.stationery       = false;
  new.appMgr.flags.allowEmbedding  = false;
  new.appMgr.defaultRect.size.w     = 450;
  new.appMgr.defaultRect.size.h     = 200;

  strcpy(new.appMgr.name, "Handwriting Calculator");
  strcpy(new.appMgr.company, "PenPoint Programming");

  ObjCallJmp(msgNew, clsAppMgr, &new, s, Error);

  return stsOK;

Error:
  return s;
}
```

Application Initialization clsHWXCalcApp objects respond to msgAppInit with the following method. It follows the same format as the application initialization method from the box calculator example.

```
MsgHandler(HWXCalcAppAppInit)
{
  INSTANCE_DATA  inst;
  APP_METRICS    am;
  WIN_METRICS    wm;
  CSTM_LAYOUT_NEW   cln;
```

```
  STATUS       s;

  BuildEntryWin( self, entryTag, &inst.entryWin );
  BuildResultsWin( resultsTag, &inst.resultsWin );

  ObjectWrite(self, ctx, &inst);

  ObjCallWarn(msgNewDefaults, clsCustomLayout, &cln );
  cln.border.style.backgroundInk = bsInkGray33;
  ObjCallWarn(msgNew, clsCustomLayout, &cln );

  wm.parent     = cln.object.uid;
  wm.options    = wsPosTop;

  ObjCallRet(msgWinInsert, inst.entryWin, &wm, s);
  ObjCallRet(msgWinInsert, inst.resultsWin, &wm, s);

  AlignChildren( cln.object.uid, &inst );

  ObjCallWarn(msgAppGetMetrics, self, &am);
  ObjCallJmp(msgFrameSetClientWin,am.mainWin,
  cln.object.uid,s, Error);

  return stsOK;
Error:
  return s;
  MsgHandlerParametersNoWarning;
}
```

In the same vein, the child windows are arranged by a call to

```
STATUS LOCAL
AlignChildren(OBJECT cstmLayoutObj,P_INSTANCE_DATA pInst)
{
  CSTM_LAYOUT_CHILD_SPEC clcs;
  STATUS s;

  CstmLayoutSpecInit(&clcs.metrics);
  clcs.metrics.h.constraint    = clPctOf;
  clcs.metrics.h.value         = 35;
  clcs.metrics.w.constraint    = clPctOf;
  clcs.metrics.w.value         = 80;
  clcs.metrics.x.constraint    =
              ClAlign(clMinEdge,clPctOf,clMaxEdge);
  clcs.metrics.x.value         = 10;
  clcs.metrics.y.constraint    =
              ClAlign(clMinEdge,clPctOf,clMaxEdge);
```

```
    clcs.child            = pInst->entryWin;
    clcs.metrics.y.value  = 55;
    ObjCallRet(msgCstmLayoutSetChildSpec, cstmLayoutObj,
               &clcs, s);
    clcs.child            = pInst->resultsWin;
    clcs.metrics.y.value  = 10;
    ObjCallRet(msgCstmLayoutSetChildSpec, cstmLayoutObj,
               &clcs, s);

    return stsOK;
}
```

Restoring the Application Once again, the instance variables are restored using the predefined tags to access the various components when their values change. The method that performs this behavior is

```
MsgHandler(HWXCalcRestore)
{
  INSTANCE_DATA  inst;
  APP_METRICS    am;
  OBJECT         frmWin;
  STATUS         s;

  ObjCallWarn(msgAppGetMetrics, self, &am);
  ObjCallJmp(msgFrameGetClientWin, am.mainWin, &frmWin,
                  s, Error);

  inst.entryWin=(WIN)ObjectCall(msgWinFindTag,
                           frmWin,(P_ARGS)entryTag);
  inst.resultsWin=(WIN)ObjectCall(msgWinFindTag,
                          frmWin,(P_ARGS)resultsTag);

  ObjectWrite(self, ctx, &inst);

  ObjCallRet(msgAddObserver, inst.entryWin, self, s );

  return stsOK;
Error:
  return s;
  MsgHandlerParametersNoWarning;
}
```

There is one additional step in restoring the scratch paper calculator: reestablishing the application as a client of the calculator scratch paper object using the statement

```
ObjCallRet(msgAddObserver, inst.entryWin, self, s );
```

Creating the Display Components Two display components need to be built. First is the results window, built using

```
STATUS LOCAL
BuildResultsWin( TAG uTag, P_OBJECT pResWin )
{
  LABEL_NEW ln;
  STATUS    s;

  ObjCallRet(msgNewDefaults, clsLabel, &ln, s);
  ln.win.tag                    = uTag;
  ln.label.style.scaleUnits     = bsUnitsFitWindowProper;
  ln.label.style.xAlignment     = lsAlignRight;
  ln.border.style.edge          = bsEdgeAll;
  ln.label.pString              = "0";
  ObjCallRet(msgNew, clsLabel, &ln, s);

  *pResWin = ln.object.uid;
  return stsOK;
}
```

Second is the calculator scratch pad window, built using

```
STATUS LOCAL
BuildEntryWin(OBJECT clientObj, TAG uTag, P_OBJECT
pEntryWin)
{
  STATUS         s;
  CALCSPAPER_NEW    spn;

  ObjectCall(msgNewDefaults, clsCalcSPaper, &spn);
  spn.win.tag               = uTag;
  spn.border.style.resize = bsResizeNone;
  spn.sPaper.listener       = clientObj;
  ObjCallRet(msgNew, clsCalcSPaper, &spn, s);

  *pEntryWin = spn.object.uid;
  return stsOK;
}
```

Notice that the scratch paper initialization structure has a special field for the listener. This is because it doesn't inherit from clsControl and therefore must manage its own client notification. When the scratch paper completes a translation it sends the listener the message, msgSPaperXlate-Completed. The listener then retrieves the translated data from the scratch paper and processes it according to the application's needs.

Computing the Results The following method responds to the msgSPaperXlateCompleted. It processes the input data and then displays the appropriate information in the results window.

```
MsgHandlerWithTypes(HWXCalcCompute,P_ARGS,P_INSTANCE_DATA)
{
  STATUS        s;
  XLATE_DATA    xdata;
  X2STRING      x2sData;
  char          resval[50];

  xdata.heap = osProcessHeapId;
  ObjCallRet(msgSPaperGetXlateData,
             pData->entryWin,&xdata,s );
  XList2Text(xdata.pXList);
  XList2StringLength( xdata.pXList, &x2sData.count );
  OSHeapBlockAlloc(osProcessHeapId, x2sData.count,
                                         &x2sData.pString);
  Xist2String(xdata.pXList, &x2sData );

  if ( preprocessString( x2sData.pString ) ) {
     itoa( computeValue( x2sData.pString ), resval, 10 );
     ObjCallRet(msgLabelSetString, pData->resultsWin,
                resval, s );
     }
  else
     ObjCallRet(msgLabelSetString, pData->resultsWin,
                                     x2sData.pString, s);

ObjCallWarn(msgOkToResetSPaper,pData->entryWin,(P_ARGS)0);
  OSHeapBlockFree(x2sData.pString);
  XListFree(xdata.pXList);

  return stsOK;
  MsgHandlerParametersNoWarning;
}
```

This method begins by retrieving the translation data from the scratch paper object. Since the quantity of data is dynamic, I elected to use space on my heap to hold the information. I indicate this by setting

```
xdata.heap = osProcessHeapId;
```

The data is returned from the msgSPaperGetXlateData in raw translation list format. PenPoint provides several filters to aid in the management of translation data. One of these, XList2Text(), can be applied to the translation data because I'm expecting characters (not gestures) that are members of the set 0123456789+-.

Once the XList2Text() filter is applied, you proceed to transform the information in the filtered XList into a string of ASCII characters. Again, the process requires that memory be allocated for storage, so the osProcessHeapId predefine is used to indicate that memory should be allocated from the application's heap.

Once the string is generated, a local function filters it further, removing white space and checking for unrecognized characters. If it finds unrecognized characters, the erroneous translation is displayed in the results window. Otherwise, the string is parsed for its components, a value is computed, and the result is displayed in the results window.

The method then sends a reset message to the calculator scratch paper so it clears itself the next time the user applies the pen to it. Finally, the method deallocates the temporary storage that it created directly (x2sData.pString) and relies on the XList support function `XListFree()` to clean up any memory it has allocated on the applications stack.

Computing Support Functions The following two functions use standard C utilities to validate the translated string, parse it, and compute a result.

```
U32 LOCAL preprocessString( char *pString )
{
  char *pPrcStr = pString;

  while( *pString )
     switch( *pString ) {
       case '0': case '1': case '2': case '3':
       case '4': case '5': case '6': case '7':
       case '8': case '9': case '-': case '+':
          *pPrcStr++ = *pString++;
          break;

       case ' ': case '\n':
          pString++;
          break;

       default:
          return false;
       }

  *pPrcStr = '\0';
  return true;
}
```

```
U32 LOCAL computeValue( char *pString )
{
  U32 val=0;
  S16 nextOp = 1;

  while (*pString) {
     if ( *pString == '+' )
       pString++;
     else if ( *pString == '-' ) {
       nextOp *= -1;
       pString++;
       }
     else {
       val += nextOp * (U32)strtoul (pString, & pString,10);
       nextOp = 1;
       }
     }

  return val;
}
```

clsCalcSPaper

clsCalcSPaper, a subclass of clsSPaper, implements a special form of scratch paper that

- Has a built-in translator restricting input to 0123456789+-.
- Keeps the stroke data visible on the display device until the first stroke after the clsCalcSPaper object has been reset.

The External Interface The external interface to clsCalcSPaper is defined in clcspapr.h:

```
#ifndef CLCSPAPR_INCLUDED
#define CLCSPAPR_INCLUDED

#ifndef CLSMGR_INCLUDED
#include <clsmgr.h>
#endif
#ifndef SPAPER_INCLUDED
#include <spaper.h>
#endif

#define clsCalcSPaper MakeGlobalWKN( 4148, 1 )

define msgOkToResetSPaper MakeMsg( clsCalcSPaper, 1 )
```

```
STATUS ClsCalcSPaperInit( void );

#define CalcSPaperNewFields \
   sPaperNewFields

typedef struct CALCSPAPER_DATA {
   CalcSPaperNewFields
} CALCSPAPER_NEW, *P_CALCSPAPER_NEW;

#endif
```

Definitions Required for clsCalcSPaper's Implementation The following header files contain definitions, message identifiers, and function prototypes for services the Calculator Scratch Paper subclass requires:

```
#ifndef FS_INCLUDED
#include <fs.h>
#endif

#ifndef PEN_INCLUDED
#include <pen.h>
#endif

#ifndef XLATE_INCLUDED
#include <xlate.h>
#endif

#ifndef XLFILTER_INCLUDED
#include <xlfilter.h>
#endif

#ifndef XTEMPLT_INCLUDED
#include <xtemplt.h>
#endif

#ifndef CLCSPAPR_INCLUDED
#include <clcspapr.h>
#endif

#ifndef DEBUG_INCLUDED
#include <debug.h>
#endif

#include <method.h>
```

This code fragment introduces several new definition files: pen.h is the definition file for the input events associated with the pen; xtemplt.h contains several useful functions for manipulating templates used by stroke translators.

In addition to the definitions, there is the instance data structure:

```
typedef struct INSTANCE_DATA {
  U32 okToReset;
  } INSTANCE_DATA, *P_INSTANCE_DATA;
```

okToReset, when true, instructs this class to clear the contents of the scratch paper the next time the user writes on it.

The Class Registration Routine Calling the following function registers clsCalcSPaper with the Class Manager.

```
STATUS ClsCalcSPaperInit (void)
{
  CLASS_NEW new;
  STATUS    s;

  ObjCallJmp(msgNewDefaults, clsClass, &new, s, Error);

  new.object.uid      = clsCalcSPaper;
  new.cls.pMsg        = clsCalcSPaperTable;
  new.cls.ancestor    = clsSPaper;
  new.cls.size        = SizeOf(INSTANCE_DATA);
  new.cls.newArgsSize = SizeOf(CALCSPAPER_NEW);

  ObjCallJmp(msgNew, clsClass, &new, s, Error);

  return stsOK;

Error:
  return s;
}
```

Initialization A clsCalcSPaper object is initialized to contain a custom translator with a special filter attached. The initialization method is

```
MsgHandlerArgType(CalcSPaperInit, P_CALCSPAPER_NEW)
{
  INSTANCE_DATA  inst;
  STATUS         s;
  P_UNKNOWN      pNewTemplate;
  XLATE_NEW      xNewTrans;
  U16            xlateFlags;
  XTM_ARGS       xtmArgs;
```

```
ObjectCall(msgNewDefaults, clsXText, &xNewTrans);

 xtmArgs.xtmType    = xtmTypeCharList;
 xtmArgs.xtmMode    = 0; // no special modes
 xtmArgs.pXtmData   = "0123456789+-";    // ascii template
 StsRet(XTemplateCompile(&xtmArgs, osProcessHeapId,
                                        &pNewTemplate), s);

 xNewTrans.xlate.pTemplate    = pNewTemplate;
 xNewTrans.xlate.hwxFlags     &=
 ~(alphaNumericEnable|punctuationEnable|verticalEnable);

 ObjCallRet(msgNew, clsXText, &xNewTrans, s);

 ObjCallRet(msgXlateGetFlags, xNewTrans.object.uid,
               &xlateFlags, s);
 xlateFlags |= xTemplateVeto | spaceDisable;
 ObjCallRet(msgXlateSetFlags, xNewTrans.object.uid,
                                    (P_ARGS)xlateFlags,s);«

 pArgs->sPaper.translator    = xNewTrans.object.uid;
 pArgs->sPaper.flags         |= spProx;

 ObjectCallAncestorCtx(ctx);

 inst.okToReset = true;
 ObjectWrite(self, ctx, &inst );

 return stsOK;
 MsgHandlerParametersNoWarning;
}
```

The translator and filter are created as a result of the msgInit message being sent. Until now, specifying when the inherited behavior for a method should be called has been done in the definition of the method table. In this example, I specify the filter/translator pair by explicitly calling the ancestor class's initialization method before calling the ancestor's msgInit handling method and after the default initialization structure was filled in per the needs of the Calculator Scratch Pad class.

```
ObjectCallAncestorCtx(ctx);
```

At initialization, the initialization structure for a text translator is filled in by first creating a new template using

```
xtmArgs.xtmType    = xtmTypeCharList;
```

```
  xtmArgs.xtmMode    = 0;               // no special modes
  xtmArgs.pXtmData   = "0123456789+-"; // ascii template
  StsRet(XTemplateCompile(&xtmArgs, osProcessHeapId,
                                     &pNewTemplate), s);
```

Once the template is compiled, it is added to the initialization information used to create the translator. The translator is then told to not recognize all alphanumerics and punctuation:

```
xNewTrans.xlate.hwxFlags &=
 ~(alphaNumericEnable | punctuationEnable | verticalEnable);
```

Next, the translator is created and told to allow the template values to veto the success of the translation, in effect restricting the valid characters to only those contained in the template. Once the translator is complete, it is used as a parameter in the new structure that will be passed to the ancestor's initialization routine. Finally, the subclass's one instance variable is initialized to true.

Saving and Restoring Instance Data The next two methods maintain the state of the okToReset flag when the object is saved.

```
MsgHandlerArgType(CalcSPaperSave, P_OBJ_SAVE)
{
  STREAM_READ_WRITE fsWrite;
  STATUS          s;

  fsWrite.numBytes  = SizeOf(INSTANCE_DATA);
  fsWrite.pBuf     = pData;
  ObjCallRet(msgStreamWrite, pArgs->file, &fsWrite, s);

  return stsOK;
  MsgHandlerParametersNoWarning;
}
MsgHandlerArgType(CalcSPaperRestore, P_OBJ_RESTORE)
{
  INSTANCE_DATA     inst;
  STREAM_READ_WRITE fsRead;
  STATUS            s;

  fsRead.numBytes = SizeOf(INSTANCE_DATA);
  fsRead.pBuf     = &inst;
  ObjCallRet(msgStreamRead, pArgs->file, &fsRead, s);

  ObjectWrite(self, ctx, &inst);
```

```
    return stsOK;
    MsgHandlerParametersNoWarning;
}
```

Clearing the Scratch Paper The behavior for clearing the contents of the scratch paper is shared by two methods. The application uses the first to indicate when the scratch paper should be freed:

```
MsgHandler(CalcSPaperReset)
{
    INSTANCE_DATA     inst;

    inst = IDataDeref(pData, INSTANCE_DATA);
    inst.okToReset = true;
    ObjectWrite(self, ctx, &inst);

    return stsOK;
    MsgHandlerParametersNoWarning;
}
```

The second method monitors input events, waiting for the first user stroke after the okToReset flag has been set to true. This method responds to the msgInputEvent message.

```
MsgHandlerWithTypes(CalcSPaperInputEvent, P_INPUT_EVENT,
P_INSTANCE_DATA)
{
    INSTANCE_DATA     inst;

    if (pArgs->devCode == msgPenStroke ) {
       if ( pData->okToReset ) {
         ObjCallWarn(msgSPaperClear, self, (P_ARGS)0);
         inst = IDataDeref(pData, INSTANCE_DATA);
         inst.okToReset = false;
         ObjectWrite(self, ctx, &inst);
         }
       }

    return stsInputContinue;
    MsgHandlerParametersNoWarning;
}
```

Makefiles and Method Tables Until now, the applications in this book required only two libraries (penpoint and app) to link successfully. In this example, you must also include in the makefile the xtemplt and xlist libraries to link with the code that supports the utility functions they contain.

As always, the final step in supplying source code are the method tables. For the scratch paper calculator, the method table file is defined

```
#ifndef APP_INCLUDED
#include <app.h>
#endif

#ifndef CLSMGR_INCLUDED
#include <clsmgr.h>
#endif

#ifndef INPUT_INCLUDED
#include <input.h>
#endif

#ifndef CLCSPAPR_INCLUDED
#include <clcspapr.h>
#endif


MSG_INFO clsHWXCalcAppMethods[] = {
  msgAppInit,  "HWXCalcAppAppInit", objCallAncestorBefore,
  msgRestore,  "HWXCalcRestore",   objCallAncestorBefore,
  msgSPaperXlateCompleted,"HWXCalcCompute",  0,
  0
};

MSG_INFO clsCalcSPaperMethods[] = {
  msgInit,       "CalcSPaperInit", 0,
  msgSave,       "CalcSPaperSave",
  objCallAncestorBefore,
  msgRestore,    "CalcSPaperRestore",
  objCallAncestorBefore,
  msgOkToResetSPaper,"CalcSPaperReset", 0,

msgInputEvent,"CalcSPaperInputEvent",
   objCallAncestorAfter,
  0
};

CLASS_INFO classInfo[] = {
  "clsHWXCalcAppTable",  clsHWXCalcAppMethods,    0,
  "clsCalcSPaperTable",  clsCalcSPaperMethods,    0,
  0
}
```

Wrap-up

By now, you realize that I've only scratched the surface of the handwriting recognition system's capabilities. However, I have covered most of what you need to know to build many form-based applications, because you can manage most work associated with handwritten input using the predefined components in the toolkit.

If you're looking for something to do, consider extending the hwxcalc example so that it

- Allows parenthesis, multiplication, and division.
- Works with input lines that look like
  <operator> <number> <comment>
  so you could have annotated worksheets, like
  +5 income
  -4 taxes

The <operator> <number> <comment> example brings up an interesting point. Most applications written for traditional WYSIWYG (What you see is what you get) user interfaces attempt to manage data entry so a bad value is flagged and the user is notified immediately. This insures the integrity of much of the application's data before the entire operation is completed. This isn't the case with handwriting-based input.

When the user writes something, it usually expresses a complete thought, which is then handed to your application in one piece, much like a compiler handles a source file. It is then your responsibility to analyze the data, like a compiler analyzes a program, checking for accuracy and completeness before committing to the operation. There is actually a lot of help out there for doing this type of analysis. You could start by reading a primer on the Unix tools Lex and Yacc which provide support for building a compiler-like translator based on predefined grammar.

Finally, I ask you to consider carefully before adopting a user interaction metaphor for the PenPoint applications you build. As with the calculator, many metaphors might work. It's up to you to decide on the one that's best based on pen and paper.

8

A Crossword Puzzle

The first seven chapters of this book concentrated on concepts, using small applications to illustrate how to write programs for PenPoint. Starting in this chapter and continuing through the remainder of the book, I'm shifting gears to build a single application that resembles an actual product: a simple crossword puzzle application.

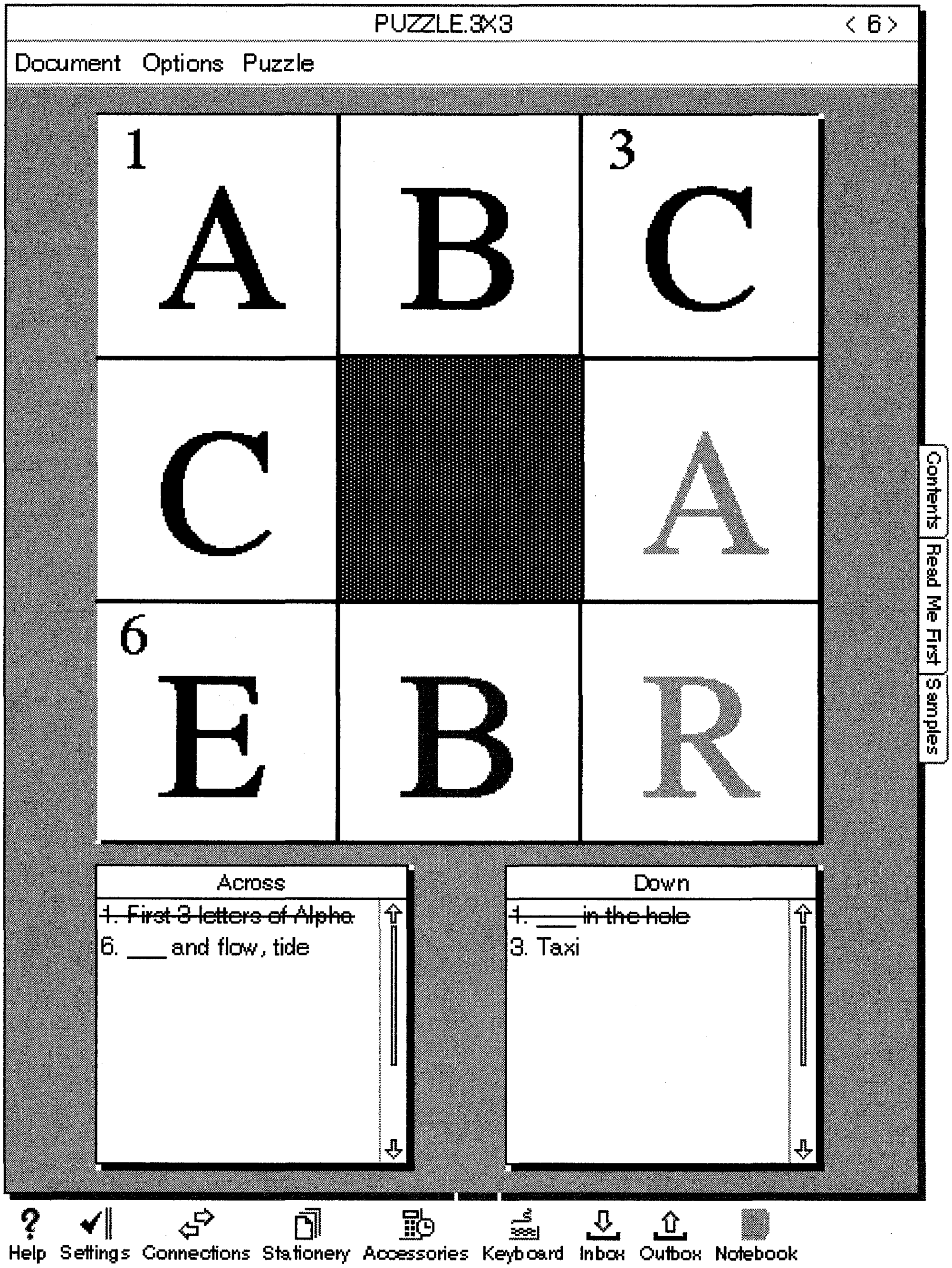
My goal in presenting this application is threefold. First, I want to illustrate more accurately the level of effort required to write PenPoint applications, including the use of Dynamic Link Libraries (DLLs). Second, I want to add to your bag of tricks by giving you alternative means of presenting information, such as drawing contexts, menus, and notes. Third, I want to leave you with a project to complete—a final lesson of sorts—that will let you extend the book's sample application into a more robust product.

Chapter 8 describes the crossword puzzle application by providing a User's Guide that lists the application's functionality and its look and feel. Following the User's Guide is a description of the various components used to implement the application's look and feel and convey it to the user. Finally, the source code used to implement the application's clsXWordApp and clsXWordData classes is presented along with the entries in the method table that supports them.

The Crossword Puzzle User's Guide

When I sat down to think about a final example, it didn't take long to decide on a crossword puzzle. Most of us have seen and worked on a crossword puzzle at least once or twice. The paper-based version of a crossword puzzle maps directly to what a pen-based version would look like. Finally, it's fun.

FIGURE 8.1 The Crossword Puzzle Application



The crossword puzzle application shown in Figure 8.1 consists of a grid for entering letters, two clue lists, and menu selections for presenting commands to the application. You work the crossword by importing puzzles from outside PenPoint, and then writing the answers to the listed clues directly on the grid. Progress in solving the puzzle is checked by selecting a command from the Puzzle menu's Check submenu. You can also instruct the clue lists to draw a line through any clue you tap to help you track your progress.

The puzzle's maximum size is 10x10. The application sizes the individual character boxes so that the entire puzzle grid takes the same amount of screen space, regardless of the total number of letter boxes in the grid. The puzzles themselves are created by producing an ASCII text file in the format described in the next section. You can obtain a blank 10x10 grid by creating a piece of crossword paper that has no puzzle attached to it. This will help you to create new puzzles for other people to use.

Loading Puzzles

The crossword application relies on PenPoint's import facility to load new puzzles. The puzzle provider creates a file that contains ASCII text descriptions of clues, positions, and answers for the puzzle. When the user drags a puzzle file into the Notebook, the import manager asks each loaded application if it knows how to process the file. The crossword application recognizes the puzzle and responds to the Import Manager that it wants to work on the file. Once the Import Manager polls each application, it presents the user with a list of applications that can work with the file. The user must then select the crossword puzzle application.

Take a moment and consider the implications of this mechanism on other applications. For example, you are writing a program that tracks the movement of stock prices. You could easily have a PenPoint application that imports data from a stock information retrieval service and converts the data into pages in your PenPoint Notebook. You could then embed the price-charting pages in other documents you might send to a client later.

Writing Your Answers

You interact with the grid shown in Figure 8.1 just as you would with a paper crossword puzzle. You can write a single uppercase character or multiple uppercase characters in either the horizontal or vertical direction. The default unknown character globe replaces unrecognized handwritten characters. You can remove a letter at any time by drawing a horizontal line through it.

The characters themselves are rendered in black or one of two gray shades that relate to the entry's accuracy. A dark gray letter is one that has been entered but not checked. A black letter is a character that has been confirmed as correct, while a light gray character is one that has been determined to be wrong.

Menu Commands

The crossword application has a menu bar that has been altered by removing standard functionality that the crossword application won't use and adding new functionality that it will use. You instruct the crossword puzzle to perform certain actions by selecting items from the Puzzle menu. These actions include showing the puzzle's solution, starting the puzzle over, changing the clue list's behavior, and checking the validity of letters that have been entered into the puzzle grid.

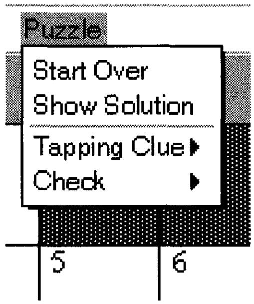
By the way, the Puzzle menu is not a good example of user interface design, because it actually implements three separate types of functionality in one place. However, I claim writer's license because I want to demonstrate nested menus that include choices.

The *Puzzle* Menu The Puzzle menu shown in Figure 8.2 contains the following items:

- **Start Over**—Select this command from the menu to erase the grid and redisplay any stricken clues without a line through them.
- **Show Solution**—Select this command from the menu to fill the grid with the correct answers, overwriting any letters that might have already been present.
- **Tapping Clue**—Select this item from the menu to see a submenu that allows you to change the behavior of the clue lists.
- **Check**—Select this item from the menu to see a submenu that allows you to check the current contents of the puzzle grid against the correct answers.

The *Tapping Clue* Sub-Menu The Tapping Clue submenu shown in Figure 8.3 lets you choose how the clue lists behave when you tap on an item they contain. The choices are mutually exclusive, and PenPoint marks the option currently in effect with a check mark. The two choices are

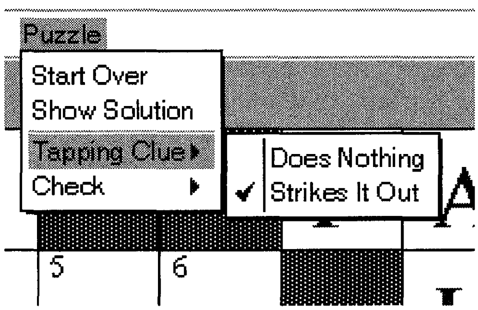
FIGURE 8.2 The Puzzle Menu



- **Does Nothing**—Select this item from the submenu to instruct the clue list boxes to ignore any pen taps that occur on one of the items they contain. It also removes any stricken lines that might have been placed there already.
- **Strike it Out**—Select this item from the submenu to instruct the clue list boxes to draw a line through any clue you tap on. To remove the line, you tap on the clue a second time. The clue lists remember which clues have been stricken so it can redraw the lines when the user selects Strike it Out after choosing Does Nothing.

The Check Sub-Menu The Check submenu shown in Figure 8.4 contains commands for checking the current contents of the puzzle grid against the correct answers. You can check your answers with

FIGURE 8.3 The Tapping Clue Submenu Text



- **Puzzle ...**—Select this item from the submenu to see the note shown in Figure 8.5. This note provides you with the total number of words in the puzzle, along with the number of words you have correct; and the total number of letters in the puzzle, along with the number of letters you have correct. You dismiss the note by tapping the OK button at the bottom.
- **Words**—Select this item from the submenu to instruct the crossword application to compare each complete word you entered in the grid against the correct words in the the puzzle. Once the check is complete, the grid is redrawn, with the letters of correct words drawn in black, and everything else drawn in light gray. This command draws correct letters in incorrect words in light gray.
- **Letters**—Select this item from the submenu to instruct the crossword application to compare each letter in the grid to the correct letter for that position and to redraw correct letters in black and incorrect letters in light gray. Unlike the Words option, any letter that's correct will be drawn as correct, even if the word or words it belongs to is incorrect.

Creating New Puzzles

You create new puzzles by building an ASCII file that the crossword puzzle application can import. For example, the file that describes the puzzle shown in Figure 8.1 contains the lines

```
pip-xwordpuzzle
3,2,2
1,0,0,ABC,First 3 letters of Alpha
6,0,2,EBB,___ and flow, tide
1,0,0,ACE,___ in the hole
3,2,0,CAB,Taxi
```

The first line contains a string expected by the crossword puzzle application to indicate that the file contains the description of a puzzle and can be imported by the application.

The second line lists the size of the grid (3x3), the number of across clues (2), and the number of down clues (2).

The remaining lines list the word/clue entries that make up the puzzle. Clues are listed in numerical order, with all the across clues first, followed by the down clues. Each clue consists of five fields: number, x position, y position, word, and clue. For example, the first entry shown is for 1 across, which is located at x position 0, y position 0 (the top of the puzzle). The word at this location is ABC, and the clue for 1 across is "First 3 letters of Alpha."

FIGURE 8.4 The Check Submenu

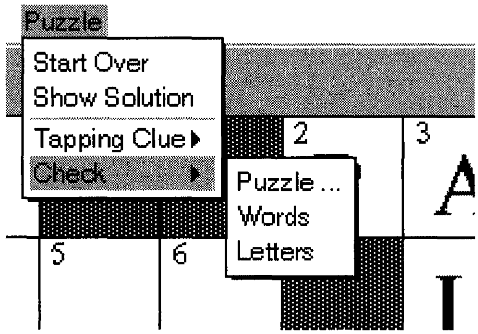
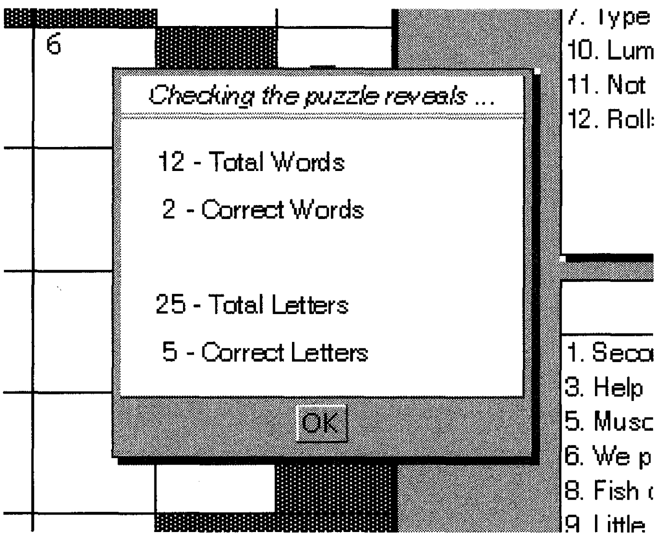


FIGURE 8.5 The Check Puzzle Note Window



You must use the comma (,) field delimiter as required.

Implementing the Crossword Application

Once the user interactions have been specified, designing and implementing the classes necessary to implement the crossword puzzle is a straightforward process. The crossword application consists of five custom components implemented as PenPoint classes:

- **clsXWordApp**—The Crossword Application class is a descendant of clsApp responsibile for managing the acceptance of queries from the import manager, managing user interactions through the menu interface, and maintaining the application's main() routine used to manage the document lifecycle.
- **clsXWordData**—The Crossword Application Data Model class inherits from clsObject and is the keeper of clues, words, and the grid template that indicate legal blocks that the user can write in. It also maintains the functionality for translating ASCII files in the crossword format into data objects that are then attached to a view and filed as crossword puzzle documents.
- **clsXWordView**—The Crossword Puzzle view (described in Chapter 9) inherits from clsView and is a composite object that builds the crossword puzzle's user interface by creating a grid object and two clue list objects for display to the user. It also checks system preferences to find out if the display is in landscape or portrait mode and adjusts its layout accordingly. Finally, clsXWordView serves as the interface between what the user enters in the grid and what the model says is correct. It is the object that resolves the difference between what's entered and what's correct and then informs the grid object how to update itself.
- **clsXWordClueList**—The Clue List display (described in Chapter 9) inherits from clsCustomLayout and provides a simple mechanism for displaying a list of entries that can be told to strike themselves out when the user taps them. This component has been implemented as a Dynamic Link Library for this project.
- **clsXWordGrid**—The Crossword Puzzles Interactive Grid (described in Chapter 10) inherits from clsSPaper and provides the view that the user interacts with while working the puzzle. It translates scribbles into uppercase letters and displays the letters in a color-coded scheme that indicates whether a particular character entry is correct.

PenPoint Classes Used in clsXWordApp and clsXWordData

The Crossword Puzzle class, clsXWordApp, is a combination of components from the PenPoint library that implement importing, menu handling, and user notification, as well as custom components that handle puzzle-specific issues such as the crossword model (clsXWordData) and view (clsXWordView).

In addition to the components clsXWordApp uses, several data manipulation components pass information between the crossword model and the view. The following sections describe the general capabilities of these PenPoint components used to implement the crossword puzzle application and model classes. The application-specific view classes are described in the next chapters.

Importing Files

PenPoint provides a set of classes that manage importing and exporting information to and from the PenPoint environment. These classes serve as the primary interface between application-oriented operating systems, such as the PC and Macintosh, to the document-oriented PenPoint environment. PenPoint manages most of the functionality of importing files such as prompting the user, querying applications on whether they can handle a particular file, moving or copying a file, and so on. clsImport does, however, require your application to respond to certain messages in certain ways in order for it to use importing.

clsImport The first step in working with clsImport is to implement a method in your application class that responds to the msgImportQuery class message. This is one of the few times that the class, and not an instance of the class, is responsibile for responding to a particular message.

Each application is sent this message when the user imports a new file into PenPoint. The message contains a pointer to a data structure that provides an open file handle to the file wishing to be imported. It's up to the application class to respond yes if it can import the file or no if it cannot. If the application can import the file, it can also indicate goodness of fit for handling the data which the Import Manager uses to order the list of applications responding positively to msgImportRequest.

Once the user selects an application, a document for that application is created (including the sending of the msgAppInit message) and then sent a msgImport message. This message contains information that includes an open file handle to use to finish creating the new document. If the application that receives the msgImport message can't create the document (for example, the file's contents are garbled at some point), it should return a status other than stsOK.

Reading Data Files One difficult problem of exchanging information between operating systems and platforms is that even ASCII information is not stored in the same format. For example, some systems use a carriage return to delineate lines, others use a line feed, while others believe in redundancy and require both! In addition most operating systems have their own preferred format for storing information.

PenPoint is no exception to the unique data format philosophy. Thankfully, however, GO provided a compatibility layer that allows you to use the stdio library functions such as fprintf() and fgets() when working with imported data files. These functions allow you to write data-importing code that matches the code used to write the file, even if the file was not written on a PenPoint machine.

Menu Support

The second new set of components used in the crossword puzzle application involves menu support. Menus are built as special cases of clsTkTable and therefore can be expected to behave in a similar fashion. Because menus are windows, it's possible to change their appearance dynamically throughout their document's lifecycle.

clsMenu In PenPoint the clsMenu class implements menus as a subclass of clsTkTable that has been optimized to provide a table whose default entries are clsMenuButton objects. Menus can be oriented both horizontally (the application's menu bar) or vertically (a pull-down menu selected from the main menu bar). A special kind of submenu, a pull-right submenu, is also created vertically.

You can specify the contents of a menu by providing the `MENU_NEW` structure with a `TK_TABLE_ENTRY` structure that contains information describing the menu. Included in this specification are definitions for decorations such as right arrows and check marks, and flags for indicating that a pull-right submenu has been defined inline.

clsMenuButton clsMenuButton is a subclass of clsButton that has been implemented to support the concept of submenus that they can pop up and take down. As a subclass of clsButton, clsMenuButton is also a subclass of clsControl and therefore able to respond to the Preview protocol implemented in clsControl. When selected, a clsMenuButton object displays its submenu, if it has one. For example, child menus can take the form of menus that appear when the user pulls down (mbMenuPull-Down), pulls right (mbMenuPullRight), or selects a menu button to have a list of selections pop up (mbMenuPopup).

clsChoice Sometimes it's desirable to display a list of items from which the user can choose one, and only one, item. PenPoint implements this functionality in the clsChoice class. clsChoice, also a descendant of clsTk-Table, builds its child windows using buttons that have the bsContact-LockOn style flag set to true.

Notifying the User

Sometimes it's necessary for the application to inform the user about a change in status. One option you have is to create a window (including components), insert it into the hierarchy, make it visible, and then monitor the user's interaction with it. This option is so popular that PenPoint provides this functionality in the clsNote component.

clsNote clsNote is an easy-to-use mechanism for providing the user with timely information about an application's status. You create a cls-Note object by specifying a `TK_TABLE_ENTRY` structure that contains information to be displayed. You can then display the note in a model fashion with your application blocked until the user acknowledges the message the note contains. Or, you can make the note model, allowing it to stay visible until

- The user taps on a button or
- Something in your application tells it to terminate or
- A specified time interval passes and the note times out.

Maintain Lists of Data

Stop and reflect for a moment: consider the amount of time you spend building data structures and writing case statements that execute different instructions to perform the same functionality based on the type of data structure you're working with. One powerful feature of object-based environments that support dynamic binding (like PenPoint), is the ability to manipulate lists of objects without knowing what those objects are until runtime. This allows code you've written, tested, and placed into production to be reused in other applications—you don't need to add case clauses to your switch statements.

Two PenPoint classes, clsList and clsString, illustrate this concept very well.

clsString clsString provides a standard mechanism for maintaining a null-terminated ASCII string. It allocates and frees the space necessary to store the string and can respond to save and restore messages. It also is an observable object, so clients can observe a string object and receive notification messages when it changes.

Keep several facts in mind when working with string objects. First, when you file and restore the object, there is no guarantee that pointers to the object's internal byte buffer will remain constant. You should maintain indexes into the string if you need to manipulate its subcomponents. Second, string objects are not locked. Anyone with the string object's UID can request its buffer and change its contents.

There are several ways to avoid ownership conflicts. One is to make copies of objects before giving them to another object. This way, you know that whenever an object is passed to you, you must free it when you're through. Unfortunately, a system performance penalty tends to be associated with constantly allocating and freeing objects.

A second method for avoiding ownership conflicts is to provide a shared object mechanism capable of freeing the object when all references to it are removed. Unfortunately, this often requires either operating system support or writing an additional piece of object management software.

A third method is to establish a verbal agreement on who owns what objects. For instance, when I create a string object I might agree to provide that information to you by passing you the object's UID with the understanding that you will only read from the string. At other times, I might send you information in one form that you understand, like a string, even though that information is kept differently in my object. In this case, I'm relying upon you to free the object when you are through with it, because I created it just for you.

clsList Groups of strings are very commonly manipulated as a single entity. For example, menus, file lists, and so on all can be thought of as a group or list of strings. PenPoint provides a special class, clsList, that optimizes the maintenance and manipulation of lists of objects, including but not limited to string objects.

Lists implement the generic behavior necessary to save and restore their contents, locate an item they contain, modify a list entry, and enumerate the objects contained in the list. Your code can manipulate the list without being concerned about the components the list contains.

One common problem with lists is the confusion caused by figuring out what is being destroyed when you free a list. This is very important, because multiple lists often share pointers to common items. In these cases, you don't want to free the actual items, only the list of pointers to them. At other times, however, you might want to free both the individual items in the list and the list itself. PenPoint supports both forms of freeing.

clsXWordApp: The Crossword Application Class

clsXWordApp is the application class for the crossword puzzle application. It is responsible for creating a new crossword puzzle document including the interaction with clsImport required to bring in new puzzles. clsXWordApp also manages the Puzzle menu and its submenus used in working the crossword puzzle. Finally, clsXWordApp's main() routine contains the function calls necessary to initialize the rest of the required classes, except clsXWordClue which is contained in a DLL.

clsXWordApp is implemented using two files (xwordapp.h and xwordapp.c) in addition to having a set of entries in the method table file (method.tbl).

xwordapp.h

xwordapp.h is the external interface for the crossword puzzle application class clsXWordApp. The file begins by checking to make sure the file hasn't already been included:

```
#ifndef XWORDAPP_INCLUDED
#define XWORDAPP_INCLUDED
```

If this is the first access to this file, the first action taken is to include the interface files for the other components it relies upon, using the statements

```
#ifndef GO_INCLUDED
#include <go.h>
#endif

#ifndef CLSMGR_INCLUDED
#include <clsmgr.h>
#endif
```

Following the include directives is the definition of the Well Known ID used to represent clsXWordApp:

```
#define clsXWordApp MakeGlobalWKN( 4149, 1 )
```

The next statements define message selectors for each application-specific message clsXWordApp requires.

```
#define msgXWordAppStartOver     MakeMsg( clsXWordApp, 1 )
#define msgXWordAppShowSoln      MakeMsg( clsXWordApp, 2 )
#define msgXWordAppSetClueTap    MakeMsg( clsXWordApp, 3 )
#define msgXWordAppDoCheck       MakeMsg( clsXWordApp, 4 )
```

Finally,

```
#endif
```

at the end of the file closes out the #ifdef statement at the beginning of the file.

xwordapp.c

xwordapp.c contains the actual implementation for the clsXWordApp crossword application class. It begins by including the familiar header files:

```
#ifndef APP_INCLUDED
#include <app.h>
#endif

#ifndef APPMGR_INCLUDED
#include <appmgr.h>
#endif

#ifndef FRAME_INCLUDED
#include <frame.h>
#endif

#ifndef FS_INCLUDED
#include <fs.h>
#endif

#ifndef RESFILE_INCLUDED
#include <resfile.h>
#endif

#ifndef TKTABLE_INCLUDED
#include <tktable.h>
#endif

#ifndef DEBUG_INCLUDED
```

```
#include <debug.h>
#endif

#include <string.h>
#include <stdio.h>
```

The header file that describes the interface to clsMenu is

```
#ifndef MENU_INCLUDED
#include <menu.h>
#endif
```

The header file that describes the interface to clsImport is

```
#ifndef IMPORT_INCLUDED
#include <import.h>
#endif
```

The header file that describes the interface to clsNote is

```
#ifndef NOTE_INCLUDED
#include <note.h>
#endif
```

Another file contains the tag definitions for objects such as default menus that PenPoint defines but the application modifies:

```
#ifndef APPTAG_INCLUDED
#include <apptag.h>
#endif
```

Finally, the interfaces to the other custom classes that implement the crossword application are included:

```
#ifndef XWORDAPP_INCLUDED
#include <xwordapp.h>
#endif

#ifndef XWRDVIEW_INCLUDED
#include <xwrdview.h>
#endif

#ifndef XWRDCLUE_INCLUDED
#include <xwrdclue.h>
#endif

#ifndef XWRDDATA_INCLUDED
#include <xwrddata.h>
```

```
#endif

#include <method.h>
```

Instance Variables clsXWordApp maintains one instance variable, xwView, in the structure

```
typedef struct INSTANCE_DATA {
  OBJECT xwView;
} INSTANCE_DATA, *P_INSTANCE_DATA;
```

I've adopted the convention of maintaining a handle to windows that I'm going to use in more than one method in a class. In this case, an alternative to using an instance variable would be to ask the application window's metrics structure to tell me the UID of the current view whenever I needed it.

Application Initialization xwordapp.c contains a standard main() routine that the Application Manager calls while maintaining crossword puzzle documents. It is defined

```
void CDECL
main(
  int                  argc,
  char *               argv[],
  U16                  processCount)
{
  if (processCount == 0) {
      ClsXWordAppInit();
      ClsXWordDataInit();
      ClsXWordViewInit();
      ClsXWordGridInit();
      AppMonitorMain(clsXWordApp, objNull);
      }
  else
      AppMain();

  Unused(argc); Unused(argv);
}
```

This routine calls the initialization functions for each of the classes the application requires. The initialization function for the application class itself is

```
STATUS ClsXWordAppInit (void)
{
  APP_MGR_NEW new;
```

```
    STATUS      s;

    ObjCallRet(msgNewDefaults, clsAppMgr, &new, s );

    new.object.uid              = clsXWordApp;
    new.cls.pMsg                = clsXWordAppTable;
    new.cls.ancestor            = clsApp;
    new.cls.size                = SizeOf(INSTANCE_DATA);
    new.cls.newArgsSize         = SizeOf(APP_NEW);

    new.appMgr.flags.accessory  = FALSE;

    strcpy(new.appMgr.name, "Crossword Puzzle");
    strcpy(new.appMgr.company, "PenPoint Programming");

    ObjCallRet(msgNew, clsAppMgr, &new, s );

    return stsOK;
}
```

Initialization Initialization of the application is handled by the XWordAppAppInit method which responds to the msgAppInit message. It is defined

```
MsgHandler(XWordAppAppInit)
{
    INSTANCE_DATA           inst;
    XWORDVIEW_NEW           vn;
    APP_METRICS             am;
    OBJECT                  mWin;
    STATUS                  s;

    ObjCallRet(msgNewDefaults, clsXWordView, &vn, s );
    ObjCallRet(msgNew, clsXWordView, &vn, s);

    inst.xwView = vn.object.uid;
    ObjectWrite( self, ctx, &inst);

    ObjCallRet(msgAppGetMetrics, self, &am, s );
    ObjCallRet(msgFrameSetClientWin, am.mainWin, inst.xwView, s );

    StsRet( XWABuildMenus( self, &mWin ), s );
    ObjCallRet(msgFrameSetMenuBar, am.mainWin, mWin, s );

    return stsOK;
    MsgHandlerParametersNoWarning;
}
```

This method initializes the application by inserting a default clsXWord-View object as the application frame's client window. Next it calls the `XWABuildMenus()` function to set up the Application's menu. If the menu is successfully created, it is used as the frame's menu bar.

Restoring clsXWordApp uses the XWordAppRestore method to respond to msgRestore by finding the UID of the xwordview object and copying it into an instance variable for future use. The XWordAppRestore method is defined

```
MsgHandlerArgType(XWordAppRestore, P_OBJ_RESTORE )
{
   INSTANCE_DATA             inst;
   APP_METRICS               am;
   STATUS                    s;

   ObjCallRet(msgAppGetMetrics, self, &am, s);
   ObjCallRet(msgFrameGetClientWin,am.mainWin,&inst.xwView,s);
   ObjectWrite( self, ctx, &inst);

   return stsOK;
   MsgHandlerParametersNoWarning;
}
```

Building the Menu Bar The clsXWordApp application implements menu management using a set of predefined tags, a predefined `TK_TABLE_ENTRY` structure, the `XWABuildMenus()` function, and a set of methods that handles the messages sent when the user selects a menu item.

Included in xwordapp.c are the predefined tags

```
#define tagXWordMenuPuzzle   MakeTag( clsXWordApp, 1 )
#define tagClueTapMenu       MakeTag( clsXWordApp, 2 )

#define mnStartOverTag       MakeTag( clsXWordApp, 3 )
#define mnShowSolnTag        MakeTag( clsXWordApp, 4 )

#define mnNothingTag         MakeTag( clsXWordApp, 5 )
#define mnStrikeOutTag       MakeTag( clsXWordApp, 6 )

#define mnPuzzleTag          MakeTag( clsXWordApp, 7 )
#define mnWordsTag           MakeTag( clsXWordApp, 8 )
#define mnLettersTag         MakeTag( clsXWordApp, 9 )
```

used by the predefined `TK_TABLE_ENTRY` structure:

```
static TK_TABLE_ENTRY XWordAppMenuBar[] = {
  {"Puzzle", 0, 0,tagXWordMenuPuzzle, tkMenuPullDown,
                          clsMenuButton},
     {"Start Over", msgXWordAppStartOver, mnStartOverTag },
     {"Show Solution", msgXWordAppShowSoln, mnShowSolnTag,
                          0, tkBorderEdgeBottom},
     {"Tapping Clue", 0, 0, 0, tkMenuPullRight},
        { 0, 0, 0, tagClueTapMenu, 0, clsChoice },
           {"Does Nothing",msgXWordAppSetClueTap,
                          mnNothingTag,
                          mnNothingTag,tkButtonOn},
           {"Strikes It Out",msgXWordAppSetClueTap,
                          mnStrikeOutTag},
           {pNull},
        {pNull},
     {"Check", 0, 0, 0, tkMenuPullRight},
          {"Puzzle ...",    msgXWordAppDoCheck, mnPuzzleTag},
          {"Words",         msgXWordAppDoCheck, mnWordsTag},
          {"Letters",       msgXWordAppDoCheck, mnLettersTag},
          {pNull},
     {pNull},
  {pNull}
};
```

Each member of the TK_TABLE_ENTRY structure contains information used to specify the menu structure. In addition to providing the type of item to be placed in the table, and the message and tag to be sent when an item is selected, there is information about the item's attributes.

Notice the use of predefined values that start with the letters tk. These values provide additional information about the menu's layout and functionality. For example,

```
{"Puzzle", 0, 0,tagXWordMenuPuzzle, tkMenuPullDown,
                          clsMenuButton},
```

uses tkMenuPullDown to indicate that the following items in the structure describe a pull-down menu that's activated using a clsMenuButton object.

Another example is

```
{"Tapping Clue", 0, 0, 0, tkMenuPullRight},
```

which indicates that the items that follow should be used to build a menu that pulls out to the right. The next line

```
{ 0, 0, 0, tagClueTapMenu, 0, clsChoice },
```

indicates that the menu will actually be a clsChoice object used to maintain a list of mutually exclusive options. In the case of this menu, the next entry uses tkButtonOn to indicate that it should be the first item selected.

Certain entries also specify one of the predefined tags as part of their information. There are two uses for this information. First, it's used as a qualifier to a single message that's meant to do more than one thing. For example, the entries each send the same message when selected by the user:

```
{"Puzzle ...",  msgXWordAppDoCheck, mnPuzzleTag},
{"Words",       msgXWordAppDoCheck, mnWordsTag},
{"Letters",     msgXWordAppDoCheck, mnLettersTag},
```

To differentiate between the entries, the method relies on the value of the third parameter (in this case a unique tag) to indicate what should be done.

The second use is to identify certain parts of a menu structure so it can be modified dynamically. For example, the structure

```
static U32 removeMenuTags[] = {
  tagAppMenuCheckpoint,
  tagAppMenuRevert,
  tagAppMenuEdit,
  0
};
```

contains tags to menus that are part of the Default Application menu. `XWABuildMenus()` uses this information to remove unwanted items from the menu it eventually displays.

Given this default information, XWordAppAppInit calls `XWABuildMenus()` to create the default application menu. `XWABuildMenus()` is defined

```
STATUS LOCAL XWABuildMenus(OBJECT self, P_OBJECT pMenuWin)
{
  MENU_NEW      mn;
  OBJECT        w;
  STATUS        s;
  U16           i;

  ObjCallRet(msgNewDefaults, clsMenu, &mn, s );
  mn.tkTable.client          = self;
  mn.tkTable.pEntries        = XWordAppMenuBar;
  ObjCallRet(msgNew, clsMenu, &mn, s );
  ObjCallRet(msgAppCreateMenuBar, self, &mn.object.uid, s );
```

```
    *pMenuWin = mn.object.uid;
    for( i=0; removeMenuTags[i]; i++ ) {
       w = (WIN)ObjectCall(msgWinFindTag, *pMenuWin,
    (P_ARGS)removeMenuTags[i] );
       ObjCallWarn( msgTkTableRemove, *pMenuWin, (P_ARGS)w );
                               }
    return stsOK;
 }
```

First, XWABuildMenus() created an instance of clsMenu by sending msgNewDefaults to clsMenu and adding the application as the client and specifying the default menu structure. Once the menu object is built, the items corresponding to the tags listed in the REMOVEMENUTAGS[] structure are removed and the function returns.

Responding to Menu Selections When the user selects a menu item, the item responds by doing one of several things. For instance, if the user selects Check, the menu manager automatically pops up the Check submenu. Some items, however, specify that a message should be sent to the menu's client. This section describes the methods that respond to the user selecting an item from a menu.

First, when the user selects Start Over, the application is sent the message msgXWordAppStartOver. The method that handles that message is defined

```
MsgHandlerWithTypes(XWordAppStartOver,P_ARGS,P_INSTANCE_DATA)
{
   return ObjCallWarn(msgXWordViewStartPlayOver,
                              pData->xwView, NULL );
   MsgHandlerParametersNoWarning;
```

This method processes the message by passing it to the crossword application view object. The message msgXWordAppShowSoln is handled in the same way:

```
MsgHandlerWithTypes(XWordAppShowSoln, P_ARGS,P_INSTANCE_DATA)
{
   return ObjCallWarn( msgXWordViewShowSoln,
                              pData->xwView, NULL );
   MsgHandlerParametersNoWarning;
}
```

Next, the message msgXWordSetClueTap is handled by the method

```
MsgHandlerWithTypes( XWordAppSetClueTap,
                                P_ARGS, P_INSTANCE_DATA )
{
  STATUS s;

  switch( (U32)pArgs ) {
      case mnNothingTag:
          ObjCallRet( msgXWordViewClueTapNothing,
                             pData->xwView,NULL, s);
          break;

      case mnStrikeOutTag:
              ObjCallRet( msgXWordViewClueTapStrikeOut,
                                 pData->xwView, NULL, s);
              break;
      }

  return stsOK;
  MsgHandlerParametersNoWarning;
}
```

This method looks at the argument passed to determine which menu item actually caused the message to be sent. It then takes appropriate action to inform the view of the user's requirements.

In the same manner, the next method processes the msgXWordAppDoCheck message when the user selects an item from the Check submenu.

```
MsgHandlerWithTypes(XWordAppDoCheck,P_ARGS,P_INSTANCE_DATA )
{
  STATUS s;

  switch( (U32)pArgs ) {
    case mnPuzzleTag:
      StsRet( XWAShowCheckPuzzleStats( pData ), s );
      break;

    case mnWordsTag:
      ObjCallRet(msgXWordViewCheckWords,
      pData->xwView, NULL, s );
      break;

    case mnLettersTag:
      ObjCallRet(msgXWordViewCheckLetters,
      pData->xwView, NULL, s );
      break;
    }
```

```
    return stsOK;
    MsgHandlerParametersNoWarning;
}
```

Note Management The previous method contained a call to the function `XWAShowCheckPuzzleStats()` when the user selected the Puzzle option from the Check submenu. This function has the responsibility of creating a clsNote object that will display information about how correct the crossword puzzle currently is.

In order to complete its task, `XWAShowCheckPuzzleStats()` needs to use predefined structures to create the note:

```
static U8 twBuff[25], cwBuff[25], tlBuff[25], clBuff[25];

static TK_TABLE_ENTRY ChkPuzzleTb[] = {
    { twBuff, 0, 0, 0, 0, clsLabel },
    { cwBuff, 0, 0, 0, 0, clsLabel },
    {" ",     0, 0, 0, 0, clsLabel },
    { tlBuff, 0, 0, 0, 0, clsLabel },
    { clBuff, 0, 0, 0, 0, clsLabel },
    {pNull}
};

static TK_TABLE_ENTRY ChkPuzzleCmdBar[] = {
    {"OK", 0, 0, 0, 0, clsButton},
    {pNull}
};
```

The first `TK_TABLE_ENTRY` structure is a place holder because the value of its contents changes each time it's displayed to the user. The `XWAShowCheckPuzzleStats()` function is defined

```
STATUS LOCAL XWAShowCheckPuzzleStats( P_INSTANCE_DATA pData )
{
    U32                     aMsg;
    NOTE_NEW                nn;
    XWORDVIEW_STATS         xvs;
    STATUS                  s;

    ObjCallRet(msgXWordViewCheckPuzzle,pData->xwView,&xvs,s);

    sprintf( twBuff, "%3d - Total Words", xvs.wordCount );
    sprintf( cwBuff, "%3d - Correct Words", xvs.okWords );
    sprintf( tlBuff, "%3d - Total Letters", xvs.letterCount );
    sprintf( clBuff, "%3d - Correct Letters", xvs.okLetters );
```

```
    ObjCallRet( msgNewDefaults, clsNote, &nn, s );
    nn.note.metrics.flags = nfSystemModal | nfUnformattedTitle;
    nn.note.pTitle = "Checking the puzzle reveals ...";
    nn.note.pContentEntries = ChkPuzzleTb;
    nn.note.pCmdBarEntries = ChkPuzzleCmdBar;
    ObjCallRet( msgNew, clsNote, &nn, s );

    ObjCallRet( msgNoteShow, nn.object.uid, (P_ARGS)&aMsg, s );

    ObjCallWarn( msgDestroy, nn.object.uid, pNull );

    return stsOK;
}
```

The first thing the method does is to send the msgXWordViewCheckPuzzle message to the view object to request current statistics for the puzzle. It then takes that information, formats it, and prints it in the predefined buffers the `TK_TABLE_ENTRY` structure points to.

Next, it creates the note as modal by 'or'ing in the nfSystemModal flag into the nn.note.metrics.flag variable. Once created, the note is made visible to the user using the msgNoteShow message. The application blocks at this point and awaits the return from the displayed note. The user signals completion by selecting the OK button at the bottom of the panel.

Importing Data Files The last two methods to be discussed in this chapter are `XWordAppImportQuery` and `XWordAppImport` which are used to support importing puzzles into PenPoint.

The `XWordAppImportQuery` method is invoked in response to the application being sent the msgImportQuery message when the user attempts to drag a non-PenPoint document into PenPoint. The method is defined

```
MsgHandlerArgType(XWordAppImportQuery, P_IMPORT_QUERY)
{

    if (ObjectCall(msgIsXWordFile, clsXWordData, pArgs->file )
            == stsOK ) {
        pArgs->canImport            = true;
        pArgs->suitabilityRating    = 100;
                                }

    return stsOK;
    MsgHandlerParametersNoWarning;
}
```

`XWordAppImportQuery` uses

```
if (ObjectCall(msgIsXWordFile, clsXWordData, pArgs->file )
        == stsOK )
```

to ask clsXWordData if the file handle references a crossword file. If the message returns stsOK, then the information

```
pArgs->canImport            = true;
pArgs->suitabilityRating    = 100;
```

is used to indicate that the file can be processed with the highest suitability rating possible.

If the user selects the crossword application to process the imported file, a new document is created, sent the msgAppInit message, and then sent the msgImport message with a handle on the open file. clsX-WordApp responds to the msgImport message with

```
MsgHandlerWithTypes(XWordAppImport, P_IMPORT_DOC,
                                P_INSTANCE_DATA)
{
   INSTANCE_DATA      inst;
   APP_METRICS        am;
   XWORDDATA_NEW      xwn;
   XWORDVIEW_NEW      vn;
   OBJECT             oldView;
   STATUS             s;

   inst = IDataDeref( pData, INSTANCE_DATA );
   oldView = inst.xwView;

   ObjCallRet(msgNewDefaults, clsXWordData, &xwn, s );
   xwn.xword.file = pArgs->file;
   ObjCallRet(msgNew, clsXWordData, &xwn, s );

   ObjCallRet(msgNewDefaults, clsXWordView, &vn, s );
   vn.view.dataObject = xwn.object.uid;
   ObjCallRet(msgNew, clsXWordView, &vn, s );

   inst.xwView = vn.object.uid;
   ObjectWrite( self, ctx, &inst);

   ObjCallRet(msgAppGetMetrics, self, &am, s );
   ObjCallRet(msgFrameSetClientWin, am.mainWin,
              inst.xwView, s);

   ObjCallWarn( msgDestroy, oldView, NULL );
```

```
    return stsOK;
    MsgHandlerParametersNoWarning;
}
```

This method responds to the import message by instructing the clsX-WordData class to make an instance of itself using the data file referenced by the open handle. Next, a view object is created for the new crossword model and replaces the default view already there. The default view is then freed.

method.tbl

method.tbl contains the following `MSG_INFO` structure for mapping messages to methods in clsXWordApp:

```
MSG_INFO clsXWordAppMethods[] = {
    msgImportQuery,   "XWordAppImportQuery", objClassMessage,
    msgImport,        "XWordAppImport",      0,
    msgAppInit,       "XWordAppAppInit",objCallAncestorBefore,
    msgRestore,       "XWordAppRestore",objCallAncestorBefore,
    msgXWordAppStartOver,   "XWordAppStartOver", 0,
    msgXWordAppShowSoln,    "XWordAppShowSoln",  0,
    msgXWordAppSetClueTap,  "XWordAppSetClueTap",0,
    msgXWordAppDoCheck,     "XWordAppDoCheck",   0,
    0
};
```

clsXWordData: The Crossword Puzzle Model Class

clsXWordData is the model class for the crossword puzzle application. It is responsible for managing the clues, words, and positional information that make each crossword puzzle document unique. clsXWordData is the class that actually contains the functionality for converting an imported data file into a model object that the user interacts with when working a crossword puzzle.

clsXWordData is implemented using the files xwrddata.h and xwrddata.c. It also has a set of entries in the method table file method.tbl.

xwrddata.h

xwrddata.h is the external interface for the Crossword Puzzle Model class clsXWordData. The beginning of the file checks to make sure it hasn't already been included:

```
#ifndef XWRDDATA_INCLUDED
#define XWRDDATA_INCLUDED
```

Next, xwrddata.h uses include directives to access the external interfaces of the other components it needs:

```
#ifndef CLSMGR_INCLUDED
#include <clsmgr.h>
#endif

#ifndef GEO_INCLUDED
#include <geo.h>
#endif
```

geo.h contains the macros and type definitions used to manipulate logical geometric constructs such as points and rectangles.

Following the include directives is the definition of the Well Known UID

```
#define clsXWordData MakeGlobalWKN( 4152, 1)
```

used to identify the clsXWordData class to the PenPoint Class Manager followed by

```
STATUS ClsXWordDataInit (void);
```

which the main() routine uses in clsXWordApp to register the clsXWordData class with the Class Manager.

Next come the message selectors for the methods unique to clsXWordData:

```
#define msgXWordDataIsXWordFile          MakeMsg(clsXWordData,1)
#define msgXWordDataGetInfo              MakeMsg(clsXWordData,2)
#define msgXWordDataGetLetters           MakeMsg(clsXWordData,3)
#define msgXWordDataGetAcrossCount       MakeMsg(clsXWordData,4)
#define msgXWordDataGetDownCount         MakeMsg(clsXWordData,5)
#define msgXWordDataGetAcrossWord        MakeMsg(clsXWordData,6)
#define msgXWordDataGetDownWord          MakeMsg(clsXWordData,7)
```

Following the message identifiers are a set of data structures used to transfer information to and from the crossword puzzle model. The first set of structures is used during the creation of a new class:

```
typedef struct XWORDDATA_NEW_ONLY {
  FILE_HANDLE       file;
  U32               size;
} XWORDDATA_NEW_ONLY, *P_XWORDDATA_NEW_ONLY;

#define xworddataNewFields \
  objectNewFields \
  XWORDDATA_NEW_ONLY xword;

typedef struct XWORDDATA_NEW {
  xworddataNewFields
} XWORDDATA_NEW, *P_XWORDDATA_NEW;
```

These structures retrieve information about a certain part of the crossword puzzle:

```
typedef U8 XWORD_DATA, *P_XWORD_DATA;

#define XWORD_MAX_WORD_SIZE 10
#define XWORD_MAX_CLUE_SIZE 40
#define XWORD_MAX_GRID_SIZE 100

typedef struct XWORDDATA_LETTER {
  U32 x;
  U32 y;
  U8  letter;
} XWORDDATA_LETTER, *P_XWORDDATA_LETTER;

typedef struct XWORDDATA_WORD {
  U32    index;
  XY32   origin;
  U8     word[XWORD_MAX_WORD_SIZE+1];
} XWORDDATA_WORD, *P_XWORDDATA_WORD;

typedef struct XWORDDATA_INFO {
  U32            size;
  XWORD_DATA     template[XWORD_MAX_GRID_SIZE];
  XWORD_DATA     numbers[XWORD_MAX_GRID_SIZE];
  OBJECT         acrossClues;
  OBJECT         downClues;
} XWORDDATA_INFO, *P_XWORDDATA_INFO;
```

Finally, at the end of the file, the statement

```
#endif
```

closes the initial #ifndef clause.

xwrddata.c

xwrddata.c contains the actual implementation for the clsXWordData Crossword Puzzle Model class. It begins by including the familiar header files:

```
#ifndef CLSMGR_INCLUDED
#include <clsmgr.h>
#endif

#ifndef FS_INCLUDED
#include <fs.h>
#endif

#ifndef OSHEAP_INCLUDED
#include <osheap.h>
#endif

#ifndef DEBUG_INCLUDED
#include <debug.h>
#endif

#include "string.h"
#include "stdio.h"
```

The interface file that describes the external interface to clsString follows:

```
#ifndef STROBJ_INCLUDED
#include <strobj.h>
#endif
```

Then the interface to clsList:

```
#ifndef LIST_INCLUDED
#include <list.h>
#endif
```

Finally, the interfaces to the Crossword Puzzle Data class itself along with the interface to the method table is included with the statements

```
#ifndef XWRDDATA_INCLUDED
#include <xwrddata.h>
```

```
#endif

#include "method.h"
```

Instance Variables clsXWordData uses this data structure to maintain information about each individual word/clue pair used in the puzzle:

```
typedef struct XWORD_ENTRY {
  U32     number;
  U32     x, y;
  U8      word[XWORD_MAX_WORD_SIZE+1];
  U8      clue[XWORD_MAX_CLUE_SIZE+1];
} XWORD_ENTRY, *P_XWORD_ENTRY;
```

Next, these statements provide clsXWordData with a structure for maintaining information on the overall organization of the crossword puzzle model:

```
#define MAX_INPUT_REC_SIZE\

   (XWORD_MAX_WORD_SIZE+XWORD_MAX_CLUE_SIZE+20)

typedef struct METRICS {
  U32 size,
      gridSize,
      acrossCnt,
      downCnt;
  U8  grid[XWORD_MAX_GRID_SIZE];
} METRICS, *P_METRICS;
```

The grid array maintains a flattened list of the characters that make up the puzzle. This array is used, for example, to generate the template that instructs the grid object which squares to black out. To get the character at position x, y, you compute:

```
x + y * size
```

Finally, a combination of the `METRICS` and and `XWORD_ENTRY` members are maintained as clsXWordData's instance data using the structure

```
typedef struct INSTANCE_DATA {
  METRICS           metrics;
  P_XWORD_ENTRY     pEntries;
} INSTANCE_DATA, *P_INSTANCE_DATA;
```

Class Registration The main() routine in clsXWordApp uses the `ClsXWordDataInit()` function to register clsXWordData with the Class Manager. It is defined

```
STATUS ClsXWordDataInit (void)
{
  CLASS_NEW   new;
  STATUS      s;

  ObjCallRet(msgNewDefaults, clsClass, &new, s );

  new.object.uid       = clsXWordData;
  new.cls.pMsg         = clsXWordDataTable;
  new.cls.ancestor     = clsObject;
  new.cls.size         = SizeOf(INSTANCE_DATA);
  new.cls.newArgsSize  = SizeOf(XWORDDATA_NEW);

  ObjCallRet(msgNew, clsClass, &new, s );

  return stsOK;
}
```

Checking Imported Files clsXWordApp invokes the class method to verify if an ASCII file contains information necessary to construct a crossword puzzle.

```
MsgHandlerArgType(XWordDataIsXWordFile, FILE_HANDLE)
{
  FILE      *fp;
  STATUS    s;

  fp = StdioStreamBind( pArgs );

  if ( !strncmp( getData(fp), XWORDDATA_LINE_1,
                        strlen(XWORDDATA_LINE_1)) )
      s = stsOK;
  else
      s = stsFailed;

  StdioStreamUnbind( fp );

  return s;
  MsgHandlerParametersNoWarning;
}
```

The check is made by seeing if the first line in the file matches

```
#define XWORDDATA_LINE_1 "pip-xwordpuzzle"
```

and returning stsOK if the match is made and stsFailed if it is not.

Notice that an extra step was needed to use the function contained in stdio.h. This step consists of binding the file handle to a file pointer using

```
fp = StdioStreamBind( pArgs );
```

prior to any file I/O to obtain a file pointer, and then calling

```
StdioStreamUnbind( fp );
```

when the file pointer is no longer needed.

Both this method and the method for initializing an object from an imported file use the utility routine

```
static P_U8 getData( FILE *fp )
{
  static U8buff[MAX_INPUT_REC_SIZE];

  return fgets( buff, MAX_INPUT_REC_SIZE, fp );
}
```

to read in the data line by line.

Creating a clsXWordData Object There are several ways to create a new model object for a crossword puzzle document, including starting from scratch or importing the initial information from outside the PenPoint environment. Either way, the following method handles the first message sent to the object, msgNewDefaults:

```
MsgHandlerArgType(XWordDataNewDefaults, P_XWORDDATA_NEW)
{
  memset( &(pArgs->xword), 0, SizeOf(XWORDDATA_NEW_ONLY) );

  return stsOK;
  MsgHandlerParametersNoWarning;
}
```

During initialization, the contents of the xword component of the XWORDDATA_NEW structure is checked for non-zero values. If the value is zero, a default set of instance data is built. Otherwise, the object is initialized from information read in from the file the xword.file value specifies.

The method that responds to msgInit is defined

```
MsgHandlerArgType(XWordDataInit, P_XWORDDATA_NEW)
{
  INSTANCE_DATA inst;
  STATUS        s;

  if ( pArgs->xword.file )
   StsRet(XWDBuildXWordFromFile(&inst,pArgs->xword.file ),s);
  else {
   memset( &inst, 0, SizeOf(INSTANCE_DATA) );
   inst.metrics.size          = pArgs->xword.size;
   inst.metrics.gridSize =
        inst.metrics.size*inst.metrics.size;
   memset( inst.metrics.grid, ' ', inst.metrics.gridSize );
                          }

  ObjectWrite(self, ctx, &inst);

  return stsOK;
  MsgHandlerParametersNoWarning;
}
```

This method checks to see if it's being asked to create the object from an imported file. If so, it uses the function

```
STATUS LOCAL
XWDBuildXWordFromFile(P_INSTANCE_DATA pData,FILE_HANDLE file)
{
  U32                        i1, j, len;
  P_XWORD_ENTRY              pEnt;
  P_METRICS                  pMet;
  P_U8                       pGrid;
  U32                        entSize;
  FILE                       *fp;
  STATUS                     s;

  fp = StdioStreamBind( file );

  getData( fp ); // ignore first line (importable check)

  pMet = &(pData->metrics);
  sscanf( getData( fp ), "%u,%u,%u",
       &(pMet->size), &(pMet->acrossCnt), &(pMet->downCnt) );
  pMet->gridSize             = pMet->size * pMet->size;

  entSize = (pMet->acrossCnt + pMet->downCnt)
               * SizeOf(XWORD_ENTRY);
```

```
    StsRet(
      OSHeapBlockAlloc(osProcessHeapId, entSize,
                                  &(pData->pEntries)) ,        s);
    memset( pData->pEntries, 0, entSize );

    pEnt = pData->pEntries;
    pGrid = pData->metrics.grid;
    memset( pGrid, 0, XWORD_MAX_GRID_SIZE );
    for ( i1=0; i1<(pMet->acrossCnt); i1++, pEnt++) {
      sscanf( getData( fp ), "%u,%u,%u,%[^,],%[^\n\r]",
            &(pEnt->number), &(pEnt->x), &(pEnt->y),
            pEnt->word, pEnt->clue );
      strncpy( &pGrid[ pEnt->y * pMet->size + pEnt->x],
                   pEnt->word, strlen( pEnt->word ) );
      }

    for ( i1=0; i1<(pMet->downCnt); i1++, pEnt++ ) {
      sscanf( getData( fp ), "%u,%u,%u,%[^,],%[^\n\r]",
            &(pEnt->number), &(pEnt->x), &(pEnt->y),
            pEnt->word, pEnt->clue );
      for ( j=0, len=strlen(pEnt->word); j<len; j++ )
        pGrid[(pEnt->y + j) * pMet->size + pEnt->x] =
            pEnt->word[j];
      }

    StdioStreamUnbind( fp );

    return stsOK;
}
```

Freeing Instance Data The following method frees any memory allocated from the heap to store the word/clue entries:

```
MsgHandlerWithTypes(XWordDataFree, P_ARGS, P_INSTANCE_DATA)
{
    if ( pData->pEntries )
      OSHeapBlockFree( pData->pEntries );

    return stsOK;
    MsgHandlerParametersNoWarning;
}
```

Responding to Save and Restore Instances of clsXWordData save their state by using this method to save their instance data:

```
MsgHandlerWithTypes(XWordDataSave,P_OBJ_SAVE,P_INSTANCE_DATA)
{
  STREAM_READ_WRITE   fsWrite;
  U32                 entCnt;
  STATUS              s;

  fsWrite.numBytes    = SizeOf(METRICS);
  fsWrite.pBuf        = &(pData->metrics);
  ObjCallRet(msgStreamWrite, pArgs->file, &fsWrite, s);

  if ( pData->pEntries ) {
    entCnt =
      pData->metrics.acrossCnt + pData->metrics.downCnt;
  fsWrite.numBytes          = entCnt * SizeOf(XWORD_ENTRY);
  fsWrite.pBuf              = pData->pEntries;
  ObjCallRet(msgStreamWrite, pArgs->file, &fsWrite, s);
  }

  return stsOK;
  MsgHandlerParametersNoWarning;
}
```

This method restores the instance data:

```
MsgHandlerArgType(XWordDataRestore, P_OBJ_RESTORE)
{
  INSTANCE_DATA             inst;
  STREAM_READ_WRITE         fsRead;
  STATUS                    s;
  U32                       entSize;
  U32                       entCnt;

  fsRead.numBytes           = SizeOf(METRICS);
  fsRead.pBuf               = &inst.metrics;
  ObjCallRet(msgStreamRead, pArgs->file, &fsRead, s);

  entCnt = inst.metrics.acrossCnt + inst.metrics.downCnt;
  if ( entCnt ) {
    entSize = entCnt * SizeOf(XWORD_ENTRY);
    StsRet(
      OSHeapBlockAlloc(osProcessHeapId, entSize,
                &inst.pEntries),    s);
```

```
      fsRead.numBytes          = entSize;
      fsRead.pBuf              = inst.pEntries;
      ObjCallJmp(msgStreamRead, pArgs->file, &fsRead,
                        s, Error);
    }

  ObjectWrite(self, ctx, &inst);

  return stsOK;
Error:
  OSHeapBlockFree( inst.pEntries );
  return s;
  MsgHandlerParametersNoWarning;
}
```

Asking for Model Information clsXWordData provides a method for returning all information about the model in a form that the view can use. The method that responds to this message is defined

```
MsgHandlerWithTypes(XWordDataGetInfo, P_XWORDDATA_INFO,
P_INSTANCE_DATA)
{
  U32                   i, l;
  P_XWORD_ENTRY         pEnt;
  P_METRICS             pMet;
  LIST_NEW              ln;
  STROBJ_NEW            son;
  U8                    buff[XWORD_MAX_CLUE_SIZE+5];
  STATUS                s;

  pArgs->size = pData->metrics.size;

  pMet = &(pData->metrics);
  memset( pArgs->template, 0, pMet->gridSize );
  memset( pArgs->numbers, 0, pMet->gridSize );

  for ( i=0; i<pMet->gridSize; i++ )
    pArgs->template[i] = pMet->grid[i] ? 1 : 0;

  pEnt = pData->pEntries;
  for(i=0, l=pMet->acrossCnt+pMet->downCnt; i<l; i++,pEnt++)
    pArgs->numbers[pEnt->x + pEnt->y * pMet->size] =
        (U8)(pEnt->number);

  ObjCallRet( msgNewDefaults, clsList, &ln, s );
  ObjCallRet( msgNew, clsList, &ln, s );
  pArgs->acrossClues = ln.object.uid;
```

```
    ObjCallRet( msgNewDefaults, clsList, &ln, s );
    ObjCallRet( msgNew, clsList, &ln, s );
    pArgs->downClues = ln.object.uid;

    pEnt = pData->pEntries;
    for ( i=0; i<pMet->acrossCnt; i++, pEnt++ ) {
      ObjCallRet( msgNewDefaults, clsString, &son, s );
      sprintf( buff, "%u. %s", pEnt->number, pEnt->clue );
      son.strobj.pString = buff;
      ObjCallRet( msgNew, clsString, &son, s );
      ObjCallRet( msgListAddItem, pArgs->acrossClues,
                  son.object.uid, s );
    }

    for ( i=0; i<pMet->downCnt; i++, pEnt++ ) {
      ObjCallRet( msgNewDefaults, clsString, &son, s );
      sprintf( buff, "%u. %s", pEnt->number, pEnt->clue );
      son.strobj.pString = buff;
      ObjCallRet( msgNew, clsString, &son, s );
      ObjCallRet( msgListAddItem, pArgs->downClues,
                  son.object.uid, s );
    }

    return stsOK;
    MsgHandlerParametersNoWarning;
}
```

This method provides information concerning the size of the puzzle:

```
pArgs->size = pData->metrics.size;
```

a template that indicates which blocks should be black:

```
for ( i=0; i<pMet->gridSize; i++ )
   pArgs->template[i] = pMet->grid[i] ? 1 : 0;
```

and a template indicating which blocks should be numbered:

```
pEnt = pData->pEntries;
for(i=0, l=pMet->acrossCnt+pMet->downCnt; i<l; i++,pEnt++)
  pArgs->numbers[pEnt->x + pEnt->y * pMet->size] =
        (U8)(pEnt->number);
```

This method also constructs two clue lists (across and down) by creating two list objects:

```
ObjCallRet( msgNewDefaults, clsList, &ln, s );
ObjCallRet( msgNew, clsList, &ln, s );
pArgs->acrossClues = ln.object.uid;
```

It then fills them with new string objects: that represent the puzzle's clues

```
pEnt = pData->pEntries;
for ( i=0; i<pMet->acrossCnt; i++, pEnt++ ) {
  ObjCallRet( msgNewDefaults, clsString, &son, s );
  sprintf( buff, "%u. %s", pEnt->number, pEnt->clue );
  son.strobj.pString = buff;
  ObjCallRet( msgNew, clsString, &son, s );
  ObjCallRet( msgListAddItem, pArgs->acrossClues,
              son.object.uid, s );
  }
```

The clsXWordData class also implements methods to return the solution:

```
MsgHandlerWithTypes(XWordDataGetLetters,
                         P_XWORD_DATA, P_INSTANCE_DATA)
{
  memcpy(pArgs,pData->metrics.grid,pData->metrics.gridSize);

  return stsOK;
  MsgHandlerParametersNoWarning;
}
```

the number of across entries:

```
MsgHandlerWithTypes(XWordDataGetAcrossCount,
                         P_U32, P_INSTANCE_DATA)
{
  *pArgs = pData->metrics.acrossCnt;

  return stsOK;
  MsgHandlerParametersNoWarning;
}
```

the number of down entries:

```
MsgHandlerWithTypes(XWordDataGetDownCount,
                         P_U32, P_INSTANCE_DATA)
{
  *pArgs = pData->metrics.downCnt;

  return stsOK;
```

```
    MsgHandlerParametersNoWarning;
}
```

the correct spelling of a word in the across direction:

```
MsgHandlerWithTypes(XWordDataGetAcrossWord,
                              P_XWORDDATA_WORD, P_INSTANCE_DATA)
{
   if ( pData->metrics.acrossCnt ) {
      pArgs->origin.x = pData->pEntries[pArgs->index].x;
      pArgs->origin.y = pData->pEntries[pArgs->index].y;
      strcpy( pArgs->word, pData->pEntries[pArgs->index].word );
      }
   else
      memset( pArgs, 0, SizeOf(XWORDDATA_WORD) );

   return stsOK;
   MsgHandlerParametersNoWarning;
}
```

and the correct spelling of a word in the down direction:

```
MsgHandlerWithTypes(XWordDataGetDownWord,
                              P_XWORDDATA_WORD, P_INSTANCE_DATA)
{
if ( pData->metrics.downCnt ) {
   pArgs->origin.x =
   pData->pEntries[pData->metrics.acrossCnt+pArgs->index].x;
   pArgs->origin.y =
   pData->pEntries[pData->metrics.acrossCnt+pArgs->index].y;
   strcpy( pArgs->word,
   pData->pEntries[pData->metrics.acrossCnt+pArgs->index].word
      );
   }
else
   memset( pArgs, 0, SizeOf(XWORDDATA_WORD) );

return stsOK;
MsgHandlerParametersNoWarning;
}
```

method.tbl

method.tbl contains the following `MSG_INFO` structure for mapping messages to methods in clsXWordData:

```
MSG_INFO clsXWordDataMethods[] = {
  msgNewDefaults,                  "XWordDataNewDefaults",
      objCallAncestorBefore,
  msgInit,                         "XWordDataInit",
      objCallAncestorBefore,
  msgFree,                         "XWordDataFree",
      objCallAncestorAfter,
  msgSave,                         "XWordDataSave",
      objCallAncestorBefore,
  msgRestore,                      "XWordDataRestore",
      objCallAncestorBefore,
  msgXWordDataIsXWordFile,         "XWordDataIsXWordFile",
      objClassMessage,
  msgXWordDataGetInfo,             "XWordDataGetInfo",              0,
  msgXWordDataGetLetters,          "XWordDataGetLetters",           0,
  msgXWordDataGetAcrossCount       "XWordDataGetAcrossCount",       0,
  msgXWordDataGetDownCount,        "XWordDataGetDownCount",         0,
  msgXWordDataGetAcrossWord,       "XWordDataGetAcrossWord",        0,
  msgXWordDataGetDownWord,         "XWordDataGetDownWord",          0,
  0
};
```

Wrap-up

This chapter presents the foundation for building a functional crossword application for PenPoint. It includes a small User's Guide, a description of the classes created to complete the project, an explanation of the new library classes used, and, finally, a description of the application class itself.

By now, you're probably thinking you can make several additions to the application's functionality. If so, here's a topic to think about that isn't covered in this chapter: stationery. Stationery is predefined templates used to create new documents. One area in which the crossword puzzle would benefit from using stationery is creating new puzzles. For instance, you could create a new puzzle starting with a piece of 5x5 stationery, filling in the answers, and then filling in the clues. Once the document is fully created, you could select Start Over from the Puzzle menu and then make a copy of that document available on some form of distribution media.

For now, if you want to create a new puzzle, you have to build an ASCII file with the information in the format described at the beginning of the chapter. Here's a second puzzle so you're not stuck with just one:

```
pip-xwordpuzzle
6,6,6
2,4,0,PA,Ma and ___
4,0,1,DEBT,Owe
7,1,2,BRAND,Type
10,2,3,AXE,Lumberjack's Tool
11,2,4,WET,Not dry
12,0,5,BUNS,Rolls
1,1,0,FEB,Second month
3,5,0,AID,Help
5,2,1,BRAWN,Muscle
6,3,1,TAXES,We pay too many
8,4,2,NET,Fish or butterfly
9,0,3,FIB,Little lie
```

9

Coordinating Views

Back in the early '80s, Apple did something rather bold. It released a machine whose internals couldn't be accessed. When you bought an early Macintosh, you received one kind of keyboard, one kind of mouse, one kind of display, and so on. This was very different from the PC market—where assembling a machine could take days because you could choose so many different components from so many different vendors. Of course, time, the market, and screaming users demanded that the Macintosh be opened up, and so Apple produced the Mac II.

And lots of things broke.

For instance, applications written to take advantage of Motorola's addressing scheme for the 68000 by stuffing attributes into the unused eight bits of an address would run for a while on the new hardware and then suddenly go belly up. Or, a window would grow beyond a certain size and look very strange, all because someone forgot to check the screen resolution assuming it would always be the same.

I wouldn't have told this story except for one thing: its historical significance. People drive technology. The more people that use a technology the more that technology tends to change. PenPoint is going to open personal computing to a large number of people for the first time. Past experience shows that these users will demand change and that hardware vendors eager to differentiate their products will happily oblige.

GO anticipated this by building PenPoint to be scaleable, both in size and in functionality. What this means to you, me, and others who write programs for PenPoint is that we must take care to write well-behaved code that avoids using knowledge of the layers underlying the Application Programmer's Interface.

For example, if you need to know whether to lay an application out in a horizontal or vertical format based on the screen orientation, you should ask PenPoint. Layout is also a concern to keep in mind when transferring documents from one pen computer to another because a document originally laid out horizontally and then saved might be restored on a vertically oriented machine!

The two classes described in this chapter illustrate several PenPoint features that help manage change in the user environment your application runs in. The first class, clsXWordView, manages the components the user needs to work the crossword puzzle. clsXWordView asks PenPoint for its screen orientation and then decides how to lay itself out in the most visually appealing manner. The second class, clsXWordClue, is a simple class that displays a header followed by a list of strings. I have implemented this class as a DLL to illustrate one way of writing and distributing interchangeable and scaleable components.

User Preference Support

Most new operating environments come with a way for the user to customize the working environment. PenPoint supports user preferences by providing a preferences application that the user interacts with to select the system preferences. PenPoint then stores this information in a resource file that all PenPoint applications can access to find out what defaults the user wants.

This section touches briefly on the concept of resource files and how they are used to store the user preferences. It then talks about the user preference resource file in particular and the type of information it contains.

Resources

A resource in PenPoint is a collection of data identified with a unique ID. Programs use resources to maintain information such as string tables, persistent objects, component descriptions for option sheets, and any other data an application wishes to maintain.

Resources are managed by the **Resource Manager**, can be of any size, and can contain any type of information. The Resource Manager provides the functionality for reading, writing, and accessing particular resource items. In addition to the predefined types of resources such as objects, you can define custom agents that allow the Resource Manager to efficiently manipulate resources composed of custom types.

Another feature of a resource is that you can describe it in a text file, as you would a program, and then use a GO tool to compile it. This is useful in areas such as internationalization where you might wish to supply different string resources based on the language used. You can pre-build different resource files and then let the user select the one that makes the most sense.

System Preferences

PenPoint keeps user preferences in a resource file called **generic** that standard Resource Manager calls can access. Preferences are organized as a set of different resources that can be accessed by using a set of Well Known Resource IDs available to any application.

Screen Orientation The crossword puzzle clsXWordView uses the screen orientation resource **prOrientation** to lay out its components. The prOrientation resource contains an unsigned 8-bit (U8) value set to one of two values:

- **prPortrait** indicates that the long edge of the screen is vertical.
- **prLandscape** indicates that the long edge of the screen is horizontal.

Other Preferences In addition to prOrientation, PenPoint also supports the user preferences listed in Table 9.1.

TABLE 9.1 Other User Preferences.

Preference	*Sets default for*
prSystemFont	Font used to display text the system maintains
prUserFont	Font used to display text the user enters
prHandPreference	Display layout—left- or right-handed writing
prWritingStyle	Writing style—Mixed or uppercase
prGestureTimeout	Length of time waited before detecting the end of a gesture
prHWXTimeout	Length of time waited before detecting the end of a written entry

prPenHoldTimeout	Length of time the user must hold the pen to the screen for a pen event to be generated
prInputPadStyle	Type of input pad (boxed or ruled) presented to the user
prLineHeight	Line height for writing pads
prCharBoxWidth	Width of character-entry boxes
prCharBoxHeight	Height of character-entry boxes
prTimeFormat	Time display—12- or 24-hour format
prTimeSeconds	Displaying or not displaying seconds field
prTime	System time value
prDateFormat	Format for displaying dates
prDocFloating	Document can or can not be floated
prDocZooming	Document can or can not be zoomed
prBell	Bell is on or off
prInactivityPowerDown	Length of time to wait from the user's last interaction before automatically powering down
prPenCursor	Visible or invisible cursor
prPrimaryInput	Primary input—the pen or a keyboard

Packaging Components for Reuse: DLLs

Every now and then, you build a component with a usefulness that transcends the application you started writing it for. When this happens, you need to think about packaging that component for reuse. Traditionally, C programmers have packaged reusable code in the form of libraries that can be linked in when the executable application is built.

Unfortunately, this method of code sharing leaves each application with its own copy of the library. Several operating systems provide a packaging technology that reduces code duplication by keeping one copy of the library loaded and allowing applications that need it to link to it at runtime. Supported by PenPoint, these reusable libraries, called Dynamic Link Libraries (or DLLs), are compatible with the code sharing inherent in PenPoint's support of object-based inheritance.

Dynamic Link Libraries

DLLs in PenPoint are self-contained units of functionality that different applications can share. The linker and loader work in concert to provide information necessary to resolve all references when the application is loaded, including loading the DLLs required by the application if they're not already present.

The build process for using DLLs is very simple and straightforward. I refer you to the PenPoint tools manual for more details about compiling and linking DLLs.

DLLMain The one addition necessary to support DLLs is that the module to be placed in the library must have a special entry point. By convention, this entry point is normally named DLLMain and is called without arguments. The loader expects the routine to return sysOK if it initialized without error. Normally, the initialization functions for classes defined in the DLL are called at this point.

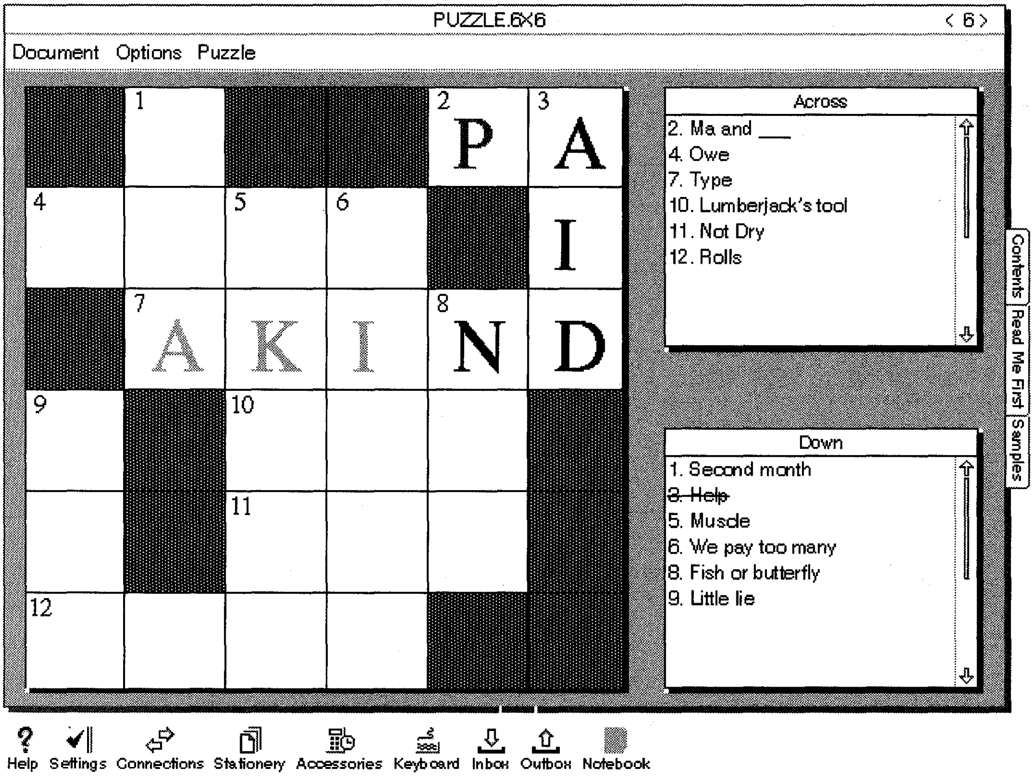
clsXWordView: The Crossword Puzzle View Class

The clsXWordView class serves as the liaison between the application model (clsXWordData) and the user. A subclass of clsView, it is a composite object that provides instances of clsXWordClueList to give the user a list of clues and an instance of clsXWordGrid for the user to work the puzzle on.

The clsXWordView object is created as a result of the user creating a puzzle document. It can be created for a particular clsXWordData object that might have been imported, or it can create a default display that can help in building new puzzles. Once created, it is inserted into the application's frame and becomes the document's main view.

clsXWordView's layout code has been built to look at the system preferences and lay itself out differently based on the orientation of the display. For example, Figure 8.1 showed the crossword puzzle in portrait layout, while Figure 9.1 shows a crossword document in landscape layout. In addition to coordinating the building of display components, clsXWordView also manages the correctness checks that the user selects from the Application menu.

FIGURE 9.1 Horizontal (Landscape) Layout of the Crossword Puzzle Application



clsXWordView is implemented in three parts: the interface file xwrdview.h; the implementation file xwrdview.c; and a structure residing in method.tbl that maps messages sent to clsXWordView objects to the methods that handle those messages.

xwrdview.h

xwrdview.h is the external interface for the Crossword Puzzle View class clsXWordView. The file begins by checking to make sure the file hasn't already been included

```
#ifndef XWRDVIEW_INCLUDED
#define XWRDVIEW_INCLUDED
```

If this is the first access to the file, the first action taken is to include the interface files for the other components it relies upon, using the statements

```
#ifndef GO_INCLUDED
#include <go.h>
#endif

#ifndef CLSMGR_INCLUDED
#include <clsmgr.h>
#endif

#ifndef VIEW_INCLUDED
#include <view.h>
#endif
```

Following the include directives is the definition of the Well Known ID

```
#define clsXWordView     MakeGlobalWKN(4150,1)
```

used to identify the clsXWordView class to the PenPoint Class Manager, followed by

```
STATUS ClsXWordViewInit(void);
```

which the main() routine uses in clsXWordApp to register the clsXWordView class with the Class Manager.

Next come the message selectors used to define messages new to clsXWordView. They are defined

```
#define msgXWordViewStartPlayOver   MakeMsg(clsXWordView,1)
#define msgXWordViewShowSoln        MakeMsg(clsXWordView,2)
#define msgXWordViewClueTapNothing  MakeMsg(clsXWordView,3)
#define msgXWordViewClueTapStrikeOut\
                                    MakeMsg(clsXWordView,4)
#define msgXWordViewCheckPuzzle     MakeMsg(clsXWordView,5)
#define msgXWordViewCheckLetters    MakeMsg(clsXWordView,6)
#define msgXWordViewCheckWords      MakeMsg(clsXWordView,7)
```

Following the message selectors are a set of data structures used to transfer information to and from the crossword puzzle model. The first set of structures is used during the creation of a new class:

```
#define xwordviewNewFields \
  viewNewFields

typedef struct XWORDVIEW_NEW {
```

```
  xwordviewNewFields
} XWORDVIEW_NEW, *P_XWORDVIEW_NEW;
```

The next structure is used by another class (in this example the clsX-WordApp object) to find out how many letters and words the user has correctly entered on the grid:

```
typedef struct XWORDVIEW_STATS {
  U32  wordCount,
       okWords,
       letterCount,
       okLetters;
} XWORDVIEW_STATS, *P_XWORDVIEW_STATS;
```

Finally at the end of the file, the statement

```
#endif
```

closes the initial #ifndef clause.

xwrdview.c

xwrdview.c contains the actual implementation for the clsXWordView Crossword User View class. It begins by including the familiar header files:

```
#ifndef WIN_INCLUDED
#include <win.h>
#endif

#ifndef FS_INCLUDED
#include <fs.h>
#endif

#ifndef DEBUG_INCLUDED
#include <debug.h>
#endif

#include <stdio.h>
#include <string.h>
```

Following the reader files is the interface file that describes the external interface to the preferences resource file:

```
#ifndef PREFS_INCLUDED
#include <prefs.h>
#endif
```

Finally, the interfaces to the Crossword Puzzle classes used by the View class, the View class itself, and the entries generated by the method compiler are included with the statements:

```
#ifndef XWRDVIEW_INCLUDED
#include <xwrdview.h>
#endif

#ifndef XWRDGRID_INCLUDED
#include <xwrdgrid.h>
#endif

#ifndef XWRDCLUE_INCLUDED
#include <xwrdclue.h>
#endif

#ifndef XWRDDATA_INCLUDED
#include <xwrddata.h>
#endif

#include <method.h>
```

Component Window Tags clsXWordView relies on the Window Manager to save and restore the state of the list and grid components it creates. The following tags are defined so that the window IDs can be located after the windows themselves have been restored:

```
#define   gridWinTag     MakeTag( clsXWordView, 1 )
#define   acrossWinTag   MakeTag( clsXWordView, 2 )
#define   downWinTag     MakeTag( clsXWordView, 3 )
```

Instance Variables clsXWordView maintains its instance data using the structure

```
typedef struct INSTANCE_DATA {
    U8      dispOrientation;
    U32     size;
    U32     gridSize;
    OBJECT  model;
    OBJECT  grid;
    OBJECT  acrossClues;
    OBJECT  downClues;
} INSTANCE_DATA, *P_INSTANCE_DATA;
```

clsXWordView uses

- **dispOrientation** to indicate whether the puzzle should be laid out for portrait or landscape viewing
- **size** and **gridSize** to store the length and total number of squares in the grid, respectively
- **model** to identify the instance of clsXWordData that contains the solution to the puzzle
- **grid, acrossClues,** and **downClues** to hold the IDs of the components used to display information and interact with the user.

Class Registration The main() routine in clsXWordApp uses the ClsXWordViewInit() function to register clsXWordView with the Class Manager. It is defined

```
STATUS ClsXWordViewInit(void)
{
  CLASS_NEW c;
  STATUS    s;

  ObjCallRet(msgNewDefaults, clsClass, &c, s );
  c.object.uid        = clsXWordView;
  c.cls.pMsg          = clsXWordViewTable;
  c.cls.ancestor      = clsView;
  c.cls.size          = SizeOf(INSTANCE_DATA);
  c.cls.newArgsSize   = SizeOf(XWORDVIEW_NEW);
  ObjCallRet(msgNew, clsClass, &c, s );

  return stsOK;
}
```

Creating a clsXWordView Object A new instance of clsXWordView is created by sending msgNewDefaults, to initialize the `XWORDVIEW_NEW` structure. The method that responds to this message is

```
MsgHandlerArgType(XWordViewNewDefaults, P_XWORDVIEW_NEW)
{
  pArgs->view.createDataObject = TRUE;

  return stsOK;
  MsgHandlerParametersNoWarning;
}
```

Next, the `XWORDVIEW_NEW` structure is filled out and used as the parameter when msgNew is sent to clsXWordView. The method that responds to the msgInit message that is sent is

```
MsgHandlerArgType(XWordViewInit, P_XWORDVIEW_NEW)
{
  INSTANCE_DATA   inst;
  WIN_METRICS     wm;
  BORDER_STYLE    bs;
  XWORDDATA_NEW   xwn;
  RES_READ_DATA   read;
  STATUS          s;
  XWORDDATA_INFO  xwrdInfo;

  if ( !(pArgs->view.dataObject)
       && pArgs->view.createDataObject ) {
     ObjCallRet(msgNewDefaults, clsXWordData, &xwn, s );
     xwn.xword.size = 10;
     ObjCallRet(msgNew, clsXWordData, &xwn, s);
     ObjCallRet(msgViewSetDataObject,self,
     xwn.object.uid,s);
     inst.model = xwn.object.uid;
     }
  else
     inst.model = pArgs->view.dataObject;

  read.resId       = prOrientation;
  read.heap        = 0;
  read.pData       = &inst.dispOrientation;
  read.length      = SizeOf(U8);
  ObjCallRet(msgResReadData,theSystemPreferences,&read,s);

  ObjCallRet(msgXWordDataGetInfo, inst.model, &xwrdInfo,s);

  inst.size        = xwrdInfo.size;
  inst.gridSize    = inst.size * inst.size;

  StsRet( XWVBuildClueList( "Across", xwrdInfo.acrossClues,
                      acrossWinTag,&inst.acrossClues),s);
  StsRet( XWVBuildClueList( "Down", xwrdInfo.downClues,
                      downWinTag, &inst.downClues ), s);

  StsRet( XWVBuildGrid(inst.size, inst.gridSize,
                      xwrdInfo.template,xwrdInfo.numbers,
                      gridWinTag, &inst.grid ), s);

  ObjectWrite(self, ctx, &inst);

  ObjCallRet(msgBorderGetStyle, self, &bs, s );

  bs.backgroundInk = bsInkGray33;
  ObjCallWarn(msgBorderSetStyle, self, &bs );
```

```
    wm.parent   = self;
    wm.options = wsPosTop;
    ObjCallRet( msgWinInsert, inst.acrossClues, &wm, s );
    ObjCallRet( msgWinInsert, inst.downClues, &wm, s );
    ObjCallRet( msgWinInsert, inst.grid, &wm, s );

    return stsOK;
    MsgHandlerParametersNoWarning;
}
```

First, the XWordViewInit method checks to see if a data object has been created. If so, it's used to initialize the view. Otherwise, an empty 10x10 default crossword puzzle is created. This will be used to display a blank grid to the user who could then use it to help generate new puzzles.

After the model issue is resolved, the preferences resource file is queried to find out the orientation of the display. A read structure is set up and then sent to the globally defined object theSystemPreferences.

Next, the data object is asked to fill in an information structure that indicates the size of the puzzle and provides a list of across clues and a list of down clues. Using that information, gridSize is computed, and then the grid, acrossList, and downList objects are created using the `XWVBuildClueList()` and `XWVBuildGrid()` functions.

When the instance data is completely initialized, it is written back to protected memory. The last thing the method does is to set its background color to light gray, and then insert each of its component windows as its children.

The `XWVBuildClueList()` function is defined

```
STATUS LOCAL
XWVBuildClueList(P_STRING pTitle,OBJECT clueList,
                 TAG winTag, P_OBJECT pList )
{
  XWORDCLUE_NEW  xwc;
  STATUS         s;

  ObjCallRet(msgNewDefaults, clsXWordClueList, &xwc, s);
  xwc.win.tag                  = winTag;
  xwc.border.style.edge        = bsEdgeAll;
  xwc.border.style.shadow      = bsShadowThickBlack;
  xwc.xwclue.pTitle            = pTitle;
  xwc.xwclue.clueList          = clueList
  ObjCallRet(msgNew, clsXWordClueList, &xwc, s);
```

```
    *pList = xwc.object.uid;
    return stsOK;
}
```

This function creates a clsXWordCLueList window named pTitle that is surrounded with an edge, given a thick black shadow, displays the list of clues in clueList, and can be identified by the tag winTag.

The XWVBuildGrid() function is defined

```
STATUS LOCAL
XWVBuildGrid( U32 size, U32 gridSize,
          P_XWORD_DATA pTemplate, P_XWORD_DATA pNumbers,
          TAG winTag, P_OBJECT pGrid )
{
    XWORDGRID_NEW  xwc;
    STATUS         s;
    U32            i;

    ObjCallRet(msgNewDefaults, clsXWordGrid, &xwc, s);
    xwc.win.tag              = winTag;
    xwc.border.style.shadow = bsShadowThickBlack;
    xwc.xwgrid.size          = size;
    for ( i=0; i<gridSize; i++ ) {
       xwc.xwgrid.template[i] = pTemplate[i];
       xwc.xwgrid.numbers[i]  = pNumbers[i];
       }
    ObjCallRet(msgNew, clsXWordGrid, &xwc, s);

    *pGrid = xwc.object.uid;
    return stsOK;
}
```

This function uses the code

```
 for ( i=0; i<gridSize; i++ ) {
    xwc.xwgrid.template[i] = pTemplate[i];
    xwc.xwgrid.numbers[i]  = pNumbers[i];
    }
```

to initialize an array that tells clsXWordGrid which squares should be blacked out in the grid (pTemplate), and which squares should be numbered (pNumbers). The pNumbers array is set up so an entry will be zero, unless it has a number to be displayed.

Responding to Save and Restore Instances of clsXWordView save part of their state, but also rely on the Window Manager to save the component windows used to construct the view. The method that responds to msgSave is

```
MsgHandlerWithTypes(XWordViewSave,P_OBJ_SAVE,
                                  P_INSTANCE_DATA)
{
  STREAM_READ_WRITE fsWrite;
  STATUS            s;


  fsWrite.numBytes  = SizeOf(U32);
  fsWrite.pBuf      = &(pData->size);
  ObjCallRet(msgStreamWrite, pArgs->file, &fsWrite, s );

  return stsOK;
  MsgHandlerParametersNoWarning;
}
```

The method that responds to msgRestore is implemented as

```
MsgHandlerArgType(XWordViewRestore, P_OBJ_RESTORE)
{

  INSTANCE_DATA      inst;
  RES_READ_DATA      read;
  STREAM_READ_WRITE  fsRead;
  STATUS             s;

  fsRead.numBytes = SizeOf(U32);
  fsRead.pBuf     = &inst.size;
  ObjCallRet(msgStreamRead, pArgs->file, &fsRead, s );

  inst.gridSize = inst.size * inst.size;

  read.resId    = prOrientation;
  read.heap     = 0;
  read.pData    = &inst.dispOrientation;
  read.length   = SizeOf(U8);
  ObjCallRet(msgResReadData,theSystemPreferences,&read,s);

  inst.grid =
  (WIN)ObjectCall(msgWinFindTag,self,(P_ARGS)gridWinTag);
  inst.acrossClues =
 (WIN)ObjectCall(msgWinFindTag,self,(P_ARGS)acrossWinTag);
  inst.downClues =
  (WIN)ObjectCall(msgWinFindTag,self,(P_ARGS)downWinTag);
```

```
    ObjCallRet(msgViewGetDataObject,self,&inst.model,s);

    ObjectWrite(self, ctx, &inst);

    return stsOK;
    MsgHandlerParametersNoWarning;
  }
```

At first, it might seem redundant to check for the screen orientation on a document that has been filed out. However, the check is important because the document could have been saved on one tablet, transferred to another, and then restored. It's possible that the user on the new machine might have oriented the tablet's screen differently. Actually, now is a good time to point out that the screen resolution might not even be the same for the two machines!

clsXWordView also overrides the msgViewSetDataObject with the method

```
MsgHandlerArgType( XWordViewSetDataObject, OBJECT )
{
  INSTANCE_DATA inst;

  inst = IDataDeref( pData, INSTANCE_DATA );
  inst.model = pArgs;
  ObjectWrite( self, ctx, &inst );

  return stsOK;
  MsgHandlerParametersNoWarning;


}
```

to keep track of when the model changes.

Window Layout As a subclass of clsView, which inherits from clsCustomLayout, instances of clsXWordView receive notification that a window layout episode has started. clsXWordView responds to this notification using the method

```
MsgHandlerWithTypes(XWordViewCLGetChildSpec,
                 P_CSTM_LAYOUT_CHILD_SPEC,P_INSTANCE_DATA)
{
  if ( pData->dispOrientation == prLandscape )
     XWVLandscapeLayout( pData, pArgs );
  else
     XWVPortraitLayout( pData, pArgs );
```

```
  return stsOK;
  MsgHandlerParametersNoWarning;
}
```

This method uses the dispOrientation instance variable to choose which layout function (`XWVLandscapeLayout()` or `XWVPortrait-Layout()`) it should call. These rather verbose functions are defined

```
LOCAL
XWVLandscapeLayout( P_INSTANCE_DATA pData,
                    P_CSTM_LAYOUT_CHILD_SPEC pSpec)
{
  if ( pSpec->child == pData->grid ) {
     pSpec->metrics.h.constraint = clPctOf;
     pSpec->metrics.h.value      = 96;
     pSpec->metrics.w.constraint = clSameAs | clOpposite;
     pSpec->metrics.w.relWin     = pSpec->child;
     pSpec->metrics.x.constraint =
          ClAlign( clMinEdge, clPctOf, clMaxEdge );
     pSpec->metrics.x.value      = 2;
     pSpec->metrics.y.constraint =
          ClAlign(clCenterEdge, clSameAs, clCenterEdge);
     }
  else if ( pSpec->child == pData->acrossClues ) {
     pSpec->metrics.w.constraint =
          ClExtend(clPctOf, clMaxEdge);
     pSpec->metrics.w.value      = 98;
     pSpec->metrics.h.constraint = clPctOf;
     pSpec->metrics.h.value      = 44;
     pSpec->metrics.h.relWin     = pData->grid;
     pSpec->metrics.x.constraint =
          ClAlign(clMinEdge, clPctOf, clMaxEdge);
     pSpec->metrics.x.value      = 106;
     pSpec->metrics.x.relWin     = pData->grid;
     pSpec->metrics.y.constraint =
          ClAlign(clMaxEdge, clSameAs, clMaxEdge);
     pSpec->metrics.y.relWin     = pData->grid;
     }
  else if ( pSpec->child == pData->downClues ) {
     pSpec->metrics.w.constraint =
          ClExtend(clPctOf, clMaxEdge);
     pSpec->metrics.w.value      = 98;
     pSpec->metrics.h.constraint = clPctOf;
     pSpec->metrics.h.value      = 44;
     pSpec->metrics.h.relWin     = pData->grid;
     pSpec->metrics.x.constraint =
          ClAlign(clMinEdge, clPctOf, clMaxEdge);
```

```
        pSpec->metrics.x.relWin     = pData->grid;
        pSpec->metrics.x.value      = 106;
        pSpec->metrics.y.constraint =
             ClAlign(clMinEdge, clSameAs, clMinEdge);
        pSpec->metrics.y.relWin     = pData->grid;
        }
  return stsOK;
}
```

Notice that in several places I use a value of 106 percent to place the alignment coordinate outside the relative window.

Next, the function that manages portrait layout is defined

```
LOCAL
XWVPortraitLayout(  P_INSTANCE_DATA pData,
                    P_CSTM_LAYOUT_CHILD_SPEC pSpec )
{
  if ( pSpec->child == pData->grid ) {
     pSpec->metrics.h.constraint = clSameAs | clOpposite;
     pSpec->metrics.h.relWin     = pSpec->child;
     pSpec->metrics.w.constraint = clPctOf;
     pSpec->metrics.w.value      = 80;
     pSpec->metrics.x.constraint =
          ClAlign(clCenterEdge, clSameAs, clCenterEdge);
     pSpec->metrics.y.constraint =
          ClAlign( clMaxEdge, clPctOf, clMaxEdge );
     pSpec->metrics.y.value      = 98;
     }
  else if ( pSpec->child == pData->acrossClues ) {
     pSpec->metrics.h.constraint =
          ClExtend(clPctOf, clMinEdge);
     pSpec->metrics.h.value      = 94;
     pSpec->metrics.h.relWin     = pData->grid;
     pSpec->metrics.w.constraint = clPctOf;
     pSpec->metrics.w.value      = 44;
     pSpec->metrics.w.relWin     = pData->grid;
     pSpec->metrics.y.constraint =
          ClAlign(clMinEdge, clPctOf, clMaxEdge);
     pSpec->metrics.y.value      = 2;
     pSpec->metrics.x.constraint =
          ClAlign(clMinEdge, clSameAs, clMinEdge);
     pSpec->metrics.x.relWin     = pData->grid;
     }


 else if ( pSpec->child == pData->downClues ) {
     pSpec->metrics.h.constraint =
          ClExtend(clPctOf, clMinEdge);
```

```
        pSpec->metrics.h.value       = 94;
        pSpec->metrics.h.relWin      = pData->grid;
        pSpec->metrics.w.constraint  = clPctOf;
        pSpec->metrics.w.value       = 44;
        pSpec->metrics.w.relWin      = pData->grid;
        pSpec->metrics.y.constraint  =
             ClAlign(clMinEdge, clPctOf, clMaxEdge);
        pSpec->metrics.y.value       = 2;
        pSpec->metrics.x.constraint  =
             ClAlign(clMaxEdge, clSameAs, clMaxEdge);
        pSpec->metrics.x.relWin      = pData->grid;
        }
     return stsOK;
   }
```

Controlling Play clsXWordView defines several methods that control how the user plays or works the crossword puzzle. They are XWordViewShowSoln and XWordViewStartOver, which respond to the messages msgXWordViewShowSoln and msgXWordViewStartOver, respectively.

XWordViewShowSoln gets the correct answers from the model object and forwards them to the grid object. It is defined

```
MsgHandlerWithTypes(XWordViewShowSoln,
                    P_ARGS, P_INSTANCE_DATA)
{
  XWORD_DATA solution[XWORD_MAX_GRID_SIZE];
  GRID_DATA  gridData[GRID_MAX_GRID_SIZE];
  U32        i;
  STATUS     s;

  ObjCallRet( msgXWordDataGetLetters, pData->model,
             &solution, s );
  for ( i=0; i<pData->gridSize; i++ )
     gridData[i] = solution[i];

 ObjCallRet(msgXWordGridSetLetters,pData>grid,gridData,s);

  return stsOK;
  MsgHandlerParametersNoWarning;
}
```

XWordViewStartPlayOver instructs the grid and clue objects to reset themselves as if the user never worked the puzzle. It is defined

```
MsgHandlerWithTypes(XWordViewStartPlayOver,
                         P_ARGS, P_INSTANCE_DATA)
{
```

```
  STATUS s;

  ObjCallRet(  msgXWordGridStartPlayOver,pData->grid,
               NULL, s );
  ObjCallRet(  msgXWordClueStartPlayOver, pData>acrossClues,
               NULL, s );
  ObjCallRet(  msgXWordClueStartPlayOver, pData->downClues,
               NULL, s );

  return stsOK;
  MsgHandlerParametersNoWarning;
}
```

Controlling the Clue Lists clsXWordView defines several methods that control the user's interaction with the items on the clue lists. These methods are invoked by messages sent from the clsXWordApp object when it receives messages from the menu bar requesting a user command be executed.

Two methods, XWordViewClueTapNothing, which responds to the message msgXWordViewClueTapNothing, and XWordViewClueTapStrikeOut, which responds to the message msgXWordViewClueTapStrikeOut, act as forwarders to the clue list component objects. They are defined

```
MsgHandlerWithTypes(XWordViewClueTapNothing,
                P_ARGS, P_INSTANCE_DATA)
{
  STATUS s;

  ObjCallRet(msgXWordClueClueTapNothing,pData->acrossClues,
             NULL, s );
  ObjCallRet(msgXWordClueClueTapNothing,pData->downClues,
             NULL, s );

  return stsOK;
  MsgHandlerParametersNoWarning;
}


MsgHandlerWithTypes(XWordViewClueTapStrikeOut, P_ARGS,
                      P_INSTANCE_DATA)


{
  STATUS s;

  ObjCallRet( msgXWordClueClueTapStrikeOut,pData->acrossClues,
              NULL, s );
```

```
  ObjCallRet( msgXWordClueClueTapStrikeOut,pData->downClues,
              NULL, s );

  return stsOK;
  MsgHandlerParametersNoWarning;
}
```

The Puzzle Statistics The clsXWordView method XWordViewCheckPuzzle responds to the msgXWordViewCheckPuzzle message by requesting copies of the model's correct data and the grid's user-supplied data, and then comparing the two sets of information in a meaningful manner. The method is defined

```
MsgHandlerWithTypes(XWordViewCheckPuzzle,
                    P_XWORDVIEW_STATS, P_INSTANCE_DATA)
{
  XWORD_DATA      solution[XWORD_MAX_GRID_SIZE];
  GRID_DATA       frGrid[GRID_MAX_GRID_SIZE];
  U32             i, len, cnt, index;
  XWORDDATA_WORD  xdw;
  STATUS          s;

  ObjCallRet(  msgXWordDataGetLetters, pData->model,
               &solution, s );
  ObjCallRet(msgXWordGridGetLetters, pData->grid,&frGrid,s);

  pArgs->letterCount = pArgs->okLetters = 0;
  for ( i=0, len=pData->gridSize; i<len; i++ )
     if ( solution[i] ) {
       pArgs->letterCount++;
       if ( solution[i] == frGrid[i] )
          pArgs->okLetters++;
       }

  pArgs->okWords = 0;
  ObjCallRet(msgXWordDataGetAcrossCount,pData->model,&cnt,s);
  pArgs->wordCount = cnt;
  for ( i=0; i<cnt; i++ ) {
     xdw.index = i;
     ObjCallRet( msgXWordDataGetAcrossWord, pData->model,
                 &xdw, s );
     index = xdw.origin.x + xdw.origin.y*pData->size;
     if (XWVaccStrEqu(&frGrid[index], pData->size,xdw.word))
       pArgs->okWords++;
     }
```

```
    ObjCallRet(msgXWordDataGetDownCount, pData->model,&cnt,s);
    pArgs->wordCount += cnt;
    for ( i=0; i<cnt; i++ ) {
       xdw.index = i;
       ObjCallRet(msgXWordDataGetDownWord,pData->model,&xdw,s);
       index = xdw.origin.x + xdw.origin.y*pData->size;
       if (XWVdwnStrEqu(&frGrid[index], pData->size,xdw.word))
         pArgs->okWords++;
       }
    return stsOK;
    MsgHandlerParametersNoWarning;
}
```

In addition to requesting more information from the model, the XWordViewCheckPuzzle uses two local functions to do the equivalent of !strcmp(). The first function

```
XWVaccStrEqu( P_U8 gStr, U32 size, P_U8 word )
{
  return( !strncmp( gStr, word, strlen(word) ) ? 1 : 0 );
  Unused( size );
}
```

is somewhat redundant. However, it provides a matching function for

```
XWVdwnStrEqu( P_U8 gStr, U32 size, P_U8 word )
{
  U32 len, i;

  for ( i=0, len=strlen(word); i < len; i++ ) {
    if ( *gStr != word[i] )
       break;
    gStr += size;
    }

 return( i == len );
}
```

This function is necessary to navigate the puzzle data which is kept as a flattened array layout of the two-dimensional data.

Visual Feedback on the Grid In addition to the statistics display, the user also has the option of requesting that the grid display its letters in differently shaded fonts indicating which letters are correct and which are incorrect. clsXWordView supports this functionality by implementing two methods that, with one difference, perform work similar to

XWordViewCheckPuzzle. The difference is that instead of returning information to the application class for display to the user, the view sends a message to the grid indicating which letters are correct. It then relies on the grid to do something intelligent with the information.

The first method, XWordViewCheckLetters, responds to the message msgXWordViewCheckLetters and is defined

```
MsgHandlerWithTypes( XWordViewCheckLetters,
                     P_ARGS, P_INSTANCE_DATA)
{
  XWORD_DATA   solution[XWORD_MAX_GRID_SIZE];
  GRID_DATA    frGrid[GRID_MAX_GRID_SIZE],
               toGrid[GRID_MAX_GRID_SIZE];
  U32          i;
  STATUS       s;

  ObjCallRet( msgXWordDataGetLetters, pData->model,
              &solution,s);
 ObjCallRet(msgXWordGridGetLetters,pData->grid,&frGrid,s);

  for ( i=0; i<pData->gridSize; i++ )
     if ( frGrid[i] )
       toGrid[i] = (solution[i] == frGrid[i]);
     else
       toGrid[i] = 0;

ObjCallRet(msgXWordGridSetOkLetters,pData->grid,toGrid,s);

  return stsOK;
  MsgHandlerParametersNoWarning;
}
```

The second method, XWordViewCheckWords, responds to the message msgXWordViewCheckWords and is defined

```
MsgHandlerWithTypes( XWordViewCheckWords,
                     P_ARGS, P_INSTANCE_DATA)
{
  U32            i, j, len, cnt, index;
  GRID_DATA      frGrid[GRID_MAX_GRID_SIZE],
                 toGrid[GRID_MAX_GRID_SIZE];
  XWORDDATA_WORD xdw;
  STATUS         s;

  ObjCallRet(msgXWordGridGetLetters, pData->grid,&frGrid,s);
  memset( toGrid, 0, pData->gridSize * SizeOf(GRID_DATA) );
```

```
ObjCallRet(msgXWordDataGetAcrossCount,pData->model,&cnt,s);
for ( i=0; i<cnt; i++ ) {
   xdw.index = i;
   ObjCallRet( msgXWordDataGetAcrossWord, pData->model,&xdw, s );
   index = xdw.origin.x + xdw.origin.y*pData->size;
   if (XWVaccStrEqu(&frGrid[index],pData->size,xdw.word)){
     len = strlen( xdw.word );
     for ( j=0 ; j<len ; j++ )
        toGrid[index+j] = 1;
     }
   }

ObjCallRet(msgXWordDataGetDownCount,pData>model,&cnt,s);
for ( i=0; i<cnt; i++ ) {
   xdw.index = i;
   ObjCallRet(msgXWordDataGetDownWord,pData->model,&xdw,s);
   index = xdw.origin.x + xdw.origin.y*pData->size;
   if (XWVdwnStrEqu(&frGrid[index],pData->size,xdw.word)){
     len = strlen( xdw.word );
     for ( j=0 ; j<len ; j++ ) {
        toGrid[index] = 1;
        index += pData->size;
        }
     }
   }

  ObjCallRet(msgXWordGridSetOkLetters,pData->grid,toGrid,s);

  return stsOK;


 MsgHandlerParametersNoWarning;
}
```

method.tbl

method.tbl contains the following MSG_INFO structure for mapping messages to methods in clsXWordView:

```
MSG_INFO clsXWordViewMethods[] = {

 msgNewDefaults,"XWordViewNewDefaults",objCallAncestorBefore,
 msgInit,       "XWordViewInit",       objCallAncestorBefore,
 msgSave,       "XWordViewSave",       objCallAncestorBefore,
 msgRestore,    "XWordViewRestore",    objCallAncestorBefore,
```

```
msgCstmLayoutGetChildSpec,"XWordViewCLGetChildSpec",
      objCallAncestorBefore,
msgXWordViewStartPlayOver,    "XWordViewStartPlayOver",      0,
msgXWordViewShowSoln,         "XWordViewShowSoln",           0,
msgXWordViewClueTapNothing,   "XWordViewClueTapNothing",     0,
msgXWordViewClueTapStrikeOut,"XWordViewClueTapStrikeOut",    0,
msgXWordViewCheckPuzzle,      "XWordViewCheckPuzzle",        0,
msgXWordViewCheckLetters,     "XWordViewCheckLetters",       0,
msgXWordViewCheckWords,       "XWordViewCheckWords",         0,
 0
};
```

clsXWordClueList: The Clue List View Class

The clsXWordClueList class is a control-style component that displays a titled, scrollable list of items to the user. The class is a subclass of clsCustomLayout and is constructed by combining a clsListBox object with a clsLabel object. The clue list can be programmatically instructed to respond to a single tap on a list item by toggling as strikeout line through the item. I choose to implement this class as a DLL because I'm sure there will be other times when I want a list with a title on the top.

clsListBox

clsXWordClueList uses an instance of clsListBox to manage the display of clues. In addition to managing the display of items it contains, clsListBox also allows you to register to receive notification when the user performs a gesture on one of the items. clsListBox items also have the ability to associate application-specific data with each object. This information is maintained, but not interpreted, for the object that is using clsListBox.

Incidentally, there are other specialized subclasses of clsListBox for managing lists of strings. They are clsStringListBox and clsFontListBox. At first, clsStringListBox seemed a likely candidate for clsXWordClueList, because it displays a list of strings. Unfortunately, it doesn't allow gesture forwarding, which is used to indicate that the user is tapping on the clue. The second subclass of clsListBox, clsFontListBox, is actually a subclass of clsStringListBox. It automatically fills in its contents from the list of available fonts.

xwrdclue.h

xwrdclue.h is the external interface for the clue list display class clsX-WordClueList. The file begins by checking to make sure the file hasn't already been included

```
#ifndef XWRDCLUE_INCLUDED
#define XWRDCLUE_INCLUDED
```

If this is the first access to the file, the first action taken is to include the interface files for the other components it relies upon, using the statements

```
#ifndef GO_INCLUDED
#include <go.h>
#endif

#ifndef CLSMGR_INCLUDED
#include <clsmgr.h>
#endif

#ifndef CLAYOUT_INCLUDED
#include <label.h>
#endif
```

Following the include directives is the definition of the Well Known ID:

```
#define clsXWordClueList MakeGlobalWKN(4153,1)
```

This identifies the clsXWordClueList class to the PenPoint Class Manager.
Next come the message selectors used to define messages new to clsX-WordClueList:

```
#define msgXWordClueStartPlayOver \
                    MakeMsg( clsXWordClueList, 1 )
#define msgXWordClueClueTapNothing \
                    MakeMsg( clsXWordClueList, 2 )
#define msgXWordClueClueTapStrikeOut \
                    MakeMsg( clsXWordClueList, 3 )
```

Following the message selectors are a set of data structures used during the creation of a new clsXWordClueList object:

```
typedef struct {
  U32                size;
  P_STRING           pTitle;
  OBJECT             clueList;
} XWORDCLUE_NEW_ONLY, *P_XWORDCLUE_NEW_ONLY;
```

```
#define xwordclueNewFields \
  customLayoutNewFields \
  XWORDCLUE_NEW_ONLY xwclue;

typedef struct XWORDCLUE_NEW {
  xwordclueNewFields
} XWORDCLUE_NEW, *P_XWORDCLUE_NEW;
```

Finally, at the end of the file, the statement

```
#endif
```

closes the initial #ifndef clause.

xwrdclue.c

xwrdclue.c contains the actual implementation for the clsXWordClueList class. It begins by including the familiar header files:

```
#ifndef GO_INCLUDED
#include <go.h>
#endif

#ifndef WIN_INCLUDED
#include <win.h>
#endif

#ifndef STROBJ_INCLUDED
#include <strobj.h>
#endif

#ifndef LIST_INCLUDED
#include <list.h>
#endif

#ifndef FS_INCLUDED
#include <fs.h>
#endif

#ifndef CLAYOUT_INCLUDED
#include <clayout.h>
#endif

#ifndef LABEL_INCLUDED
#include <label.h>
#endif
```

```
#ifndef GWIN_INCLUDED
#include <gwin.h>
#endif

#ifndef XGESTURE_INCLUDED
#include <xgesture.h>
#endif

#ifndef DEBUG_INCLUDED
#include <debug.h>
#endif

#include <stdio.h>
#include <string.h>
```

The interface file that describes the external interface to the clsListBox object follows:

```
#ifndef LISTBOX_INCLUDED
#include <listbox.h>
#endif
```

Finally, the interface to the clsXWordClueList class itself and the entries generated by the method compiler are included with the statements

```
#ifndef XWRDCLUE_INCLUDED
#include <xwrdclue.h>
#endif

#include <xclu_mth.h>
```

Notice that a second method table file has been introduced. This is necessary to support placing clsXWordClueList in its own separate Dynamic Link Library.

Component Window Tags clsXWordClueList relies on the Window Manager to save and restore the state of the title and list components it creates. The following tags

```
#define titleWinTag  MakeTag( clsXWordClueList, 1 )
#define listWinTag   MakeTag( clsXWordClueList, 2 )
```

are defined so the window IDs can be located after the windows themselves have been restored.

Instance Variables clsXWordClueList maintains its instance data by using the structure

```
typedef struct INSTANCE_DATA {
  U8      clueTapMode;
  U16     clueCnt;
  OBJECT  titleWin;
  OBJECT  listWin;
} INSTANCE_DATA, *P_INSTANCE_DATA;
```

clsXWordClueList uses clueTapMode to keep track of whether it should monitor user taps on items in the clue list. It uses the definitions

```
#define MODE_NOTHING     0
#define MODE_STRIKEOUT   1
```

to indicate its state.

The rest of the instance variables keep information that is commonly used to process requests. Keeping this information as instance data is a performance consideration; this information could be requested from the listBox object or Window Manager as needed.

DLL Initialization xwrdclue.c contains a standard DLLMain() routine that PenPoint calls when the DLL is loaded into the operating environment. It is defined

```
STATUS EXPORTED DLLMain (void)
{
  STATUSs;

  StsRet(ClsXWordClueListInit(), s);

  return stsOK;
}
```

and is responsible for calling

```
STATUS ClsXWordClueListInit(void)
{
  CLASS_NEW c;
  STATUS    s;

  ObjCallRet(msgNewDefaults, clsClass, &c, s );
  c.object.uid        = clsXWordClueList;
  c.cls.pMsg          = clsXWordClueListTable;
  c.cls.ancestor      = clsCustomLayout;
  c.cls.size          = SizeOf(INSTANCE_DATA);
```

```
    c.cls.newArgsSize   = SizeOf(XWORDCLUE_NEW);
    ObjCallRet(msgNew, clsClass, &c, s );

    return stsOK;
  }
```

Initializing a clsXWordClueList Object The method that responds to the msgInit message for clsXWordClueList is

```
  MsgHandlerArgType(XWordClueInit, P_XWORDCLUE_NEW)
  {
    INSTANCE_DATA inst;
    WIN_METRICS   wm;
    LIST_FREE     lf;
    STATUS        s;

    inst.clueTapMode = MODE_NOTHING;
    ObjCallRet( msgListNumItems, pArgs->xwclue.clueList,
                   &inst.clueCnt, s );

    StsRet( XWCCreateListTitle( pArgs->xwclue.pTitle,
                        titleWinTag, &inst.titleWin ), s );
    StsRet( XWCCreateListBox( self, pArgs->xwclue.clueList,
                        listWinTag, &inst.listWin ), s );

    lf.key = (OBJ_KEY)clsList;
    lf.mode = listFreeItemsAsObjects;
    ObjCallWarn( msgListFree, pArgs->xwclue.clueList, &lf );

    ObjectWrite(self, ctx, &inst );

    wm.parent = self;


    wm.options = wsPosTop;
    ObjCallRet( msgWinInsert, inst.titleWin, &wm, s );
    ObjCallRet( msgWinInsert, inst.listWin, &wm, s );

    return stsOK;
    MsgHandlerParametersNoWarning;
  }
```

This method begins by setting the clueCnt instance variable with the value returned from the message sent asking the list for that information. Next, it creates the title and list objects that comprise a clsXWordClueList object. It then frees the list it was given as part of XWORDCLUE_NEW structure. Finally, it writes the instance data back to protected memory and inserts the new child windows in the window hierarchy.

In the process of creating the list component, XWordClueInit uses the `XWCCreateListTitle()` and `XWCCreateListBox()` functions.

XWCCreateListTitle() is defined

```
STATUS LOCAL
XWCCreateListTitle(P_U8 pTitle,TAG tag, P_OBJECT pTitleWin)
{
  LABEL_NEW         ln;
  STATUS            s;

  ObjCallRet(msgNewDefaults, clsLabel, &ln, s);
  ln.win.tag                    = tag;
  ln.label.style.scaleUnits = bsUnitsFitWindowProper;
  ln.label.style.xAlignment = lsAlignCenter;
  ln.label.pString              = pTitle;
  ln.border.style.edge          = bsEdgeAll;
  ObjCallRet(msgNew, clsLabel, &ln, s);
  *pTitleWin = ln.object.uid;

  return stsOK;
}
```

XWCCreateListBox() is defined

```
STATUS LOCAL
XWCCreateListBox( OBJECT self, OBJECT list, TAG tag,
                  P_OBJECT pListBox)
{
  LIST_BOX_NEW      lbn;
  LIST_BOX_ENTRY    lbe;
  LABEL_NEW         ln;
  LIST_ENTRY        le;
  STATUS            s;
  U16               cnt;
  U32               i;

  ObjCallRet( msgListNumItems, list, &cnt, s );

  ObjCallRet(msgNewDefaults, clsListBox, &lbn, s);
  lbn.win.tag                    = tag;
  lbn.border.style.edge          = bsEdgeAll;
  lbn.listBox.client             = self;
  lbn.listBox.nEntries           = cnt;
  lbn.listBox.nEntriesToView = cnt;
  ObjCallRet(msgNew, clsListBox, &lbn, s);
  *pListBox = lbn.object.uid;
```

```
    memset( &lbe, 0, SizeOf(LIST_BOX_ENTRY) );
    lbe.listBox = *pListBox;
    lbe.freeEntry= lbFreeDataWhenDestroyed;
    for ( i=0; i<cnt; i++ ) {
      lbe.position = le.position = i;
      ObjCallRet( msgListGetItem, list, &le, s );
      ObjCallRet( msgNewDefaults, clsLabel, &ln, s );
      ln.border.style.edge = bsEdgeNone;
    ObjCallRet(msgStrObjGetStr,le.item,&ln.label.pString,s);
      ObjCallRet( msgNew, clsLabel, &ln, s );
      lbe.win = ln.object.uid;
     ObjCallRet(msgListBoxInsertEntry, *pListBox, &lbe, s );
      }

      return stsOK;
}
```

The first part of this function uses the code

```
lbn.listBox.client         = self;
lbn.listBox.nEntries       = cnt;
lbn.listBox.nEntriesToView = cnt;
```

to create a clsListBox object that will create and display all possible entries. Additionally, the clsXWordClueList object being created will be notified of all changes made to the ListBox, including forwarded gestures.

Next, a `LIST_BOX_ENTRY` structure is initialized to describe the type of labels that will be inserted into the ListBox to display the clues. The initialization includes the line

```
lbe.freeEntry = lbFreeDataWhenDestroyed;
```

which instructs the list not to free any objects until the ListBox itself is destroyed. You can also specify that the items be freed when they are no longer visible to the user.

Once the structure is initialized, the input list is traversed, and individual clsLabel objects are created and then inserted into the ListBox for each clue.

Responding to Save and Restore Instances of clsXWordClueList save the current clueCnt and clueTapMode and rely on the Window Manager to save the component windows used to construct the clue list. The method that responds to msgSave is

```
MsgHandlerWithTypes(XWordClueSave,P_OBJ_SAVE,P_INSTANCE_DATA)
{
    STREAM_READ_WRITE   fsWrite;
    STATUS              s;

    fsWrite.numBytes    = SizeOf(U16);
    fsWrite.pBuf        = &(pData->clueCnt);
    ObjCallRet(msgStreamWrite, pArgs->file, &fsWrite, s );

    fsWrite.numBytes    = SizeOf(U8);
    fsWrite.pBuf        = &(pData->clueTapMode);
    ObjCallRet(msgStreamWrite, pArgs->file, &fsWrite, s );

    return stsOK;
    MsgHandlerParametersNoWarning;
}
```

The method that responds to msgRestore is

```
MsgHandlerArgType(XWordClueRestore, P_OBJ_RESTORE)
{
  INSTANCE_DATA       inst;
  LIST_BOX_METRICS    lbm;
  STREAM_READ_WRITE   fsRead;
  STATUS              s;

  fsRead.numBytes     = SizeOf(U16);
  fsRead.pBuf         = &inst.clueCnt;
  ObjCallRet(msgStreamRead, pArgs->file, &fsRead, s );

  fsRead.numBytes     = SizeOf(U8);
  fsRead.pBuf         = &inst.clueTapMode;
  ObjCallRet(msgStreamRead, pArgs->file, &fsRead, s );
  inst.titleWin =
  (WIN)ObjectCall(msgWinFindTag,self,(P_ARGS)titleWinTag);
  inst.listWin =
  (WIN)ObjectCall(msgWinFindTag,self,(P_ARGS)listWinTag);

  ObjectWrite(self, ctx, &inst );

  ObjCallRet( msgListBoxGetMetrics,inst.listWin, &lbm, s );
  lbm.client = self;
  ObjCallRet( msgListBoxSetMetrics,inst.listWin, &lbm, s );

  return stsOK;
  MsgHandlerParametersNoWarning;
}
```

The statements

```
ObjCallRet( msgListBoxGetMetrics, inst.listWin, &lbm, s );
lbm.client = self;
ObjCallRet( msgListBoxSetMetrics, inst.listWin, &lbm, s );
```

are necessary to re-establish the clsXWordClueList parent object as a dependent of the ListBox object after restoring is complete.

Window Layout As a subclass of clsCustomLayout, instances of clsXWordClueList receive notification that a window layout episode has started. clsXWordClueList responds to this notification using the method

```
#define TEXT_SIZE                12

MsgHandlerWithTypes(XWordClueCLGetChildSpec,
          P_CSTM_LAYOUT_CHILD_SPEC, P_INSTANCE_DATA)
{
  if ( pArgs->child == pData->titleWin ) {
     pArgs->metrics.w.constraint   = clSameAs;
     pArgs->metrics.h.constraint   = clAbsolute;
     pArgs->metrics.h.value        = TEXT_SIZE;
     pArgs->metrics.x.constraint   =
          ClAlign(clMinEdge, clSameAs, clMinEdge);
     pArgs->metrics.y.constraint   =
          ClAlign(clMaxEdge, clSameAs, clMaxEdge);
     }
  else if ( pArgs->child == pData->listWin ) {
     pArgs->metrics.w.constraint   = clSameAs;
     pArgs->metrics.h.relWin       = pData->titleWin;
     pArgs->metrics.h.constraint   =
          ClExtend(clSameAs, clMinEdge);
     pArgs->metrics.x.constraint   =
          ClAlign(clMinEdge, clSameAs, clMinEdge);
     pArgs->metrics.y.constraint   =
          ClAlign(clMinEdge, clSameAs, clMinEdge);
     }

  return stsOK;
  MsgHandlerParametersNoWarning;
}
```

Notice that I specify an absolute size for the label. This causes the clue list to take from the space for displaying clues as it gets smaller. Otherwise, you could end up with two small boxes, neither of which are readable.

Controlling Play clsXWordClueList defines several methods that control how the user plays or interacts with the list of clues. They are XWordClueStartPlayOver, XWordClueClueTapNothing, and XWordClueClueTapStrikeOut, which respond to the messages msgXWordClueStartPlayOver, msgXWordClueClueTapNothing, and msgXWordClueClueTapStrikeOut, respectively.

XWordClueStartPlayOver is defined

```
MsgHandlerWithTypes( XWordClueStartPlayOver,
                     P_ARGS, P_INSTANCE_DATA )
{
  LIST_BOX_ENTRY lbe;
  U32            i;
  STATUS         s;

  for ( i=0; i<pData->clueCnt; i++ ) {
    lbe.listBox  = pData->listWin;
    lbe.position = i;
    ObjCallRet(msgListBoxGetEntry, pData->listWin, &lbe,s);
    if ( lbe.data == MODE_STRIKEOUT ) {
       StsRet(XWCSetClueEntryStyle(lbe.win, MODE_NOTHING),s);
       lbe.data = MODE_NOTHING;
       ObjCallRet(msgListBoxSetEntry, pData->listWin,&lbe,s);
       }
    }

  return stsOK;
  MsgHandlerParametersNoWarning;
}
```

XWordClueStartPlayOver goes through each item in the list and removes the strikeout attribute from the displayed item if it's there. It uses the function

```
STATUS LOCAL
XWCSetClueEntryStyle( OBJECT clueEnt, U8 style )
{
  LABEL_STYLE          ls;
  STATUS               s;

  ObjCallRet( msgLabelGetStyle, clueEnt, &ls, s );
  ls.strikeout = (style == MODE_STRIKEOUT) ? 1 : 0;
  ObjCallRet( msgLabelSetStyle, clueEnt, &ls, s );
  ObjCallRet( msgWinDirtyRect, clueEnt, pNull, s );

  return stsOK;
}
```

to remove the strikeout mark. The next two methods also use this function for setting and removing the strikeout attribute.

The XWordClueClueTapNothing method is used to disable the strikeout feature from the clue list. It is implemented by

```
MsgHandlerArgType( XWordClueClueTapNothing, P_ARGS )
{
  INSTANCE_DATA  inst;
  LIST_BOX_ENTRY lbe;
  U32            i;
  STATUS         s;

  inst = IDataDeref( pData, INSTANCE_DATA );
  inst.clueTapMode = MODE_NOTHING;
  ObjectWrite(self, ctx, &inst );

  for ( i=0; i<inst.clueCnt; i++ ) {
     lbe.listBox = inst.listWin;
     lbe.position = i;
     ObjCallRet(msgListBoxGetEntry,inst.listWin,&lbe, s);
     if ( lbe.data == MODE_STRIKEOUT )
       StsRet(XWCSetClueEntryStyle(lbe.win,MODE_NOTHING),s);
     }

  return stsOK;
  MsgHandlerParametersNoWarning;
}
```

Although the strikeout attribute is removed from the displayed Label item, it is not removed from the list item's data. This allows the user to turn off the strikeout features without losing track of the items that have been marked. When the XWordClueClueTapStrikeout method is used, all items that were marked as having a line through them will be restored to that state.

The implementation for the XWordClueClueTapStrikeout method is

```
MsgHandlerArgType( XWordClueClueTapStrikeOut, P_ARGS )
{
  INSTANCE_DATA  inst;
  LIST_BOX_ENTRY lbe;
  U32            i;
  STATUS         s;

  inst = IDataDeref( pData, INSTANCE_DATA );
  inst.clueTapMode = MODE_STRIKEOUT;
  ObjectWrite(self, ctx, &inst );
```

```
    for ( i=0; i<inst.clueCnt; i++ ) {
       lbe.listBox = inst.listWin;
       lbe.position = i;
       ObjCallRet( msgListBoxGetEntry, inst.listWin, &lbe, s);
       if ( lbe.data == MODE_STRIKEOUT )
         StsRet(XWCSetClueEntryStyle( lbe.win,MODE_STRIKEOUT),
                   s);
       }

    return stsOK;
    MsgHandlerParametersNoWarning;
  }
```

Handling Forwarded Gestures clsXWordClueList uses the gesture forwarding feature of clsListBox to tell when the user has tapped on an item in the ListBox. Currently, a tap causes the strikeout state to toggle between On and Off. The code that implements this functionality responds to msgListBoxEntryGesture and is defined

```
  MsgHandlerWithTypes( XWordClueEntryGesture,
                       P_LIST_BOX_ENTRY, P_INSTANCE_DATA)
  {
    STATUS s;

    if ( !( ((P_GWIN_GESTURE)(pArgs->arg))->msg == xgs1Tap ) )
       return stsOK;

    if ( pData->clueTapMode == MODE_NOTHING )
       return stsOK;

    pArgs->data = (P_UNKNOWN)((pArgs->data == MODE_NOTHING)
                              ? MODE_STRIKEOUT : MODE_NOTHING);
    StsRet(XWCSetClueEntryStyle(pArgs->win,(U8)pArgs->data),s);
    ObjCallRet( msgListBoxSetEntry,pArgs->listBox, pArgs,s);

    ObjCallRet( msgWinDirtyRect, pArgs->win, pNull, s );

    return stsOK;
    MsgHandlerParametersNoWarning;
  }
```

The first conditional

```
    if ( !( ((P_GWIN_GESTURE)(pArgs->arg))->msg == xgs1Tap ) )
```

extracts the event from the ListBox entry structure and checks to see if it's a single tap. If it is, the conditional then checks to see if the user wants the tap processed or not.

xclu_mth.tbl

xclu_mth.tbl contains the complete set of message/method mapping structures for the clsXWordClueList DLL and is implemented

```
#ifndef CLSMGR_INCLUDED
#include <clsmgr.h>
#endif

#ifndef CLAYOUT_INCLUDED
#include <clayout.h>
#endif

#ifndef LISTBOX_INCLUDED
#include <listbox.h>
#endif

#ifndef XWRDCLUE_INCLUDED
#include <xwrdclue.h>
#endif

MSG_INFO clsXWordClueListMethods[] = {
  msgInit,      "XWordClueInit",    objCallAncestorBefore,
  msgSave,      "XWordClueSave",    objCallAncestorBefore,
  msgRestore,   "XWordClueRestore", objCallAncestorBefore,
  msgCstmLayoutGetChildSpec, "XWordClueCLGetChildSpec",
          objCallAncestorBefore,
  msgXWordClueStartPlayOver, "XWordClueStartPlayOver", 0,
  msgXWordClueClueTapNothing,"XWordClueClueTapNothing",0,
  msgXWordClueClueTapStrikeOut,"XWordClueClueTapStrikeOut",0,
  msgListBoxEntryGesture,      "XWordClueEntryGesture",   0,
  0
};
CLASS_INFO classInfo[] = {
  "clsXWordClueListTable",clsXWordClueListMethods,0,
  0
};
```

Wrap-up

One powerful advantage of using objects to program is the support they provide for decoupling information. For example, what makes a crossword puzzle view is that it responds to the requests of the crossword application class. It doesn't care if its start over message was generated by a menu selection or a voice command—that's someone else's problem.

DLLs carry this concept even further. Notice that there was no compile time indication that clsXWordView uses functionality from a DLL. clsXWordView methods send messages that aren't bound to the receiving method until runtime anyway, so it makes no difference when the linking of behavior with the request for its usage occurs. This is why it's safe to say that the use of deferred binding in conjunction with DLLs is going to open a brand new market for reusable components for application building.

Finally, although I only profiled Strings and Lists, several other utility classes included in the PenPoint SDK are worth looking into before you start large-scale development. The time you spend learning the component hierarchy will be regained by the time you save reusing the PenPoint components and therefore you should consider it a high priority item.

10

WYSIWYG GUIs

The focus of this chapter is to place PenPoint's concept of WYSIWYG GUIs (What You See Is What You Get Graphical User Interfaces) within the framework of direct manipulation metaphors. Basically, the tightly integrated combination of pen and screen open the door to many possible direct manipulation style metaphors never before produced. For example, suppose I have a program for maintaining a phone list attached to a tablet machine. One possible metaphor for a dialing feature would be to render an old style rotary dial telephone ring onto the screen so the user could place a pen inside a finger circle and proceed to pull the dial around, just as you would dial a real rotary phone with a pen.

In the tradition of saving the best for last, Chapter 10 finishes the crossword puzzle application by presenting the code necessary to render the grid using a PenPoint drawing context. The grid is then used by the crossword puzzle view graph to provide the user with a direct manipulation style object for working the puzzle. This chapter also touches on memory mapped file I/O as one means of saving space on a tablet computer. Finally, I close the chapter and the book with several suggestions for extending the crossword puzzle that would be both fun and informative.

The ImagePoint Imaging Model

ImagePoint is a multiple coordinate system imaging model supported by PenPoint for rendering images onto hardware display devices. It provides a rich model for placing graphical images on a display device (a screen, a plotter, or a printer, for example) that is well beyond the scope of a single chapter in this book. With that in mind, I would like to present some highlights to give you an idea of what's happening behind the screens.

Coordinate Systems

ImagePoint supports a simple two-dimensional coordinate system based upon the concept that any place in the plane can be described as a distance in units from a point designated as the origin. What the units are depends upon the coordinate system in use at the time of the drawing request. To facilitate image rendering, ImagePoint supports several different coordinate systems, each with its own strengths and weaknesses.

Unit Definitions ImagePoint supports five integer-based coordinate systems ranging from direct mapping to hardware pixels to user-defined logical space. Starting from the highest level and moving to the hardware are

- **Logical Unit Coordinates (LUC)**, an abstract set of coordinates ImagePoint uses in rendering primitives on behalf of an application.
- **Logical Window Coordinates (LWC)**, a translated set of pixel coordinates in which the origin of the window is mapped to (0,0).
- **Parent Window Coordinates (PWC)**, a translated set of pixel coordinates in which (0,0) is mapped to the origin of a window's parent.
- **Logical Device Coordinates (LDC)**, a translated set of pixel coordinates in which (0,0) is mapped to the origin (lower left corner) of the display device.
- **Device Units (DU4)**, the physical coordinate system of the hardware device. This coordinate system is fixed and varies from hardware platform to hardware platform. Its abbreviation comes from the fact that many hardware displays implement a fourth-quadrant (0,0 in upper left corner) coordinate system.

Transformations ImagePoint supports coordinate transformation through a set of utility routines. The transformation routines are not limited to transforming a coordinate in one space to its coordinate in another. In addition to that functionality, transformations include scaling, translation, and rotation of coordinates.

Drawing Contexts

Drawing contexts are logical imaging models bound to physical display devices. You create a drawing context (an instance of class clsDrwCtx) and then set its attributes based on the stylistic requirements of your application. You then draw your user interface in the drawing context where ImagePoint performs the operations necessary to translate logical requests into a physical device update.

Drawing contexts provide support for clipping, drawing, image rendering, hit detections, color support, fonts, and several other pieces of functionality. Even though drawing contexts are bound to a particular window, that binding is not an exclusive relationship. Multiple drawing contexts can be shared between multiple windows, usually as a performance consideration.

Graphics Primitives

ImagePoint supports a simple set of drawing primitives for rendering images on the display. For example, there are open figures such as polylines, bezier curves, and arcs. There are also closed figures such as rectangles, polygons, and chords. Each primitive uses the information contained in the drawing context, such as foreground color, background color, and so on when rendering its image onto the display. In addition to primitive drawing operations, ImagePoint also supports rendering images in various formats, such as TIFF, to the screen.

Text

Text support is probably one of the most underrated features of PenPoint's drawing model. GO has adopted outline font technology for use in rendering text to display devices. This allows decent-looking fonts to be rendered at any size and greatly aids in building portable user interfaces. ImagePoint also supports font bitmaps as a performance enhancement for rapid text rendering. In addition, there is automatic font selection based on the closest match to a requested set of attributes.

clsXWordGrid: A Direct Manipulation Crossword Grid

The clsXWordGrid class implements the visual component the user interacts with to work the crossword puzzle. It is a subclass of clsSPaper, from which it inherits its ability to gather user strokes into scribbles that are translated to letters. The translator object built for the clsXWordGrid contains a template that recognizes uppercase A-Z and a straight line. With the straight line, the user can "draw through" or erase a character in the grid.

The grid works in units of blocks that are given the logical coordinate size of 100 units by 100 units. The block is then scaled so that size by size (where size is the number of blocks across or down) can be drawn in the space available to the grid. The grid manages hit detection based on information passed back with the xlist when translation occurs, and can effectively deal with a character string in the horizontal or vertical direction.

The grid manages user feedback by maintaining an attribute field for each displayable block in a memory-mapped file. The file is actually a flattened representation of the two-dimensional grid that stores the information in the grid, one row following the next. Each block has its own attribute field that indicates whether the block should be blacked out or not. Further, if a block is capable of containing a letter, the attribute tracks whether the user has filled one in and, if so, whether it's correct, incorrect, or untested. Feedback is provided by rendering the characters in different shaded fonts; black for correct, dark gray for untested, and light gray for tested and found incorrect.

clsXWordGrid maintains the list of block entries inside a memory-mapped file. The reason for using a memory-mapped file scheme is to reduce memory usage in the tablet machines. Current implementations of PenPoint contain the entire operating system and storage volume inside the tablet's RAM. This means that data that exists in both a file and an internal memory structure is using twice as much memory as it needs to. This isn't a problem with the crossword application, but it might be a significant factor in building a word processor. Either way, it's a useful addition to a programmer's bag of tricks.

The following sections present the files xwrdgrid.h and xwrdgrid.c which implement the interface and implementation of the clsXWordGrid class, respectively. The method.tbl source, including the entries for clsXWordApp, clsXWordView, clsXWordData, and clsXWordGrid, is presented in its entirety near the end of this chapter.

xwrdgrid.h

xwrdgrid.h is the external interface for the crossword puzzle's Grid View class clsXWordGrid. The file begins by checking to make sure that it hasn't been included already:

```
#ifndef XWRDGRID_INCLUDED
#define XWRDGRID_INCLUDED
```

If this is the first access to the file, the first action taken is to include the interface files for the other components it relies upon, using the statement

```
#ifndef GO_INCLUDED
#include <go.h>
#endif

#ifndef CLSMGR_INCLUDED
#include <clsmgr.h>
#endif

#ifndef SPAPER_INCLUDED
#include <spaper.h>
#endif
```

Following the include directives is the definition of the Well Known ID:

```
#define clsXWordGrid     MakeGlobalWKN(4151,1)
```

used to identify the clsXWordGrid class to the PenPoint Class Manager, followed by:

```
STATUS ClsXWordGridInit(void);
```

which the main() routine in clsXWordApp uses to register the clsXWord-Grid class with the Class Manager.

Next come the message selectors used to define messages new to clsX-WordGrid. They are defined

```
#define msgXWordGridStartPlayOver  MakeMsg(clsXWordGrid, 1)
#define msgXWordGridGetLetters     MakeMsg(clsXWordGrid, 2)
#define msgXWordGridSetLetters     MakeMsg(clsXWordGrid, 3)
#define msgXWordGridSetOkLetters   MakeMsg(clsXWordGrid, 4)
```

Following the message selectors are the data structures used to specify the initialization of the crossword puzzle's grid. They are defined

```
#define GRID_MAX_GRID_SIZE 100

typedef U8 GRID_DATA, *P_GRID_DATA;

typedef struct {
```

```
    U8          size;
    GRID_DATA   numbers[GRID_MAX_GRID_SIZE];
    GRID_DATA   template[GRID_MAX_GRID_SIZE];
} XWORDGRID_NEW_ONLY, *P_XWORDGRID_NEW_ONLY;

#define xwordgridNewFields \
  sPaperNewFields \
  XWORDGRID_NEW_ONLY xwgrid;


typedef struct XWORDGRID_NEW {
  xwordgridNewFields
} XWORDGRID_NEW, *P_XWORDGRID_NEW;
```

Finally, at the end of the file, the statement

```
#endif
```

closes the initial #ifndef clause.

xwrdgrid.c

xwrdgrid.c contains the actual implementation for the clsXWordGrid Crossword Puzzle Grid View class.

Include statements xwrdgrid.c begins by including the familiar header files:

```
#ifndef WIN_INCLUDED
#include <win.h>
#endif

#ifndef GEO_INCLUDED
#include <geo.h>
#endif

#ifndef FS_INCLUDED
#include <fs.h>
#endif

#ifndef SPAPER_INCLUDED
#include <spaper.h>
#endif

#ifndef OSHEAP_INCLUDED
#include <osheap.h>
#endif
```

```
#ifndef XLATE_INCLUDED
#include <xlate.h>
#endif

#ifndef XLFILTER_INCLUDED
#include <xlfilter.h>
#endif

#ifndef XTEMPLT_INCLUDED
#include <xtemplt.h>
#endif

#ifndef DEBUG_INCLUDED
#include <debug.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

The header files are followed by the interface file that describes the external interface to the graphics primitives:

```
#ifndef SYSGRAF_INCLUDED
#include <sysgraf.h>
#endif
```

the font primitives:

```
#ifndef SYSFONT_INCLUDED
#include <sysfont.h>
#endif
```

and the fixed-point math primitives:

```
#ifndef GOMATH_INCLUDED
#include <gomath.h>
#endif
```

Finally, the interfaces to the Crossword Puzzle Grid and the entries generated by the method compiler are included with the statements

```
#ifndef XWRDGRID_INCLUDED
#include <xwrdgrid.h>
#endif

#include <method.h>
```

xwrdgrid.c Constants The first defined constant is the name of the file used to hold data for the grid:

```
#define GRID_DATAFILE    "gridDataFile"
```

Next, xwrdgrid.c uses a set of defined values for computing the logical coordinates for the layout of a character block:

```
#define BLOCK_SIZE        100
#define BLOCK_LTR_X_OFF    25
#define BLOCK_LTR_Y_OFF    20
#define BLOCK_NUM_X_OFF     5
#define BLOCK_NUM_Y_OFF     5
```

Finally, the attributes attached to each block are a set of flags that are 'or'ed into the attribute variable. The possible flag values are

```
#define beNull      0x00
#define beBlack     0x01
#define beNumber    0x02
#define beLetter    0x04
#define beRight     0x08
#define beWrong     0x10
```

Instance Variables clsXWordGrid maintains its instance data using two different structures. The first structure is used to describe each entry in the grid:

```
typedef struct GRID_ENTRY {
  U8    number;
  U8    letter;
  U8    status;
} GRID_ENTRY, *P_GRID_ENTRY;
```

The actual instance data is described by the structure

```
typedef struct INSTANCE_DATA {
  U32               size;
  U32               gridSize;
  U32               screenBlockSize;
  SYSDC             gridDC;
  OBJECT            gdFileHandle;
  P_GRID_ENTRY      pEntries;
} INSTANCE_DATA, *P_INSTANCE_DATA;
```

clsXWordGrid uses the size instance variable to keep track of the number of blocks in a row or column and the gridSize instance variable to keep track of the total number of blocks in the grid. screenBlockSize is set whenever the grid window resizes and is used by clsXWordGrid to help calculate hit detection and rectangles that need to be redrawn. Next, the gridDC instance variable holds the ID of the drawing context used by the grid to render itself into its window.

The final two entries are used by clsXWordGrid to maintain the underlying model data for the grid in a memory-mapped file. gdFileHandle identifies the file that contains the data, while pEntries is a memory pointer to the start of the array.

Registration The `ClsXWordGridInit()` function is used by the main() routine in clsXWordApp to register clsXWordGrid with the Class Manager. It is defined

```
STATUS ClsXWordGridInit(void)
{
  CLASS_NEW c;
  STATUS    s;

  ObjCallRet(msgNewDefaults, clsClass, &c, s);
  c.object.uid        = clsXWordGrid;
  c.cls.pMsg          = clsXWordGridTable;
  c.cls.ancestor      = clsSPaper;
  c.cls.size          = SizeOf(INSTANCE_DATA);
  c.cls.newArgsSize   = SizeOf(XWORDGRID_NEW);
  ObjCallRet(msgNew, clsClass, &c, s );

  return stsOK;
}
```

Creating a clsXWordGrid Object A new instance of clsXWordGrid is created by first initializing the `XWORDGRID_NEW` structure by sending msgNewDefaults. The method that responds to this message is

```
MsgHandlerArgType(XWordGridNewDefaults, P_XWORDGRID_NEW)
{
  pArgs->border.style.edge     = bsEdgeAll;
 memset( &(pArgs->xwgrid), 0, SizeOf(XWORDGRID_NEW_ONLY) );

  return stsOK;
  MsgHandlerParametersNoWarning;
}
```

This method zeros out the parameters contained in the XWORDGRID_-NEW portion of the initialization structure, and sets the border style to have an edge all around the grid.

Next, the XWORDGRID_NEW structure is filled out and used as the parameter when msgNew is sent to clsXWordGrid. The method that responds to the msgInit message is

```
MsgHandlerArgType(XWordGridInit, P_XWORDGRID_NEW)
{
  INSTANCE_DATA      inst;
  FS_NEW             fsn;
  STREAM_READ_WRITE  fsWrite;
  GRID_ENTRY         ge[GRID_MAX_GRID_SIZE];
  STATUS             s;
  U32                i;

  StsRet(XWGBuildTranslator(&(pArgs->sPaper.translator)),s);

  pArgs->sPaper.flags &= ~spRuling;
  pArgs->sPaper.flags |= spProx;

  ObjectCallAncestorCtx(ctx);

  inst.size     = pArgs->xwgrid.size;
  inst.gridSize = (inst.size * inst.size);

  ObjCallRet( msgNewDefaults, clsFileHandle, &fsn, s );
  fsn.fs.locator.pPath = GRID_DATAFILE;
  fsn.fs.locator.uid   = theWorkingDir;
  ObjCallRet(msgNew, clsFileHandle, &fsn, s );
  inst.gdFileHandle = fsn.object.uid;

  fsWrite.numBytes = inst.gridSize * SizeOf(GRID_ENTRY);
  memset( ge, 0, fsWrite.numBytes );
  fsWrite.pBuf = ge;
  ObjCallRet(msgStreamWrite, inst.gdFileHandle, &fsWrite,s);
  ObjCallRet( msgFSMemoryMap, inst.gdFileHandle,
               &inst.pEntries, s );

  for ( i=0; i<inst.gridSize; i++ ) {
     if ( !(pArgs->xwgrid.template[i]) )
       inst.pEntries[i].status |= beBlack;
     else
     if (inst.pEntries[i].number = pArgs->xwgrid.numbers[i])
       inst.pEntries[i].status |= beNumber;
     }
```

```
  StsRet( XWGBuildGridDC( &inst.gridDC ), s );

  ObjectWrite(self, ctx, &inst);

  ObjectCall(msgDcSetWindow, inst.gridDC, (P_ARGS)self);

  return stsOK;
  MsgHandlerParametersNoWarning;
}
```

The XWordGridInit method is responsible for initializing the four major parts of the the grid object. If you look ahead to the method.tbl section, you will notice that the ancestor class is not called automatically. Instead, the ancestor is called explicitly so that a translator built using `XWGBuildTranslator()` can be inserted as part of the initialization structure.

The `XWGBuildTranslator()` function is defined

```
STATUS LOCAL
XWGBuildTranslator( P_OBJECT pTranslator )
{
  P_UNKNOWN    pNewTemplate;
  XLATE_NEW    xNewTrans;
  U16          xlateFlags;
  XTM_ARGS     xtmArgs;
  STATUS       s;

  ObjCallRet(msgNewDefaults, clsXText, &xNewTrans, s );

  xtmArgs.xtmType  = xtmTypeCharList;
  xtmArgs.xtmMode  = 0;
  xtmArgs.pXtmData = "ABCDEFGHIJKLMNOPQRSTUVWXYZ-";
  StsRet( XTemplateCompile(&xtmArgs,
                  osProcessHeapId, &pNewTemplate), s);

  xNewTrans.xlate.pTemplate = pNewTemplate;
  xNewTrans.xlate.hwxFlags &=
     ~(xltCaseEnable|xltPunctuationEnable|xltVerticalEnable);

  ObjCallRet(msgNew, clsXText, &xNewTrans, s);

  ObjCallRet(msgXlateGetFlags, xNewTrans.object.uid,
             &xlateFlags, s);
  xlateFlags |= xTemplateVeto | xltSpaceDisable;
  ObjCallRet(msgXlateSetFlags, xNewTrans.object.uid,
(P_ARGS)xlateFlags, s);
```

```
  *pTranslator = xNewTrans.object.uid;

  return stsOK;
}
```

After the translator is built, the ancestor class is given a chance to initialize. This creates the appropriate scribble object and sets up the XWordGrid for receiving translated handwriting.

The next step in the XWordGridInit method is to set the size instance variable from the XWORDGRID_NEW structure and then create and memory map the file that will monitor the grid's contents. File creation and memory mapping are done using

```
fsWrite.numBytes = inst.gridSize * SizeOf(GRID_ENTRY);
memset( ge, 0, fsWrite.numBytes );
fsWrite.pBuf = ge;
ObjCallRet(msgStreamWrite, inst.gdFileHandle, &fsWrite, s);
ObjCallRet( msgFSMemoryMap, inst.gdFileHandle,
              &inst.pEntries, s );
```

The file handle to the memory-mapped file is retained as an instance variable. Once the file is initialized, XWordGridInit uses the template and number data from XWORDGRID_NEW to initialize the grid entries.

The last step before writing the instance data back into protected memory is to call the function XWGBuildDC(), which builds the graphics context that is bound to the window and used to render the grid on the display.

The XWGBuildDC() function is defined

```
STATUS LOCAL
XWGBuildGridDC( P_SYSDC pDC )
{
  SYSDC_NEW        dn;
  SYSDC_FONT_SPEC  fs;
  STATUS           s;

  ObjCallRet(msgNewDefaults, clsSysDrwCtx, &dn, s );
  ObjCallRet(msgNew, clsSysDrwCtx, &dn, s );
  *pDC = dn.object.uid;

  ObjCallWarn(msgDcSetLineThickness, *pDC, (P_ARGS)2);

  fs.id              = 0;
  fs.attr.group      = sysDcGroupUserInput;
  fs.attr.weight     = sysDcWeightNormal;
```

```
  fs.attr.aspect     = sysDcAspectNormal;
  fs.attr.italic     = 0;
  fs.attr.monospaced = 0;
  fs.attr.encoding   = sysDcEncodeGoSystem;
  ObjCallRet(msgDcOpenFont, *pDC, &fs, s );

  return stsOK;
}
```

This function creates a default instance of class clsSysDrwCtx, sets its line thickness to 2, and then opens a font that has been specified by setting the attributes in the `SYSDC_FONT_SPEC` structure.

Freeing Instances of clsXWordGrid The next method is responsible for responding to msgFree to de-allocate any resources allocated by the clsXWordGrid object:

```
MsgHandlerWithTypes(XWordGridFree, P_ARGS, P_INSTANCE_DATA)
{
  STATUS s;

  ObjCallRet(msgFSMemoryMapFree, pData->gdFileHandle,NULL,s);
  ObjCallWarn( msgDestroy, pData->gdFileHandle, NULL );

  ObjCallWarn( msgDestroy, pData->gridDC, NULL );

  return stsOK;
  MsgHandlerParametersNoWarning;
}
```

In this example, it is necessary to unmap the memory-mapped file by sending the file handle the msgFSMemoryFree message. The file handle itself must also be destroyed by sending it the msgDestroy message. Note, however, that this doesn't destroy the file's contents. That won't happen until the puzzle document itself is freed, and all associated files are also freed.

In addition to the file handle, XWordGridFree must also de-allocate the resources used to maintain the drawing context for the grid window.

Saving and Restoring The next method is used by clsXWordGrid to respond to msgSave by filing the size of the grid and the screenSize of a block used to hold a character in the grid:

```
MsgHandlerWithTypes(XWordGridSave,
                          P_OBJ_SAVE, P_INSTANCE_DATA)
{
```

```
STREAM_READ_WRITE fsWrite;
STATUS            s;

  fsWrite.numBytes = SizeOf(U32);
  fsWrite.pBuf     = &(pData->size);
  ObjCallRet(msgStreamWrite, pArgs->file, &fsWrite, s );

  fsWrite.numBytes = SizeOf(U32);
  fsWrite.pBuf     = &(pData->screenBlockSize);
  ObjCallRet(msgStreamWrite, pArgs->file, &fsWrite, s );

  return stsOK;
  MsgHandlerParametersNoWarning;
}
```

The next method is used to restore the state of the grid:

```
MsgHandlerArgType( XWordGridRestore, P_OBJ_RESTORE )
{
  STREAM_READ_WRITE   fsRead;
  INSTANCE_DATA       inst;
  FS_NEW              fsn;
  STATUS              s;

  fsRead.numBytes  = SizeOf(U32);
  fsRead.pBuf      = &inst.size;
  ObjCallRet(msgStreamRead, pArgs->file, &fsRead, s );

  inst.gridSize = inst.size * inst.size;

  fsRead.numBytes = SizeOf(U32);
  fsRead.pBuf     = &inst.screenBlockSize;
  ObjCallRet(msgStreamRead, pArgs->file, &fsRead, s );

  ObjCallRet( msgNewDefaults, clsFileHandle, &fsn, s );
  fsn.fs.locator.pPath  = GRID_DATAFILE;
  fsn.fs.locator.uid    = theWorkingDir;
  ObjCallRet(msgNew, clsFileHandle, &fsn, s );
  inst.gdFileHandle = fsn.object.uid;

  ObjCallRet( msgFSMemoryMap, inst.gdFileHandle,
                &inst.pEntries, s );

  StsRet( XWGBuildGridDC( &inst.gridDC ), s );

  ObjectWrite(self, ctx, &inst);

  ObjCallWarn(msgDcSetWindow, inst.gridDC, (P_ARGS)self);
```

```
    ObjCallWarn( msgSPaperClear, self, NULL );

    return stsOK;
    MsgHandlerParametersNoWarning;
}
```

In addition to reading in the horizontal and vertical size of the grid and the block size, it also computes the number of blocks in the grid (gridSize) and remaps the grid data file that contains the current entry information.

Next it rebuilds the device context using XWGBuildGridDC() and then writes the instance data back into protected memory. Finally, the device context ID is mapped onto the window used to display the grid and the ancestor class behavior managing the scribbles is used to clean up any stray, unprocessed scribbles that remain from the user prior to the page turn.

Rendering the Grid The Window Manager sends the msgWinRepaint message to inform a PenPoint window that it needs repainting. The window processes this message by notifying the Window Manager that it's about ready to begin updating the display. Next it updates the display by sending commands to a drawing context mapped to a window. Finally, when the repainting is done, the window sends the Window Manager a message indicating that the update episode is now over.

The method that responds to msgWinRepaint for clsXWordGrid is defined

```
MsgHandlerWithTypes(XWordGridRepaint,P_ARGS,P_INSTANCE_DATA)
{
    RECT32 r;
    SIZE32 sz;
    STATUS s;

    ObjCallRet(msgWinBeginRepaint, pData->gridDC, pNull, s);

    ObjCallWarn(msgDcIdentity, pData->gridDC, pNull );
    sz.w = sz.h = pData->size*BLOCK_SIZE;
    ObjCallRet(msgDcScaleWorld, pData->gridDC, &sz, s );

    ObjCallRet(msgBorderGetBorderRect, self, &r, s );
    ObjCallRet(msgDcLWCtoLUC_RECT32, pData->gridDC, &r, s );
    ObjCallRet(msgDcClipRect, pData->gridDC, &r, s );

    ObjCallWarn(msgDcFillWindow, pData->gridDC, pNull );
    StsRet( XWGDrawGrid( pData ), s );
    StsRet( XWGDrawTemplate( pData ), s );
```

```
  StsRet( XWGDrawLetters( pData ), s );

  ObjCallRet(msgWinEndRepaint, self, Nil(P_ARGS), s );
  return stsOK;
  MsgHandlerParametersNoWarning;
}
```

The first action this method takes is to set up the translation matrix inside the drawing context so that each block—no matter how many are in the grid—is of size BLOCK_SIZE. From this time on, all drawing operations occur in this logical coordinate space.

The next step is to protect the grid's shadow by asking self for its border rectangle, transforming it into world coordinates, and then using it as the clipping rectangle for all subsequent operations. Once the drawing rectangle is determined, it is cleared and then repainted in several layers. When the repainting is done, the episode is marked closed.

The actual drawing takes place using several local functions responsible for different layers of the rendering process. Although not necessary, I chose to implement drawing this way since rendering using drawing contexts tends to result in very verbose code that's hard to follow and hence hard to debug.

The first function, XWGDrawGrid(), is defined

```
STATUS LOCAL
XWGDrawGrid( P_INSTANCE_DATA pData )
{
  SYSDC_POLYLINE  pl;
  XY32            pnts[2];
  STATUS          s;
  U16             i;
  U32             gridWorldSize;

  ObjCallWarn( msgDcSetForegroundRGB, pData->gridDC,
               (P_ARGS)sysDcRGBBlack );
  gridWorldSize = pData->size * BLOCK_SIZE;

  pl.count      = 2;
  pl.points     = pnts;
  pnts[0].y     = gridWorldSize;
  pnts[1].y     = 0;
  for ( i=BLOCK_SIZE; i<gridWorldSize; i+=BLOCK_SIZE ) {
    pnts[0].x = pnts[1].x = i;
    ObjCallRet( msgDcDrawPolyline, pData->gridDC, &pl, s );
    }
```

```
    pnts[0].x = 0;
    pnts[1].x = gridWorldSize;
    for ( i=BLOCK_SIZE; i<gridWorldSize; i+=BLOCK_SIZE ) {
      pnts[0].y = pnts[1].y = i;
      ObjCallRet( msgDcDrawPolyline, pData->gridDC, &pl, s );
      }

    return stsOK;
}
```

This function sets the foreground color to black, so that the polylines it is about to draw are colored black, and then draws two sets of lines that serve to render the grid.

Next, the function `XWGDrawTemplate()` is defined:

```
STATUS LOCAL
XWGDrawTemplate( P_INSTANCE_DATA pData )
{
    SYSDC_TEXT_OUTPUT tx;
    SYSDC_PATTERN     oldPat;
    U8                c[3];
    U32               x, y;
    SCALE             fontScale;
    RECT32            blackOut;
    P_GRID_ENTRY      pGridEntry;
    STATUS            s;

    ObjCallWarn(msgDcIdentityFont, pData->gridDC, pNull );
    fontScale.x=fontScale.y =FxMakeFixed(((BLOCK_SIZE*1)/4),0);
    ObjCallWarn(msgDcScaleFont, pData->gridDC, &fontScale);

    ObjCallWarn(msgDcSetForegroundRGB, pData->gridDC,
                (P_ARGS)sysDcRGBBlack );
    oldPat = ObjCallWarn( msgDcSetFillPat, pData->gridDC,
                          (P_ARGS)sysDcPat75);
    blackOut.size.w = blackOut.size.h = BLOCK_SIZE;

    pGridEntry = pData->pEntries;

    memset( &tx, 0, sizeof(SYSDC_TEXT_OUTPUT));
    tx.alignChr = sysDcAlignChrTop;
    tx.pText     = c;
    tx.lenText  = 2;
    for( y=0; y<pData->size; y++ ) {
      tx.cp.y= (pData->size - y)*BLOCK_SIZE - BLOCK_NUM_Y_OFF;
      blackOut.origin.y = (pData->size - y - 1)*BLOCK_SIZE;
```

```
        for( x = 0; x<pData->size; x++, pGridEntry++ )
            if ( pGridEntry->status & beBlack ) {
              blackOut.origin.x = x*BLOCK_SIZE;
              ObjCallRet(msgDcDrawRectangle, pData->gridDC,
                          &blackOut, s);
              }
            else if ( pGridEntry->status & beNumber ) {
              sprintf( c, "%2d", pGridEntry->number );
              tx.cp.x = x*BLOCK_SIZE + BLOCK_NUM_X_OFF;
              ObjCallRet(msgDcDrawText, pData->gridDC, &tx, s );
              }
        }

    ObjCallWarn(msgDcSetFillPat,pData->gridDC, (P_ARGS)oldPat);

    return stsOK;
  }
```

This function scales the font to one-quarter the size of a block, sets the fill pattern color to 75 percent foreground color, 25 percent background color, and prepares generic text and block structures.

This method scans the attributes of each entry in the grid data, to determine if the block should be blacked out or drawn with or without a number. When through, this method tries to be nice by setting the fill pattern to what it previously was. This is an unnecessary step, because by convention the drawing context is used only by a single instance of this object, and any method that uses the context follows the assumption that nothing can safely be assumed.

Finally, the `XWGDrawLetters()` function is used to render the text to the screen. It is defined

```
STATUS LOCAL XWGDrawLetters( P_INSTANCE_DATA pData )
{
  SYSDC_TEXT_OUTPUT tx;
  U32               x, y;
  SCALE             fontScale;
  U8                str[2];
  P_GRID_ENTRY      pGridEntry;
  STATUS            s;

  ObjCallWarn(msgDcIdentityFont, pData->gridDC, pNull );
  fontScale.x = fontScale.y =
FxMakeFixed(((BLOCK_SIZE*3)/4),0);
  ObjCallWarn(msgDcScaleFont, pData->gridDC, &fontScale);

  memset( &tx, 0, sizeof(SYSDC_TEXT_OUTPUT));
```

```
    tx.alignChr = sysDcAlignChrBaseline;
    tx.lenText  = 1;
    tx.pText    = str;

    pGridEntry = pData->pEntries;
    for( y=0; y<pData->size; y++ ) {
       tx.cp.y = (pData->size - y -1)*BLOCK_SIZE
                     + BLOCK_LTR_Y_OFF;
       for( x = 0; x<pData->size; x++, pGridEntry++ ) {
            tx.cp.x = x*BLOCK_SIZE + BLOCK_LTR_X_OFF;
            *tx.pText = pGridEntry->letter;
            if ( pGridEntry->status & beWrong ) {
              ObjCallWarn(msgDcSetForegroundRGB, pData->gridDC,
                           (P_ARGS)sysDcRGBGray33 );
              ObjCallRet(msgDcDrawText, pData->gridDC, &tx, s );
              }
            else if ( pGridEntry->status & beRight ) {
              ObjCallWarn(msgDcSetForegroundRGB, pData->gridDC,
                           (P_ARGS)sysDcRGBBlack );
              ObjCallRet(msgDcDrawText, pData->gridDC, &tx, s );
              }
            else if ( pGridEntry->status & beLetter ) {
              ObjCallWarn(msgDcSetForegroundRGB, pData->gridDC,
                                (P_ARGS)sysDcRGBGray66 );
              ObjCallRet(msgDcDrawText, pData->gridDC, &tx, s );
              }
            }
       }
    return stsOK;
  }
```

The function checks the various attribute flags of the letter in question and, based on that information, decides which font to use to render the letter.

Managing User Input Rendering a display window to look like a crossword puzzle is only half the problem that clsXWordGrid solves. The other half is managing user input handwritten on the grid. clsXWordGrid receives a tremendous amount of assistance in this area from its ancestor class clsSPaper.

Handwriting recognition is managed by the ancestor class which sends itself the message msgXlateCompleted when translated writing is available. The XWordGridTransWriting method is used to retrieve the translated list of data, filter that list, and locate the block in which writing began. It is defined

```
MsgHandlerWithTypes( XWordGridTransWriting, P_ARGS,
                     P_INSTANCE_DATA )
{
  STATUS         s;
  XLATE_DATA     xdata;
  X2STRING       x2sData;
  XLIST_ELEMENT  xe;
  XY32           penLoc;
  RECT32         dr;
  U32            index;
  P_U8           pStr;

  xdata.heap = osProcessHeapId;
  ObjCallRet(msgSPaperGetXlateData, self, &xdata, s );

  XList2Text(xdata.pXList);
  XListGet( xdata.pXList, 0, &xe );

  StsRet(

     XWGFindGridPos(pData,
                &(((P_XLATE_BDATA)(xe.pData))->box.origin),
                &penLoc), s );
  ObjCallRet(msgWinDirtyRect, self,
                &(((P_XLATE_BDATA)(xe.pData))->box), s);

  XList2StringLength( xdata.pXList, &x2sData.count );
  StsRet( OSHeapBlockAlloc(osProcessHeapId, x2sData.count,
                           &x2sData.pString), s );
  XList2String(xdata.pXList, &x2sData );
  StsJmp(
     XWGFilterTransData( x2sData.pString, x2sData.count ),
     s, Error);

  index = penLoc.x + penLoc.y * pData->size;
  for ( pStr = x2sData.pString; *pStr; pStr++ ) {
     if ( *pStr == '\n' ) {
       index += pData->size - 1;
       penLoc.y++;
       penLoc.x--;
         }
     else {
        if ( !(pData->pEntries[index].status & beBlack) ) {
           pData->pEntries[index].status &=
                ~(beLetter |beWrong |beRight);
           if ( *pStr == '-' )
```

```
                pData->pEntries[index].letter = 0;
            else {
                pData->pEntries[index].letter = *pStr;
                pData->pEntries[index].status |= beLetter;
                }
            }

        StsJmp( XWGGridPosToRect( pData, &penLoc, &dr ),
                                      s, Error );
        ObjCallJmp( msgWinDirtyRect, self, &dr, s, Error );
        index++;
        penLoc.x++;
        }
      }

  s = stsOK;
Error:
  OSHeapBlockFree(x2sData.pString);
  XListFree(xdata.pXList);

  return s;
  MsgHandlerParametersNoWarning;
}
```

XWordGridTransWriting uses several local functions to help with its responsibilities. The first two are

```
STATUS LOCAL
XWGFindGridPos( P_INSTANCE_DATA pData, P_XY32 pIn,
                            P_XY32 pOut )
{
   pOut->x = pIn->x / pData->screenBlockSize;
   pOut->y = pData->size - pIn->y / pData->screenBlockSize -1;

   return stsOK;
}
```

and

```
STATUS LOCAL
XWGGridPosToRect ( P_INSTANCE_DATA pData, P_XY32 pIn,
                   P_RECT32 pOut )
{
  pOut->origin.x = pIn->x * pData->screenBlockSize;
  pOut->origin.y = (pData->size - pIn->y - 1)
                         * pData->screenBlockSize;
```

```
  pOut->size.w   = pOut->size.h = pData->screenBlockSize;

  return stsOK;
}
```

The other function confirms that the list of translated data exactly matches what this application expects:

```
STATUS LOCAL
XWGFilterTransData( P_U8 pStr, U32 len )
{
  U32  i, j;

  for ( i=0, j=0; i<len; i++ )
    if (  (pStr[i] == '\n' )
        ||(pStr[i] == xltCharUnknownDefault )
        ||(pStr[i] == '-' )
        ||((pStr[i]>='A') && (pStr[i]<='Z')) )
        pStr[j++] = pStr[i];
  pStr[j] = '\0';

  return stsOK;
}
```

In addition to the utility functions already described, handwriting recognition makes use of size knowledge gained as a result of responding to the msgWinSized message with the method

```
MsgHandlerArgType( XWordGridWinSized, P_WIN_METRICS )
{
  INSTANCE_DATA   inst;
  WIN_METRICS     wm;
  STATUS          s;

  ObjCallRet( msgWinGetMetrics, self, &wm, s );

  inst = IDataDeref( pData, INSTANCE_DATA );
  inst.screenBlockSize = wm.bounds.size.w / inst.size;
  ObjectWrite(self, ctx, &inst);

  return stsOK;
  MsgHandlerParametersNoWarning;
}
```

This method then stores the new window extent value divided by the number of blocks in the appropriate instance variable.

Responding to the Outside World clsXWordGrid defines several methods that respond to external messages asking it to get and/or set the state of letter attributes inside the grid. It also responds to a request to start over. The start over message msgXWordGridStartPlayOver request is handled by the method

```
MsgHandlerWithTypes( XWordGridStartPlayOver,
                          P_ARGS, P_INSTANCE_DATA )
{
  U32     i;
  STATUS  s;

  for ( i=0; i<pData->gridSize; i++ )
    if ( !(pData->pEntries[i].status & beBlack) ) {
      pData->pEntries[i].status &=
                         ~( beLetter | beWrong | beRight );
      pData->pEntries[i].letter = '\0';
      }

  ObjCallRet( msgWinDirtyRect, self, pNull, s );

  return stsOK;
  MsgHandlerParametersNoWarning;
}
```

In addition to starting over, the grid can be told to accept an array of items as the solution to the puzzle, and therefore renders them as correct onto the display. The method that accomplishes this is defined

```
MsgHandlerWithTypes( XWordGridSetLetters,
                          P_GRID_DATA, P_INSTANCE_DATA)
{
  U32     i;
  XY32    penLoc;
  RECT32  dr;
  STATUS  s;

  for ( i=0; i<pData->gridSize; i++ )
    if ( pData->pEntries[i].letter = pArgs[i] ) {
      pData->pEntries[i].status |= beLetter | beRight;
      pData->pEntries[i].status &= ~beWrong;
      penLoc.x = i % pData->size;
      penLoc.y = i / pData->size;
      StsRet( XWGGridPosToRect( pData, &penLoc, &dr ), s );
      ObjCallRet( msgWinDirtyRect, self, &dr, s );
      }
```

```
    return stsOK;
    MsgHandlerParametersNoWarning;
}
```

An additional method has the same functionality, but takes as its input a subset of correct letters, not the entire grid:

```
MsgHandlerWithTypes(XWordGridSetOkLetters,
                              P_GRID_DATA, P_INSTANCE_DATA)
{
    U32       i;
    XY32      penLoc;
    RECT32    dr;
    STATUS    s;

    for ( i=0; i<pData->gridSize; i++ )
      if ( pData->pEntries[i].status & beLetter ) {
        pData->pEntries[i].status &= ~( beWrong | beRight );
        pData->pEntries[i].status |=
                          pArgs[i] ? beRight : beWrong;
        penLoc.x = i % pData->size;
        penLoc.y = i / pData->size;
        StsRet( XWGGridPosToRect( pData, &penLoc, &dr ), s );
        ObjCallRet( msgWinDirtyRect, self, &dr, s );
        }

    return stsOK;
    MsgHandlerParametersNoWarning;
}
```

The next method returns a copy of the letters the user has actually filled in. This is used as information when trying to determine which, if any, of the user's letters are correct.

```
MsgHandlerWithTypes( XWordGridGetLetters,
                              P_GRID_DATA, P_INSTANCE_DATA)
{
    U32i;

    for ( i=0; i<pData->gridSize; i++ )
      if ( pData->pEntries[i].status & beLetter )
        pArgs[i] = pData->pEntries[i].letter;
      else
        pArgs[i] = 0;

    return stsOK;
```

```
    MsgHandlerParametersNoWarning;
  }
```

method.tbl

method.tbl contains the following `MSG_INFO` structure for mapping messages to methods in clsXWordGrid:

```
MSG_INFO clsXWordGridMethods[] = {
 msgNewDefaults,                "XWordGridNewDefaults"
    objCallAncestorBefore,
 msgInit,                       "XWordGridInit",
    0,
 msgFree,                       "XWordGridFree",
    objCallAncestorAfter,
 msgSave,                       "XWordGridSave",
    objCallAncestorBefore,
 msgRestore,                    "XWordGridRestore",
    objCallAncestorBefore,
 msgWinRepaint,                 "XWordGridRepaint",
    objCallAncestorBefore,
 msgWinSized,                   "XWordGridWinSized",
    objCallAncestorBefore,
 msgXlateCompleted,             "XWordGridTransWriting",
    objCallAncestorBefore,
 msgXWordGridStartPlayOver,     "XWordGridStartPlayOver",
    0,
 msgXWordGridGetLetters,        "XWordGridGetLetters",
    0,
 msgXWordGridSetLetters,        "XWordGridSetLetters",
    0,
 msgXWordGridSetOkLetters,      "XWordGridSetOkLetters",
    0,
 0

};
```

The Complete method.tbl File

The complete definition of the crossword puzzle application's method.tbl file is

```
#ifndef CLSMGR_INCLUDED
#include <clsmgr.h>
#endif
```

```
#ifndef APP_INCLUDED
#include <app.h>
#endif

#ifndef WIN_INCLUDED
#include <win.h>
#endif

#ifndef IMPORT_INCLUDED
#include <import.h>
#endif

#ifndef CLAYOUT_INCLUDED
#include <clayout.h>
#endif

#ifndef XLATE_INCLUDED
#include <xlate.h>
#endif

#ifndef XWORDAPP_INCLUDED
#include <xwordapp.h>
#endif

#ifndef XWRDDATA_INCLUDED
#include <xwrddata.h>
#endif

#ifndef XWRDVIEW_INCLUDED
#include <xwrdview.h>
#endif

#ifndef XWRDGRID_INCLUDED
#include <xwrdgrid.h>
#endif

MSG_INFO clsXWordAppMethods[] = {
 msgImportQuery,                "XWordAppImportQuery",
    objClassMessage,
 msgImport,                     "XWordAppImport",
    0,
 msgAppInit,                    "XWordAppAppInit",
    objCallAncestorBefore,
 msgRestore,                    "XWordAppRestore",
    objCallAncestorBefore,
```

```
 msgXWordAppStartOver,          "XWordAppStartOver",
    0,
 msgXWordAppShowSoln,           "XWordAppShowSoln",
    0,
 msgXWordAppSetClueTap,         "XWordAppSetClueTap",
    0,
 msgXWordAppDoCheck,            "XWordAppDoCheck",
    0,
 0
};

MSG_INFO clsXWordDataMethods[] = {
 msgNewDefaults,                "XWordDataNewDefaults",
    objCallAncestorBefore,
 msgInit,                       "XWordDataInit",
    objCallAncestorBefore,
 msgFree,                       "XWordDataFree",
    objCallAncestorAfter,
 msgSave,                       "XWordDataSave",
    objCallAncestorBefore,
 msgRestore,                    "XWordDataRestore",
    objCallAncestorBefore,
 msgXWordDataIsXWordFile,       "XWordDataIsXWordFile",
    objClassMessage,
 msgXWordDataGetInfo,           "XWordDataGetInfo",
    0,
 msgXWordDataGetLetters,        "XWordDataGetLetters",
    0,
 msgXWordDataGetAcrossCount,    "XWordDataGetAcrossCount",
    0,
 msgXWordDataGetDownCount,      "XWordDataGetDownCount",
    0,
 msgXWordDataGetAcrossWord,     "XWordDataGetAcrossWord",
    0,
 msgXWordDataGetDownWord,       "XWordDataGetDownWord",
    0,
 0
};

MSG_INFO clsXWordViewMethods[] = {
 msgNewDefaults,                "XWordViewNewDefaults",
    objCallAncestorBefore,
 msgInit,                       "XWordViewInit",
    objCallAncestorBefore,
 msgSave,                       "XWordViewSave",
    objCallAncestorBefore,
 msgRestore,                    "XWordViewRestore",
    objCallAncestorBefore,
```

```
 msgCstmLayoutGetChildSpec,    "XWordViewCLGetChildSpec",
    objCallAncestorBefore,
 msgXWordViewStartPlayOver,    "XWordViewStartPlayOver",
    0,
 msgXWordViewShowSoln,         "XWordViewShowSoln",
    0,
 msgXWordViewClueTapNothing,   "XWordViewClueTapNothing",
    0,
 msgXWordViewClueTapStrikeOut,"XWordViewClueTapStrikeOut",
    0,
 msgXWordViewCheckPuzzle,      "XWordViewCheckPuzzle",
    0,
 msgXWordViewCheckLetters,     "XWordViewCheckLetters",
    0,
 msgXWordViewCheckWords,       "XWordViewCheckWords",
    0,
 0
};

MSG_INFO clsXWordGridMethods[] = {
 msgNewDefaults,              "XWordGridNewDefaults",
    objCallAncestorBefore,
 msgInit,                     "XWordGridInit",
    0,
 msgFree,                     "XWordGridFree",
    objCallAncestorAfter,
 msgSave,                     "XWordGridSave",
    objCallAncestorBefore,
 msgRestore,                  "XWordGridRestore",
    objCallAncestorBefore,
 msgWinRepaint,               "XWordGridRepaint",
    objCallAncestorBefore,
 msgWinSized,                 "XWordGridWinSized",
    objCallAncestorBefore,
 msgXlateCompleted,           "XWordGridTransWriting",
    objCallAncestorBefore,
 msgXWordGridStartPlayOver,   "XWordGridStartPlayOver",
    0,
 msgXWordGridGetLetters,      "XWordGridGetLetters",
    0,
 msgXWordGridSetLetters,      "XWordGridSetLetters",
    0,
 msgXWordGridSetOkLetters,    "XWordGridSetOkLetters",
    0,
 0
};
```

```
CLASS_INFO classInfo[] = {
 "clsXWordAppTable",           clsXWordAppMethods,
    0,
 "clsXWordDataTable",          clsXWordDataMethods,
    0,
 "clsXWordViewTable",          clsXWordViewMethods,
    0,
 "clsXWordGridTable",          clsXWordGridMethods,
    0,
 0
};
```

Wrap-up

I believe that the only way to truly learn something is to experience it firsthand. If you have been following the examples and building the sample programs you have started on that journey, but you still have a way to go. Now is a good time to branch out and explore other areas of the PenPoint API by extending the crossword puzzle application. With that in mind, I offer you a short list of possible enhancements to the crossword puzzle application.

- Complete the clsXWordView component so it responds to receiving a new model without being destroyed.
- Add clients to clsXWordClue and clsXWordGrid.
- Have clsXWordGrid notify its observers when a user has entered a letter or completed a word. This behavior could be tied to clsXWordView so that clsXWordView performs an accuracy check automatically and then sends the results back to the grid.
- Put back the Edit menu and support Undo, Cut, Paste.
- Be brave—support spell checking.
- Add help to the document.
- Add a create mode in which the user would enter letters on the grid and then select a menu command to generate clue numbers automatically. Then, while in build mode, the user could tap on a clue to bring up an insertion pad to enter the text for the clue.
- And so on.

Appendix A
Background Reading

Being able to effectively work with PenPoint requires experience in C programming, object-oriented programming, graphical user interface design, and small systems understanding. There are many good books written on the topics of small systems, graphical user interfaces, and C programming, and I urge you to seek them out.

To learn more about object-oriented programming in general, and how it's implemented in PenPoint, you might consider reading the following books.

Beck, K., W. Cunningham. "A Diagram for Object-Oriented Programs." Proceedings of OOPSLA '86: 361-367.

Carr, R., D. Shafer. *The Power of PenPoint.* Reading, MA: Addison-Wesley, 1991.

Cox, B., A. Novobilski. *Object-Oriented Programming: An Evolutionary Approach, Second Edition.* Reading, MA: Addison-Wesley, 1991.

Jacobson, I. "Object-Oriented Development in an Industrial Environment." Proceedings of OOPSLA '87: 183-191. ACM Press.

Krasner, G., S. Pope (1988). "A Cookbook Approach for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80." Journal of Object-Oriented Programming, 1(3).

Novobilski, A. (1992). "NeXTstep and Me." Object Magazine 1(4).

Wirfs-Brock, R., R. Johnson (1990). "Surveying Current Research in Object-Oriented Design." Communications of the ACM, 33(9), 104-124.

Additionally, Addison-Wesley publishes the GO Technical Library which consists of the following titles.

- *PenPoint Application Writing Guide* (1992)
- *PenPoint User Interface Design Reference* (1992)
- *PenPoint Development Tools* (1992)
- *PenPoint Architectural Reference, Volumes I and II* (1992)
- *PenPoint Application Programming Interface Reference, Volumes I and II* (1992)

Appendix B

Source Code for Crossword Application

This appendix contains the complete source listings for the crossword puzzle application discussed in Chapters 8, 9, and 10.

makefile

```
!ifdef %PENPOINT_PATH
PENPOINT_PATH = $(%PENPOINT_PATH)
!else
PENPOINT_PATH = d:\penpoint
!endif

#  The DOS name of your project directory
PROJ = xwordapp

#  Standard defines for sample code
!INCLUDE $(PENPOINT_PATH)\sdk\sample\sdefines.mif

#  The PenPoint name of your application
EXE_NAME = Crossword Puzzle

#  The linker name for your executable : company-name-V<major>(<minor>)
EXE_LNAME   = pip-xwordapp-v1(0)

#  Object files needed to build your app
EXE_OBJS =  method.obj xwordapp.obj xwrddata.obj xwrdview.obj xwrdgrid.obj
```

```
#  Libs needed to build your app
EXE_LIBS = $(DLL_NAME) penpoint app xtemplt xlist

EXE_DLC = xwordapp.dlc

DLL_LNAME = pip-xwrdclue-v1(0)

DLL_OBJS = xclu_mth.obj xwrdclue.obj

DLL_LIBS = penpoint

# Targets

all: $(APP_DIR)\$(PROJ).exe $(APP_DIR)\$(PROJ).dll .SYMBOLIC

#  The clean rule must be :: because it is also defined in srules
clean :: .SYMBOLIC
 -del method.h
 -del xclue_mth.h
 -del xwordapp.lib

#  Dependencies
xwordapp.obj:   xwordapp.c method.h xwordapp.h xwrddata.h xwrdview.h
xwrdgrid.h

xwrddata.obj: xwrddata.c method.h xwrddata.h

xwrdview.obj: xwrdview.c method.h xwrdview.h xwrddata.h xwrdgrid.h
xwrdclue.h

xwrdgrid.obj: xwrdgrid.c method.h xwrdgrid.h

xwrdclue.obj: xwrdclue.c xclu_mth.h xwrdclue.h

#  Standard rules for sample code
!INCLUDE $(PENPOINT_PATH)\sdk\sample\srules.mif
```

xwordapp.dlc

pip-xwrdclue-v1(0)	xwordapp.dll
pip-xwordapp-v1(0)	xwordapp.exe

xwordapp.h

```
#ifndef XWORDAPP_INCLUDED
#define XWORDAPP_INCLUDED

#ifndef GO_INCLUDED
#include <go.h>
#endif

#ifndef CLSMGR_INCLUDED
#include <clsmgr.h>
#endif

#define clsXWordApp MakeGlobalWKN( 4149, 1 )

#define msgXWordAppStartOver    MakeMsg( clsXWordApp, 1 )
#define msgXWordAppShowSoln     MakeMsg( clsXWordApp, 2 )
#define msgXWordAppSetClueTap   MakeMsg( clsXWordApp, 3 )
#define msgXWordAppDoCheck      MakeMsg( clsXWordApp, 4 )

#endif
```

xwordapp.c

```
#ifndef APP_INCLUDED
#include <app.h>
#endif

#ifndef APPTAG_INCLUDED
#include <apptag.h>
#endif

#ifndef APPMGR_INCLUDED
#include <appmgr.h>
#endif

#ifndef FRAME_INCLUDED
#include <frame.h>
#endif

#ifndef FS_INCLUDED
#include <fs.h>
#endif

#ifndef RESFILE_INCLUDED
#include <resfile.h>
#endif

#ifndef IMPORT_INCLUDED
#include <import.h>
#endif

#ifndef TKTABLE_INCLUDED
#include <tktable.h>
#endif

#ifndef MENU_INCLUDED
#include <menu.h>
#endif

#ifndef NOTE_INCLUDED
#include <note.h>
#endif
```

```
#ifndef XWORDAPP_INCLUDED
#include <xwordapp.h>
#endif

#ifndef XWRDVIEW_INCLUDED
#include <xwrdview.h>
#endif

#ifndef XWRDGRID_INCLUDED
#include <xwrdgrid.h>
#endif

#ifndef XWRDDATA_INCLUDED
#include <xwrddata.h>
#endif

#ifndef DEBUG_INCLUDED
#include <debug.h>
#endif

#include <method.h>
#include <string.h>
#include <stdio.h>

typedef struct INSTANCE_DATA {
    OBJECT xwView;
} INSTANCE_DATA, *P_INSTANCE_DATA;

#define tagXWordMenuPuzzle  MakeTag( clsXWordApp, 1 )
#define tagClueTapMenu      MakeTag( clsXWordApp, 2 )

#define mnStartOverTag  MakeTag( clsXWordApp, 3 )
#define mnShowSolnTag   MakeTag( clsXWordApp, 4 )

#define mnNothingTag    MakeTag( clsXWordApp, 5 )
#define mnStrikeOutTag  MakeTag( clsXWordApp, 6 )

#define mnPuzzleTag     MakeTag( clsXWordApp, 7 )
#define mnWordsTag      MakeTag( clsXWordApp, 8 )
#define mnLettersTag    MakeTag( clsXWordApp, 9 )
```

```
static TK_TABLE_ENTRY XWordAppMenuBar[] = {
    {"Puzzle", 0, 0, tagXWordMenuPuzzle, tkMenuPullDown, clsMenuButton},
        {"Start Over", msgXWordAppStartOver, mnStartOverTag },
        {"Show Solution", msgXWordAppShowSoln, mnShowSolnTag,
                              0, tkBorderEdgeBottom},
        {"Tapping Clue", 0, 0, 0, tkMenuPullRight},
            { 0, 0, 0, tagClueTapMenu, 0, clsChoice },
               {"Does Nothing",msgXWordAppSetClueTap,
                  mnNothingTag,mnNothingTag, tkButtonOn},
                  {"Strikes It Out",msgXWordAppSetClueTap, mnStrikeOutTag},
                  {pNull},
            {pNull},
        {"Check", 0, 0, 0, tkMenuPullRight},
                {"Puzzle ...",  msgXWordAppDoCheck, mnPuzzleTag},
                {"Words",       msgXWordAppDoCheck, mnWordsTag},
                {"Letters",     msgXWordAppDoCheck, mnLettersTag},
                {pNull},
        {pNull},
    {pNull}
};

static U32 removeMenuTags[] = {
    tagAppMenuCheckpoint,
    tagAppMenuRevert,
    tagAppMenuEdit,
    0
};

STATUS LOCAL XWABuildMenus(OBJECT self, P_OBJECT pMenuWin)
{
    MENU_NEW    mn;
    OBJECT      w;
    STATUS      s;
    U16         i;

    ObjCallRet(msgNewDefaults, clsMenu, &mn, s );
    mn.tkTable.client   = self;
    mn.tkTable.pEntries = XWordAppMenuBar;
    ObjCallRet(msgNew, clsMenu, &mn, s );
    ObjCallRet(msgAppCreateMenuBar, self, &mn.object.uid, s );
```

```
    *pMenuWin = mn.object.uid;
    for( i=0; removeMenuTags[i]; i++ ) {
        w = (WIN)ObjectCall(msgWinFindTag, *pMenuWin,
                                    (P_ARGS)removeMenuTags[i] );
        ObjCallWarn( msgTkTableRemove, *pMenuWin, (P_ARGS)w );
        }

    return stsOK;
}

static U8 twBuff[25], cwBuff[25], tlBuff[25], clBuff[25];

static TK_TABLE_ENTRY ChkPuzzleTb[] = {
    { twBuff, 0, 0, 0, 0, clsLabel },
    { cwBuff, 0, 0, 0, 0, clsLabel },
    {" ",     0, 0, 0, 0, clsLabel },
    { tlBuff, 0, 0, 0, 0, clsLabel },
    { clBuff, 0, 0, 0, 0, clsLabel },
    {pNull}
};

static TK_TABLE_ENTRY ChkPuzzleCmdBar[] = {
    {"OK", 0, 0, 0, 0, clsButton},
    {pNull}
};


STATUS LOCAL XWAShowCheckPuzzleStats( P_INSTANCE_DATA pData )
{
    U32             aMsg;
    NOTE_NEW        nn;
    XWORDVIEW_STATS xvs;
    STATUS          s;

    ObjCallRet( msgXWordViewCheckPuzzle, pData->xwView, &xvs, s );

    sprintf( twBuff, "%3d - Total Words", xvs.wordCount );
    sprintf( cwBuff, "%3d - Correct Words", xvs.okWords );
    sprintf( tlBuff, "%3d - Total Letters", xvs.letterCount );
    sprintf( clBuff, "%3d - Correct Letters", xvs.okLetters );
```

```
    ObjCallRet( msgNewDefaults, clsNote, &nn, s );
    nn.note.metrics.flags   = nfSystemModal | nfUnformattedTitle;
    nn.note.pTitle          = "Checking the puzzle reveals ...";
    nn.note.pContentEntries = ChkPuzzleTb;
    nn.note.pCmdBarEntries  = ChkPuzzleCmdBar;
    ObjCallRet( msgNew, clsNote, &nn, s );

    ObjCallRet( msgNoteShow, nn.object.uid, (P_ARGS)&aMsg, s );

    ObjCallWarn( msgDestroy, nn.object.uid, pNull );

    return stsOK;
}


MsgHandlerArgType(XWordAppImportQuery, P_IMPORT_QUERY)
{
    if ( ObjectCall(msgXWordDataIsXWordFile, clsXWordData, pArgs->file)
         == stsOK ) {
        pArgs->canImport          = true;
        pArgs->suitabilityRating  = 100;
        }
    return stsOK;
    MsgHandlerParametersNoWarning;
}


MsgHandlerWithTypes(XWordAppImport, P_IMPORT_DOC, P_INSTANCE_DATA)
{
    INSTANCE_DATA   inst;
    APP_METRICS     am;
    XWORDDATA_NEW   xwn;
    XWORDVIEW_NEW   vn;
    OBJECT          oldView;
    STATUS          s;

    inst = IDataDeref( pData, INSTANCE_DATA );
    oldView = inst.xwView;

    ObjCallRet(msgNewDefaults, clsXWordData, &xwn, s );
```

```
    xwn.xword.file = pArgs->file;
    ObjCallRet(msgNew, clsXWordData, &xwn, s );

    ObjCallRet(msgNewDefaults, clsXWordView, &vn, s );
    vn.view.dataObject = xwn.object.uid;
    ObjCallRet(msgNew, clsXWordView, &vn, s );

    inst.xwView = vn.object.uid;
    ObjectWrite( self, ctx, &inst);

    ObjCallRet(msgAppGetMetrics, self, &am, s );
    ObjCallRet(msgFrameSetClientWin, am.mainWin, inst.xwView, s);

    ObjCallWarn( msgDestroy, oldView, NULL );

    return stsOK;
    MsgHandlerParametersNoWarning;
}


MsgHandler(XWordAppAppInit)
{
    INSTANCE_DATA   inst;
    XWORDVIEW_NEW   vn;
    APP_METRICS     am;
    OBJECT          mWin;
    STATUS          s;

    ObjCallRet(msgNewDefaults, clsXWordView, &vn, s );
    ObjCallRet(msgNew, clsXWordView, &vn, s);

    inst.xwView = vn.object.uid;
    ObjectWrite( self, ctx, &inst);

    ObjCallRet(msgAppGetMetrics, self, &am, s );
    ObjCallRet(msgFrameSetClientWin, am.mainWin, inst.xwView, s );

    XWABuildMenus( self, &mWin );
    ObjCallRet(msgFrameSetMenuBar, am.mainWin, mWin, s );

    return stsOK;
```

```
    MsgHandlerParametersNoWarning;
}


MsgHandlerArgType(XWordAppRestore, P_OBJ_RESTORE )
{
    INSTANCE_DATA   inst;
    APP_METRICS     am;
    STATUS          s;

    ObjCallRet(msgAppGetMetrics, self, &am, s);
    ObjCallRet(msgFrameGetClientWin, am.mainWin, &inst.xwView, s );
    ObjectWrite( self, ctx, &inst);

    return stsOK;
    MsgHandlerParametersNoWarning;
}


MsgHandlerWithTypes(XWordAppStartOver, P_ARGS, P_INSTANCE_DATA )
{
    return ObjCallWarn(  msgXWordViewStartPlayOver, pData->xwView, NULL );
    MsgHandlerParametersNoWarning;
}


MsgHandlerWithTypes(XWordAppShowSoln, P_ARGS, P_INSTANCE_DATA )
{
    return ObjCallWarn(  msgXWordViewShowSoln, pData->xwView, NULL );
    MsgHandlerParametersNoWarning;
}


MsgHandlerWithTypes(XWordAppSetClueTap, P_ARGS, P_INSTANCE_DATA )
{
    STATUS s;

    switch( (U32)pArgs ) {
        case mnNothingTag:
            ObjCallRet( msgXWordViewClueTapNothing, pData->xwView, NULL, s);
            break;
```

```
        case mnStrikeOutTag:
            ObjCallRet(msgXWordViewClueTapStrikeOut,pData->xwView,NULL, s);
            break;
        }

    return stsOK;
    MsgHandlerParametersNoWarning;
}


MsgHandlerWithTypes(XWordAppDoCheck, P_ARGS, P_INSTANCE_DATA )
{
    STATUS s;

    switch( (U32)pArgs ) {
        case mnPuzzleTag:
            StsRet( XWAShowCheckPuzzleStats( pData ), s );
            break;

        case mnWordsTag:
            ObjCallRet(msgXWordViewCheckWords, pData->xwView, NULL, s );
            break;

        case mnLettersTag:
            ObjCallRet(msgXWordViewCheckLetters, pData->xwView, NULL, s );
            break;
        }
    return stsOK;
    MsgHandlerParametersNoWarning;
}


STATUS ClsXWordAppInit (void)
{
    APP_MGR_NEW new;
    STATUS      s;

    ObjCallRet(msgNewDefaults, clsAppMgr, &new, s );

    new.object.uid      = clsXWordApp;
```

```
    new.cls.pMsg        = clsXWordAppTable;
    new.cls.ancestor    = clsApp;
    new.cls.size        = SizeOf(INSTANCE_DATA);
    new.cls.newArgsSize = SizeOf(APP_NEW);

    new.appMgr.flags.accessory = FALSE;

    strcpy(new.appMgr.name, "Crossword Puzzle");
    strcpy(new.appMgr.company, "Programming in Penpoint");

    ObjCallRet(msgNew, clsAppMgr, &new, s );

    return stsOK;
}


void CDECL
main(
    int         argc,
    char *      argv[],
    U16         processCount)
{
    if (processCount == 0) {
        ClsXWordAppInit();
        ClsXWordDataInit();
        ClsXWordViewInit();
        ClsXWordGridInit();
        AppMonitorMain(clsXWordApp, objNull);
        }
    else
        AppMain();

    Unused(argc); Unused(argv);
}
```

xwrddata.h

```
#ifndef XWRDDATA_INCLUDED
#define XWRDDATA_INCLUDED

#ifndef CLSMGR_INCLUDED
#include <clsmgr.h>
#endif

#ifndef GEO_INCLUDED
#include <geo.h>
#endif

#define clsXWordData MakeGlobalWKN( 4152, 1)

#define msgXWordDataIsXWordFile      MakeMsg( clsXWordData, 1 )
#define msgXWordDataGetInfo          MakeMsg( clsXWordData, 2 )
#define msgXWordDataGetLetters       MakeMsg( clsXWordData, 3 )
#define msgXWordDataGetAcrossCount   MakeMsg( clsXWordData, 4 )
#define msgXWordDataGetDownCount     MakeMsg( clsXWordData, 5 )
#define msgXWordDataGetAcrossWord    MakeMsg( clsXWordData, 6 )
#define msgXWordDataGetDownWord      MakeMsg( clsXWordData, 7 )

STATUS ClsXWordDataInit (void);

typedef struct XWORDDATA_NEW_ONLY {
    FILE_HANDLE file;
    U32         size;
} XWORDDATA_NEW_ONLY, *P_XWORDDATA_NEW_ONLY;


#define xworddataNewFields \
    objectNewFields \
    XWORDDATA_NEW_ONLY xword;

typedef struct XWORDDATA_NEW {
    xworddataNewFields
} XWORDDATA_NEW, *P_XWORDDATA_NEW;

typedef U8 XWORD_DATA, *P_XWORD_DATA;
```

```
#define XWORD_MAX_WORD_SIZE  10
#define XWORD_MAX_CLUE_SIZE  40
#define XWORD_MAX_GRID_SIZE 100

typedef struct XWORDDATA_LETTER {
    U32 x;
    U32 y;
    U8  letter;
} XWORDDATA_LETTER, *P_XWORDDATA_LETTER;

typedef struct XWORDDATA_WORD {
    U32     index;
    XY32    origin;
    U8      word[XWORD_MAX_WORD_SIZE+1];
} XWORDDATA_WORD, *P_XWORDDATA_WORD;

typedef struct XWORDDATA_INFO {
    U32         size;
    XWORD_DATA  template[XWORD_MAX_GRID_SIZE];
    XWORD_DATA  numbers[XWORD_MAX_GRID_SIZE];
    OBJECT      acrossClues;
    OBJECT      downClues;
} XWORDDATA_INFO, *P_XWORDDATA_INFO;

#endif
```

xwrddata.c

```
#ifndef CLSMGR_INCLUDED
#include <clsmgr.h>
#endif

#ifndef STROBJ_INCLUDED
#include <strobj.h>
#endif

#ifndef LIST_INCLUDED
#include <list.h>
#endif

#ifndef FS_INCLUDED
#include <fs.h>
#endif

#ifndef OSHEAP_INCLUDED
#include <osheap.h>
#endif

#ifndef XWRDDATA_INCLUDED
#include <xwrddata.h>
#endif

#ifndef DEBUG_INCLUDED
#include <debug.h>
#endif

#include "method.h"
#include "string.h"
#include "stdio.h"

/*
 * Instance Variable Definitions
 *
 */

typedef struct XWORD_ENTRY {
    U32     number;
```

```
    U32     x, y;
    U8      word[XWORD_MAX_WORD_SIZE+1];
    U8      clue[XWORD_MAX_CLUE_SIZE+1];
} XWORD_ENTRY, *P_XWORD_ENTRY;

#define MAX_INPUT_REC_SIZE (XWORD_MAX_WORD_SIZE+XWORD_MAX_CLUE_SIZE+20)

typedef struct METRICS {
    U32 size,
        gridSize,
        acrossCnt,
        downCnt;
    U8  grid[XWORD_MAX_GRID_SIZE];
} METRICS, *P_METRICS;

typedef struct INSTANCE_DATA {
    METRICS         metrics;
    P_XWORD_ENTRY   pEntries;
} INSTANCE_DATA, *P_INSTANCE_DATA;


MsgHandlerArgType(XWordDataNewDefaults, P_XWORDDATA_NEW)
{
    memset( &(pArgs->xword), 0, SizeOf(XWORDDATA_NEW_ONLY) );

    return stsOK;
    MsgHandlerParametersNoWarning;
}


static P_U8 getData( FILE *fp )
{
    static U8   buff[MAX_INPUT_REC_SIZE];

    return fgets( buff, MAX_INPUT_REC_SIZE, fp );
}


#define XWORDDATA_LINE_1 "pip-xwordpuzzle"
```

```
MsgHandlerArgType(XWordDataIsXWordFile, FILE_HANDLE)
{
     FILE        *fp;
     STATUS      s;

     fp = StdioStreamBind( pArgs );

     if ( !strncmp(getData(fp), XWORDDATA_LINE_1,strlen(XWORDDATA_LINE_1)) )
        s = stsOK;
     else
        s = stsFailed;

     StdioStreamUnbind( fp );

     return s;
     MsgHandlerParametersNoWarning;
}


STATUS LOCAL XWDBuildXWordFromFile(P_INSTANCE_DATA pData,FILE_HANDLE file )
{
    U32                i1, j, len;
    P_XWORD_ENTRY      pEnt;
    P_METRICS          pMet;
    P_U8               pGrid;
    U32                entSize;
    FILE               *fp;
    STATUS             s;

    fp = StdioStreamBind( file );

    getData( fp );  // ignore first line (importable check)

    pMet = &(pData->metrics);
    sscanf( getData( fp ), "%u,%u,%u",
                    &(pMet->size), &(pMet->acrossCnt), &(pMet->downCnt) );
    pMet->gridSize  = pMet->size * pMet->size;

    entSize = (pMet->acrossCnt + pMet->downCnt)*SizeOf(XWORD_ENTRY);
    StsRet(OSHeapBlockAlloc(osProcessHeapId,entSize,&(pData->pEntries)), s);
    memset( pData->pEntries, 0, entSize );
```

```
    pEnt = pData->pEntries;
    pGrid = pData->metrics.grid;
    memset( pGrid, 0, XWORD_MAX_GRID_SIZE );
    for ( i1=0; i1<(pMet->acrossCnt); i1++, pEnt++) {
        sscanf( getData( fp ), "%u,%u,%u,%[^,],%[^\n\r]",
             &(pEnt->number),&(pEnt->x),&(pEnt->y),pEnt->word,pEnt->clue );
        strncpy( &pGrid[ pEnt->y * pMet->size + pEnt->x], pEnt->word,
                    strlen( pEnt->word ) );
        }

    for ( i1=0; i1<(pMet->downCnt); i1++, pEnt++ ) {
        sscanf( getData( fp ), "%u,%u,%u,%[^,],%[^\n\r]",
             &(pEnt->number),&(pEnt->x),&(pEnt->y),pEnt->word,pEnt->clue );
        for ( j=0, len=strlen(pEnt->word); j<len; j++ )
            pGrid[(pEnt->y + j) * pMet->size + pEnt->x] = pEnt->word[j];
        }

    StdioStreamUnbind( fp );

    return stsOK;
}


MsgHandlerArgType(XWordDataInit, P_XWORDDATA_NEW)
{
    INSTANCE_DATA   inst;
    STATUS          s;

    if ( pArgs->xword.file )
        StsRet( XWDBuildXWordFromFile( &inst, pArgs->xword.file ), s );
    else {
        memset( &inst, 0, SizeOf(INSTANCE_DATA) );
        inst.metrics.size       = pArgs->xword.size;
        inst.metrics.gridSize   = inst.metrics.size * inst.metrics.size;
        memset( inst.metrics.grid, ' ', inst.metrics.gridSize );
        }

    ObjectWrite(self, ctx, &inst);

    return stsOK;
```

```
    MsgHandlerParametersNoWarning;
}


MsgHandlerWithTypes(XWordDataFree, P_ARGS, P_INSTANCE_DATA)
{
    if ( pData->pEntries )
        OSHeapBlockFree( pData->pEntries );

    return stsOK;
    MsgHandlerParametersNoWarning;
}


MsgHandlerWithTypes(XWordDataSave, P_OBJ_SAVE, P_INSTANCE_DATA)
{
    STREAM_READ_WRITE fsWrite;
    U32               entCnt;
    STATUS            s;

    fsWrite.numBytes    = SizeOf(METRICS);
    fsWrite.pBuf        = &(pData->metrics);
    ObjCallRet(msgStreamWrite, pArgs->file, &fsWrite, s);

    if ( pData->pEntries ) {
        entCnt = pData->metrics.acrossCnt + pData->metrics.downCnt;
        fsWrite.numBytes    = entCnt*SizeOf(XWORD_ENTRY);
        fsWrite.pBuf        = pData->pEntries;
        ObjCallRet(msgStreamWrite, pArgs->file, &fsWrite, s);
        }

    return stsOK;
    MsgHandlerParametersNoWarning;
}


MsgHandlerArgType(XWordDataRestore, P_OBJ_RESTORE)
{
    INSTANCE_DATA       inst;
    STREAM_READ_WRITE   fsRead;
    STATUS              s;
```

```
    U32                 entSize;
    U32                 entCnt;

    fsRead.numBytes= SizeOf(METRICS);
    fsRead.pBuf= &inst.metrics;
    ObjCallRet(msgStreamRead, pArgs->file, &fsRead, s);

    entCnt = inst.metrics.acrossCnt + inst.metrics.downCnt;
    if ( entCnt ) {
        entSize = entCnt * SizeOf(XWORD_ENTRY);
        StsRet(OSHeapBlockAlloc(osProcessHeapId, entSize,
                                    &inst.pEntries),s);
        fsRead.numBytes = entSize;
        fsRead.pBuf     = inst.pEntries;
        ObjCallJmp(msgStreamRead, pArgs->file, &fsRead, s, Error);
        }

    ObjectWrite(self, ctx, &inst);

    return stsOK;
Error:
    OSHeapBlockFree( inst.pEntries );
    return s;
    MsgHandlerParametersNoWarning;
}


MsgHandlerWithTypes(XWordDataGetInfo, P_XWORDDATA_INFO, P_INSTANCE_DATA)
{
    U32             i, l;
    P_XWORD_ENTRY   pEnt;
    P_METRICS       pMet;
    LIST_NEW        ln;
    STROBJ_NEW      son;
    U8              buff[XWORD_MAX_CLUE_SIZE+5];
    STATUS          s;

    pArgs->size = pData->metrics.size;

    pMet = &(pData->metrics);
    memset( pArgs->template, 0, pMet->gridSize );
```

```
    memset( pArgs->numbers, 0, pMet->gridSize );

    for ( i=0; i<pMet->gridSize; i++ )
        pArgs->template[i] = pMet->grid[i] ? 1 : 0;

    pEnt = pData->pEntries;
    for ( i=0, l = pMet->acrossCnt + pMet->downCnt; i<l; i++, pEnt++)
         pArgs->numbers[pEnt->x + pEnt->y * pMet->size] = (U8)(pEnt->number);

    ObjCallRet( msgNewDefaults, clsList, &ln, s );
    ObjCallRet( msgNew, clsList, &ln, s );
    pArgs->acrossClues = ln.object.uid;

    ObjCallRet( msgNewDefaults, clsList, &ln, s );
    ObjCallRet( msgNew, clsList, &ln, s );
    pArgs->downClues = ln.object.uid;

    pEnt = pData->pEntries;
    for ( i=0; i<pMet->acrossCnt; i++, pEnt++ ) {
        ObjCallRet( msgNewDefaults, clsString, &son, s );
        sprintf( buff, "%u. %s", pEnt->number, pEnt->clue );
        son.strobj.pString = buff;
        ObjCallRet( msgNew, clsString, &son, s );
        ObjCallRet(msgListAddItem,pArgs->acrossClues,son.object.uid,s );
        }

    for ( i=0; i<pMet->downCnt; i++, pEnt++ ) {
        ObjCallRet( msgNewDefaults, clsString, &son, s );
        sprintf( buff, "%u. %s", pEnt->number, pEnt->clue );
        son.strobj.pString = buff;
        ObjCallRet( msgNew, clsString, &son, s );
        ObjCallRet( msgListAddItem, pArgs->downClues, son.object.uid, s );
        }

    return stsOK;
    MsgHandlerParametersNoWarning;
}


MsgHandlerWithTypes(XWordDataGetLetters, P_XWORD_DATA, P_INSTANCE_DATA)
{
```

```
    memcpy( pArgs, pData->metrics.grid, pData->metrics.gridSize );

    return stsOK;
    MsgHandlerParametersNoWarning;
}


MsgHandlerWithTypes(XWordDataGetAcrossCount, P_U32, P_INSTANCE_DATA)
{
    *pArgs = pData->metrics.acrossCnt;

    return stsOK;
    MsgHandlerParametersNoWarning;
}


MsgHandlerWithTypes(XWordDataGetDownCount, P_U32, P_INSTANCE_DATA)
{
    *pArgs = pData->metrics.downCnt;

    return stsOK;
    MsgHandlerParametersNoWarning;
}


MsgHandlerWithTypes(XWordDataGetAcrossWord, P_XWORDDATA_WORD,
                    P_INSTANCE_DATA)
{
    if ( pData->metrics.acrossCnt ) {
        pArgs->origin.x = pData->pEntries[pArgs->index].x;
        pArgs->origin.y = pData->pEntries[pArgs->index].y;
        strcpy( pArgs->word, pData->pEntries[pArgs->index].word );
        }
    else
        memset( pArgs, 0, SizeOf(XWORDDATA_WORD) );

    return stsOK;
    MsgHandlerParametersNoWarning;
}
```

```
MsgHandlerWithTypes(XWordDataGetDownWord, P_XWORDDATA_WORD,
                      P_INSTANCE_DATA)
{
    if ( pData->metrics.downCnt ) {
        pArgs->origin.x =
             pData->pEntries[pData->metrics.acrossCnt+pArgs->index].x;
        pArgs->origin.y =
             pData->pEntries[pData->metrics.acrossCnt+pArgs->index].y;
        strcpy( pArgs->word,
             pData->pEntries[pData->metrics.acrossCnt+pArgs->index].word );
        }
    else
        memset( pArgs, 0, SizeOf(XWORDDATA_WORD) );

    return stsOK;
    MsgHandlerParametersNoWarning;
}


STATUS ClsXWordDataInit (void)
{
    CLASS_NEW   new;
    STATUS      s;

    ObjCallRet(msgNewDefaults, clsClass, &new, s );

    new.object.uid       = clsXWordData;
    new.cls.pMsg         = clsXWordDataTable;
    new.cls.ancestor     = clsObject;
    new.cls.size         = SizeOf(INSTANCE_DATA);
    new.cls.newArgsSize  = SizeOf(XWORDDATA_NEW);

    ObjCallRet(msgNew, clsClass, &new, s );

    return stsOK;
}
```

xwrdview.h

```
#ifndef XWRDVIEW_INCLUDED
#define XWRDVIEW_INCLUDED

#ifndef GO_INCLUDED
#include <go.h>
#endif

#ifndef CLSMGR_INCLUDED
#include <clsmgr.h>
#endif

#ifndef VIEW_INCLUDED
#include <view.h>
#endif

#define clsXWordView    MakeGlobalWKN(4150,1)

#define msgXWordViewStartPlayOver       MakeMsg( clsXWordView, 1 )
#define msgXWordViewShowSoln            MakeMsg( clsXWordView, 2 )
#define msgXWordViewClueTapNothing      MakeMsg( clsXWordView, 3 )
#define msgXWordViewClueTapStrikeOut    MakeMsg( clsXWordView, 4 )
#define msgXWordViewCheckPuzzle         MakeMsg( clsXWordView, 5 )
#define msgXWordViewCheckLetters        MakeMsg( clsXWordView, 6 )
#define msgXWordViewCheckWords          MakeMsg( clsXWordView, 7 )

STATUS ClsXWordViewInit(void);

#define xwordviewNewFields \
    viewNewFields

typedef struct XWORDVIEW_NEW {
    xwordviewNewFields
} XWORDVIEW_NEW, *P_XWORDVIEW_NEW;

typedef struct XWORDVIEW_STATS {
    U32 wordCount,
        okWords,
        letterCount,
        okLetters;
```

```
} XWORDVIEW_STATS, *P_XWORDVIEW_STATS;

#endif
```

xwrdview.c

```
#ifndef WIN_INCLUDED
#include <win.h>
#endif

#ifndef FS_INCLUDED
#include <fs.h>
#endif

#ifndef PREFS_INCLUDED
#include <prefs.h>
#endif

#ifndef XWRDVIEW_INCLUDED
#include <xwrdview.h>
#endif

#ifndef XWRDGRID_INCLUDED
#include <xwrdgrid.h>
#endif

#ifndef XWRDCLUE_INCLUDED
#include <xwrdclue.h>
#endif

#ifndef XWRDDATA_INCLUDED
#include <xwrddata.h>
#endif

#ifndef DEBUG_INCLUDED
#include <debug.h>
#endif

#include <method.h>
#include <stdio.h>
#include <string.h>

#define gridWinTag      MakeTag( clsXWordView, 1 )
#define acrossWinTag    MakeTag( clsXWordView, 2 )
#define downWinTag      MakeTag( clsXWordView, 3 )
```

```
typedef struct INSTANCE_DATA {
    U8      dispOrientation;
    U32     size;
    U32     gridSize;
    OBJECT  model;
    OBJECT  grid;
    OBJECT  acrossClues;
    OBJECT  downClues;
} INSTANCE_DATA, *P_INSTANCE_DATA;


STATUS LOCAL
XWVBuildClueList( P_STRING pTitle, OBJECT clueList, TAG winTag,
                  P_OBJECT pList )
{
    XWORDCLUE_NEW   xwc;
    STATUS          s;

    ObjCallRet(msgNewDefaults, clsXWordClueList, &xwc, s);
    xwc.win.tag                  = winTag;
    xwc.border.style.edge        = bsEdgeAll;
    xwc.border.style.shadow      = bsShadowThickBlack;
    xwc.xwclue.pTitle            = pTitle;
    xwc.xwclue.clueList          = clueList;
    ObjCallRet(msgNew, clsXWordClueList, &xwc, s);

    *pList = xwc.object.uid;
    return stsOK;
}


STATUS LOCAL
XWVBuildGrid( U32 size, U32 gridSize,
              P_XWORD_DATA pTemplate, P_XWORD_DATA pNumbers,
              TAG winTag, P_OBJECT pGrid )
{
    XWORDGRID_NEW   xwc;
    STATUS          s;
    U32             i;
```

```
    ObjCallRet(msgNewDefaults, clsXWordGrid, &xwc, s);
    xwc.win.tag              = winTag;
    xwc.border.style.shadow = bsShadowThickBlack;
    xwc.xwgrid.size          = size;
    for ( i=0; i<gridSize; i++ ) {
        xwc.xwgrid.template[i] = pTemplate[i];
        xwc.xwgrid.numbers[i]  = pNumbers[i];
        }
    ObjCallRet(msgNew, clsXWordGrid, &xwc, s);

    *pGrid = xwc.object.uid;
    return stsOK;
}


MsgHandlerArgType(XWordViewSetDataObject, OBJECT )
{
    INSTANCE_DATA   inst;

    inst = IDataDeref( pData, INSTANCE_DATA );
    inst.model = pArgs;
    ObjectWrite(self, ctx, &inst);

    return stsOK;
    MsgHandlerParametersNoWarning;
}


MsgHandlerWithTypes(XWordViewSave, P_OBJ_SAVE, P_INSTANCE_DATA)
{
    STREAM_READ_WRITE   fsWrite;
    STATUS              s;

    fsWrite.numBytes = SizeOf(U32);
    fsWrite.pBuf     = &(pData->size);
    ObjCallRet(msgStreamWrite, pArgs->file, &fsWrite, s );

    return stsOK;
    MsgHandlerParametersNoWarning;
}
```

```
MsgHandlerArgType(XWordViewRestore, P_OBJ_RESTORE)
{
    INSTANCE_DATA       inst;
    RES_READ_DATA       read;
    STREAM_READ_WRITE   fsRead;
    STATUS              s;

    fsRead.numBytes = SizeOf(U32);
    fsRead.pBuf     = &inst.size;
    ObjCallRet(msgStreamRead, pArgs->file, &fsRead, s );

    inst.gridSize = inst.size * inst.size;

    read.resId  = prOrientation;
    read.heap   = 0;
    read.pData  = &inst.dispOrientation;
    read.length = SizeOf(U8);
    ObjCallRet(msgResReadData, theSystemPreferences, &read, s );

    inst.grid        =
      (WIN)ObjectCall(msgWinFindTag,self,(P_ARGS)gridWinTag);
    inst.acrossClues =
      (WIN)ObjectCall(msgWinFindTag,self,(P_ARGS)acrossWinTag);
    inst.downClues   =
      (WIN)ObjectCall(msgWinFindTag,self,(P_ARGS)downWinTag);

    ObjCallRet( msgViewGetDataObject, self, &inst.model, s);

    ObjectWrite(self, ctx, &inst);

    return stsOK;
    MsgHandlerParametersNoWarning;
}


LOCAL
XWVLandscapeLayout( P_INSTANCE_DATA pData, P_CSTM_LAYOUT_CHILD_SPEC pSpec )
{
    if ( pSpec->child == pData->grid ) {
        pSpec->metrics.h.constraint = clPctOf;
```

```
        pSpec->metrics.h.value      = 96;
        pSpec->metrics.w.constraint = clSameAs | clOpposite;
        pSpec->metrics.w.relWin     = pSpec->child;
        pSpec->metrics.x.constraint =
                ClAlign( clMinEdge, clPctOf, clMaxEdge );
        pSpec->metrics.x.value      = 2;
        pSpec->metrics.y.constraint =
                ClAlign(clCenterEdge, clSameAs, clCenterEdge);
        }
    else if ( pSpec->child == pData->acrossClues ) {
        pSpec->metrics.w.constraint = ClExtend(clPctOf, clMaxEdge);
        pSpec->metrics.w.value      = 98;
        pSpec->metrics.h.constraint = clPctOf;
        pSpec->metrics.h.value      = 44;
        pSpec->metrics.h.relWin     = pData->grid;
        pSpec->metrics.x.constraint =
                ClAlign(clMinEdge, clPctOf, clMaxEdge);
        pSpec->metrics.x.value      = 106;
        pSpec->metrics.x.relWin     = pData->grid;
        pSpec->metrics.y.constraint =
                ClAlign(clMaxEdge, clSameAs, clMaxEdge);
        pSpec->metrics.y.relWin     = pData->grid;
        }
    else if ( pSpec->child == pData->downClues ) {
        pSpec->metrics.w.constraint = ClExtend(clPctOf, clMaxEdge);
        pSpec->metrics.w.value      = 98;
        pSpec->metrics.h.constraint = clPctOf;
        pSpec->metrics.h.value      = 44;
        pSpec->metrics.h.relWin     = pData->grid;
        pSpec->metrics.x.constraint =
                ClAlign(clMinEdge, clPctOf, clMaxEdge);
        pSpec->metrics.x.relWin     = pData->grid;
        pSpec->metrics.x.value      = 106;
        pSpec->metrics.y.constraint =
                ClAlign(clMinEdge, clSameAs, clMinEdge);
        pSpec->metrics.y.relWin     = pData->grid;
        }
    return stsOK;
}
```

```
LOCAL
XWVPortraitLayout( P_INSTANCE_DATA pData, P_CSTM_LAYOUT_CHILD_SPEC pSpec )
{
    if ( pSpec->child == pData->grid ) {
        pSpec->metrics.h.constraint = clSameAs | clOpposite;
        pSpec->metrics.h.relWin     = pSpec->child;
        pSpec->metrics.w.constraint = clPctOf;
        pSpec->metrics.w.value      = 80;
        pSpec->metrics.x.constraint =
                ClAlign(clCenterEdge, clSameAs, clCenterEdge);
        pSpec->metrics.y.constraint =
                ClAlign( clMaxEdge, clPctOf, clMaxEdge );
        pSpec->metrics.y.value      = 98;
        }
    else if ( pSpec->child == pData->acrossClues ) {
        pSpec->metrics.h.constraint = ClExtend(clPctOf, clMinEdge);
        pSpec->metrics.h.value      = 94;
        pSpec->metrics.h.relWin     = pData->grid;
        pSpec->metrics.w.constraint = clPctOf;
        pSpec->metrics.w.value      = 44;
        pSpec->metrics.w.relWin     = pData->grid;
        pSpec->metrics.y.constraint =
                ClAlign(clMinEdge, clPctOf, clMaxEdge);
        pSpec->metrics.y.value      = 2;
        pSpec->metrics.x.constraint =
                ClAlign(clMinEdge, clSameAs, clMinEdge);
        pSpec->metrics.x.relWin     = pData->grid;
        }
    else if ( pSpec->child == pData->downClues ) {
        pSpec->metrics.h.constraint = ClExtend(clPctOf, clMinEdge);
        pSpec->metrics.h.value      = 94;
        pSpec->metrics.h.relWin     = pData->grid;
        pSpec->metrics.w.constraint = clPctOf;
        pSpec->metrics.w.value      = 44;
        pSpec->metrics.w.relWin     = pData->grid;
        pSpec->metrics.y.constraint =
                ClAlign(clMinEdge, clPctOf, clMaxEdge);
        pSpec->metrics.y.value      = 2;
        pSpec->metrics.x.constraint =
                ClAlign(clMaxEdge, clSameAs, clMaxEdge);
        pSpec->metrics.x.relWin     = pData->grid;
```

```
        }
    return stsOK;
}


MsgHandlerWithTypes(XWordViewCLGetChildSpec, P_CSTM_LAYOUT_CHILD_SPEC,
                    P_INSTANCE_DATA)
{
    if ( pData->dispOrientation == prLandscape )
        XWVLandscapeLayout( pData, pArgs );
    else
        XWVPortraitLayout( pData, pArgs );

    return stsOK;
    MsgHandlerParametersNoWarning;
}


MsgHandlerArgType(XWordViewNewDefaults, P_XWORDVIEW_NEW)
{
    pArgs->view.createDataObject = TRUE;

    return stsOK;
    MsgHandlerParametersNoWarning;
}


MsgHandlerArgType(XWordViewInit, P_XWORDVIEW_NEW)
{
    INSTANCE_DATA   inst;
    WIN_METRICS     wm;
    BORDER_STYLE    bs;
    XWORDDATA_NEW   xwn;
    RES_READ_DATA   read;
    STATUS          s;
    XWORDDATA_INFO  xwrdInfo;

    if ( !(pArgs->view.dataObject) && pArgs->view.createDataObject ) {
        ObjCallRet(msgNewDefaults, clsXWordData, &xwn, s );
        xwn.xword.size = 10;
        ObjCallRet(msgNew, clsXWordData, &xwn, s);
```

```
        ObjCallRet(msgViewSetDataObject, self, xwn.object.uid, s );
        inst.model = xwn.object.uid;
        }
    else
        inst.model = pArgs->view.dataObject;

    read.resId  = prOrientation;
    read.heap   = 0;
    read.pData  = &inst.dispOrientation;
    read.length = SizeOf(U8);
    ObjCallRet(msgResReadData, theSystemPreferences, &read, s );

    ObjCallRet(msgXWordDataGetInfo, inst.model, &xwrdInfo, s);

    inst.size       = xwrdInfo.size;
    inst.gridSize   = inst.size * inst.size;

    StsRet( XWVBuildClueList( "Across", xwrdInfo.acrossClues,
                             acrossWinTag, &inst.acrossClues ), s);
    StsRet( XWVBuildClueList( "Down", xwrdInfo.downClues,
                             downWinTag, &inst.downClues ), s);

    StsRet( XWVBuildGrid(inst.size, inst.gridSize,
                         xwrdInfo.template, xwrdInfo.numbers,
                         gridWinTag, &inst.grid ), s);

    ObjectWrite(self, ctx, &inst);

    ObjCallRet(msgBorderGetStyle, self, &bs, s );
    bs.backgroundInk = bsInkGray33;
    ObjCallWarn(msgBorderSetStyle, self, &bs );

    wm.parent = self;
    wm.options = wsPosTop;
    ObjCallRet( msgWinInsert, inst.acrossClues, &wm, s );
    ObjCallRet( msgWinInsert, inst.downClues, &wm, s );
    ObjCallRet( msgWinInsert, inst.grid, &wm, s );

    return stsOK;
    MsgHandlerParametersNoWarning;
}
```

```
MsgHandlerWithTypes(XWordViewStartPlayOver, P_ARGS, P_INSTANCE_DATA)
{
    STATUS s;

    ObjCallRet( msgXWordGridStartPlayOver, pData->grid, NULL, s );
    ObjCallRet( msgXWordClueStartPlayOver, pData->acrossClues, NULL, s );
    ObjCallRet( msgXWordClueStartPlayOver, pData->downClues, NULL, s );

    return stsOK;
    MsgHandlerParametersNoWarning;
}


MsgHandlerWithTypes(XWordViewShowSoln, P_ARGS, P_INSTANCE_DATA)
{
    XWORD_DATA  solution[XWORD_MAX_GRID_SIZE];
    GRID_DATA   gridData[GRID_MAX_GRID_SIZE];
    U32         i;
    STATUS      s;

    ObjCallRet( msgXWordDataGetLetters, pData->model, &solution, s );
    for ( i=0; i<pData->gridSize; i++ )
        gridData[i] = solution[i];

    ObjCallRet( msgXWordGridSetLetters, pData->grid, gridData, s );

    return stsOK;
    MsgHandlerParametersNoWarning;
}


MsgHandlerWithTypes(XWordViewClueTapNothing, P_ARGS, P_INSTANCE_DATA)
{
    STATUS s;

    ObjCallRet( msgXWordClueClueTapNothing, pData->acrossClues, NULL, s );
    ObjCallRet( msgXWordClueClueTapNothing, pData->downClues, NULL, s );

    return stsOK;
```

```
    MsgHandlerParametersNoWarning;
}


MsgHandlerWithTypes(XWordViewClueTapStrikeOut, P_ARGS, P_INSTANCE_DATA)
{
    STATUS s;

    ObjCallRet(msgXWordClueClueTapStrikeOut,pData->acrossClues,NULL, s );
    ObjCallRet( msgXWordClueClueTapStrikeOut,pData->downClues, NULL, s );

    return stsOK;
    MsgHandlerParametersNoWarning;
}


XWVaccStrEqu( P_U8 gStr, U32 size, P_U8 word )
{
    return( !strncmp( gStr, word, strlen(word) ) ? 1 : 0 );
    Unused( size );
}


XWVdwnStrEqu( P_U8 gStr, U32 size, P_U8 word )
{
    U32 len, i;

    for ( i=0, len=strlen(word); i < len; i++ ) {
        if ( *gStr != word[i] )
            break;
        gStr += size;
        }

    return( i == len );
}


MsgHandlerWithTypes(XWordViewCheckPuzzle, P_XWORDVIEW_STATS, P_INSTANCE_DATA)
{
    XWORD_DATA      solution[XWORD_MAX_GRID_SIZE];
    GRID_DATA       frGrid[GRID_MAX_GRID_SIZE];
```

```
    U32             i, len, cnt, index;
    XWORDDATA_WORD  xdw;
    STATUS          s;

    ObjCallRet( msgXWordDataGetLetters, pData->model, &solution, s );
    ObjCallRet( msgXWordGridGetLetters, pData->grid, &frGrid, s );

    pArgs->letterCount = pArgs->okLetters = 0;
    for ( i=0, len=pData->gridSize; i<len; i++ )
        if ( solution[i] ) {
            pArgs->letterCount++;
            if ( solution[i] == frGrid[i] )
                pArgs->okLetters++;
            }

    pArgs->okWords = 0;
    ObjCallRet( msgXWordDataGetAcrossCount, pData->model, &cnt, s );
    pArgs->wordCount = cnt;
    for ( i=0; i<cnt; i++ ) {
        xdw.index = i;
        ObjCallRet( msgXWordDataGetAcrossWord, pData->model, &xdw, s );
        index = xdw.origin.x + xdw.origin.y*pData->size;
        if ( XWVaccStrEqu( &frGrid[index], pData->size, xdw.word) )
            pArgs->okWords++;
        }

    ObjCallRet( msgXWordDataGetDownCount, pData->model, &cnt, s );
    pArgs->wordCount += cnt;
    for ( i=0; i<cnt; i++ ) {
        xdw.index = i;
        ObjCallRet( msgXWordDataGetDownWord, pData->model, &xdw, s );
        index = xdw.origin.x + xdw.origin.y*pData->size;
        if ( XWVdwnStrEqu( &frGrid[index], pData->size, xdw.word ) )
            pArgs->okWords++;
        }

    return stsOK;
    MsgHandlerParametersNoWarning;
}
```

```
MsgHandlerWithTypes(XWordViewCheckLetters, P_ARGS, P_INSTANCE_DATA)
{
    XWORD_DATA  solution[XWORD_MAX_GRID_SIZE];
    GRID_DATA   frGrid[GRID_MAX_GRID_SIZE], toGrid[GRID_MAX_GRID_SIZE];
    U32         i;
    STATUS      s;

    ObjCallRet( msgXWordDataGetLetters, pData->model, &solution, s );
    ObjCallRet( msgXWordGridGetLetters, pData->grid, &frGrid, s );

    for ( i=0; i<pData->gridSize; i++ )
        if ( frGrid[i] )
            toGrid[i] = (solution[i] == frGrid[i]);
        else
            toGrid[i] = 0;

    ObjCallRet( msgXWordGridSetOkLetters, pData->grid, toGrid, s );

    return stsOK;
    MsgHandlerParametersNoWarning;
}


MsgHandlerWithTypes(XWordViewCheckWords, P_ARGS, P_INSTANCE_DATA)
{
    U32             i, j, len, cnt, index;
    GRID_DATA       frGrid[GRID_MAX_GRID_SIZE], toGrid[GRID_MAX_GRID_SIZE];
    XWORDDATA_WORD  xdw;
    STATUS          s;

    ObjCallRet( msgXWordGridGetLetters, pData->grid, &frGrid, s );
    memset( toGrid, 0, pData->gridSize * SizeOf(GRID_DATA) );

    ObjCallRet( msgXWordDataGetAcrossCount, pData->model, &cnt, s );
    for ( i=0; i<cnt; i++ ) {
        xdw.index = i;
        ObjCallRet( msgXWordDataGetAcrossWord, pData->model, &xdw, s );
        index = xdw.origin.x + xdw.origin.y*pData->size;
        if ( XWVaccStrEqu( &frGrid[index], pData->size, xdw.word) ) {
            len = strlen( xdw.word );
            for ( j=0 ; j<len ; j++ )
```

```
                toGrid[index+j] = 1;
            }
        }

    ObjCallRet( msgXWordDataGetDownCount, pData->model, &cnt, s );
    for ( i=0; i<cnt; i++ ) {
        xdw.index = i;
        ObjCallRet( msgXWordDataGetDownWord, pData->model, &xdw, s );
        index = xdw.origin.x + xdw.origin.y*pData->size;
        if ( XWVdwnStrEqu( &frGrid[index], pData->size, xdw.word ) ) {
            len = strlen( xdw.word );
            for ( j=0 ; j<len ; j++ ) {
                toGrid[index] = 1;
                index += pData->size;
                }
            }
        }

    ObjCallRet( msgXWordGridSetOkLetters, pData->grid, toGrid, s );

    return stsOK;
    MsgHandlerParametersNoWarning;
}


STATUS ClsXWordViewInit(void)
{
    CLASS_NEW c;
    STATUS    s;

    ObjCallRet(msgNewDefaults, clsClass, &c, s );
    c.object.uid        = clsXWordView;
    c.cls.pMsg          = clsXWordViewTable;
    c.cls.ancestor      = clsView;
    c.cls.size          = SizeOf(INSTANCE_DATA);
    c.cls.newArgsSize   = SizeOf(XWORDVIEW_NEW);
    ObjCallRet(msgNew, clsClass, &c, s );

    return stsOK;
}
```

xwrdgrid.h

```
#ifndef XWRDGRID_INCLUDED
#define XWRDGRID_INCLUDED

#ifndef GO_INCLUDED
#include <go.h>
#endif

#ifndef CLSMGR_INCLUDED
#include <clsmgr.h>
#endif

#ifndef SPAPER_INCLUDED
#include <spaper.h>
#endif

#define clsXWordGrid    MakeGlobalWKN(4151,1)

#define msgXWordGridStartPlayOver   MakeMsg( clsXWordGrid, 1 )
#define msgXWordGridGetLetters      MakeMsg( clsXWordGrid, 2 )
#define msgXWordGridSetLetters      MakeMsg( clsXWordGrid, 3 )
#define msgXWordGridSetOkLetters    MakeMsg( clsXWordGrid, 4 )

STATUS ClsXWordGridInit(void);

#define GRID_MAX_GRID_SIZE 100

typedef U8 GRID_DATA, *P_GRID_DATA;

typedef struct {
    U8          size;
    GRID_DATA   numbers[GRID_MAX_GRID_SIZE];
    GRID_DATA   template[GRID_MAX_GRID_SIZE];
} XWORDGRID_NEW_ONLY, *P_XWORDGRID_NEW_ONLY;

#define xwordgridNewFields \
    sPaperNewFields \
    XWORDGRID_NEW_ONLY xwgrid;
```

```
typedef struct XWORDGRID_NEW {
    xwordgridNewFields
} XWORDGRID_NEW, *P_XWORDGRID_NEW;

#endif
```

xwrdgrid.c

```
#ifndef WIN_INCLUDED
#include <win.h>
#endif

#ifndef GEO_INCLUDED
#include <geo.h>
#endif

#ifndef FS_INCLUDED
#include <fs.h>
#endif

#ifndef SPAPER_INCLUDED
#include <spaper.h>
#endif

#ifndef SYSGRAF_INCLUDED
#include <sysgraf.h>
#endif

#ifndef SYSFONT_INCLUDED
#include <sysfont.h>
#endif

#ifndef OSHEAP_INCLUDED
#include <osheap.h>
#endif

#ifndef GOMATH_INCLUDED
#include <gomath.h>
#endif

#ifndef XLATE_INCLUDED
#include <xlate.h>
#endif

#ifndef XLFILTER_INCLUDED
#include <xlfilter.h>
#endif
```

```
#ifndef XTEMPLT_INCLUDED
#include <xtemplt.h>
#endif

#ifndef XWRDGRID_INCLUDED
#include <xwrdgrid.h>
#endif

#ifndef DEBUG_INCLUDED
#include <debug.h>
#endif

#include <method.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define GRID_DATAFILE   "gridDataFile"

#define BLOCK_SIZE          100    // percent of display
#define BLOCK_LTR_X_OFF      25
#define BLOCK_LTR_Y_OFF      20
#define BLOCK_NUM_X_OFF       5
#define BLOCK_NUM_Y_OFF       5

#define beNull   0x00
#define beBlack  0x01
#define beNumber 0x02
#define beLetter 0x04
#define beRight  0x08
#define beWrong  0x10

typedef struct GRID_ENTRY {
    U8  number;
    U8  letter;
    U8  status;
} GRID_ENTRY, *P_GRID_ENTRY;

typedef struct INSTANCE_DATA {
    U32             size;
```

```
    U32             gridSize;
    U32             screenBlockSize;
    SYSDC           gridDC;
    OBJECT          gdFileHandle;
    P_GRID_ENTRY    pEntries;
} INSTANCE_DATA, *P_INSTANCE_DATA;


STATUS LOCAL
XWGBuildGridDC( P_SYSDC pDC )
{
    SYSDC_NEW       dn;
    SYSDC_FONT_SPEC fs;
    STATUS          s;

    ObjCallRet(msgNewDefaults, clsSysDrwCtx, &dn, s );
    ObjCallRet(msgNew, clsSysDrwCtx, &dn, s );
    *pDC = dn.object.uid;

    ObjCallWarn(msgDcSetLineThickness, *pDC, (P_ARGS)2);

    fs.id                   = 0;
    fs.attr.group           = sysDcGroupUserInput;
    fs.attr.weight          = sysDcWeightNormal;
    fs.attr.aspect          = sysDcAspectNormal;
    fs.attr.italic          = 0;
    fs.attr.monospaced      = 0;
    fs.attr.encoding        = sysDcEncodeGoSystem;
    ObjCallRet(msgDcOpenFont, *pDC, &fs, s );

    return stsOK;
}

MsgHandlerArgType(XWordGridNewDefaults, P_XWORDGRID_NEW)
{
    pArgs->border.style.edge     = bsEdgeAll;
    memset( &(pArgs->xwgrid), 0, SizeOf(XWORDGRID_NEW_ONLY) );

    return stsOK;
    MsgHandlerParametersNoWarning;
}
```

```
STATUS LOCAL
XWGBuildTranslator( P_OBJECT pTranslator )
{
    P_UNKNOWN           pNewTemplate;
    XLATE_NEW           xNewTrans;
    U16                 xlateFlags;
    XTM_ARGS            xtmArgs;
    STATUS              s;

    ObjCallRet(msgNewDefaults, clsXText, &xNewTrans, s );

    xtmArgs.xtmType     = xtmTypeCharList;
    xtmArgs.xtmMode     = 0;
    xtmArgs.pXtmData    = "ABCDEFGHIJKLMNOPQRSTUVWXYZ-";
    StsRet(XTemplateCompile(&xtmArgs, osProcessHeapId, &pNewTemplate), s);

    xNewTrans.xlate.pTemplate = pNewTemplate;
    xNewTrans.xlate.hwxFlags &=
            ~(xltCaseEnable | xltPunctuationEnable | xltVerticalEnable);

    ObjCallRet(msgNew, clsXText, &xNewTrans, s);

    ObjCallRet(msgXlateGetFlags, xNewTrans.object.uid, &xlateFlags, s);
    xlateFlags |= xTemplateVeto | xltSpaceDisable;
    ObjCallRet(msgXlateSetFlags,xNewTrans.object.uid,(P_ARGS)xlateFlags, s);

    *pTranslator = xNewTrans.object.uid;

    return stsOK;
}


MsgHandlerArgType(XWordGridInit, P_XWORDGRID_NEW)
{
    INSTANCE_DATA       inst;
    FS_NEW              fsn;
    STREAM_READ_WRITE   fsWrite;
    GRID_ENTRY          ge[GRID_MAX_GRID_SIZE];
    STATUS              s;
```

```
    U32                 i;

    StsRet( XWGBuildTranslator( &(pArgs->sPaper.translator) ), s );

    pArgs->sPaper.flags         &= ~spRuling;
    pArgs->sPaper.flags         |= spProx;

    ObjectCallAncestorCtx(ctx);

    inst.size       = pArgs->xwgrid.size;
    inst.gridSize   = (inst.size * inst.size);

    ObjCallRet( msgNewDefaults, clsFileHandle, &fsn, s );
    fsn.fs.locator.pPath    = GRID_DATAFILE;
    fsn.fs.locator.uid      = theWorkingDir;
    ObjCallRet(msgNew, clsFileHandle, &fsn, s );
    inst.gdFileHandle = fsn.object.uid;

    fsWrite.numBytes = inst.gridSize*SizeOf(GRID_ENTRY);
    memset( ge, 0, fsWrite.numBytes );
    fsWrite.pBuf = ge;
    ObjCallRet( msgStreamWrite, inst.gdFileHandle, &fsWrite, s);
    ObjCallRet( msgFSMemoryMap, inst.gdFileHandle, &inst.pEntries, s );

    for ( i=0; i<inst.gridSize; i++ ) {
        if ( !(pArgs->xwgrid.template[i]) )
            inst.pEntries[i].status |= beBlack;
        else if ( inst.pEntries[i].number = pArgs->xwgrid.numbers[i] )
            inst.pEntries[i].status |= beNumber;
        }

    StsRet( XWGBuildGridDC( &inst.gridDC ), s );

    ObjectWrite(self, ctx, &inst);

    ObjectCall(msgDcSetWindow, inst.gridDC, (P_ARGS)self);

    return stsOK;
    MsgHandlerParametersNoWarning;
}
```

```
MsgHandlerWithTypes(XWordGridFree, P_ARGS, P_INSTANCE_DATA)
{
    STATUS s;

    ObjCallRet( msgFSMemoryMapFree, pData->gdFileHandle, NULL, s );
    ObjCallWarn( msgDestroy, pData->gdFileHandle, NULL );

    ObjCallWarn( msgDestroy, pData->gridDC, NULL );

    return stsOK;
    MsgHandlerParametersNoWarning;
}


MsgHandlerWithTypes(XWordGridSave, P_OBJ_SAVE, P_INSTANCE_DATA)
{
    STREAM_READ_WRITE   fsWrite;
    STATUS              s;

    fsWrite.numBytes = SizeOf(U32);
    fsWrite.pBuf     = &(pData->size);
    ObjCallRet(msgStreamWrite, pArgs->file, &fsWrite, s );

    fsWrite.numBytes = SizeOf(U32);
    fsWrite.pBuf     = &(pData->screenBlockSize);
    ObjCallRet(msgStreamWrite, pArgs->file, &fsWrite, s );

    return stsOK;
    MsgHandlerParametersNoWarning;
}


MsgHandlerArgType(XWordGridRestore, P_OBJ_RESTORE)
{
    STREAM_READ_WRITE   fsRead;
    INSTANCE_DATA       inst;
    FS_NEW              fsn;
    STATUS              s;

    fsRead.numBytes = SizeOf(U32);
```

```
    fsRead.pBuf      = &inst.size;
    ObjCallRet(msgStreamRead, pArgs->file, &fsRead, s );

    inst.gridSize = inst.size * inst.size;

    fsRead.numBytes = SizeOf(U32);
    fsRead.pBuf      = &inst.screenBlockSize;
    ObjCallRet(msgStreamRead, pArgs->file, &fsRead, s );

    ObjCallRet( msgNewDefaults, clsFileHandle, &fsn, s );
    fsn.fs.locator.pPath     = GRID_DATAFILE;
    fsn.fs.locator.uid       = theWorkingDir;
    ObjCallRet(msgNew, clsFileHandle, &fsn, s );
    inst.gdFileHandle = fsn.object.uid;

    ObjCallRet( msgFSMemoryMap, inst.gdFileHandle, &inst.pEntries, s );

    StsRet( XWGBuildGridDC( &inst.gridDC ), s );

    ObjectWrite(self, ctx, &inst);

    ObjCallWarn(msgDcSetWindow, inst.gridDC, (P_ARGS)self);

    ObjCallWarn( msgSPaperClear, self, NULL );

    return stsOK;
    MsgHandlerParametersNoWarning;
}


MsgHandlerWithTypes(XWordGridGetLetters, P_GRID_DATA, P_INSTANCE_DATA)
{
    U32 i;

    for ( i=0; i<pData->gridSize; i++ )
        if ( pData->pEntries[i].status & beLetter )
            pArgs[i] = pData->pEntries[i].letter;
        else
            pArgs[i] = 0;

    return stsOK;
```

```
    MsgHandlerParametersNoWarning;
}


STATUS LOCAL
XWGGridPosToRect( P_INSTANCE_DATA pData, P_XY32 pIn, P_RECT32 pOut )
{
    pOut->origin.x  = pIn->x * pData->screenBlockSize;
    pOut->origin.y  = (pData->size - pIn->y - 1) * pData->screenBlockSize;
    pOut->size.w    = pOut->size.h = pData->screenBlockSize;

    return stsOK;
}


MsgHandlerWithTypes(XWordGridSetLetters, P_GRID_DATA, P_INSTANCE_DATA)
{
    U32     i;
    XY32    penLoc;
    RECT32  dr;
    STATUS  s;

    for ( i=0; i<pData->gridSize; i++ )
        if ( pData->pEntries[i].letter = pArgs[i] ) {
            pData->pEntries[i].status |= beLetter | beRight;
            pData->pEntries[i].status &= ~beWrong;
            penLoc.x = i % pData->size;
            penLoc.y = i / pData->size;
            StsRet( XWGGridPosToRect( pData, &penLoc, &dr ), s );
            ObjCallRet( msgWinDirtyRect, self, &dr, s );
            }

    return stsOK;
    MsgHandlerParametersNoWarning;
}


MsgHandlerWithTypes(XWordGridSetOkLetters, P_GRID_DATA, P_INSTANCE_DATA)
{
    U32     i;
    XY32    penLoc;
```

```
    RECT32  dr;
    STATUS  s;

    for ( i=0; i<pData->gridSize; i++ )
        if ( pData->pEntries[i].status & beLetter ) {
            pData->pEntries[i].status &= ~( beWrong | beRight );
            pData->pEntries[i].status |= pArgs[i] ? beRight : beWrong;
            penLoc.x = i % pData->size;
            penLoc.y = i / pData->size;
            StsRet( XWGGridPosToRect( pData, &penLoc, &dr ), s );
            ObjCallRet( msgWinDirtyRect, self, &dr, s );
            }

    return stsOK;
    MsgHandlerParametersNoWarning;
}


STATUS LOCAL
XWGDrawGrid( P_INSTANCE_DATA pData )
{
    SYSDC_POLYLINE  pl;
    XY32            pnts[2];
    STATUS          s;
    U16             i;
    U32             gridWorldSize;

    ObjCallWarn(msgDcSetForegroundRGB,pData->gridDC,(P_ARGS)sysDcRGBBlack );
    gridWorldSize = pData->size * BLOCK_SIZE;

    pl.count  = 2;
    pl.points = pnts;
    pnts[0].y = gridWorldSize;
    pnts[1].y = 0;
    for ( i=BLOCK_SIZE; i<gridWorldSize; i+=BLOCK_SIZE ) {
        pnts[0].x = pnts[1].x = i;
        ObjCallRet( msgDcDrawPolyline, pData->gridDC, &pl, s );
        }

    pnts[0].x = 0;
    pnts[1].x = gridWorldSize;
```

```
    for ( i=BLOCK_SIZE; i<gridWorldSize; i+=BLOCK_SIZE ) {
        pnts[0].y = pnts[1].y = i;
        ObjCallRet( msgDcDrawPolyline, pData->gridDC, &pl, s );
        }

    return stsOK;
}


STATUS LOCAL
XWGDrawTemplate( P_INSTANCE_DATA pData )
{
    SYSDC_TEXT_OUTPUT   tx;
    SYSDC_PATTERN       oldPat;
    U8                  c[3];
    U32                 x, y;
    SCALE               fontScale;
    RECT32              blackOut;
    P_GRID_ENTRY        pGridEntry;
    STATUS              s;

    ObjCallWarn(msgDcIdentityFont, pData->gridDC, pNull );
    fontScale.x = fontScale.y = FxMakeFixed(((BLOCK_SIZE*1)/4),0);
    ObjCallWarn(msgDcScaleFont, pData->gridDC, &fontScale);

    ObjCallWarn(msgDcSetForegroundRGB,pData->gridDC,(P_ARGS)sysDcRGBBlack );
    oldPat = ObjCallWarn(msgDcSetFillPat, pData->gridDC,
                         (P_ARGS)sysDcPat75);
    blackOut.size.w = blackOut.size.h = BLOCK_SIZE;

    pGridEntry = pData->pEntries;

    memset( &tx, 0, sizeof(SYSDC_TEXT_OUTPUT));
    tx.alignChr = sysDcAlignChrTop;
    tx.pText    = c;
    tx.lenText  = 2;
    for( y=0; y<pData->size; y++ ) {
        tx.cp.y = (pData->size - y)*BLOCK_SIZE - BLOCK_NUM_Y_OFF;
        blackOut.origin.y = (pData->size - y - 1)*BLOCK_SIZE;
        for( x = 0; x<pData->size; x++, pGridEntry++ )
            if ( pGridEntry->status & beBlack ) {
```

```
                blackOut.origin.x = x*BLOCK_SIZE;
                ObjCallRet(msgDcDrawRectangle,pData->gridDC,&blackOut,s);
                }
            else if ( pGridEntry->status & beNumber ) {
                sprintf( c, "%2d", pGridEntry->number );
                tx.cp.x = x*BLOCK_SIZE + BLOCK_NUM_X_OFF;
                ObjCallRet(msgDcDrawText, pData->gridDC, &tx, s );
                }
        }

    ObjCallWarn(msgDcSetFillPat, pData->gridDC, (P_ARGS)oldPat);

    return stsOK;
}


STATUS LOCAL XWGDrawLetters( P_INSTANCE_DATA pData )
{
    SYSDC_TEXT_OUTPUT   tx;
    U32                 x, y;
    SCALE               fontScale;
    U8                  str[2];
    P_GRID_ENTRY        pGridEntry;
    STATUS              s;

    ObjCallWarn(msgDcIdentityFont, pData->gridDC, pNull );
    fontScale.x = fontScale.y = FxMakeFixed(((BLOCK_SIZE*3)/4),0);
    ObjCallWarn(msgDcScaleFont, pData->gridDC, &fontScale);

    memset( &tx, 0, sizeof(SYSDC_TEXT_OUTPUT));
    tx.alignChr = sysDcAlignChrBaseline;
    tx.lenText  = 1;
    tx.pText    = str;

    pGridEntry = pData->pEntries;
    for( y=0; y<pData->size; y++ ) {
        tx.cp.y = (pData->size - y -1)*BLOCK_SIZE + BLOCK_LTR_Y_OFF;
        for( x = 0; x<pData->size; x++, pGridEntry++ ) {
            tx.cp.x = x*BLOCK_SIZE + BLOCK_LTR_X_OFF;
            *tx.pText = pGridEntry->letter;
            if ( pGridEntry->status & beWrong ) {
```

```
                ObjCallWarn(msgDcSetForegroundRGB, pData->gridDC,
                                (P_ARGS)sysDcRGBGray33 );
                ObjCallRet(msgDcDrawText, pData->gridDC, &tx, s );
                }
            else if ( pGridEntry->status & beRight ) {
                ObjCallWarn(msgDcSetForegroundRGB, pData->gridDC,
                                (P_ARGS)sysDcRGBBlack );
                ObjCallRet(msgDcDrawText, pData->gridDC, &tx, s );
                }
            else if ( pGridEntry->status & beLetter ) {
                ObjCallWarn(msgDcSetForegroundRGB, pData->gridDC,
                                (P_ARGS)sysDcRGBGray66 );
                ObjCallRet(msgDcDrawText, pData->gridDC, &tx, s );
                }
            }
        }

    return stsOK;
}


MsgHandlerWithTypes(XWordGridRepaint, P_ARGS, P_INSTANCE_DATA ) {
    RECT32      r;
    SIZE32      sz;
    STATUS      s;

    ObjCallRet(msgWinBeginRepaint, pData->gridDC, pNull, s);

    ObjCallWarn(msgDcIdentity, pData->gridDC, pNull );
    sz.w = sz.h = pData->size*BLOCK_SIZE;
    ObjCallRet(msgDcScaleWorld, pData->gridDC, &sz, s );

    ObjCallRet(msgBorderGetBorderRect, self, &r, s );
    ObjCallRet(msgDcLWCtoLUC_RECT32, pData->gridDC, &r, s );
    ObjCallRet(msgDcClipRect, pData->gridDC, &r, s );

    ObjCallWarn(msgDcFillWindow, pData->gridDC, pNull );
    StsRet( XWGDrawGrid( pData ), s );
    StsRet( XWGDrawTemplate( pData ), s );
    StsRet( XWGDrawLetters( pData ), s );
```

```
    ObjCallRet(msgWinEndRepaint, self, Nil(P_ARGS), s );
    return stsOK;
    MsgHandlerParametersNoWarning;
}


MsgHandlerArgType( XWordGridWinSized, P_WIN_METRICS )
{
    INSTANCE_DATA   inst;
    WIN_METRICS     wm;
    STATUS          s;

    ObjCallRet( msgWinGetMetrics, self, &wm, s );

    inst = IDataDeref( pData, INSTANCE_DATA );
    inst.screenBlockSize = wm.bounds.size.w / inst.size;
    ObjectWrite(self, ctx, &inst);

    return stsOK;
    MsgHandlerParametersNoWarning;
}


MsgHandlerWithTypes( XWordGridStartPlayOver, P_ARGS, P_INSTANCE_DATA )
{
    U32     i;
    STATUS  s;

    for ( i=0; i<pData->gridSize; i++ )
        if ( !(pData->pEntries[i].status & beBlack) ) {
            pData->pEntries[i].status &= ~( beLetter | beWrong | beRight );
             pData->pEntries[i].letter = '\0';
             }

    ObjCallRet( msgWinDirtyRect, self, pNull, s );

    return stsOK;
    MsgHandlerParametersNoWarning;
}
```

```
STATUS LOCAL
XWGFindGridPos( P_INSTANCE_DATA pData, P_XY32 pIn, P_XY32 pOut )
{
    pOut->x = pIn->x / pData->screenBlockSize;
    pOut->y = pData->size - pIn->y / pData->screenBlockSize - 1;

    return stsOK;
}


STATUS LOCAL
XWGFilterTransData( P_U8 pStr, U32 len )
{
    U32 i, j;

    for ( i=0, j=0; i<len; i++ )
        if (    (pStr[i] == '\n' )
            ||  (pStr[i] == xltCharUnknownDefault )
            ||  (pStr[i] == '-' )
            ||  ((pStr[i]>='A') && (pStr[i]<='Z')) )
            pStr[j++] = pStr[i];
    pStr[j] = '\0';

    return stsOK;
}


MsgHandlerWithTypes( XWordGridTransWriting, P_ARGS, P_INSTANCE_DATA )
{
    STATUS              s;
    XLATE_DATA          xdata;
    X2STRING            x2sData;
    XLIST_ELEMENT       xe;
    XY32                penLoc;
    RECT32              dr;
    U32                 index;
    P_U8                pStr;

    xdata.heap = osProcessHeapId;
    ObjCallRet(msgSPaperGetXlateData, self, &xdata, s );
```

```
    XList2Text(xdata.pXList);
    XListGet( xdata.pXList, 0, &xe );

    StsRet( XWGFindGridPos(pData,
                             &(((P_XLATE_BDATA)(xe.pData))->box.origin),
                        &penLoc), s );
    ObjCallRet(msgWinDirtyRect,self,&(((P_XLATE_BDATA)(xe.pData))->box),s);

    XList2StringLength( xdata.pXList, &x2sData.count );
    StsRet( OSHeapBlockAlloc(osProcessHeapId, x2sData.count,
                             &x2sData.pString), s );
    XList2String(xdata.pXList, &x2sData );
    StsJmp( XWGFilterTransData(x2sData.pString,x2sData.count), s, Error);

    index = penLoc.x + penLoc.y * pData->size;
    for ( pStr = x2sData.pString; *pStr; pStr++ ) {
       if ( *pStr == '\n' ) {
           index += pData->size - 1;
           penLoc.y++;
           penLoc.x--;
           }
       else {
           if ( !(pData->pEntries[index].status & beBlack) ) {
               pData->pEntries[index].status &=
                                   ~(beLetter|beWrong|beRight);
               if ( *pStr == '-' )
                   pData->pEntries[index].letter = 0;
               else {
                   pData->pEntries[index].letter = *pStr;
                   pData->pEntries[index].status |= beLetter;
                   }
               }
           StsJmp( XWGGridPosToRect( pData, &penLoc, &dr ), s, Error );
           ObjCallJmp( msgWinDirtyRect, self, &dr, s, Error );
           index++;
           penLoc.x++;
           }
       }

   s = stsOK;
Error:
```

```
    OSHeapBlockFree(x2sData.pString);
    XListFree(xdata.pXList);

    return s;
    MsgHandlerParametersNoWarning;
}


STATUS ClsXWordGridInit(void)
{
    CLASS_NEW c;
    STATUS    s;

    ObjCallRet(msgNewDefaults, clsClass, &c, s);
    c.object.uid        = clsXWordGrid;
    c.cls.pMsg          = clsXWordGridTable;
    c.cls.ancestor      = clsSPaper;
    c.cls.size          = SizeOf(INSTANCE_DATA);
    c.cls.newArgsSize   = SizeOf(XWORDGRID_NEW);
    ObjCallRet(msgNew, clsClass, &c, s );

    return stsOK;
}
```

method.tbl

```
#ifndef CLSMGR_INCLUDED
#include <clsmgr.h>
#endif

#ifndef APP_INCLUDED
#include <app.h>
#endif

#ifndef WIN_INCLUDED
#include <win.h>
#endif

#ifndef IMPORT_INCLUDED
#include <import.h>
#endif

#ifndef CLAYOUT_INCLUDED
#include <clayout.h>
#endif

#ifndef XLATE_INCLUDED
#include <xlate.h>
#endif

#ifndef XWORDAPP_INCLUDED
#include <xwordapp.h>
#endif

#ifndef XWRDDATA_INCLUDED
#include <xwrddata.h>
#endif

#ifndef XWRDVIEW_INCLUDED
#include <xwrdview.h>
#endif

#ifndef XWRDGRID_INCLUDED
#include <xwrdgrid.h>
#endif
```

```
MSG_INFO clsXWordAppMethods[] = {
    msgImportQuery,          "XWordAppImportQuery",  objClassMessage,
    msgImport,               "XWordAppImport",       0,
    msgAppInit,              "XWordAppAppInit",      objCallAncestorBefore,
    msgRestore,              "XWordAppRestore",      objCallAncestorBefore,
    msgXWordAppStartOver,    "XWordAppStartOver",    0,
    msgXWordAppShowSoln,     "XWordAppShowSoln",     0,
    msgXWordAppSetClueTap,   "XWordAppSetClueTap",   0,
    msgXWordAppDoCheck,      "XWordAppDoCheck",      0,
    0
};


MSG_INFO clsXWordDataMethods[] = {
    msgNewDefaults,      "XWordDataNewDefaults", objCallAncestorBefore,
    msgInit,             "XWordDataInit",        objCallAncestorBefore,
    msgFree,             "XWordDataFree",        objCallAncestorAfter,
    msgSave,             "XWordDataSave",        objCallAncestorBefore,
    msgRestore,          "XWordDataRestore",     objCallAncestorBefore,
    msgXWordDataIsXWordFile,       "XWordDataIsXWordFile",       objClassMessage,
    msgXWordDataGetInfo,           "XWordDataGetInfo",           0,
    msgXWordDataGetLetters,        "XWordDataGetLetters",        0,
    msgXWordDataGetAcrossCount,    "XWordDataGetAcrossCount",    0,
    msgXWordDataGetDownCount,      "XWordDataGetDownCount",      0,
    msgXWordDataGetAcrossWord,     "XWordDataGetAcrossWord",     0,
    msgXWordDataGetDownWord,       "XWordDataGetDownWord",       0,
    0
};


MSG_INFO clsXWordViewMethods[] = {
    msgNewDefaults, "XWordViewNewDefaults",          objCallAncestorBefore,
    msgInit,        "XWordViewInit",                 objCallAncestorBefore,
    msgSave,        "XWordViewSave",                 objCallAncestorBefore,
    msgRestore,     "XWordViewRestore",              objCallAncestorBefore,
    msgCstmLayoutGetChildSpec,"XWordViewCLGetChildSpec",
        objCallAncestorBefore,
    msgXWordViewStartPlayOver,   "XWordViewStartPlayOver",   0,
    msgXWordViewShowSoln,        "XWordViewShowSoln",        0,
```

```
    msgXWordViewClueTapNothing, "XWordViewClueTapNothing",  0,
    msgXWordViewClueTapStrikeOut,"XWordViewClueTapStrikeOut",0,
    msgXWordViewCheckPuzzle,    "XWordViewCheckPuzzle",     0,
    msgXWordViewCheckLetters,   "XWordViewCheckLetters",    0,
    msgXWordViewCheckWords,     "XWordViewCheckWords",      0,
    0
};


MSG_INFO clsXWordGridMethods[] = {
    msgNewDefaults, "XWordGridNewDefaults", objCallAncestorBefore,
    msgInit,        "XWordGridInit",        0,
    msgFree,        "XWordGridFree",        objCallAncestorAfter,
    msgSave,        "XWordGridSave",        objCallAncestorBefore,
    msgRestore,     "XWordGridRestore",     objCallAncestorBefore,
    msgWinRepaint,  "XWordGridRepaint",     objCallAncestorBefore,
    msgWinSized,    "XWordGridWinSized",    objCallAncestorBefore,
    msgXlateCompleted,"XWordGridTransWriting",objCallAncestorBefore,
    msgXWordGridStartPlayOver,  "XWordGridStartPlayOver",   0,
    msgXWordGridGetLetters,     "XWordGridGetLetters",      0,
    msgXWordGridSetLetters,     "XWordGridSetLetters",      0,
    msgXWordGridSetOkLetters,   "XWordGridSetOkLetters",    0,
    0
};


CLASS_INFO classInfo[] = {
    "clsXWordAppTable",     clsXWordAppMethods,     0,
    "clsXWordDataTable",    clsXWordDataMethods,    0,
    "clsXWordViewTable",    clsXWordViewMethods,    0,
    "clsXWordGridTable",    clsXWordGridMethods,    0,
    0
};
```

xwrdclue.h

```
#ifndef XWRDCLUE_INCLUDED
#define XWRDCLUE_INCLUDED

#ifndef GO_INCLUDED
#include <go.h>
#endif

#ifndef CLSMGR_INCLUDED
#include <clsmgr.h>
#endif

#ifndef CLAYOUT_INCLUDED
#include <clayout.h>
#endif

#define clsXWordClueList    MakeGlobalWKN(4153,1)

#define msgXWordClueStartPlayOver       MakeMsg( clsXWordClueList, 1 )
#define msgXWordClueClueTapNothing      MakeMsg( clsXWordClueList, 2 )
#define msgXWordClueClueTapStrikeOut    MakeMsg( clsXWordClueList, 3 )

typedef struct {
    U32         size;
    P_STRING    pTitle;
    OBJECT      clueList;
} XWORDCLUE_NEW_ONLY, *P_XWORDCLUE_NEW_ONLY;

#define xwordclueNewFields \
    customLayoutNewFields \
    XWORDCLUE_NEW_ONLY xwclue;

typedef struct XWORDCLUE_NEW {
    xwordclueNewFields
} XWORDCLUE_NEW, *P_XWORDCLUE_NEW;

#endif
```

xwrdclue.c

```
#ifndef GO_INCLUDED
#include <go.h>
#endif

#ifndef WIN_INCLUDED
#include <win.h>
#endif

#ifndef STROBJ_INCLUDED
#include <strobj.h>
#endif

#ifndef LIST_INCLUDED
#include <list.h>
#endif

#ifndef FS_INCLUDED
#include <fs.h>
#endif

#ifndef CLAYOUT_INCLUDED
#include <clayout.h>
#endif

#ifndef LABEL_INCLUDED
#include <label.h>
#endif

#ifndef LISTBOX_INCLUDED
#include <listbox.h>
#endif

#ifndef GWIN_INCLUDED
#include <gwin.h>
#endif

#ifndef XGESTURE_INCLUDED
#include <xgesture.h>
#endif
```

```
#ifndef XWRDCLUE_INCLUDED
#include <xwrdclue.h>
#endif

#ifndef DEBUG_INCLUDED
#include <debug.h>
#endif

#include <xclu_mth.h>
#include <stdio.h>
#include <string.h>


#define titleWinTag MakeTag( clsXWordClueList, 1 )
#define listWinTag  MakeTag( clsXWordClueList, 2 )

#define MODE_NOTHING    0
#define MODE_STRIKEOUT  1

#define TEXT_SIZE   12

typedef struct INSTANCE_DATA {
    U8      clueTapMode;
    U16     clueCnt;
    OBJECT  titleWin;
    OBJECT  listWin;
} INSTANCE_DATA, *P_INSTANCE_DATA;


STATUS LOCAL
XWCCreateListTitle( P_U8 pTitle, TAG tag, P_OBJECT pTitleWin )
{
    LABEL_NEW   ln;
    STATUS      s;

    ObjCallRet(msgNewDefaults, clsLabel, &ln, s);
    ln.win.tag                    = tag;
    ln.label.style.scaleUnits     = bsUnitsFitWindowProper;
    ln.label.style.xAlignment     = lsAlignCenter;
    ln.label.pString              = pTitle;
    ln.border.style.edge          = bsEdgeAll;
    ObjCallRet(msgNew, clsLabel, &ln, s);
```

```
    *pTitleWin = ln.object.uid;

    return stsOK;
}


STATUS LOCAL
XWCCreateListBox( OBJECT self, OBJECT list, TAG tag,
                            P_OBJECT pListBox)
{
    LIST_BOX_NEW    lbn;
    LIST_BOX_ENTRY  lbe;
    LABEL_NEW       ln;
    LIST_ENTRY      le;
    STATUS          s;
    U16             cnt;
    U32             i;

    ObjCallRet( msgListNumItems, list, &cnt, s );

    ObjCallRet(msgNewDefaults, clsListBox, &lbn, s);
    lbn.win.tag                   = tag;
    lbn.border.style.edge         = bsEdgeAll;
    lbn.listBox.client            = self;
    lbn.listBox.nEntries          = cnt;
    lbn.listBox.nEntriesToView    = cnt;
    ObjCallRet(msgNew, clsListBox, &lbn, s);
    *pListBox = lbn.object.uid;

    memset( &lbe, 0, SizeOf(LIST_BOX_ENTRY) );
    lbe.listBox = *pListBox;
    lbe.freeEntry   = lbFreeDataWhenDestroyed;
    for ( i=0; i<cnt; i++ ) {
        lbe.position = le.position = i;
        ObjCallRet( msgListGetItem, list, &le, s );
        ObjCallRet( msgNewDefaults, clsLabel, &ln, s );
        ln.border.style.edge        = bsEdgeNone;
        ObjCallRet( msgStrObjGetStr, le.item, &ln.label.pString, s );
        ObjCallRet( msgNew, clsLabel, &ln, s );
        lbe.win = ln.object.uid;
        ObjCallRet(msgListBoxInsertEntry, *pListBox, &lbe, s );
        }
```

```
    return stsOK;
}

MsgHandlerArgType(XWordClueInit, P_XWORDCLUE_NEW)
{
    INSTANCE_DATA   inst;
    WIN_METRICS     wm;
    LIST_FREE       lf;
    STATUS          s;

    inst.clueTapMode = MODE_NOTHING;
    ObjCallRet(msgListNumItems,pArgs->xwclue.clueList,&inst.clueCnt,s );

    StsRet( XWCCreateListTitle( pArgs->xwclue.pTitle, titleWinTag,
                                    &inst.titleWin ), s );
    StsRet( XWCCreateListBox( self, pArgs->xwclue.clueList, listWinTag,
                                    &inst.listWin ), s );

    lf.key = (OBJ_KEY)clsList;
    lf.mode = listFreeItemsAsObjects;
    ObjCallWarn( msgListFree, pArgs->xwclue.clueList, &lf );

    ObjectWrite(self, ctx, &inst );

    wm.parent = self;
    wm.options = wsPosTop;
    ObjCallRet( msgWinInsert, inst.titleWin, &wm, s );
    wm.parent = self;
    wm.options = wsPosTop;
    ObjCallRet( msgWinInsert, inst.listWin, &wm, s );

    return stsOK;
    MsgHandlerParametersNoWarning;
}


MsgHandlerWithTypes(XWordClueSave, P_OBJ_SAVE, P_INSTANCE_DATA)
{
    STREAM_READ_WRITE   fsWrite;
    STATUS              s;
```

```
    fsWrite.numBytes = SizeOf(U16);
    fsWrite.pBuf     = &(pData->clueCnt);
    ObjCallRet(msgStreamWrite, pArgs->file, &fsWrite, s );

    fsWrite.numBytes = SizeOf(U8);
    fsWrite.pBuf     = &(pData->clueTapMode);
    ObjCallRet(msgStreamWrite, pArgs->file, &fsWrite, s );

    return stsOK;
    MsgHandlerParametersNoWarning;
}


MsgHandlerArgType(XWordClueRestore, P_OBJ_RESTORE)
{
    INSTANCE_DATA        inst;
    LIST_BOX_METRICS     lbm;
    STREAM_READ_WRITE    fsRead;
    STATUS               s;

    fsRead.numBytes = SizeOf(U16);
    fsRead.pBuf      = &inst.clueCnt;
    ObjCallRet(msgStreamRead, pArgs->file, &fsRead, s );

    fsRead.numBytes = SizeOf(U8);
    fsRead.pBuf      = &inst.clueTapMode;
    ObjCallRet(msgStreamRead, pArgs->file, &fsRead, s );

    inst.titleWin   =
       (WIN)ObjectCall(msgWinFindTag,self,(P_ARGS)titleWinTag);
    inst.listWin    =
       (WIN)ObjectCall(msgWinFindTag,self,(P_ARGS)listWinTag);

    ObjectWrite(self, ctx, &inst );

    ObjCallRet( msgListBoxGetMetrics, inst.listWin, &lbm, s );
    lbm.client = self;
    ObjCallRet( msgListBoxSetMetrics, inst.listWin, &lbm, s );

    return stsOK;
    MsgHandlerParametersNoWarning;
}
```

```
MsgHandlerWithTypes(XWordClueCLGetChildSpec, P_CSTM_LAYOUT_CHILD_SPEC,
                    P_INSTANCE_DATA)
{
    if ( pArgs->child == pData->titleWin ) {
        pArgs->metrics.w.constraint = clSameAs;
        pArgs->metrics.h.constraint = clAbsolute;
        pArgs->metrics.h.value      = TEXT_SIZE;
        pArgs->metrics.x.constraint =
                ClAlign(clMinEdge, clSameAs, clMinEdge);
        pArgs->metrics.y.constraint =
                ClAlign(clMaxEdge, clSameAs, clMaxEdge);
        }
    else if ( pArgs->child == pData->listWin ) {
        pArgs->metrics.w.constraint = clSameAs;
        pArgs->metrics.h.relWin     = pData->titleWin;
        pArgs->metrics.h.constraint = ClExtend(clSameAs, clMinEdge);
        pArgs->metrics.x.constraint =
                ClAlign(clMinEdge, clSameAs, clMinEdge);
        pArgs->metrics.y.constraint =
                ClAlign(clMinEdge, clSameAs, clMinEdge);
        }

    return stsOK;
    MsgHandlerParametersNoWarning;
}


STATUS LOCAL
XWCSetClueEntryStyle( OBJECT clueEnt, U8 style )
{
    LABEL_STYLE ls;
    STATUS      s;

    ObjCallRet( msgLabelGetStyle, clueEnt, &ls, s );
    ls.strikeout = (style == MODE_STRIKEOUT) ? 1 : 0;
    ObjCallRet( msgLabelSetStyle, clueEnt, &ls, s );
    ObjCallRet( msgWinDirtyRect, clueEnt, pNull, s );

    return stsOK;
}
```

```
MsgHandlerWithTypes( XWordClueStartPlayOver, P_ARGS, P_INSTANCE_DATA )
{
    LIST_BOX_ENTRY  lbe;
    U32             i;
    STATUS          s;

    for ( i=0; i<pData->clueCnt; i++ ) {
        lbe.listBox = pData->listWin;
        lbe.position = i;
        ObjCallRet( msgListBoxGetEntry, pData->listWin, &lbe, s );
        if ( lbe.data == MODE_STRIKEOUT ) {
            StsRet( XWCSetClueEntryStyle( lbe.win, MODE_NOTHING ), s );
            lbe.data = MODE_NOTHING;
            ObjCallRet( msgListBoxSetEntry, pData->listWin, &lbe, s );
            }
        }

    return stsOK;
    MsgHandlerParametersNoWarning;
}

MsgHandlerArgType( XWordClueClueTapNothing, P_ARGS )
{
    INSTANCE_DATA   inst;
    LIST_BOX_ENTRY  lbe;
    U32             i;
    STATUS          s;

    inst = IDataDeref( pData, INSTANCE_DATA );
    inst.clueTapMode = MODE_NOTHING;
    ObjectWrite(self, ctx, &inst );

    for ( i=0; i<inst.clueCnt; i++ ) {
        lbe.listBox = inst.listWin;
        lbe.position = i;
        ObjCallRet( msgListBoxGetEntry, inst.listWin, &lbe, s );
        if ( lbe.data == MODE_STRIKEOUT )
            StsRet( XWCSetClueEntryStyle( lbe.win, MODE_NOTHING ), s);
        }

    return stsOK;
```

```
    MsgHandlerParametersNoWarning;
}


MsgHandlerArgType( XWordClueClueTapStrikeOut, P_ARGS )
{
    INSTANCE_DATA   inst;
    LIST_BOX_ENTRY  lbe;
    U32             i;
    STATUS          s;

    inst = IDataDeref( pData, INSTANCE_DATA );
    inst.clueTapMode = MODE_STRIKEOUT;
    ObjectWrite(self, ctx, &inst );

    for ( i=0; i<inst.clueCnt; i++ ) {
        lbe.listBox = inst.listWin;
        lbe.position = i;
        ObjCallRet( msgListBoxGetEntry, inst.listWin, &lbe, s );
        if ( lbe.data == MODE_STRIKEOUT )
            StsRet( XWCSetClueEntryStyle( lbe.win, MODE_STRIKEOUT ), s );
        }

    return stsOK;
    MsgHandlerParametersNoWarning;
}


MsgHandlerWithTypes(XWordClueEntryGesture, P_LIST_BOX_ENTRY,
P_INSTANCE_DATA)
{
    STATUS s;

    if ( !( ((P_GWIN_GESTURE)(pArgs->arg))->msg == xgs1Tap ) )
        return stsOK;

    if ( pData->clueTapMode == MODE_NOTHING )
        return stsOK;

    pArgs->data = (P_UNKNOWN) ( (pArgs->data == MODE_NOTHING)
                                ? MODE_STRIKEOUT : MODE_NOTHING );
    StsRet( XWCSetClueEntryStyle( pArgs->win, (U8)pArgs->data ), s );
```

```
    ObjCallRet( msgListBoxSetEntry, pArgs->listBox, pArgs, s );

    ObjCallRet( msgWinDirtyRect, pArgs->win, pNull, s );

    return stsOK;
    MsgHandlerParametersNoWarning;
}


STATUS ClsXWordClueListInit(void)
{
    CLASS_NEW c;
    STATUS    s;

    ObjCallRet(msgNewDefaults, clsClass, &c, s );
    c.object.uid        = clsXWordClueList;
    c.cls.pMsg          = clsXWordClueListTable;
    c.cls.ancestor      = clsCustomLayout;
    c.cls.size          = SizeOf(INSTANCE_DATA);
    c.cls.newArgsSize   = SizeOf(XWORDCLUE_NEW);
    ObjCallRet(msgNew, clsClass, &c, s );

    return stsOK;
}


STATUS EXPORTED DLLMain (void)
{
    STATUS  s;

    StsRet(ClsXWordClueListInit(), s);

    return stsOK;
}
```

dll.lbc

```
++DLLMAIN.'pip-xwrdclue-v1(0)'
```

xclu_mth.tbl

```
#ifndef CLSMGR_INCLUDED
#include <clsmgr.h>
#endif

#ifndef CLAYOUT_INCLUDED
#include <clayout.h>
#endif

#ifndef LISTBOX_INCLUDED
#include <listbox.h>
#endif

#ifndef XWRDCLUE_INCLUDED
#include <xwrdclue.h>
#endif


MSG_INFO clsXWordClueListMethods[] = {
    msgInit,         "XWordClueInit",         objCallAncestorBefore,
    msgSave,         "XWordClueSave",         objCallAncestorBefore,
    msgRestore,      "XWordClueRestore",      objCallAncestorBefore,
    msgCstmLayoutGetChildSpec,"XWordClueCLGetChildSpec",
        objCallAncestorBefore,
    msgXWordClueStartPlayOver,  "XWordClueStartPlayOver",   0,
    msgXWordClueClueTapNothing, "XWordClueClueTapNothing",  0,
    msgXWordClueClueTapStrikeOut,"XWordClueClueTapStrikeOut",   0,
    msgListBoxEntryGesture, "XWordClueEntryGesture",        0,
    0
};


CLASS_INFO classInfo[] = {
    "clsXWordClueListTable",clsXWordClueListMethods,0,
    0
};
```

Index

\penpoint\app 46
\\boot\penpoint\boot\dll 45
_NEW 37
__FILE__ 56
__LINE__ 56
!strcmp() 253
#define directive 55
#ifdef FILENAME_INCLUDED 51
#include directives 79
? help gesture 121

absolute adjustments 144
absolute pixel coordinates 9
absolute size 265
abstract superclass 120
abstractions 14
accessible 153
Accessories Menu 58, 101-102
accessory 102, 164
accumulator
  changes 148
  access methods 107
  child window 133
  display 139, 140, 148, 149
  limits 106
  view 143
acetate
  example 116
  layer 155
across clue count 196
acrossClues 242
acrossWinTag 241
across 227
activated state 47-48, 68-69, 72
add observer 91
addressing scheme 233
add stroke 156
additional functionality 230
administer value 25, 160, 162
administered value (defined) 24
agents 235
AlignChildren() 165, 178
alignment 165, 249
allowEmbedding 102
alphanumerics 187
analog data 157
analysis 89
anappdb.ini 62
ancestor class 50, 119, 121, 281, 285
ancestor 20, 21, 36, 39, 40, 70, 71, 72, 76, 80, 101
animation team 116
anObject 28
ANSI-C 23
Apollo 66
app.h 51
app.ini 45
Apple Macintosh 1, 66
Apple 66, 233
Application's Menu 208
Application Framework, 7-8, 11, 13, 22, 65, 67-68, 71, 73, 78-79, 88-89, 91, 94, 115, 127, 166
  framework API 51
Application Menu, default 8, 210
Application Menu 200, 210, 237
Application Programmers' Interface (API) 7, 10, 156, 234, 299
application
  class, 7, 8, 22, 31, 44, 47, 52, 68, 88, 90, 99, 161, 175
  code level 60
  data 69
  de-installation 54
  developer 65
  directory 71
  distribution 45
  domain 90
  header file 46
  heap 182
  identification information 54
  initialization 164, 206
  installation 45, 48, 54
  instance 66
  lifecycle 18, 67, 78
  lifetime 7
  manager API 51
  manager 48-50, 70, 101, 206
  model 95, 237
  modules 46
  monitor 48, 57, 102
  name 54
  object 47, 71
  oriented OS 199
  programmer 4
  registration 99
  source file 46-47
application-specific data 256
applications 22, 44, 66, 94, 127
AppMain() 48
appmgr.h 51
AppMonitorMain() 48
AppMonitor 102
apptag.h 205
app 188
APP_METRICS 83
APP_MGR_NEW (defined) 36
APP_MGR_NEW 49, 101
arcs 273
ASCII strings 30, 57, 176, 182, 193, 196, 200, 202
assembly level 60
assembly metaphor 21
assembly process 21, 233
ASSERT() 55-56
asynchronous environment 147
asynchronous messages 21
attribute field 34
attributes 37, 273
auto layout 124
auto sizing 124
automatic filing 73
automatic font selection 273
auxiliary storage 7
background color 273
background task 154
battery monitor 23
behavior request 31
behavior 14, 16, 21, 28, 30, 31, 90, 14
bezier curves 273
binding (defined) 18
binding options 18
bit-mapped display 6
bitmap graphic display 66
bitmap 116, 273
black letter 194
blocked 201, 214
BLOCK_SIZE 286
\\boot\penpoint\boot\dll 45
boot 46
boot procedure 43
boot sequence 44
boot time 45
boot volume 45
boot.dlc 44, 45, 60, 62, 86
border 120-121
  area 120
  object 120
  rectangle 286
  type 119
Border Layer 115
boundaries 119
boundary limits 19
box-based calculator 159, 160-161, 173, 177
BoxCalc Application 161
  methods 164-172
boxed input panels 154
breakpoints 60-61, 63, 87
bsContactLockOn 201
bsEdgeAll 135, 137
bsEdgeNone 134, 135
bsInkGray33 134-135
bsJoinSquare 135, 137
bsShadowNone 135, 137
bsUnitsFitWindowProper 133, 135, 137
BuildEntryWin() 180
BuildNumberWin() 168
BuildOperatorWin() 169
BuildResultsWin() 167-168, 180
built-in protocol 119
built-in translator 183
button-based
  interface 95
  calculator 153
buttons 8, 13, 121, 201

C++ 14, 67
C-based
  Objects 67
  programming 57
calcapp.c 102, 127
CalcAppAppInit 100, 127
calcbtvw.c 129-130
calcbtvw.h 129-130
CalcBtVwAccmChanged 148
CalcBtVwAccmError 148

CalcBtVwCLGetChildSpec 143
CalcBtVwDigit 145
CalcBtVwFnc 146
CalcBtVwFree 140
CalcBtVwInit 137-138, 140
calcbtvwNewFields macro 130
CalcBtVwRestore 141-143
CalcBtVwSave 141
CalcBtVwSetDataObject 140
CALCBTVW_NEW 130
CalcEndSetAccm 110
calceng.c 105-106, 110, 111
calceng.h 104-105
CalcEngAdd 107-108
CalcEngClr 107-108
CalcEngDiv 107, 109
CalcEngGetAccm 110
CalcEngInit 110
CalcEngMul 107, 109
CalcEngReadAccm 107
CalcEngRestore 111
CalcEngSave 111
CalcEngSetAccm 107
CalcEngSub 107-109
CALCENG_NEW 105
CALCENG_VAL 105
CalcSPaperInit 185
CalcSPaperInputEvent 188
CalcSPaperReset 188
CalcSPaperRestore 187
CalcSPaperSave 187
calculator application 96, 99, 125
  application class 137
  button view 124, 128, 139-140
  engine class 95, 98, 105, 103-104, 107, 110, 115, 128, 139, 147-148
  error statuses 132
  example 89-90, 95
  scratch pad 186
  scratch paper 179, 184
  view 126, 145
  keypad 135
Carr, Robert 4
carriage return 200
cartoonists 116
case statement 151
categories 65
centered 121
character
  boxes 158
  data 154
  recognition software 9
  set 159
check marks 200
check puzzle note 197
Check Submenu 193-195, 197, 211-213
child embedded applications 102
child window, 167
  alignment 165
  layout 123, 143
  processing loop 140, 117-119, 121-122, 126-128, 141, 178, 262
children 123
ChkPuzzleCmdBar 213
ChkPuzzleTb 213
choice selection 122
chords 273
clAbsolute 265
CLAlign() 123-124, 143-144
Class Manager 13, 22-24 28, 31-35, 37, 47, 76, 83, 111, 129, 137, 163, 176, 185, 217, 239, 242, 275, 279
class(es) 46, 90, 154
  decomposition 151
  hierarchy 21-22
  initialization 48
  interface 37
  library 21, 73, 93, 122
  manager (defined) 23
  message 199
  method 221
  name 93-95
  registration 47 50, 76, 97, 130, 137, 163, 185, 221, 242, 279
  Universal Id 24-25, 162
Class-Responsibility-Collaborator Approach 93-95, 97
  Cards 93, 95
  description 99
  Table 98
classifying 90
CLASSNAME_NEW 13, 16-25, 30, 34, 37, 66, 104, 198
CLASS_INFO 34-35, 53, 76
CLASS_NEW 36
clCenterEdge 124
clcspapr.h 176, 183
clearing 188
ClExtend() 143-145
client 123, 158, 179, 211, 299
  notification 120, 158-159, 169-170, 180
  object 120, 167, 169
  window 127, 208
clientNotifyModified 170
clipping rectangle 286
clipping 117, 119, 273
clMaxEdge 124, 143-144
clMinEdge 124, 144-145
clock application 4
closed figures 273
clPctOf 143-144
clsAccmDisplay 151
clSameAs 124, 143, 145
clsAppMgr 49-51, 68, 74, 84
clsApp 22, 44, 51, 68, 71, 79, 97, 99, 100, 164, 198
clsBorder 98, 115, 117, 119-121
ClsBoxCalcAppInit() 163, 176
clsBoxCalcAppMethods 173
clsBoxCalcApp 159, 162, 164
clsButton 93, 97-98, 117-118, 122, 200, 213
ClsCalcAppInit() 101
clsCalcAppMethods 103, 150
clsCalcApp 95, 97, 99, 101-103, 127, 149
ClsCalcBtVw() 130
ClsCalcBtVwInit() 137
clsCalcBtVwMethods 150
clsCalcBtVw 95, 97-99, 118, 126-131, 133-134, 136-137, 140-141, 143-145, 148-149, 151
ClsCalcEngInit() 111
clsCalcEngMethods 112, 150
clsCalcEng 95, 97-99, 103-106, 108, 110-111, 149
ClsCalcSPaperInit() 185
clsCalcSPaperMethods 189
clsCalcSPaper 183-185
clsChoice 123, 201, 209-210
ClsClassNameInit() 47
Clsclassnameinit() 48
clsClass 20, 22, 25, 36, 39
ClsCoinAppInit() 84
clsCoinAppMethods 78
clsCoinApp 79-80, 86
clsControl 117, 120-121, 180, 200
clsCustomLayout 98, 117, 122-123, 128, 143, 161, 165, 198, 247, 256, 265
clsDataField 157
ClsDemoAppInit() 48, 74
clsDemoAppMethods 53, 75
clsDemoAppTable 48
clsDemoApp 50, 52-53
clsDrwCtx 273
clsEmbeddedWin 98
clsField 157-159, 161, 169
clsFixedField 157
clsFontListBox 256
clsGWin 98, 120
clsHWXCalcAppMethods 189
clsHWXCalcApp 173, 175, 177
ClsHWXCalcInit() 177
clsIconWin 123
clsImport 199, 203, 205
clsIntegerField 157, 159, 158
clsKeypad 151
clsLabel keypad 126
clsLabel 25-26, 37, 82-83, 97-98, 117-178, 121-122, 133, 151, 157, 168, 213, 256, 263
clsLightFixture INSTANCE_DATA 32
ClsLightFixtureInit() 35, 39
clsLightFixtureMethods 34
clsLightFixtureTable 36
clsLightFixture 24, 32, 37-38, 40-41
clsListBox 256, 259, 263, 268
clsList 202, 219, 228
clsMenuButton 200, 209
clsMenu 115, 123, 200, 205, 211
ClsMsgToString() 31
clsNote 201, 205, 213
clsObject 20, 22, 25, 38-39, 97-97, 103, 117, 156, 165
clsPctOf 165
clsPopUpChoice 22
clsResFile 98, 103
clsSameAs 144
clsScribble 156
clsSPaper 158, 174, 176, 183, 198, 273, 289
clsStringListBox 256
clsString 202, 219, 228
clsSysDrwCtx 282-283
clsTabBar 123
clsTableLayout 117, 122-123
clsTextField 115, 157
clsText 115
ClsTimedLightFixtureInit() 38
clsTimedLightFixtureMethods 39
clsTimedLightFixture 24, 41

clsTkTable 97-98, 117-118, 123, 126, 128, 133-136, 151, 200-201
clsView 98, 117, 126-128, 130, 132, 139-140, 198, 247
clsWhatever 30
clsWin 22, 98, 117-118, 127
clsXGesture (defined) 159
clsXText (defined) 159
clsXText 186, 281
clsXWord (defined) 159
ClsXWordAppInit() 206
clsXWordAppMethods 216, 296
clsXWordApp 191, 198-199, 203-204, 206, 208, 215, 217, 221, 239-240, 242, 251, 274-275, 279
ClsXwordClueListInit() 260
clsXWordClueListMethods 269
clsXWordClueList 198, 237, 245, 256-261, 263, 265-266, 268-269
clsXWordClue 203, 234, 299
ClsXWordDataInit() 217, 221
clsXWordDataMethods 230, 297
clsXWordData 191, 198-199, 215-217, 219-222, 224, 226, 228, 237, 242, 274
ClsXWordGridInit() 275, 279
clsXWordGridMethods 295, 298
clsXWordGrid 198, 237, 245, 273-276, 278-280, 283, 289, 293, 295, 299
ClsXWordViewInit() 239, 242
clsXWordViewMethods 255, 297
clsXWordView 198, 208, 234-235, 237-242, 247, 250-253, 270, 274, 299
ClsYourClassNameInit() 35
clue count 196
clue list view class 256
clue list 193-194, 227, 251
clueCnt 261, 263
clueList 245
clueTapMode 260, 263
code 20, 87
  bulk 7
  reuse 65, 67
  sharing 7
coding 89
coin app 78-79, 83, 85-86, 88
coinapp class 78
coinapp.c 78-79, 85
coinapp.h 78, 85
CoinApp methods 80-84, 87-88
COIN_STATUS 80
collaborate 93-94, 103
collections of controls 122
color capabilities 9
color support 273
compatibility layer 200
compile time 54-55
compiler manual 64
compiler 18, 32, 64, 79, 1909
compiling 51, 62, 67, 214
completion 214
complex style sheets 122
complex views 123
components 6-7, 11, 19, 21, 31, 56, 90, 93, 95, 115-117, 119, 123, 153-154, 159, 167, 180, 190, 199-200, 233-234, 236, 241, 256, 262, 299
  hierarchy 270
  library 94, 115
  packaging 236
  window tags 259
  windows 244, 263
composite object 158, 198
computation 170
computeValue() 183
computing 171, 181-182
confirmDelete 102
conflicts 202
connectivity 8, 10
consistency of operation 65
constants 278
constraint metrics 144
constraints 128
construction 21, 31, 97, 116
constructs 217
consumer 15-17, 31, 37, 140
context 63, 80, 86
  DB 62
contextual information 9
controls 119, 121, 159
  behavior 115
  client 136
  collections 122
  data 136
  management 136
  messages 121
  model 120
  objects 72
  value 169
  style 256
controller behavior 147
controller object 90-91, 123
controlling play 250, 266
coordinate 249
  space 286
  systems 271-272
  transformation 272
correct spelling 229
cost of reuse 95
coupling 22, 270
Create Menu 5
create mode 299
created state 68-69, 199
creating instances 25
creating puzzles 196
creation 215
cross development environment 43
cross development 55, 61
crossword
  application 194, 203
  model 199, 216, 239
crossword puzzle application 191-193, 196, 200, 234, 238, 241, 250, 270-271, 274, 289, 295, 299
  class 199
  documents 198, 206
  implementation 198
  view 199, 240
CstmLayoutSpecInit() 165
CSTM_LAYOUT_CHILD_SPEC 144
ctx context 32, 63, 80
current
  call 62
  scope 62
  state 63
  task 62
  version number 30
cursor 155
custom component 19, 119
custom layout 115, 122, 128, 166
custom translator 159, 185
customization 21, 234
cutting and pasting 7
cutting data 299
CVBuildAccmDisplay() 133-134
CVBuildKeypad() 134
cyclical loop 145

dark gray letter 194
data 16
  abstraction 14
  corruption 8
  entry panels 156
  entry tool 9
  file 73
  lists 201
  object 126, 132, 141, 143, 147, 244
  security 20
  structure 37
  transfer 10
DB source level symbolic debugger 60-61, 63, 88
  commands
    ? 61, 63
    bc 88
    bl 88
    ctx 62, 87
    fs 61
    g 64, 87
    srcdir 62, 87
    sym 62, 87
    tl 63, 86
    v 87
    P,p 64
    T,t 64
  compiling 62
  contexts 62
  db.ini 62
  debugger 86
  linking 62
  manual 61
  using 62
Dbg() 55
DbgFlag() 56
DbgFlagGet() 57
de-installation 54, 70, 102
de-reference data 20, 108, 141, 146-147
deallocate storage 182
dealloc_mem 30
debug command 61, 63
debugging CoinApp 86
debugging context 86
debugging DemoApp 63
debug.h 51, 55, 57
Debugf() 57, 61
DebugFlagGet() 58
DebugFlagSet() 58
DEBUG 55-57
debugging 51, 54-55, 57, 63
  flags 55, 55, 61
  functions 57

information 26-27
log 70, 76
macros 55
messages 55
objects 62
output stream 26, 60
session 86
tools 31, 60
decoration 121, 200
decoupling 270
default
action 60
Application Menu 8
application 44
crossword puzzle 244
data structure 25
file 62
initialization structure 186
values 26
deferred
binding 13, 18-19, 23, 270
I/O 10
messaging 29
#define directive 55
definition file 38, 52-53, 78, 105
definitions 161, 175, 184
degree of freedom 155
delete stroke 156
demo application 48, 50, 52, 54, 62, 70, 73, 76
DemoApp
debugging 63
method.h 53
methods 70-73
method.tbl 48
demoapp.c 52, 70
demoapp.exe 54
demoapp.h 51-52
discrete selection 157
description 94
design 21, 89, 97, 194, 198
by construction 97
process 19
design-to-coding ratio 89
destroy 202, 263, 299
destructive testing 19
developing components 19
development
organization 19
time 14
device drivers 155
Device Units (DU4) 272
dialog boxes 156
direct manipulation 271, 273
direct mapping 272
directory 68-69
dirty bit 172
dirty windows 116
disassemble code 60
disassembled instructions 63
dismiss note 196
dispatch loop 29, 48
dispatch table 53
display 69, 233
components 167, 180
devices 8, 22, 271, 273
window 126, 289
Display Manager 66
dispOrientation 242, 248
DllMain() 45, 237, 260
document 5, 7, 47-48, 67-68, 70, 76-77, 100, 111, 125, 199, 206, 222, 230, 237, 247, 299
creation 215, 230
lifecycle 66-69, 77-78, 126, 198, 200
management 88
oriented OS 199
removal 69
transfer 234
process 69
documenting 19
Does Nothing choice 195
domain knowledge 93
dormant state 68-71
down clue count 196
downClues 242
downWinTag 241
down 227
drawing context 9, 116, 119, 156, 191, 271, 273, 279, 285-286, 288
drawing regions 115
drawing 273
dribbles 155
Dxxxx Debug flags (defined) 58
dynamic binding 14
dynamic binding 201
Dynamic Link Libraries 8, 20, 45-46, 86, 191, 198, 203, 234, 236-237, 256, 259-260, 269-270
DllMain() 45, 237, 260
initialization 260
loading 44
dynamic linking 30
dynamic link 62
dynamic loading 30
Dynamic UID (defined) 24
dynamically allocated buffer 30

edge 245, 280
Edit menu 299
eight and three character rule 54
embedded
applications 102
documents 65
objects 27
emulation 155
encapsulation 14, 20
enhancements 299
entryTag 176
enumerated list of classes 94
enumerated type 80
environ.ini 44, 62
Error 49, 60, 68, 70, 83, 148, 173
condition 146
messages 58
states 104
status 129
ET++ 67
event filtering 119
exception handling 29
executable 236
form 43
module name 62
module 63
execution behavior 63
exiting applications 67
exporting information 52, 199
extension, dlc 46
external definitions 50
external interface 41, 129, 183, 217, 240, 257, 259, 274, 277

factory 67, 88
family metaphor 118
feedback 120, 122, 154-155, 171, 253, 274
fgets() 200
field applications 8
fields 157-158, 202
figures 273
file
creation 282
handle 216, 282
I/O 271
management 23
system access 6
system classes 22-23
system 10, 60, 70-71, 133
__FILE__ 56
FILENAME_INCLUDED 51
filing data 72, 103
fill pattern 288
filter data 176
filter/translator pair 186
filter 60, 119, 181-182, 185
firstNumTag 162
fixed-point math primitives 277
flags 200, 278
floating notebooks 10
flow of control 83
flow of ink 155
focusStyle 170
fonts 22, 273
bitmaps 273
primitives 277
scale 288
selection 273
size 133
technology 273
foreground color 273, 287
form-based application 190
formatted information 57
format 200
forwarded gestures 256, 263, 268
fprintf() 57, 200
frame (defined) 116
frame 167, 208, 237
frame client window 208
framework 66
free objects 202, 263
fstNone 170
function
calls 23, 28, 55, 64, 104
defintions 13
prototypes 161
references 18
functional interface 55
FxMakeFixed() 288

gdFileHandle 279
generic (defined) 235

generic 19
  applications 9
  components 90, 95
  control model 120
  input controls 159
  objects 19
geo.h 217
geometric 217
geometries 120
gesture 4-6, 9, 23, 120, 154, 159, 182, 256, 263, 268
  data 154
  forwarding 256
  recognition 121, 158
get values 61
getData() 222
global debug flags 55
global well known UID 35, 52, 104
globally accessible memory 29
GO Corporation 4, 11, 14, 24, 29, 57, 65-67, 200, 233, 235, 273
GO PenPoint 3
go.h 52
gomath.h 277
Graphical User interface (GUI) 1-2, 271
graphics primitives 9, 273
gray background 165
gray letter 194
grid array 220
grid based grouping 122
grid size 196
grid view 274
gridDataFile 278
gridDC 279
gridSize 242, 244, 279, 285
gridWinTag 241
grid 242
GRID_DATAFILE 278
GRID_ENTRY 278

hand coding 67
handwriting based calculator 173-174
handwriting 4, 159
  classes 22-23
  prototypes 22
  recognition system (defined) 154
  recognition system 6, 9, 153-155, 158-159, 190, 289, 292
  translation 176
  input 156, 190, 289
hard drive 46
hardware
  characteristics 2, 8
  devices 8, 271
  displays 8, 271
  events 155
  initialization 44
  vendor 3, 233
hardwired interrupt key 63
Header file 73, 129
heap 60, 182
heavyweight objects 20
height 119, 144
help gesture 121
help screen 61
help 299
hidden window 118
hierarchy 23
ahit detections 273-275, 279
horizontal 274
  layout 234, 238
  menu 200
  size 285
hot mode 68, 71-72, 77, 102
hotMode 102
HWX Engine (defined) 154
HWX engine 155-156, 159
hwxcalc example 190
HWXCalcAppAppInit 177
HWXCalcCompute 181
HWXCalcRestore 179
hybrid environment 104

icon-based interfaces 1
IDataDeref() 82, 84, 140
identification information 54
identification 160
identifying reuse 94
#ifdef FILENAME_INCLUDED 51
image rendering 272-273
ImagePoint 271-273
Imaging model 271, 273
implementation 18, 90, 105, 184, 258
  file 31, 238
  options 115
  strategy 90
  techniques 13
  tool 18
implementer 31
implementing 198
import manager 193, 199
import.h 205
imported 196, 199
  data 200, 214
  documents 48
  file 221-223, 237
  puzzles 193
In Box 10
#include directives 79
increase generality 19
indexed instance variables 131
index 132, 148-149
information structure 244
infrared nets 3
inheritance hierarchy 18, 94, 115-116
inheritance 6-7, 13-14, 16, 20, 90, 115, 236
inherited behavior 21, 25, 38-41, 53, 68, 70, 79, 97, 126-127
initialization 164, 180, 186, 206-207, 260
ink 155
inner area 120
input 190
  components 153
  controls 159
  devices 155
  events 66, 155, 188
  support 158
inserting windows 123
insertion pad 157-158, 299
inside 120-121
inspecting objects 26
installation classes 22
installation 54
installed application 47, 67
Installer (defined) 45
installing applications 45, 76
installing CoinApp 86
installing at boot 45
instance (defined) 20
instance 24, 47, 53, 63, 67
  data 15, 17, 20-21, 33, 68, 71, 80, 83, 106, 108, 132, 176
    (pData) 27
    structure 74
    size limits 32
  methods (defined) 14
  methods 15, 17
  variables (defined) 14
  variables 21, 31, 77, 80, 120, 162, 164, 179, 206, 260, 278
    data structure 36
    size 101
INSTANCE_DATA 36, 185, 220, 278, 137
integer 159
  field 170, 173
integer-based 272
INTEGER_FILED_NEW 168
interchangeable 234
  components 13
interface
  architecture 10
  files 31, 50, 52, 238
internal representation 19
internals 8
internationalization 235

kernel 8, 44, 47-48
keyboard input 119, 159
keyboard-phobic 3
keyboards 3, 9, 155, 158, 233
keypad 136, 145, 153
  child window 133, 139
  paradigm 97
  view 143
  windows 140

Label item 267
label.h 26, 52
label 25-26
LABEL_NEW 25-26, 83, 133
landscape layout 237-238
landscape mode 44
layered fashion 116
layout 119-120, 133, 135, 198, 234, 237, 278
  abilities 127
  behavior 115
  capabilities 122, 128
  episode 247, 265
  function 248
lbFreeDataWhenDestroyed 263
left-justified 121
lenArgs 29
letters choice 196
letters 274
Lex 190
library 18, 154, 188, 199, 236
life cycle 65
light fixture 14-18, 24, 34

light pen 2
LIGHTFIXTURE_NEW 36, 38
LIGHTFIXTURE_NEW_ONLY 38
lightweight threads 8
__LINE__ 56
line break 200
linker 18, 64, 237
linking 20-21, 54, 62, 67, 236
List box 263, 265, 268
list objects 227
list.h 219
listbox.h 259
listbox 260
listener 180
lists of data 201
lists 202, 262
listWinTag 259
LIST_BOX_ENTRY 263
ln.object.uid 26
load module name 54
load module 54
loader 237
loading puzzle 193
local memory 80
locate 202
locator 9
lockon button 122
logical
  constraints 123
  coordinate size 274
  coordinate space 272, 286
  coordinates 278
  Device Coordinates (LDC) 272
  geomitries 217
  imaging model 273
  Screen Units 164
  Unit Coordinates (LUC) 272
  units 133
  Window Coordinates (LWC) 272
long int 63
lowercase 154

Mac II 233
MacApp 66
Machine Interface Layer (defined) 44
Machine Interface Layer 8
Macintosh 7, 10, 199, 233
  applications 67
  Mac II 233
  MacApp 66
macro definitions 13
macro-based interface 23
main window 71, 165
main() 45, 47-48, 74, 76-77, 79, 85, 97, 99, 101-103, 111, 163, 176, 198, 203, 206, 217, 221, 239, 242, 275, 279
maintenance methods 110
maintaining components 19
Make utility 64
makefile 64, 188
MakeGlobalWKN() 24, 52
MakeMsg() 24, 34
MakeNonErr() 25
MakeStatus() 25, 104
MakeTag() 25
MakeWKN() 24
manager (defined) 123
manager 128
managing data 126
managing notification 103
managing objects 23
mapping 285
margin area 120, 121
maximum size 193
maxS32 27
mbMenuPopup 201
mbMenuPullDown 201
memory mapped file I/O 271
memory mapping 282
memory model 44
memory protection 8
memory usage 58, 60, 274
memory-mapped file 274, 279, 283
memory 26
memset() 224
menu 116, 119, 157, 191, 200, 202, 210
  bar 194, 208, 251
  client 211
  handling 199
  interface 198
  nested 194
  selections 211
  support 200
menu.h 205
MENU_NEW 200
message 7, 17-21, 25-26, 28, 30-31, 34, 48, 50, 104
  binding 21
  definitions 105
  dispatch table 50
  field 34
  functions (defined) 28
  identifiers 24, 28, 31, 161
  logging 61
  macros (defined) 29
  mapping 229
  sending macros 133
  sending type 28
  send 18, 37, 39, 104
  tracing 27
  type 29
message to method translation table 37
message-based interface 22
message-sending macros 104
message/method mapping 53, 269
messages to superclass 40
messaging
  facility 28
  overhead 162
meta-class 20, 25
method 18, 20-21, 28, 33-35, 37, 40-41, 48-49, 53
  arguments (pArgs) 27
  definitions 129
  dispatch table 71
  field 34
  lookup 19
  macros (defined) 32
method table (defined) 31
method table 34-35, 39, 46, 51-52, 54, 67, 72-73, 75, 173, 188, 216, 219
  attribute field 34
  compiler mt.exe 34, 48, 53-54, 241, 259, 277
  definitions 78, 141
  entries 76
  message field 34
  method field 34
  structure 53
method.h 48, 51
method.tbl 48, 52, 78, 103, 112, 129, 149, 203, 216, 229, 238, 255, 281, 295
method/message mapping 53, 269
methodology 16, 22
METRICS 220
metrics 144
Microsoft
  RTF 154
  Windows-NT 3
  Windows 66
mil.ini 44
mini-debugger 60-61
minS32 27
miscellaneous classes 22-23
MIT 2
mmsgWinEndRepaint 286
mnLettersTag 208
mnNothingTag 208
mnPuzzleTag 208
mnShowSolnTag 208
mnStartOverTag 208
mnStrikeOutTag 208
mnWordsTag 208
modal 201, 214
model 14, 140, 199, 226, 237, 239, 242, 253
  class 90, 99, 103, 216-219
  data 279
  objects 90-91
modify registers 60
modify 202
module 18, 30
momentary contact button 122
monitor 188
Motorola 233
mouse input 159
mouse-based interfaces 2
mouse 1, 9, 66, 155, 158, 233
MS-DOS 10, 46, 54, 61
msgAdded 92
msgAddObserverAt 91
msgAddObserver 91-92, 179
msgAppClose 70, 72, 84
msgAppGetMetrics 83, 127
msgAppInit 53, 70-71, 77, 82, 99-100, 127, 164, 177, 207, 215
msgAppOpen 70, 72, 82
msgBorderGetBorderRect 285
msgBorderGetStyle 139
msgBorderSetStyle 139
msgCalcBtVwDigit 130, 145
msgCalcBtVwFnc 130, 146
msgCalcEngAccmChanged 105, 107, 148
msgCalcEngAdd 105
msgCalcEngClr 104
msgCalcEngDiv 105
msgCalcEngError 105, 148

msgCalcEngGetAccm 104
msgCalcEngMul 105
msgCalcEngSetAccm 104
msgCalcEngSub 105
msgClass 27
msgControlAcceptPreview 121
msgControlSetDirty 170
msgCstmLayoutGetChildSpec 143
msgCstmLayoutSetChildSpec 166
msgDcClipRect 285
msgDcDrawPolyline 286
msgDcDrawRectangle 288
msgDcDrawText 288
msgDcFillWindow 285
msgDcIdentityFont 287
msgDcIdentity 285
msgDcLWCtoLUC_RECT32 285
msgDcScaleFont 287
msgDcScaleWorld 285
msgDcSetFillPat 287
msgDcSetFontForegroundColor 287
msgDcSetForegroundRGB 286
msgDestroy 26, 140, 283
msgDump 26, 31, 55
msgEnumObservers 93
msgFieldModified 169-171
msgFieldValidateEdit 170
msgFrameSetClientWin 83, 127
msgFree 70-71, 78-79, 283
msgFSMemoryFree 283
msgFSMemoryMap 282
msgGetObserver 93
MsgHandler() 33-34
MsgHandlerArgType() 33, 81
MsgHandlerParametersNoWarning 32
MsgHandlerPrimitive() 32-33
MsgHandlerWithTypes() 83
msgImportQuery 199, 214
msgImportRequest 199
msgImport 199, 215
msgInit 70-71, 77, 80, 186, 222, 242, 261, 280
msgInputEvent 188
msgIsA 27
msgIsXWordFile 215
msgLabelGetString 171
msgLabelSetString 139
msgListBoxEntryGesture 268
msgNewDefaults 26, 36, 49-50, 74, 105, 133, 135, 211, 222, 242, 279
msgNew 26, 30, 36, 50, 74, 105, 123, 126, 242
msgNoteShow 214
msgNotifyObservers 92, 107
msgNumObservers 93
msgOkToResetSPaper 183
msgPostObservers 92
msgRemoved 92
msgRemoveObserver 91
msgResReadData 243
msgRestore 70, 73, 81, 141, 208, 246, 264, 284
msgSave 70, 72, 77, 81, 126, 141, 246, 263, 283
msgScrAddStroke 158
msgScrComplete 158
msgSomeMessage 28
msgSPaperComplete 158
msgSPaperGetXlateData 181, 290
msgSPaperXlateCompleted 180-181
msgStreamRead 81, 142
msgStreamWrite 81, 141
msgTrace 27
msgUpdate 92
msgVersion 27
msgViewGetDataObject 142
msgViewSetDataObject 140, 247
msgWinFindTag 167
msgWinInsert 140
msgWinRepaint 285
msgWinSized 292
msgXlateComplete 158, 289
msgXlateGetFlags 281
msgXlateSetFlags 281
msgXWordAppDoCheck 204, 212
msgXWordAppSetClueTap 204
msgXWordAppShowSoln 204, 211
msgXWordAppStartOver 204, 211
msgXWordClueClueTapNothing 257, 266
msgXWordClueClueTapStrikeOut 257, 266
msgXWordClueStartPlayOver 257, 266
msgXWordDataGetAcrossCount 217
msgXWordDataGetAcrossWord 217
msgXWordDataGetDownCount 217
msgXWordDataGetDownWord 217
msgXWordDataGetInfo 217
msgXWordDataGetLetters 217
msgXWordDataIsXWordFile 217
msgXWordGridGetLetters 275
msgXWordGridSetLetters 275
msgXWordGridSetOkLetters 275
msgXWordGridStartPlayOver 275
msgXWordGridStartPlayOver 293
msgXWordSetClueTap 211
msgXWordViewCheckLetters 239
msgXWordViewCheckLetters 254
msgXWordViewCheckPuzzle 214, 239, 252
msgXWordViewCheckWords 239, 254
msgXWordViewClueStrikeOut 239
msgXWordViewClueTapNothing 239
msgXWordViewClueTapStrikeOut 251
msgXWordViewShowSoln 239, 250
msgXWordViewStartOver 250
msgXWordViewStartPlayOver 239
msg 32
MSG_INFO 35, 216, 229, 255, 295
MS_DOS 43
multiple views 91, 103, 126

name scope 62
name space 93
named volume 46
nested menus 194
network based computing 23
networking technologies 3, 10
neural net technology 6
_NEW 37
NEW 50
new data structure 26, 36
NEWDEFAULTS 31
new puzzles 196
next statement 64
nextOp 132, 147
NeXTstep 67
nfSystemModal 214
Nil(SIZEOF) 50
nn.note.metrics.flag 214
non-discrete data 157
non-error messages 58
non-pen 155
nonexistent state 68-71
not activated 77
note management 213
note.h 205
noteboard example 90-91, 95
Notebook User Interface (NUI) 4-6, 8, 22-23, 115-117, 119, 122-125
Notebook User Interface class library 122
Notebook 4, 70-71, 76
note 191, 196
notification 91, 159-160, 169-170, 180, 199, 265
  behavior 122
  client 123
  list 91
  messages 123, 202
notifies 154
notifying observers 92
null object 26
numbers 154
numeric 159

objCallAncestorAfter 34, 40
objCallAncestorBefore 34, 40
ObjCallJmp() 30, 49, 83
ObjCallRet() 26, 30, 104
objClassMessage 216
object
  creation 25, 223
  freeing 25
  management 202
  manipulating 25
  model 67
  recreation 73
object-oriented (defined) 13
object-oriented 4, 11, 15-16
  analysis 89
  architecture 6
  design 19, 89-90, 95
  enhancements 151
  environment 14, 65, 201
  framework 37
  implementation 10
  paradigm 115
  programming 17, 21, 23, 41, 66-67, 126, 147
  programming languages 23
Object-Pascal 66-67
ObjectCall() 27-29
ObjectCallAncestor() 40
ObjectCallAncestorCtx() 40, 186
objecthood 90
Objective-C 14, 67
ObjectPost() 29
ObjectPostAsync() 29
ObjectSend() 29
ObjectSendUpdate() 29

ObjectWrite() 80, 82, 108
object 13-14, 18, 20, 28, 30
objNull 26
OBJ_NOTIFY_OBSERVERS 92, 107
observer 91, 97, 103, 105
list 92-93
notification 91, 98, 103, 106, 108, 299
registration 91, 126
observable object 202
obsrObj 107
OK Button 196, 214
okToReset 187
on-screen cursor 155
open figures 273
opened state 68-69, 72
operating environment 44
operating system 3, 4, 6, 8, 10, 13, 43, 48, 66-67, 80, 155, 199-200, 202, 234, 236, 260
operatorTag 162
operator 153, 159
opt-label 30
opt-status 30
optimum performance 119
option cards 25
option sheets 25, 123m 157, 234
ordinal 104
OS/2 8
OSHeapBlockAlloc() 224, 290
OSHeapBlockFree() 224, 226, 291
OSI Networking Standard 10
osProcessHeapId 181-182, 224
Out Box 10
out-of-proximity 155, 158, 160
outline font technology 273
output debug messages 74
output stream 55
overriding methods 17, 39-40, 53, 70, 76
overwrite fields 158
ownership 202

packaging 236
panels 154
paper 153-154
paradigm 120
parent boundaries 119
Parent Window Coordinates (PWC) 272
parent window 117-119, 122, 128, 133, 135
pArgs (method arguments) 27, 29, 32
pArgStructure 28
parsing 176, 182
Pascal 66
Paste 299
Pause key 61, 86
PC market 233
PC 199
pData (method data) 27, 32, 81, 83
pen and paper metaphor 65, 154, 174, 190
pen and screen 271
pen 2-3, 5, 10, 91, 95, 116, 120, 122, 153-155, 158, 160, 174, 184, 195, 271
computer (see tablet) 234
events 155-156
input 119, 154
movement 116, 154
stroke 23, 158
tap 116, 195
tracking systems 8
pen-based
computing 1, 3-4
machines 10, 45
metaphor 11
systems 2, 9, 46, 153
version 191
pen.h 184
PenPoint 188
classes 31, 199
library 199
Object Model 23
Seminar 153
Software Developer's Kit 73
windows 116
Definition of Objects 20
\penpoint\app 46
pEntries 279
performance
considerations 119, 273
enhancement 273
persistent
data 14, 103, 141
objects 234
personal computing 233
phone list 271
physical
address 63
display device 273
movement 154
pip-xwordpuzzle 196, 222
pixel 272
characteristics 44
plotter 272
point and click metaphor 1
pointing devices 9
polygons 273
polylines 273, 287
pop-up 200-201
portability issues 8portrait layout 237
portrait mode 44
positions 9
potential reusers 151
Power of PenPoint, The 4
pre-built component library 115
pre-coding 89
pre-defined grammar 190
pre-framework era 66
pre-loading 45
predefined
classes 65
components 154, 190
resources 234
tags 179, 208, 210
preferences 235-236, 240, 244
prefs.h 240
preprocessor directives 50
preprocessor 18
preprocessString() 182
preview
behavior 122
mode 120-121
protocol 200
primitives 273, 277
print messages 54
printf() 55
printing 23, 282
private data 14
prLandscape 235
process 71
boundaries 20-21, 29
space 29
processCount 47-48, 75-77
processing loop 140
producer 15, 17, 37
production objects 62
programmatic
disabling 121
enabling 121
programming community 13
programmatically 121, 256
prOrientation 235
protected data 32
protected memory 20, 31-32, 80, 82, 84, 108, 139, 141, 143, 146-147, 165, 167, 262, 282, 285
proximity (defined) 155
proximity data 9
proximity 158, 160
prPortrait 235
pString 83
pTitle 245
pull down 201
pull-down menu 200
pull-right submenu 200
punctuation 154, 159, 187
puzzle
document 237
size 193, 227
statistics 252
second sample 231
Puzzle Menu 193-195, 203
Puzzle menu option 213
Puzzle... submenu 196
P_ARGS 83
P_BUTTON_NEW 134
P_CALCVAL 108
P_CUSTOM_LAYOUT_CHILD_SPEC 143
P_GWIN_GESTURE 268
P_INSTANCE_DATA 83

Quick Help strings 25

radio buttons 122, 157
RAM volume 44
raw translation 181
re-install 61
re-run 61
reactivated state 69
read-only capacity 83
reading data files 200
real estate agent 3
real world abstraction 14
real-time 154
receive notification 256
recompile 61
rectangle 9, 115, 217, 273, 279
rectangular region 116, 119

recursive dispatch loop 27
recursive live embedding 6-7, 164
registration 48
relative positioning 123, 128, 144
relative Window positions 124
reliability 19
relink 61
relWin 145
remote
  interface classes 22-23
  object 21
remove observer 91
RemoveMenuTags 210-211
remove 194
removing objects 26
rendering 271
render 9, 156, 194, 272, 274, 279, 285-286, 289
reordering cards 94
repainting 116, 285
  protocol 119
requested behavior 18
research groups 66
reset message 182
resizing layout 117
resizing 119, 279
resource
  file 48, 69, 71, 77, 110, 234, 235, 240, 244
  Ids 235
  manager (defined) 234
  manager 235
responding 293
responsibility/collaborator pairs 94
restore state 110, 141
restoring data 68, 101, 111, 127, 162, 166, 179, 187, 245, 263, 283
resultsTag 162, 176
reusable
  classes 89
  components 8, 19
reusabilty 7, 11, 14, 16, 18-21, 38, 66, 73, 89-90, 93-95, 121, 201, 236
Rich Text Format 154
right arrows 200
right justified 121
root ancestor 25
rotation 272
ruled paper 154
running demoapp 76
runtime installation 46
runtime 45

sample programs 299
sample puzzle 231
saving
  data 77, 101, 127, 187, 245, 263, 283
  state 67, 78, 110, 147
scalable operating system 7
scaleable 233-234
scaling 272
scratch pad window 180
scratch paper based calculator 173-176
scratch paper 158, 183, 188
scratch-pad 154
screen 272
  devices 9
  layout 119
  orientation 234-235, 247
  resolutiion 233
  type 44
screenBlockSize 279
scribbles (defined) 154
scribbles keys 23
scribbles 156, 158-159, 198, 274, 285
scrollable list 256
scrollbars 116
secondNumTag 162
segmentation 154
segment 60
selected volume 45
self-sent message 120
self 32-33, 83, 132, 135-136, 165
sending messages 28, 91
set values 61
shaded fonts 274
shadow 120-121, 245
Shafer, Dan 4
shared
  code 237
  information 15
  libraries 20
  objects 202
  part 15
  pointers 202
Show Solution Submenu 194
shrink-wrap (defined) 119
shrink-wrap 122
sibling windows 118
signal 214
simple calculator 95
single inheritance 20
single model 126
single process 21
single tap 269
68000 233
64K limit 32
size limits 32
size, instance variable 101
size 242, 265, 279
Smalltalk 14, 23, 90
SoftTalk 23
Software Developer's Kit (defined) 26
Software Developer's Kit 43, 49, 54-55, 73, 83, 270
software
  emulation 155
  engineering 22
  quality 13-14
  reuse 10
solution 194, 293
source
  code 62-63, 67, 78, 104, 129, 190
  information 87
source level debugger 55, 61
SPaper Calculator 174
specifications 16, 155
spell checking 299
spelling manager 23
spelling 229
sprintf() 146
sscanf() 224
stack contents 63
stack frame 62
stages of reuse 19
stamp utility 54
standard pen events 155-156
staples 123
start over Submenu 194, 211
starting over 194
starting PenPoint 44
state data 77
stateful data 15, 68, 77-78, 127, 132, 143
stateful objects 71
stateless entity 107
stateless objects 72
states 68-69, 260
static binding 14
stationary 5, 48, 102, 164, 230
  notebook 102
  notebook application 4
statistics display 253
status
  error 25
  handler 29
  information 24
  non-error 25
  update 87
  values 28, 27
STATUS 30, 49, 104, 133-134
stdio.h 222
StdioStreamBind() 221-223
StdioStreamUnbind() 221-222, 224
stdio 200
stdlib.h 176
step into 64
step over 64
stock prices 193
storage format 200
!strcmp() 253
STREAM_READ_WRITE 81, 141
strike it out choice 195
strikeout
  attribute 267
  mark 267
  state 268
string 182
  manager 23
  object 56
  pointer 63
  tables 234-235
strncpy() 224
strobj.h 219
stroke 9, 156, 174, 274
  analysis 6
  data 158, 183
  translator 185
strokelist 156
stsFailed 222
stsNonValidOp 162
stsOK 30, 71, 134, 171, 222, 237
stub structure 46
style flag 201
style sheets 119
stylistic requirements 273
stylus device 4, 44
subclass 20-21, 31, 38, 95, 143, 174
subhierarchies 22
superclass 16-17, 20-22, 36-40, 50, 94, 143
support functions 106, 129, 167, 182

support routines 51
symbolic
  debugger, DB 60, 86
  information 60, 87
  level 61, 63
symbols 62, 154
synchronous
  functions 29
  messages 21
sysapp.ini 45
syscopy.ini 45
SYSDC_FONT_SPEC 282-283
SYSDC_NEW 282
SYSDC_PATTERN 287
SYSDC_TEXT_OUTPUT 287
sysfont.h 277
sysgraf.h 277
system
  applications 44-45, 102
  components 4
  debug flags 61
  faults 8
  flags 57
  information 58
  interface 6
  Log 59
  Messages 58
  performance 202
  preferences 198, 234-235, 237
systemApp 102

Table layout 115, 123, 126
  windows 122
  manager 123
Table Of Contents 4-5, 70, 77
table-based groupings 123
tablet (see pen computer) 2-4, 7, 26, 44, 154-155, 234, 247, 271, 274
tabular grid 136
tags (defined) 24-25, 160-161, 166-167, 179, 208, 210, 241, 245, 259
  definitions 205
  identifier 25
tagClueTapMenu 208
tagXWordMenuPuzzle 208
take down 200
Tapping Clue Submenu 194-195
tap 116, 120, 145, 201, 256, 268-269
tasks 60
  ID 27, 62-63, 86-87
  list 58, 61
taxonomy 46, 94
telephone example 121
tellObsAccmChanged() 106, 108
tellObsError() 107-108
template (defined) 159
template 185-186, 227, 230
  compiling 159
testing 89
text 273
  classes 22-23
  field 9
  file 235
  layout facilities 121
  rendering 273
  rotation 121
  strings 121
theSystemPreferences 244
thick black shadow 245
TIFF 273
timed light fixture 16-17, 24
title bar 116, 121
titleWinTag 259
title 256
tkButtonOn 210
tkfield.h 161
tkMenuPullDown 209
tkMenuPullRight 209
TkTable 131, 136
TK_TABLE_ENTRY 134, 145, 200-201, 208-209, 213-214
TK_TABLE_NEW 134-135
tlAbsolute 134
tlMaxFit 134-135
toggle button 122
toggle 157, 256
toolkit 190
top level dispatch loop 29
TOPS 23
translator 190
transformations 272, 286
translate scribbles 198
translated handwriting 282
translation 155, 158, 181, 187, 272, 274
  data 176
  matrix 286
  mechanism 31
  success 187
  table 37, 48
translator 156, 159, 174, 183, 185
transparent 119
turning away 77
turning the page 70
two-dimensional 272

underlying
  architecture 60
  framework 166
  model 279
Undo 299
unique
  data format 200
  message number 24
  module indentifier 54
  part 15
  tag 167, 210
Unit Definitions 272
Universal Identifier (defined) 24
Universal Identifier 26, 28, 31-33, 91, 100, 132-133, 160-161, 167, 202, 234
Unix 190
unknown character globe 193
unmapping 283
unrecognized characters 182
Unused() 32
update
  episode 285
  handling 148
  mechanism 91
  message 103, 147, 173
  observer 91
uppercase 154
user
  acceptance 11
  environment 44, 234
  events 116
  feedback 274
  guide 191
  input 23, 289
  interaction metaphor 65, 145, 190
  menus 122
  notification 199, 201
  preferences 22, 234
User Interface 4, 6, 8-9, 66, 72, 90-91, 115, 124, 154, 190, 194, 273
  components 23, 93, 117
  management hierarchy 116
  Toolkit classes 22-23, 44
user-friendly 66
user-supplied data 252
using CoinApp 86
utility functions 30, 159, 222, 272

validation 154, 157, 170-171, 173, 194
  messages 160
value 132, 146, 148
variable contents 63
vendor 3, 233
verbal agreement 202
verbose
  code 286
  output 27
versioning (defined) 30
  information 54
  number 54
vertical applications 45
vertical layout 234
vertical size 285
vertical 274
vertical menus 200
veto success 187
view
  behavior 147
  class 101, 199, 233, 238, 240-241, 270
  freeing 140
  hierarchy 118
  initialization 140
  objects 90-91
  updating 126
view/data approach 90
viewNewFields macro 130
virtual machine 8
visibility 119
visible window 118
visual
  attributes 119
  component 273
  feedback 10, 155, 171, 253
  hierarchy (defined) 116
volume name 46
volumes 46

warm-reboot mechanism 8
warning status 171
watchpoint 63
Well known
  constants 25
  Id (defined) 24

Id 25, 28, 39, 50-51, 85, 102-104, 129, 176, 275
lifecycles 67
resource Ids 235
well-defined protocol 67
whitebox testing 56
width 119, 144
wigits 90
window (defined) 116
Window 7, 9, 72, 118, 126, 200, 244, 279
alignment 165
boundaries 119
classes 22-23, 116, 119
components 115
hierarchy 100, 117-118, 127, 141, 143, 160, 167, 262
identification 160
Id 161, 241, 259
layout episode 247, 265
repainting protocol 119
tag 161, 241, 259
Window Manager (defined) 116
Window Manager 93, 241, 245, 260, 263, 285
windowing system 45, 115
winTag 245
wire frame view 118
wknGlobal 24
word choice 196
word/clue entries 196
working environmnet 234
working vocabulary 93
world coordinates 286
wsSendFile 133-135
wsTransparent 134-135
WYSIWYG (What You See Is What You Get) 190, 271

x2sData.pString 182
X2STRING 181, 290
xclu_mth.tbl 269
Xerox Star 1
Xerox 1, 2
xlate.h 176
XLATE_DATA 181
XLATE_NEW 185, 281
xlfilter.h 176
XList2String() 290
XList2StringLength() 290
XList2Text() 181-182, 290
XListFree() 182, 291
XlistGet() 290
XList 159, 182, 188,, 274
XLIST_ELEMENT 290
XTemplateCompile() 187, 281
xtemplt.h 184
xtemplt 188
XTM-ARGS 185, 281
xtmTypeCharList 186
XWABuildMenus() 208, 210-211
XWAShowCheckPuzzleStats() 213
XWCCreateListBox() 262
XWCCreateListTitle() 262
XWCSetClueEntryStyle() 266
XWDBuildXWordFromFile() 223
XWGBuildDC() 282
XWGBuildGridDC() 285
XWGBuildTranslator() 280-281
XWGDrawGrid() 286
XWGDrawLetters() 288
XWGDrawTemplate() 287
XWGFilterTransData 292
XWGFindGridPos() 291
XWGGridPosToRect() 291
xword file 222
xwordapp.c 203-204, 206
xwordapp.h 203
XWordAppAppInit 207, 210
XWordAppDoCheck 212
XWordAppImportQuery 214-215
XWordAppImport 214-215
XWordAppMenuBar 209
XWordAppRestore 208
XWordAppSetClueTap 212
XWordAppShowSoln 211
XWordAppStartOver 211
XWordClueCLGetChildSpec 265
XWordClueClueTapNothing 266-267
XWordClueClueTapStrikeOut 266-267
XWordClueEntryGesture 268
XWordClueInit 261-262
XWordClueRestore 264
XWordClueSave 264
XWordClueStartPlayOver 266
XWORDCLUE_NEW 258, 261
XWORDCLUE_NEW_ONLY 257
XWordDataFree 224
XWordDataGetAcrossCount 228
XWordDataGetAcrossWord 229
XWordDataGetDownCount 228
XWordDataGetDownWord 229
XWordDataGetInfo 226
XWordDataGetLetters 228
XWordDataInit 223
XWordDataIsXWordFile 221
XWordDataNewDefaults 222
XWordDataRestore 225
XWordDataSave 225
XWORDDATA_INFO 218
XWORDDATA_LETTER 218
XWORDDATA_LINE_1 222
XWORDDATA_NEW 218, 222
XWORDDATA_NEW_ONLY 218
XWORDDATA_WORD 218
XWordGridFree 283
XWordGridGetLetters 294
XWordGridInit 280-282
XWordGridNewDefaults 279
XWordGridRepaint 285
XWordGridRestore 284
XWordGridSave 283
XWordGridSetLetters 293
XWordGridSetOkLetters 294
XWordGridStartPlayOver 293
XWordGridTransWriting 289-291
XWordGridWinSized 292
XWordGrid 282
XWORDGRID_NEW 276, 279-280, 282
XWordViewCheckLetters 254
XWordViewCheckPuzzle 252-254
XWordViewCLGetChildSpec 247
XWordViewClueTapNothing 251
XWordViewClueTapStrikeOut 251
XWordViewInit 243-244
XWordViewNewDefaults 242
XWordViewRestore 246
XWordViewSave 246
XWordViewSetDataObject 247
XWordViewShowSoln 250
XWordViewStartOver 250
XWORDVIEW_NEW 239-240, 242
XWORDVIEW_STATS 240
XWORD_ENTRY 220
xwrdclue.c 258
xwrdclue.h 257
xwrddata.c 216, 219
xwrddata.h 216-127
xwrdgrid.c 274, 276, 278
xwrdgrid.h 274
xwrdview.c 238, 240
xwrdview.h 238
XWVaccStrEqu() 253
XWVBuildClueList() 244
XWVBuildGrid() 244-245
XWVdwnStrEqu() 253
XWVLandscapeLayout() 248
XWVPortraitLayout() 248-249

Yacc 190

Operating Systems

>$26.95 USA
>$34.95 CANADA

PENPOINT™ PROGRAMMING

Andy Novobilski

"Required reading for all PenPoint programmers."
—Gary T. Downing, Manager
GO Educational Services, GO Corporation

PenPoint Programming is a hands-on tutorial showing how to develop applications for the revolutionary operating system from GO Corporation. The author introduces PenPoint from a programmer's perspective and demonstrates how following PenPoint's predefined framework for applications building will result in software with standardized behavior, reusable system-level components, and greatly increased functionality.

Through numerous in-depth examples, the book explores the different approach required to write programs that take full advantage of the pen-based model. You'll learn about PenPoint's Application Framework, the Notebook User Interface, and PenPoint's handwriting recognition process. Topics include:

- demonstration of over 30 PenPoint classes in the context of working applications
- object-oriented design techniques for writing applications
- importing and converting outside data into information that PenPoint understands
- object-oriented extensions used by PenPoint to implement the operating system
- how pen-centric application design differs from mouse-based design

For programmers and developers working with the PenPoint environment, *PenPoint Programming* is an invaluable and essential resource.

Andy Novobilski is a consultant and developer specializing in object-oriented programming and pen-based technology. He is the coauthor of *Object Oriented Programming: An Evolutionary Approach*, Second Edition, with Brad Cox, and is a contributor to *Object* Magazine.

52695
9 780201 608335
ISBN 0-201-60833-2
60833

Cover design by Ned Williams

Addison-Wesley Publishing Company