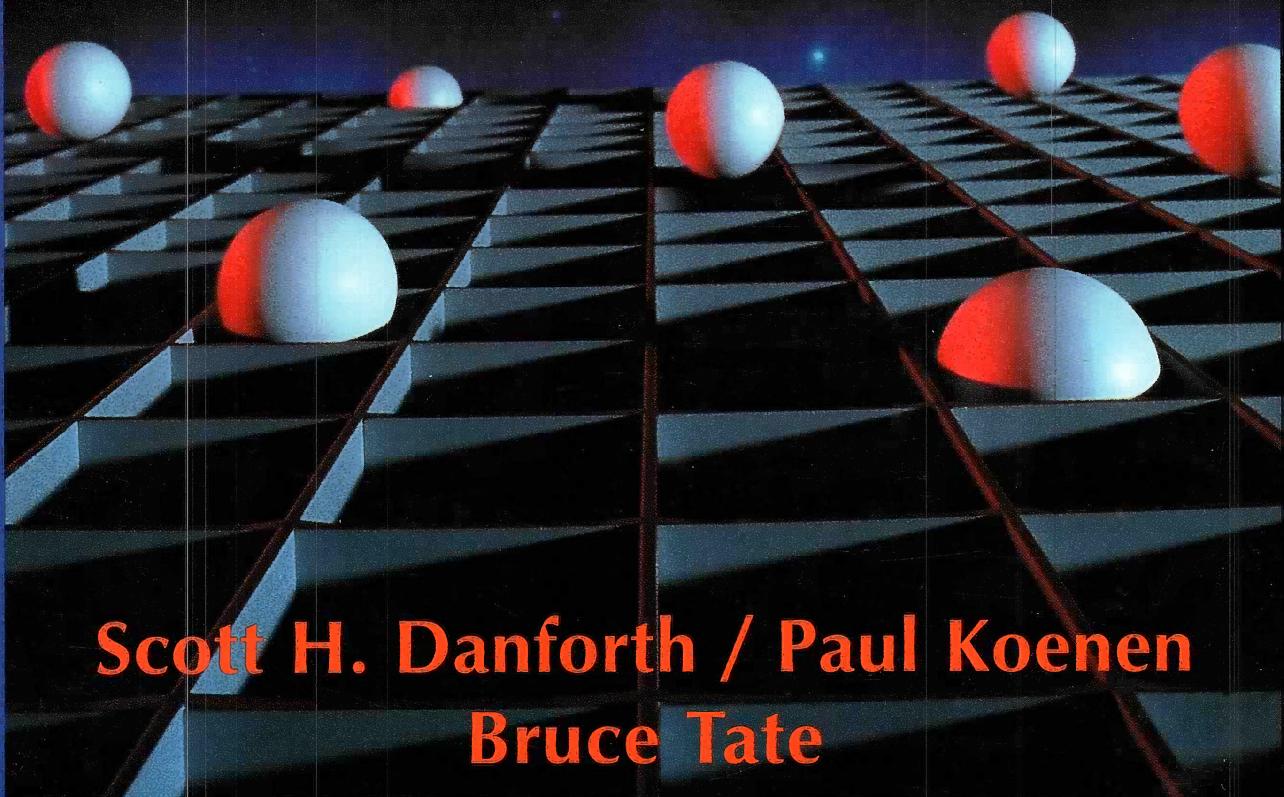




Includes Disk!

OBJECTS for OS/2®



Scott H. Danforth / Paul Koenen
Bruce Tate

VINR COMPUTER LIBRARY

OBJECTS FOR OS/2®

VNR's OS/2 Series

- OS/2 Presentation Manager GPI Graphics
by Graham C.E. Winn
- The COBOL Presentation Manager Programming Guide
by David M. Dill
- Learning to Program OS/2 2.0 Presentation Manager by Example:
Putting the Pieces Together
by Stephen A. Knight
- Comprehensive Database Performance for OS/2 2.0's Extended Services
by Bruce Tate, Tim Malkemus, and Terry Gray
- Client/Server Programming with OS/2 2.1, Third Edition
by Robert Orfali and Dan Harkey
- Now that I have OS/2 2.1 On My Computer--What Do I Do Next?, Second Edition
by Steven Levenson
- The OS/2 Handbook, Second Edition: Applications, Integration, and Optimization
by William H. Zack
- OS/2 2.X Notebook: Best of IBM OS/2 Developer
Edited by Dick Conklin
- Using Workplace OS/2: Power User's Guide to IBM OS/2 Version 2.1
by Lori Brown and Jeff Howard
- The Shell Collection: OS/2 2.X Utilities
by Steven Levenson
- Writing OS/2 2.1 Device Drivers in C, Second Edition
by Steven Mastrianni
- The OS/2 2.1 Corporate Programmer's Handbook
by Nora Scholin, Marty Sullivan, and Robin Scragg
- OS/2 2.1 REXX Handbook: Basics, Applications and Tips
by Hallet German
- OS/2 2.1 Application Programmer's Guide
by Jody Kelly, Craig Swearingen, Dawn Bezviner, and Theodore Shrader
- OS/2 and Netware Programming: Using the Netware Client API for C
by Lori Gauthier
- Object-Oriented Programming Using SOM and DSOM
by Christina Lau
- OS/2 V2 C++ Class Library: Power GUI Programming with C Set ++ Interface Class Library
by Bill Law, Kevin Leong, Bob Love, Hiroshi Tsuji, and Bruce Olson
- OS/2 Remote Communications: Asynchronous to Synchronous--Tips and Techniques
by Ken Stonecipher
- Objects for OS/2
by Scott Danforth, Paul Koenen, and Bruce Tate
- Developing C and C++ Software in the OS/2 Environment
by Mitra Gopaul
- WIN Functions
OS/2 Quick Reference Library, Vol. 1
by Nora Scholin
- The GUI--OOUI War--Windows vs. OS/2: The Designer's Guide to Human -Computer Interfaces
by Theo Mandel
- Client/Server Survival Guide with OS/2
by Robert Orfali and Dan Harkey
- Message Functions
OS/2 Quick Reference Library, Vol. 2
by Nora Scholin

OBJECTS FOR OS/2®

**Scott Danforth
Paul Koenen
Bruce Tate**

VAN NOSTRAND REINHOLD
I(T)P A Division of International Thomson Publishing Inc.



New York • Albany • Bonn • Boston • Detroit • London • Madrid • Melbourne
Mexico City • Paris • San Francisco • Singapore • Tokyo • Toronto

Copyright © 1994 by Van Nostrand Reinhold

Library of Congress Catalog Card Number 94-20064
ISBN 0-442-01738-3

All rights reserved. No part of this work covered by the copyright hereon may be reproduced or used in any form or by any means—graphic, electronic, or mechanical, includin photocopying, recording, taping, or information storage and retrieval systems—with the written permission of the publisher.

I(T)P Van Nostrand Reinhold, is an International Thomson Publishing Company. ITP logo is a trademark under license.

Printed in the United States of America

Van Nostrand Reinhold
115 Fifth Avenue
New York, NY 10003

International Thomson Publishing GmbH
Konigswinterer Strasse 418
53227 Bonn
Germany

International Thomson Publishing
Berkshire House
168-173 High Holborn
London WC1V7AA
England

International Thomson Publishing
221 Henderson Road
#05 10 Henderson Building
Singapore 0315

Thomas Nelson Australia
102 Dodds Street
South Melbourne, Victoria 3205
Australia

International Thomson Publishing Japan
Hirakawacho Kyowa Building, 3F
2-2-1 Hirakawa-cho, Chiyoda-ku
Tokyo 102
Japan

Nelson Canada
1120 Birchmount Road
Scarborough, Ontario
M1K 5G4, Canada

OS/2 Accredited logo is a trademark of IBM Corp. and is used by Van Nostrand Reinhold under license: "Objects for OS/2" is independently published by Van Nostrand Reinhold. IBM Corp. is not responsible in any way for the contents of this publication.

Van Nostrand Reinhold is an accredited member of the IBM Independent Vendor League.

RRDHB 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

Library of Congress Cataloging in Publication Data

Danforth, Scott

Objects for OS/2 / Scott Danforth, Paul Koenen, Bruce Tate.
p. cm.

Includes index.

ISBN 0-442-01738

1. Operating systems (Computers) 2. OS/2 (Computer file)

I. Koenen, Paul. II. Tate, Bruce, 1965-. III. Title.

QA76.76.O63D343 1994

005.265—dc20

94-20064

CIP

Contents

Foreword	ix
Preface	xi
Acknowledgments	xiii
Section I: Basics	1
1. Basic Objects	3
Introduction	3
The Benefits of Object Orientation	4
Encapsulated Objects	5
Classes	7
Conclusion	14
2. An OS/2 Overview	15
Organization of OS/2	15
Interprocess Communication	19
OS/2 Memory Model	23
Exception Management	27
Files	28
Dynamic Link Libraries	29
Conclusion	30
3. Presentation Manager	32
Introduction	32
PM Basics	33
Graphics	43
Conclusion	47
4. Object-Oriented Programming using C++	48
Introduction	48
Inheritance in C++	50
Generic Procedures in C++	60
Multiple Inheritance	66
Interfacing C++ Objects with other Domains	70
Conclusion	71

5. Object-Oriented Programming using SOM	72
Introduction	72
Introduction to SOM Terminology	74
SOM Classes	75
Using SOM Classes	76
Implementing SOM Classes — The Basics	81
6. Using the SOM API	88
Introduction	88
Method Resolution	89
Conclusion	108
7. A Guided Tour through the SOM API	109
Introduction	109
Non-Object Include Files	110
Object Methods	113
Class Methods	115
Class Manager Methods	121
8. The Workplace Shell	125
Introduction	125
WPS: An OO User Interface	126
Writing WPS Programs	127
IDL vs OIDL	132
Levels of Compliance	133
Related Topics	134
Conclusion	134
Section II: Samples	135
9. SOM Examples	137
Introduction	137
Using the System Object Model, SOM	138
Example One	138
The Implementation File	139
The Client Program	142
Building and Running Your Program	143
10. SOM Attributes and Overrides	145
Introduction	145

Attributes	145
The override	152
11. The SOM Metaclass	153
Introduction	153
Metaclass	153
A Constructor Example	155
Running Example One	158
example: Object administration	158
Running it	161
Accessing other data	161
12. SOM Multiple Inheritance	163
Introduction	163
Example. Speaking in tongues	166
13. Taming PM with OOP	178
Introduction	178
Is PM OO?	179
Windows and Messages	180
PM, Your Traffic Cop	181
PMAp Example	184
14. PMAp Divided Into Classes	191
Introduction	191
15. Graphics: Boxes on your Client	203
The PM Graphics Interface	203
16. Wrapping PM Controls	222
Introduction	222
PM Controls	223
17. A Piano of PushButtons	234
Introduction	234
The Window Class	234
18. Let's Wrap!	261
Introduction	261
Possible Enhancement toward better extensibility	291
19. A File System Example	292
OS/2 OPERATING SYSTEM	292

The OS/2 File System	292
Conclusion	310
20. IPC with Semaphores	311
A Semaphore Example	311
Semaphores	312
Section III: Experts	322
21. ObjectPM: A Consultant's View	324
Introduction	324
ObjectPM	329
ObjectPM Architecture	340
Drawing Tools	365
The Author	374
22. A Practical Methodology for OOU Design	375
Introduction	375
Object Models and Views	377
The OVTT Methodology	378
A Simple Calendar Example	381
Summary	397
The Author	398
Section IV: SOM 2.1	399
23. Constructors and Destructors in SOM	401
Introduction	401
Quick Review of Initializers in SOM	402
Initializers in ESOM	403
Implementing New Classes with Initializers	405
Using Initializers	407
Destructors in ESOM	408
A Complete Example	409
24. Metaclass Programming in SOM	417
Introduction	417
Examples	421

Foreword

An innovative idea without development is quite useless.

Peter Goldmark, American Inventor

This book is first and foremost a practical guide for software developers who want to use advanced techniques to solve their business problems. It describes how to use object oriented programming to exploit the power of OS/2. It points out how features of OS/2—features like the workplace shell and the System Object Model—make object oriented programming practical and valuable.

I hate people who are wise during the event.

Kenneth Tynan, British critic

In contrast to the perspective style of this book, I thought I would take a few minutes to explain, with only a sprinkling of hindsight, why and how the OS/2 development team added support for object technology.

First, why. Increased competition and technology advancement are making OS and subsystem APIs grow larger and more complex. PC DOS had approximately 200 programming calls, while OS/2 PM and Windows each have more than 2000. Techniques like OLE (C) add about 1000 more, and we see no end. Heterogeneity, function distribution, collaboration, and time-based media each adds to the API list. These expanding APIs have a multiplier effect and cause an exponential growth in programming complexity.

Meanwhile, commercial applications—running large and small businesses—are surpassing personal productivity as the new growth area. A larger, less skilled community is being deployed into a more complex programming environment. This is not only a problem for application programmer, it also means that developers of these systems must revise them more frequently to keep pace with new developments from active competitors. Larger, more complex systems must be revised more frequently.

Objects have unique properties that can address this problem:

- They allow generalized APIs to be produced, which can be extended by others. (In other words, not all system extensions need to be managed by the original developer.)
- They can be combined in frameworks that codify the elements of design—a GUI, for example, or a communications subsystem—allowing developers to implement unique requirements more easily.
- They are adept to change. They suffer less from the ripple effects that plague procedural programming.

So, how? The OS/2 team set the following design points for the object oriented support:

- Language neutrality—people use many languages to build their applications
- High performance—always a requirement, especially with GUI programming
- Binary compatibility from release to release—the operating system must be able to present stable interfaces as it evolves.

They quickly found that the current crop of object-oriented languages all failed the first test, and most failed the third. A single language, and fragile implementation was usually the price for good performance. The System Object Model (SOM) was designed to meet these requirements. This book provides ample detail on SOM in OS/2, so I'll take just a little more space to broaden the context. The IBM work on SOM found an echo in the work being carried out by the Object Management Group (OMG). The OMG was defining standards which enabled objects to be located and interoperate across networks. Put simply, the OMG was solving the problems of objects separated by location, while SOM dealt with objects separated by language. The two problems had a great deal in common, and IBM combined the two problems in its Distributed SOM (DSOM). DSOM compiles with the standard that the OMG laid down—the Common Object Request Broker Architecture (CORBA). DSOM provides a single way to build objects that can be accessed from within an address space, across address spaces, and across a network, without sacrificing performance.

In addition, the SOM has been implemented on OS/2, AIX/6000, HP/UX, and Windows 3.1. It is fast becoming the defacto standard for defining objects between languages. It is giving rise to new compilers and enhanced language environments like IBM's CSET++, MetaWare's High C++, IBM's Visual Age, Digitalk's Parts, and ParcPlace's Object Works all of which directly support SOM.

With this in mind, I hope that you not only find practical benefit from this book, but that you enjoy its description of the operating system that is the springboard for a broader view of what object technology can deliver—an empowering but unifying technology for a new breed of systems and applications.

Cliff Reeves

Preface

Objects in the mirror are larger than they appear.

- My car

I saw that Cliff used a few catchy quotes. Not to be outdone, I decided to try my own. This one says it all. It seems that object-oriented programming promises to solve all of our problems. Some may not agree. Like objects in the mirror, things for them usually turn out to be a little larger and a little more complicated than they intended.

Oops.

- Many people with the above experience

I have had a dramatically different experience. Object-oriented (OO) programming has revolutionized my programming. Object-oriented programming helps me to organize. It helps me to visualize. It saves me time. It saves IBM money. It makes my code easier to maintain. Then why do so many people have problems with it? I think these people probably see object oriented programming systems as a *tool*. It isn't. *It is a way of thinking*. We want to teach you to think this way.

Different people learn in different ways. Recently, I have been learning about personality types. Intuitives tend to need lots of information to support the big picture. They tend to think of sensory types as simpletons. Once they have the big picture, they need samples to show how things are done. Sensory types need lots of lists and procedures and facts. They tend to think of intuitives as air heads. They also like examples. We provide both. Our first section covers the basic theory with plenty of samples sprinkled in. The next section contains many examples that are liberally augmented with text from beginning to end. You will get this information from different perspectives. We have three very different authors to bring you this subject. Scott is a SOM designer. He helped to invent and implement the metaclass framework and the constructors, as well as the C++ support. Paul is a frameworks guy. He was instrumental in the design and implementation of a prototype of a graphics framework using SOM on PM, AIX, and Windows. I have talked with many customers who live or die by this technology and have the scars to prove it. This is my second effort with VNR.

We were not satisfied to bring another SOM programming manual for beginners to market, though you can certainly learn SOM from this book. We also wanted to bring you the cutting edge of object technology. The third section is for experts. Scott provided three excellent chapters on the SOM API, the constructors, and the metaclass framework. Dick Berry, a leading user interface de-

signer, provided a chapter on a user interface design methodology. John Pompei wrote one on his C++ and PM frameworks.

We don't think that you will find a better treatment of objects in the context of OS/2. It just doesn't exist. We tried to keep it *lively*—from Paul's OO SOM programmer to his push button piano. We tried to provide different perspectives—from Scott's academic treatments with the flavor of a battle-hardened C programmer to Paul's light, informal examples to my 500-foot-high cruises over the material. You may not like all of it, but we are confident that if you are an OO programmer and an OS/2 programmer, you *will* find something that you like.

Bruce A. Tate

Acknowledgments

The authors would like to give a special acknowledgement to Mike Connor who originally focused on the importance of binary packaging technology for language neutral class libraries, to IBM research fellow Larry Loucks, who has consistently championed the cause of SOM, and to the following people who worked with Mike to realize the initial version of SOM: Larry Raper, Andy Martin, Nurcan Coskun, and Roger Sessions. We also owe special thanks to John Pompei and Dick Berry, who both contributed excellent chapters. They had very little to gain, but both contributions are invaluable to this book. Thanks to Maggie Tate for all of the long editing hours. Thanks also to Greg Fox for the programming assistance.

If real programmers don't eat quiche, then they don't ever back things up either. We certainly didn't, and thanks to Mike Philpot who saved the day at the last minute with his nifty OS/2 utilities. Thanks to IBM and our managers, who supported this effort much more than we could have believed.

Bruce would like to thank Maggie for daring to share the dream. Paul would like to thank Kathleen, Brigid, Deirdre, and Liam for all of the support at home, and his parents Bill and Delores Koenen.

Trademarks

IBM, OS/2, Presentation Manager, Workplace Shell, PS/2, SOMobjects, and SOM are trademarks of International Business Machines Corporation.

OMG, and CORBA are registered trademarks of Object Management Group.

UNIX is a registered trademark of UNIX System Laboratories, Inc.

MS-DOS, MS, Microsoft, and Windows are registered trademarks of Microsoft Corporation.

PostScript is a registered trademark of Adobe Systems Incorporated.

Times and Helvetica are registered trademarks of Allied Corporation.

X Windows and X-Window are trademarks of Massachusetts Institute of Technology.

Section I: Basics

1

Basic Objects

INTRODUCTION

The benefits are real. Perhaps you have heard them. Be more productive. Reuse your code. Understand what you have written. But object orientation is more than following a few rules to crank out bug-free programs at break-neck speeds. It grows beyond languages or tools—or even algorithms. It is a philosophy, a paradigm that changes the way we approach programming. It is a whole new way of thinking.

Years ago, we saw the same kind of revolution. Rules and conventions were created that allowed programmers to structure their code in a way that encouraged organized decomposition into functions or procedures. Known as structured programming, this *era* spawned a generation of supporting products. Researchers began to create languages and theory to support the new paradigm. Universities graduated a seemingly endless supply of computer scientists, most of whom were well armed to carry the structured torch to corporate America. It changed the way we program, and it revolutionized the way we *think* about programming.

Today, we are in the midst of a similar revolution. Object-Oriented Programming (OOP) is rapidly changing the way we think about all aspects of software development, from design, to implementation, to test to use. As before, tools and languages are being defined to support this paradigm. Universities are learning about the new model, and they are beginning to enlighten us through conferences such as ACM's OOPSLA, or Object Oriented Programming Systems, Lan-

guages, and Applications. User interfaces, such as OS/2's Workplace Shell, are supporting new and innovative ways to use object technology. A mad scramble in the industry to learn C++ and SmallTalk has ensued. Hopefully, this is why many of you bought this book.

We would like to introduce you to this brave new world as we see it, as it pertains to the OS/2 operating system. Many aspects of OS/2 support Object-Oriented Programming Systems (OOPS) in new and innovative ways. For the first time, system object model (known as SOM) is available within an operating system that allows us to implement objects using the language of our choice. The user interface is integrated into this model in some ways, as well. Some aspects of Presentation Manager also support OO concepts to various degrees. Before we dive into the details of OS/2, we would like to present our view of the benefits of Object Oriented Programming, and then present our conception of OOP.

THE BENEFITS OF OBJECT ORIENTATION

This book will mean nothing to you if you do not understand the benefits of OOP. Many projects have been buried beneath the latest fad or newest programming tool. We wrote this book because we believe that the technology is here to stay, and that the benefits of this technology are real. In fact, we have bet our careers on it. Object technology can increase productivity by speeding up high level design, increasing output throughout implementation, and reducing the number of objects implemented through better sharing. Testing is easier because more objects are shared, and encapsulated objects can only be accessed through standard interfaces. The use of inheritance reduces the need for dual maintenance, in many cases, and reduces its impact in others. Encapsulated objects with rigid interfaces of methods and attributes simplify the documentation process. Finally, OS/2's implementation of OOP is rich and powerful. Here is a more detailed look at why this technology is so important:

Encouragement of code sharing is built into the programming model. Of course, tools such as function libraries allow code sharing today. OOP takes sharing into a whole new dimension. We can use classes that are already ideal for our purposes. We can also override the behavior of a class to specialize it for our purpose, without undue maintenance, implementation, or testing burdens. Our specialized classes do not need to impact others using the same library.

Rigid interfaces; flexible implementation. Though OOP tends to encourage fairly rigid interfaces early in the design cycle, this is not true of the implementation. In fact, many times the easiest implementation is chosen for scaffolding and changed later. This type of decision making is ideal for a rapid prototyping environment, when complex or obscure functions can be simplified or left out.

Encapsulated objects are a natural way to break down complex systems. Many advanced software development tools incorporated data flow diagrams before OOP was in vogue. Databases are designed with entity-relationship diagrams. These schemes were developed over time because they work. They fit a natural thought process for the design in complex systems.

Though OOP is different from structured programming, OOP is a natural way to approach programming. Programming shops often experience five-fold or even ten-fold productivity improvements. Much of this improvement can be directly attributed to the intuitive model of encapsulated objects.

Object-Oriented Programming helps to partition larger projects. Objects make a natural partitioning for the delegation of work within large projects. Further, since OOP encourages the design of interface and behavior before implementation, system dependencies seem to be easier to manage. Changes of implementation can usually be confined to the objects that must change. OO concepts fit larger scales as well, with client and server objects.

Objects tend to handle rapid prototyping well. Many of the rapid prototyping tools and languages tend to be implementations of OO systems. Small-Talk, for example, is a language and environment that is ideal for rapid prototypes. The model seems to favor iterative development. This, in turn, favors early customer involvement, which is extremely important for designing an optimal user interface.

Objects fit modern graphic user interfaces well. Modern interfaces have characteristics that naturally map to the OO programming paradigm. Controls and windows map onto objects, events map onto messages, and icons and system objects, such as printers, are obvious visual representations of objects. On the OS/2 side, Presentation Manager supports many OO concepts. The Workplace Shell is implemented in SOM, which is the object model for OS/2.

ENCAPSULATED OBJECTS

Structured programs and object-oriented programs may use the same algorithm. They may be developed using the same tools, and they may even be written in the same language. The key difference is in the way we approach the task of programming. When we create a structured program, we define a task. If the task is complex, we simply decompose it into simpler tasks. When it is decomposed far enough, we can write procedures or functions that accomplish our desired operation. The key point is this: for structured programming, we *start with the whole* and decompose the problem, until we have a manageable level of complexity. We concentrate on *behavior*.

For object-oriented programming, we create a system (or program) by defining the objects that make up the system, and the relationships between them. Then, we implement the objects. The key point is this: for OOP, we start with the *object definitions* and concentrate on the *composition of the objects*.

That's it. Instead of the behavior of the program, you think about the objects, and the way that the objects will be composed to make up the system. After you have defined and implemented the objects and the relationships between them, you are finished. This paradigm is ideal for many of the problems that we encounter today. We will first look at the fundamental unit of OOP, the object. Then, we will look at the relationships between objects.

Objects, at their most fundamental level, are composed of data and methods. Data is expressed much like private variables in structured programming.

Methods are like functions. Both the implementation of methods and data within objects is hidden. Only the *method signature*, or the name of the method and its corresponding return value and parameters, is exposed to other users and objects. This concept is known as *encapsulation*. Indeed, many of the benefits of OOP can be directly derived from object encapsulation.

Instance Variables

Instance variables define the state of an object. OO programs, like structured programs, store data in *variables*. We call them *instance variables*, because the variable values may have different values for each different object of the same type. They are usually declared much like variables in any structured language. There are some key characteristics.

- The variables exist to describe the state of the object.
- The variables tend to be simple in nature. This means that as a rule, we will not use complex variables (such as structures) as instance variables. Complexity is achieved through object composition.
- The variables exist for the life of the object.
- The instance variables are only accessible by the object that declares them.
- Each different object of the same type (called an *instance*) has its own set of instance variables.

Instance variables can only be directly accessed by the object that creates them. We can only access the value of an instance variable by using methods designed to return their value. This may sound like an unnecessary restriction, but this discipline will give us enormous power. For example, consider an object called *person*. This object has instance variables for the person's *name* and *dateOfBirth*. There are three methods: *getName*, *getDateOfBirth*, and *getAge*. The first two methods return the value of the corresponding instance variables. The third computes the age by using the system date and instance data *dateOfBirth*. After some thought, the implementer decides that performance is more important than the space required to allocate an instance variable and decides to create an instance variable called *e*. No problem. Even though others may already be using this interface, they will not need to change their code because no other object even knows that the age instance variable ever existed. This is an example of the power of encapsulation.

Methods

Methods are the chief communication mechanism between objects. In SOM and many other OOPS, they are represented as functions. To declare a method, we need to specify the following:

- **Method name.** Users of this method will invoke it with this method name.
- **Parameters.** Unlike a function, a method always has at least one parameter. By convention, in most OOPS the first parameter is the object on which we wish to invoke the method. Some OOPS, like SmallTalk, force all parameters to be of type Object. Others like C++ allow many different data types.
- **Return value.** As with functions, the return value is optional. It is used to pass data back to the object that invoked the method.

Since methods can be implemented using function calls, most procedural languages can be used to produce OO code, though they may not enforce the discipline of encapsulation. In this case, it is up to the programmer to enforce encapsulation by limiting the scope of the instance variables to the objects that can see them.

Methods do not have to be implemented with functions. In this case, invoking a method is sometimes referred to as *passing a message*. OS/2 assists the programmer with message routing through a variety of subsystems. *Presentation Manager*, or PM, is a *graphic user interface* (GUI) that routes messages to the appropriate object, which is usually a window. PM programs are not always object oriented, but PM programs have many OO characteristics. The *System Object Model* (SOM) also assists the programmer with message routing, among other things. The *Workplace Shell* (WPS) uses components of SOM and PM. Messages may be sent to another program or process by using stacks, queues, semaphores, or shared memory. On the more complex side, many Client/Server programs are object-oriented. Both the client and server systems can be treated as objects. They may contain objects that send messages to objects that exist on a different system. Distributed programs may send messages using remote procedure calls. Database programming may be simplified by treating components of a database management system as objects. For each of these programming systems, OOP can be used to make life a whole lot easier.

CLASSES

Most of the time, when we define the methods and data that make up an object, we need more than one of them. For this reason, when we define the characteristics of an object, we are defining a *class*. A *class* is really an object type. When we need an object of a class, we create an object (called an *instance* of the class) with a complete copy of all instance variables that we define within the class. We can access the object by using any of the methods we define in the class definition. Before looking at some of the complex aspects of classes, it is time to look at a typical hypothetical and completely unrealistic example of a class with some instance data and methods.

Example: StackOfPizzas

We are designing a complex organization program for a pizza delivery store. The owner tells us that the more a pizza gets moved, the more cheese gets stuck to the top of the box. He wants to plan the way that he stacks up the pizzas in advance. We know that we will need a stack object. Of course, we computer programmers know life is much easier if the first thing that goes onto the stack is the last thing that comes off. The first example shows what our object data and methods will look like. The pseudo-code language is my own and does not reflect OS/2 syntax.

Class: StackofPizzas**Data:**

```
Integer TopOfStack  
Pizza Stack[10]; /* at most 10 pizzas on our stack */
```

Methods:

```
AddPizza(in Pizza thePizzaToAdd) returns an Integer;  
RemoveAPizza() returns a Pizza;
```

We should point out a couple of things about this example. First, when we specify instance variables, we also specify a type. Obviously, our system will probably not support a type called *Pizza*. This is a *class*, meaning that all elements of the array *Stack* will be objects that belong to the class *Pizza*. Since all *Pizzas* are encapsulated objects, I really don't care what the object looks like. You give me a pizza, and I'll put it on top of this stack. It can even be a special kind of pizza: one with extra cheese, anchovies, and pineapples. My object does not need to know what kind of pizza.

Second, there are some elements of the implementation that we might want to change. For example, we may want to put more than 10 *Pizza* objects on a stack. We can just change the implementation by changing the *AddAPizza* method and the dimension of the array. No user of this stack will have to change his or her code. We can also add interfaces without impacting existing code. Later, we will learn that with SOM, he or she won't even have to recompile! For example, we may decide to add an interface that will tell us which pizza is on the top of the stack, without taking it off of the stack.

Since we conveniently hid the implementation and data from other objects, we can make some radical changes without any impact to the rest of the system. For example, if we want to make the stack infinitely large, using a linked list instead of an array may be a better choice. No problem, even if our stack is used by many other objects. We change the data to provide the head of a linked list, change the add method to add new nodes to the head of our list, and change the remove method to remove nodes from the head of our list. We have just made sweeping changes to our *StackOfPizzas* methods, but we will not need to modify a single line of code elsewhere in our system!

Classes and Inheritance

So far, we have learned that objects communicate with each other primarily through methods. This is a relationship of *use*. Objects can have other relationships as well. In particular, there are *constructive* relationships. Constructive relationships can be split into two categories: *inheritance* or *specialization*, and *containment*.

Inheritance is a fundamental concept of OOP. Loosely speaking, it involves the creation of a new class by the modification of an existing class. The class that we wish to modify is called the *parent class* or *superclass*. The new class is called the *child class* or *subclass*. Creating a new class which is based on a superclass is called *subcassing*. Usually, we are limited to a couple of options when subclassing any given class. We can:

- **Change, or *override*, the implementation of a method.** This change can involve radical or minute changes to the parent class.
- **Introduce new instance variables.** The instance variables contained in the parent may not be sufficient for the purposes of the new class.
- **Add new methods.** If there are new instance variables that must be accessed by other classes, then new methods must be created to access them. New methods may be needed for other reasons as well.

This concept tends to revolutionize the way that we program for several reasons.

It lets us adapt classes which would normally be inadequate for our use. This tends to encourage reuse of the classes that we create. It also makes it easier for us to appropriate other classes! It encourages one version of the interfaces (the superclass's), which makes use of the new classes easier, and allows existing code to use the new objects (though they may not make use of the new functionality).

We can inherit the behavior, data, and interfaces that are useful to us without modification. OO programmers are basically lazy. We like to take advantage of our earlier work, or use anyone else's that suits our purposes. When we use inheritance, we don't have to write as much code. Since we don't even have to cut and paste the existing code, we don't have to test it twice or maintain it twice.

We can specialize general objects for more specific use. This problem comes up frequently in the programming of applications. For example, we may use a general part class liberally in our program. It may have interfaces to get the part name, part price, and part number. There may also be occasion for a gear class. We probably still need part names, numbers, and prices: information that is relevant to a part. In addition, we may also need instance variables for a gear ratio, and methods to access these instance variables. No problem. Create a part class and a gear decendent of part. **Gear** inherits all of part's interfaces, and defines the necessary specialized interfaces. To code that needs only generic part interfaces, any instance of part can be used. Instances of

any descendent of part may also be used, be it a gear or a screw or any other part. As they say, “parts is parts.”

OOP makes us more efficient. Programmers like to become more efficient, especially when we don't have to tell our managers that we are more efficient. There are several types of inheritance to consider.

Inheritance of Implementation

When we inherit from a class, we inherit the implementation of all of the methods and instance variables. We can choose to override, or change, the implementation of some or all of the methods, or we can leave it the same. We can add instance variables. With most OOPS, we can also choose to use the parent's method as part of our implementation, or ignore it all together. Let's look at our pizza example again. Suppose some user of the StackOfPizzas class needed an interface to count the pizzas in a stack. He or she did not want to modify the existing class, since existing users had no need for the new interface. Instead, this user decides to create a new class, NumberedPizzaStack.

Class: NumberedPizzaStack

Parent: StackOfPizzas;

Data:

 Integer PizzaCount = 0 initially;

Methods:

 GetPizzaCount() returns Integer;
 Override AddAPizza;
 Override RemoveAPizza;

Code:

```
AddAPizza (pizzaToAdd)
Begin
    PizzaCount = PizzaCount + 1;
    Return (Parent_AddAPizza(pizzaToAdd));
End

RemoveAPizza ()
Begin
    PizzaCount = PizzaCount - 1;
    Return (Parent_RemoveAPizza());
End

GetPizzaCount ()
Begin
    Return (PizzaCount);
End;
```

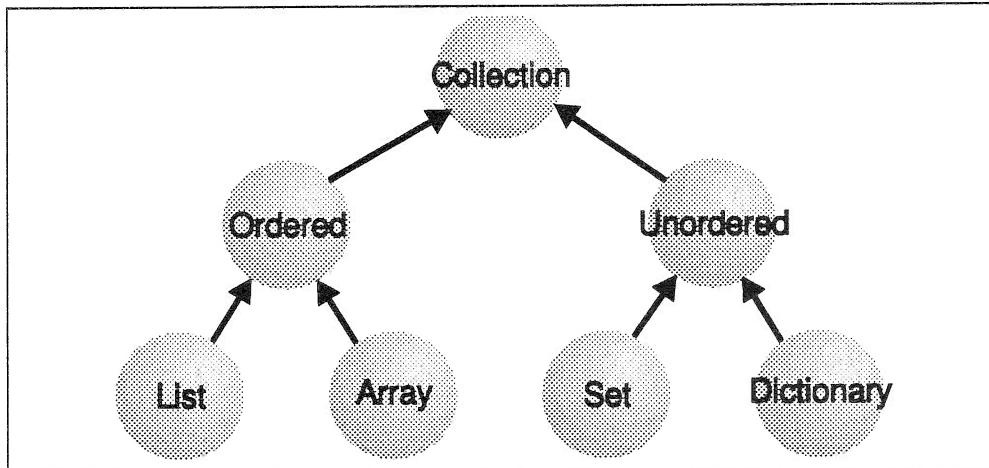


Figure 1.1. Collection Class Relationships

That is power. On the surface it may not appear to be, but look at what we have accomplished. We returned a count of the number of elements on the stack without knowing anything about the implementation of the stack. We modified the behavior of the `AddAPizza` and `RemoveAPizza` without knowing anything about the implementation. We kept the interface intact. We certainly do not have to extensively retest this class, though some regression testing and a brief test of the new method is in order. We do not have dual maintenance of the stack add and remove functions. The implementation of the stack can be completely overhauled, without affecting the new class at all!

Inheritance of Interface

There are times when inheritance of implementation is not what we are after. Sometimes, we need to inherit only the interface of an existing object. It may be that we want existing programs to work with our new objects, or we may want the users of our new class to feel comfortable by using interfaces with which they are already familiar. Either way, our new object inherit the interface from existing objects.

A *collection class* is an interesting utility class that can be used to hold other objects. Collection classes tend to be divided into collections that have order, like a list or array, and those that don't, like a set. In either case, many of the method interfaces will be the same. We will need methods to clear the collection, destroy the collection, add an object (a list would add the object to the end, though there may be specialized methods to insert an object into the middle of a list). We don't want to ask our users to relearn the interface each time they want to use a different type of collection. We should strive to make the interfaces as similar as possible, and only supply different functions to take advantage of the unique features of individual collections. We can do this by having a base class called a *collection* as in Figure 1.1. The collection will have all of the meth-

ods defined that are appropriate for all of the different collection types. The more specialized classes, *set* and *list*, can inherit the interface from the *collection* class. We will not actually write any code for the *collection* parent class, only the *set* and *list* children—we really do not expect anyone to use *collection* per se, only the *set* and *list* classes. We created the parent class for the sake of having a common interface for our collections. The *collection* class, then, is known as an abstract class.

Polymorphism

Combining implementation inheritance and interface inheritance can give us an incredibly useful feature known as *polymorphism*. This word means “many forms.” We can write code that uses a method of some general class. We know that subclasses will inherit the same interface. We can modify the method’s behavior by overriding it, but the interface remains unchanged. We can have many specialized implementations of the method, but the interface remains the same. In fact, the same code can invoke this method on different objects and get entirely different behavior.

Consider a light switch on a wall. This switch could be hooked up to any number of different appliances or devices. The light switch is the *interface* that remains constant, and the appliance is the implementation which can take many different forms. One interface, many different implementations.

The OS/2 Workplace Shell (WPS) has some good examples of this behavior. Items on the workplace desktop are objects. When you drag an object and drop it on the printer, the object is printed. Actually, when you drop the object on the printer, the *print* method is invoked on the object. The document object then knows how to handle itself. The WPS only knows how to handle a generic document object. The specialized documents may be graphic images, spread sheets, or complex word processor documents. In any case, the same print method can handle many different forms of the same class. The same concept is used to implement the shredder. Dropping an object on the shredder sends a *destroyYourself* message to the object. The object may have complex memory management and many different associated files, or it may be only a single ASCII file with no other resources, but the same method can handle all of the cases. This is because the WPS deals with the general class interface to invoke the object’s specialized method implementations.

Containment Relationships

Another type of constructive relationship is the *containment relationship*. Simply put, an object can contain other objects, much like an element of a C structure can itself be another structure. An instance variable of an object can be another object. Containment usually means that we can painlessly create complex structures without formal data structures. In fact, many object-oriented programmers shun the use of data structures. In practice, there is not much need for them, as object containment can satisfy all but the most stringent of performance demands. We can easily do a linked list with objects:

Class: link;

Data:

```
Object    Value;  
link      Next;
```

Methods:

```
getValue() returns Object;  
setValue(Object) returns nothing;  
getNext() returns link;  
setNext(link) returns nothing;
```

This C code fragment could traverse the list, printing all of the objects, assuming that *obj* has the first element in the list, and the list is NULL-terminated.

```
while (obj = getNext(obj)) {  
    print(getValue(obj));  
}
```

The first line gets the next link object. The second line gets the object stored in *Value* from the link object and prints it. This general example shows how to use objects in the place of data structures. In a real object-oriented world, we would probably not use a linked list. Some type of collection class would be more appropriate. We could add elements to a generic *list* and then forget about them until we needed them again. The *list* class would handle all memory management, finds, and things like ordering and sorting.

Multiple Inheritance

Sometimes a class needs characteristics of more than one parent. To provide a solution to this need, many OOPS provide a feature known as *multiple inheritance*. Several problems must be confronted for an OOPS to provide an effective way to inherit from more than one parent.

- We must provide an interface that will allow the child to specify more than one parent.
- We have the potential for name conflicts between the methods or attributes supplied by each of the parents.
- We have the potential for both parents to override the same method of a common parent. In this case, the new class must decide which implementation to invoke. This is known as the *diamond inheritance problem*.

When implemented effectively, multiple inheritance can open new worlds for the programmer who chooses to master it. The concept can be used to construct objects with very complex, but intuitive, behavior from existing classes. For ex-

ample, we may want to present the `StackOfPizzas` class to the user as a Workplace Shell object. In this case, the most natural implementation would be for the stack interface to inherit from the `WorkplaceShellObject` class, but there may be compelling reasons not to do so. If we want to have a light version of the stack class that does not have the overhead of a WPS object, then we can declare the stack class and a WPS stack class, which has `StackOfPizzas` and `WorkplaceShellObject` as its parents.

Multiple inheritance has allowed us to create the basic classes that we need. It is the final piece of the object composition puzzle. To build a practical and useful class library for our programs or our users, we need only provide the basic, essential classes. Our users can easily override methods to change the behavior or add interfaces or instance variables to add new features that specialize any class. They can create complex objects by creating instance variables that are objects. They can mix two or more classes together to form a unified, complex interface. Finally, we can use different implementations of the same method to make more efficient use of our classes through polymorphism.

CONCLUSION

The change to OOPS involves more of a shift in our way of thinking than new tools and languages. Grasping the concepts is not too demanding. Central concepts to the object-oriented programming model are:

- Basic units of objects are made up of methods and instance data.
- Strict adherence to encapsulation—methods are the only exposed interface.
- Inheritance allows specialization of both implementation and interface.

Though OO languages can make it easier to adhere to these concepts, with effort, they can also be implemented with procedural languages. Strict application of these basic philosophies can secure many benefits, including: increased productivity, shared code, tighter integration of programming model and design, easier use of today's GUIs, to list a few.

In the following chapters, we will attempt to put all of this in the context of OS/2. We will introduce the object model for OS/2, SOM. We will also discuss Presentation Manager, which implements some of the key concepts of OOP and simplifies the graphical rendering of internal objects of the operating system. We will talk about the communication mechanisms that are available to the OS/2 programmer, and we will discuss some of the other system interfaces. Most of all, we will try to orient you more closely to the world of objects.

2

An OS/2 Overview

ORGANIZATION OF OS/2

In the last chapter, we explored object-oriented concepts that will shape our programming process. In the next few chapters, we will begin to look at OS/2 2.x from a programmer's perspective. This chapter will contain the basic characteristics of OS/2 programs, and the organization of the services provided within OS/2's API (Application Programming Interface).

This chapter will seem to have little to do with object-oriented programming. Don't worry. Many of the earliest APIs within OS/2 were not developed with OOPS in mind. In many OO programming languages (such as C++), there is nothing particularly OO about the code within a method implementation. We will show you how to encapsulate the API into an OO framework. We will later show you how to seamlessly integrate the OS/2 programming services into OO programs.

Before we proceed, we should note that the topics in this chapter alone provide more than enough information for an entire book. There are, in fact, many excellent programming guides and references for the various OS/2 APIs. For this reason, we will present only the basic functionality and the most important function names here. For more detailed information on APIs, consult one of the many references or the excellent on-line text provided with the OS/2 2.1 programmer's toolkit.

Characteristics of OS/2

First, the basics. OS/2 is a *32-bit* operating system that supports preemptive, prioritized multitasking. By 32-bit, we mean that pointers have 32 bits of resolution to specify a memory address (actually, this is not *quite* true, for reasons that we will see later). *Multitasking* means that OS/2 is designed from the ground up to handle multiple concurrent tasks. *Prioritized* means that we can assign priorities to these concurrent tasks, and OS/2 will make sure that the CPU clock cycles will be allocated accordingly. *Preemptive* means that the OS/2 kernel has the power to take the CPU from one of these threads when necessary. (By contrast, Windows is a *16-bit, cooperative multitasking* operating environment. *Cooperative* means that each process must voluntarily relinquish the CPU.)

Units of Execution and Protection

OS/2 supports two units of execution. The first is a *thread*. A thread is simply a series of program steps running under OS/2. Like a single fiber of cotton within a rope, it cannot be divided any further. A distinguishing characteristic is that the thread cannot own resources.

The next unit of execution is the *process*. Like a piece of yarn, a process can be made up of one or more threads. A *process* consists of the code, data, and resources that make up an application. Of course, a process must have at least one thread. It can spin off new threads and kill existing ones. It can also own resources, such as file handles, semaphores, pipes, queues, windows, and memory. Some of these resources are shared with other processes, and some are private. In any case, resources of one process are *protected* from others. However, all resources within a process are accessible by all threads within the process.

This multi-threaded design has been quite popular among programmers. Since a thread owns no resource, it does not consume much system power. And since it has access to the same resources as all the other threads within a process, it can be easily implemented, without the complex interprocess communication overhead inherent in some systems. Both of these facts make it easy to take advantage of multiprocessing on OS/2.

One of the basic differences between DOS and OS/2 is the notion of protection. In DOS, nothing prevents one application from accessing the resources of another. For this reason, a rogue pointer can crash the entire operating system. This can also happen in OS/2, but the risk is only from applications that run at privilege levels. Since well written applications do not usually run at such high privilege levels, OS/2 is much more resistant to this type of error than Windows. For this reason, OS/2 has achieved great popularity for corporate mission critical applications.

APIs for Execution and Control

OS/2 requires a set of functions that allow users to create and control concurrency. In addition, facilities should allow different types multitasking within a

single application. OS/2 provides APIs for the process and thread, and sessions to help consistently manage the user interface of multiprocessing applications.

Process

A process is executed whenever an .EXE file is executed. Processes can be created programmatically as well through *DosExecProgram*. When this interface is used, the new process will be a *child* process of the creating process. There are various ways to handle a child processes resources and environment. The parent can choose whether to allow the child to *inherit* resources. Similarly, the child can be allowed to *inherit* the parent's environment. In any case, the child process will run in its own private address space. *Shared memory* is the only recourse available if the same memory is to be accessed by both processes. Alternatively, the problem may be solvable by using some of the interprocess communication (IPC) APIs provided by OS/2. The parent process can decide to continue to run concurrently (*asynchronously*) or wait until the child completes before continuing (*synchronously*).

Even if the child is started asynchronously, the parent process may later decide that it needs to wait on the completion of the child. For example, the child may create a file that the parent process needs before it may continue. In this case, any thread within the parent can issue a *DosWaitChild* process. The parent will wait for the completion of the child.

There are several functions that are related to the termination of a process. *DosKillProcess* is used to kill another process. A multitasking operating system must make provisions for some cleanup processing after a process has been prematurely terminated because a kill-process can leave a system in an unstable state. In OS/2 this is accomplished by *DosExitList*, which establishes a list of functions that are called when the process ends. *DosGetInfoBlocks* is used to get information about a process and the threads within a process, such as a process or thread identification.

Thread

Like a process, a thread can be launched to do some concurrent task. Threads may be more appropriate than processes for the following reasons.

- A thread does not consume as many resources as a process, and generally starts more quickly than a process. For this reason, performance can be better with threads.
- All threads within a process share the same resources, and reside in the same address space. This can often drastically simplify processing. Conversely, it is up to the thread programmer to make sure that concurrency is managed correctly.
- The thread API makes it easier to launch a function rather than an .EXE file, which can simplify processing.

Threads are created with *DosCreateThread*. This function is generally passed to an address of a function to execute. This address is usually a single parameter which may be an address of a data structure or object. Each thread uses a differ-

ent execution stack. The stack is created automatically by the *DosCreateThread* call, and you can specify the size.

Managing concurrency can include managing resources. *Mutex semaphores* are discussed later in this chapter, and they are excellent for this purpose. Managing concurrency can also require sequential access to a section of code. These code sections can be bounded by *DosEnterCritSec* and *DosExitCritSec*. When an OS/2 thread is within a critical section, it will not switch to another thread until the corresponding *DosExitCritSec* is encountered. Threads can be suspended with *DosSuspendThread*. Suspended threads are resumed with—you guessed it—*DosResumeThread*, and terminated using *DosKillThread*.

OS/2 is prioritized on a thread level. Thread priority is set with *DosSetPriority*. Setting an appropriate priority is extremely important for performance in a multitasking system. If the thread sets a priority that is too low, the process will not get enough clock cycles to accomplish its task. Usually setting a thread's priority is not necessary, because OS/2 does a sufficient job of setting priority by default. If the priority is too high, then other processes within the system will not get their required clock cycles, which defeats the purpose of multitasking. OS/2 boosts the priority of threads belonging to processes running in the foreground automatically. Similarly in PM, the thread containing the main message loop is given an automatic priority boost, keeping the UI snappy.

Multithreading is especially important in a PM application. Since the priority of a thread is boosted whenever a thread creates a message queue (which happens when you create a window), it is not a good idea to do much else within that thread. The priority is boosted so that the user interface can be responsive and smooth. If too many other tasks are done, then the user interface will not get enough clock cycles. Since the priority of a thread is set high, and it is not predominantly used for user interface, something will suffer. First, the user interface will be sluggish. Second, other applications, which already have a lower priority because they are in the background, will grind to a halt for all practical purposes. This is not good behavior for a multitasking system! Whenever you need to do a task that is longer than a fraction of a second in a PM thread, start a new thread to do it. OS/2 makes this easy.

Sessions

An application may be composed of more than one process. For this reason, *sessions* are defined so that a single application can have a single set of buffers for keyboard, mouse, and screen management. A session is started using *DosStartSession* and terminated using *DosStopSession*. Like a process, a session can be a *child* of the session that creates it, but no relationship is required. The new session can be one of five types. They are:

- Full screen, protect mode
- Full screen DOS mode
- Presentation Manager
- Text window, protect mode
- Text window, DOS mode

It is possible to programmatically switch to a session using either *DosSelectSession* or the *OS/2 Task List*. Since launching a program usually automatically creates a session when necessary, it is generally not necessary to manage sessions, but these calls are available for the times that you do!

INTERPROCESS COMMUNICATION

Since processes do not share the same address space, they cannot use common objects or (gasp) Global Variables to communicate. They must have other ways to synchronize and communicate. Files are one possibility, but OS/2 provides many other services that allow better performing, less static interprocess communication (IPC). In addition to files, OS/2 provides four ways of doing interprocess communication: *queues*, *threads*, *pipes*, *shared memory*, and *semaphores*. Each is best suited for a different type of communication. *Semaphores* make effective signals or controls. Like a traffic light controls access to an intersection, semaphores control access to various system resources. Since processes resources of each process are protected from other processes, *shared memory* in the form of *named shared segments* must be used to provide common memory segments. *Named pipes* can be used to send a stream of data from one process to another. Like a hose, a pipe has a source and an outlet. One process opens the writing end of a pipe and gives it a name, much like a file. Another process uses this name to open the pipe, and reads the data supplied by the writing process. *Queues* are basically pipes that can be read by only one process, but queues may be written by many processes. Data elements on a queue may be simple messages or pointers to memory segments that contain application-specific data. Timers can be used to signal a process of given time intervals.

Interprocess Communication API

To review, *semaphores* are usually used like a control flag or Boolean variable to send a signal or denote ownership. *Shared Memory*, discussed under memory management, can be used to provide access to more than one process. *Pipes* can be used to send a stream of data from one process to another, and *queues* can be used to send one 32 bit word (which can be an address) from one process to another. *Timers* can be used for synchronization.

Semaphores

The semaphore API changed drastically between OS/2 2.0 and its predecessors. In OS/2 versions 1.x, the same semaphore manages different logical operations. For OS/2 2.x, several different semaphore types were established. In general, the function names take the following form

DosVerbTypeSem

where *Verb* is the action applied to the semaphore, and *Type* is one of the semaphore types defined below.

Event semaphores are used to manage events, much like a traffic light. The paradigm here is a signal. An owning process creates the semaphore. Other processes can then open and use the semaphore. One or more processes can use the semaphore to wait on a signal via a *DosWaitEventSem* call. The processes will release control if the semaphore is in a *reset* state and will resume once the semaphore is *posted* by some other process via a *DosPostEventSem*. This type of semaphore is excellent for process synchronization. When a process has no more need for the semaphore, it is closed via a *DosCloseEventSem* command. The verbs that may be applied to event semaphores are *Create*, *Open*, *Wait*, *Post*, *Reset*, *Query*, and *Close*. For example, the command that is used to create an event semaphore is *DosCreateEventSem*.

Mutual Exclusion semaphores, denoted by the type *Mutex*, are used to protect resources such as files or shared memory from undesirable concurrent access. A good analogy here is the narrow part of an hour glass, which limits access to a single grain of sand. The process is the sand, and the space separating the top and bottom of the hourglass is the resource. Mutex semaphores can restrict access to code segments that must be sequentially executed. The paradigm here is ownership. A process that obtains ownership of a semaphore through *DosRequestMutexSem* has exclusive access to the corresponding resource until that semaphore is released. Some of the verbs that can be applied to a mutex semaphore are *Create*, *Open*, *Request*, *Release*, *Query*, and *Close*.

Multiple Resource Mutex semaphores, denoted by the type *MuxWait*, are like mutex semaphores, except that instead of requiring a single resource, the process requires a list of resources. A *Muxwait list* must be created of existing, open semaphores. This list is just an array of semaphores. The list must contain either event or mutex semaphores, but not both. Of course, the semantics of the muxwait semaphore is different, depending on the type of semaphore contained. For example, for a file to be printed, a program must have access to the file and the printer queue. Without both, the print program cannot continue. The verbs that can be applied to muxwait semaphores are *Create*, *Open*, *Add*, *Delete*, *Query*, *Wait*, and *Close*.

Like most of the interprocess communication mechanisms, semaphores may be named or anonymous. Named semaphores are given a name of the form `\SEM32\NameIdentifier`. The `\SEM32\` is a case insensitive prefix required by the operating system for uniqueness. Anonymous semaphores must be managed by handle only.

Whenever a semaphore (or other IPC mechanism) is created or a named one opened, a handle is created. Subsequent accesses of the semaphore occur through the handle, so that OS/2 can achieve adequate levels of performance and obscurity. If a semaphore is opened more than once, a count is maintained, and the semaphore will not be released until an equal number of closes occurs.

Pipes

Pipes are like files that can be written on one end and read from another end. Consequently, the API looks much like OS/2's file API. Like semaphores, pipes may be named or anonymous. Named pipes are required to have the case insensitive prefix `\PIPE\`, much like named semaphores.

Anonymous pipes are accessed through functions of the form

`DosVerbPipe`
or
`DosVerbNPipe`

for named pipes where *verb* is one of the operations defined below.

Create allows the creation of a pipe. *Open* allows a pipe to be opened for reading. *Close* terminates the usage of an opened or created pipe. The creating process can *write* to a pipe, and the opening process can *read* from the pipe.

In addition, several functions are only available to named pipes. First, the server creates the named pipe, at which time it is in a *disconnected* state. *DosConnectNPipe* is used by the server process to put the named pipe in a *listening* state, at which time client processes can open the named pipe. The server and client can then communicate through reads and writes. The client then issues a *DosCloseNPipe* to sever the connection on the client side after which a *DosDisconnectNPipe* issued by the server acknowledges the close on the server side, once again leaving the pipe in a disconnected state.

The named pipe API also contains additional functions that allow the server process to effectively manage concurrency through semaphores and manage duplex message pipes, as well as many other functions. Named pipes can also be used for communication in a client/server network environment.

Queues

Queues, like pipes, are used to transfer data from one process to another. The difference is that pipes are used to transfer a continuous stream of data while queues are used to transfer one message at a time. The message can be a word of data or a pointer to a segment of memory. The queue owner creates the queue via *DosCreateQueue* and reads from it via *DosReadQueue*, or looks at elements without removing them via *DosPeekQueue*. Only the owner can read from the queue. Information is read from the queue in first-in-first-out order (FIFO), last-in-last-out order, or in the order of the priority specified by the writing process, regardless of the order in which they were written (ties are read FIFO). Like semaphores and pipes, queue names have a case insensitive prefix: `"\QUEUES\"`.

Once the queue is created, other processes can open it by name via *DosOpenQueue*. This returns a handle, which is used to access the queue in all subsequent accesses. These processes are then free to write elements to the queue, which are 32-bit words. If the written word is a pointer, it is important to make

sure that the queue reader has addressability to the memory segment. The writing process can also specify a priority, if needed.

Some additional functions are also available to the server. *DosPurgeQueue* can be used to remove unconditionally all elements from the queue. *DosQueryQueue* can be used to return the number of elements in a queue.

One final note: These queues are not to be confused with PM Message Queues, which are discussed later in this book, or with printer queues.

Timers

For a multitasking operating system, timing operations are frequently important to relinquish control of the CPU or coordinate activities with other processes within a system. Several types of timers can help make these things happen.

In OS/2, unrestricted *wait loops* or *spin loops* that do nothing but wait for events can have a devastating effect on system performance. Of course, the best solution is to structure programming logic to make this unnecessary through other types of interprocess communication. Another solution is to suspend the thread for a short time using *DosSleep* within the spin loop. *DosSleep* is a function which is managed by the operating system that suspends an application for a specified amount of time. It only suspends the current thread, not the entire process.

While *DosSleep* can provide some level of timing, *asynchronous timers* provide better precision. *DosAsyncTimer* can be used to post an event semaphore at a specified time. Alternatively, *DosStartTimer* can be used to start a timer which repeatedly posts an event semaphore whenever the specified time interval expires. *DosStopTimer* is the complementary function that stops the timer.

Object Strategy for IPC Services

There are two basic strategies for integrating the various IPC services (which are procedural in nature) into an OO architecture. The minimalist approach is to use the API "as is" within the method implementation. Basically, we treat the API as a primitive within our language. This answer is frequently adequate, as the C++ language has an extensive function library that is often used liberally within method implementation code.

Another possibility is to use the API to implement objects such as named pipes or timers. For example, an *eventSemaphore* object could have instance data for the handle, name, and timeout. It could support methods to query, post, wait, open, and close the semaphore. Pipes, queues, and timers are similarly well designed to take advantage of this design.

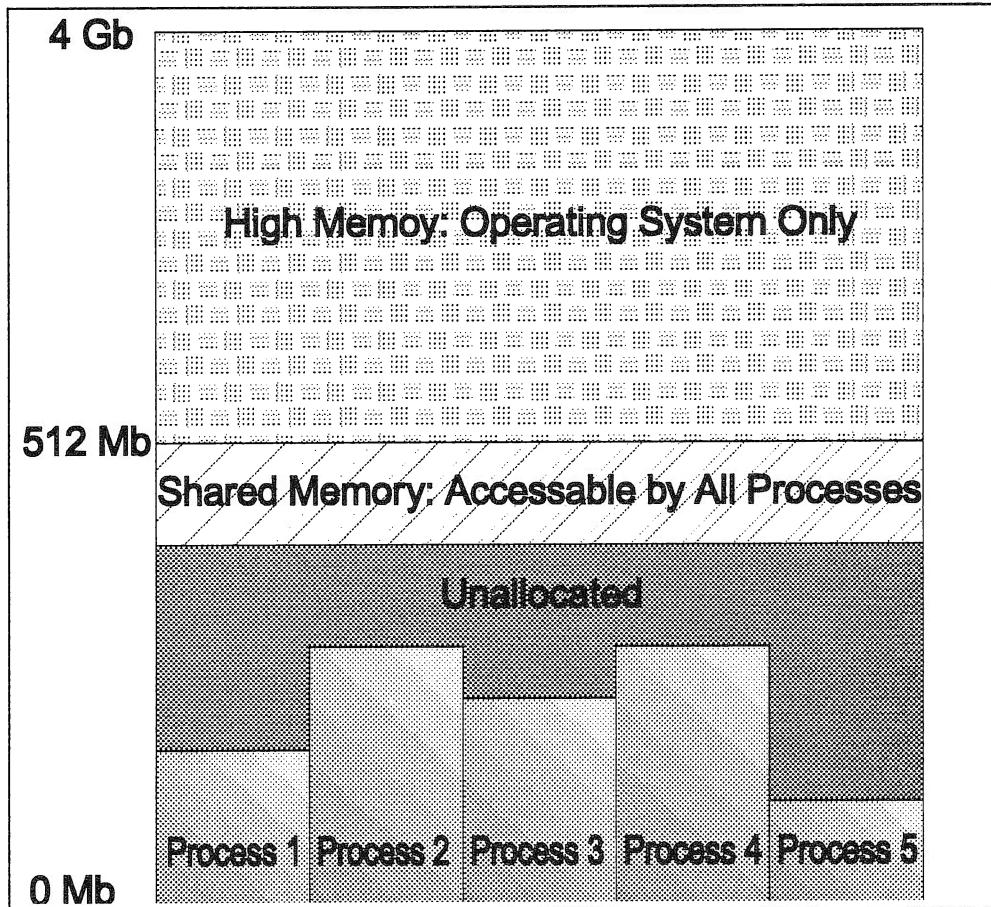


Figure 2.1. OS/2 Memory Management

OS/2 MEMORY MODEL

The OS/2 memory model is designed to provide a framework for portable, well performing applications. OS/2 offers each process a flat memory space that is accessed by 32 bit addresses. Memory is protected and manipulated in terms of *pages*. Both of these characteristics are similar to a wide range of platforms today. While a 32-bit pointer is sufficient to access four gigabytes of memory in theory, an application is limited to only 512 megabytes of memory for reasons that we will explore later. At least 64MB of this memory is dedicated to shared memory, leaving a scarce 448MB of memory for each application.

Each process runs in its own private address space. This means that many processes may have data stored at pointer address 0 though most C compilers reserve address 0 for NULL which is an invalid pointer. Each process's private space begins at physical address 0 and goes up to 512MB (minus the shared memory). Figure 2.1 shows the OS/2 memory model. The shared memory ends

at the 512MB line and encroaches on at least 64MB of each application's private address space. All memory above the 512MB line is dedicated to the operating system.

Paging

All 32-bit programs will be implemented using *paging*. Each page in an applications private address space is mapped onto physical memory by the operating system. If memory allocation ever exceeds the amount of free memory within the system, some pages will be written to disk to make room for the new allocations, if adequate free disk space is available.

Memory is allocated and manipulated in units of memory objects. Each memory object is composed of one or more pages. The size of the memory object is then rounded up to the next multiple of 4KB (the size of a page). Memory is reserved for use in two steps. Allocation of memory reserves an address range of pages for use, but the memory is not yet assigned to physical storage. *Committing* memory actually assigns it to physical storage. Multiple pages can be committed at the same time. When this happens, the pages will have sequential addresses.

When performance is a critical consideration, some key factors should be considered. First, the number of allocations should be minimized. Ideally, one large segment should be allocated. Pages can be committed as needed. Also, since at least 4KB will be used for each page even when less is allocated, pages should be allocated in 4KB intervals. Too many small allocations can eat up memory in a hurry. For these reasons, many applications allocate a larger heap and suballocate from their heap. OOP makes effective memory management easy because one object can be created to handle memory for the entire application. This object can have a consistent interface, even if the memory management scheme or implementation changes dramatically.

You can also take advantage of understanding OS/2 memory management when you link. When linking OS/2 applications, several .obj files are linked together into a single DLL. If these .objs are packaged in 4Kb multiples, fewer pages will be required to hold them. You can also assume that many small memory objects packaged separately is a bad design, because a minimum of 4KB is allocated per segment.

Object Considerations

Memory management must obviously be considered whether the programming language is procedural or object-oriented, though some OOPS, like Lisp and SmallTalk, do handle memory management automatically. In fact, within the method implementation, the memory management considerations for objects and functions are nearly identical.

And now some good news. Many compilers will do this memory management for you. Some do it quite well and are completely transparent to you. In fact, the memory management routine within SOM is based on the C routines malloc()

and free(). For this reason, we will not discuss OS/2 memory management in any more detail, but some historical elements should be explored.

History

When the first version of OS/2 was developed (OS/2 1.0), IBM, in its infinite wisdom, thought that OS/2 1.0 needed to be made to work with the many 80286 systems that were available. These systems supported segmentation instead of paging as a memory management technique. With segmentation, memory segments with 16 bits of addressability were allocated. Only a single segment could be allocated for any given memory object. The operating system achieved greater than 16 bits of addressability through page switching. A page allocation table was maintained. Any complete memory address, then, consisted of a *selector*, which determined the correct memory segment, and an *offset*, which determined the appropriate location within the segment.

OS/2 2.x maintains complete compatibility with the older versions of OS/2. This is done by dedicating a bit within an address pointer to the type of address, which is either a flat 32-bit address or a selector/offset pair. In addition, two bits are used to determine the privilege levels that an application must have to access these memory segments.

This information is important for three reasons. First, this means that an application cannot access a full four gigabytes of memory, as it could if the full 32-bits were dedicated to determining a unique address location, because it has access to only 29 of the 32 bits. Second, it determines the way in which addresses are interpreted. To determine a selector/offset type address, the offset is used as the least significant portion of a 32-bit flat address, so the hex formula to convert a selector:offset address to a 32-bit address:

$$(1FFF \text{ AND } \text{selector}) \times 10000 + \text{offset}.$$

(1FFF AND selector) serves to mask off the three most significant bits of the selector. Then, we multiply by 10000 hex to shift the address left 16 bits. Third, when writing programs that mix the memory models (such as a flat 32-bit addressing type application calling a selector/offset type function), it is important to take these factors into consideration. Usually this is accomplished with a layer between the dissimilar function and the calling application called a *thunk*. And no, this is not the past participle of the IBM motto.

Thunks

A *thunk* is a layer which is responsible for handling the considerations associated with mixing flat 32-bit addresses with 16-bit addresses. It must consider these factors.

- Addresses will need to be converted. Flat 32-bit addresses will be converted to a selector and an offset and vice versa. This conversion is easily accomplished using the formula above.
- Memory objects which are larger than 640KB will need to be decomposed. Sometimes this can be handled within a thunk, but often the basic program architecture needs to be revisited.
- Memory objects cannot span two 640KB segments. Given the two architectures, flat 32-bit objects can span two 640KB segments. This is not permitted within the selector:offset addressing scheme.

The program which requires the conversion calls the *thunk*, which converts all addresses within the parameters to the appropriate scheme. The thunk also converts addresses which are contained in objects or data structure parameters. Additionally, the address must not point to a memory address which spans two 640KB segments, or a segment larger than 640KB.

Swapping

With both types of memory management, using OS/2 a program can allocate more storage than is available within RAM. Either a page or a segment can be temporarily written to disk to accomplish this. In either case, memory resources are still not infinite. The total amount of memory available to the operating system is determined by the amount of system memory combined with the unused disk space within the disk partition that contains the swapper. This partition is determined by the *SWAPPATH* environment variable. When this space is exhausted, subsequent allocations will fail (or *commits* for flat 32-bit allocations).

One of the reasons that OS/2 2.x provides better performance than previous releases is that pages are all the same size, while segment sizes vary. Varied sizes for segments means fragmentation is a concern and the math is usually more involved. In addition, the flat memory address does not need to be translated, but we have to do some conversion to translate the selector/offset to an address.

Memory Management API

Using memory within OS/2 consists of the following steps. They are:

- **Allocating** the memory object using *DosAllocMem*. This allocates entries in the page allocation table, but does not commit physical storage. Memory segments are allocated in 4KB pages. Smaller allocations are rounded up.
- **Committing** the page or range of pages using *DosSetMem*. This commits the physical memory needed for the page. *DosSetMem* with different parameters is also used to decommit a page or range of pages.
- **Using** the memory object.
- **Decommitting** the pages.
- **Freeing** the memory object using *DosFreeMem*.

In addition, *DosQueryMem* can be used to obtain information about a range of pages within a process's address space. After memory objects are allocated, they may be suballocated as needed. This is done using the following steps. They are:

- **Initialize** memory objects for suballocation using *DosSubSetMem*.
- **Suballocate** from the object using *DosSubAllocMem*.
- **Use** the suballocated memory.
- **Subfree** the memory using *DosSubFreeMem*.
- **Terminate** the use of a memory object for suballocation using *DosSubUnsetMem*.

You can remember these steps through the acronyms *ACUDF* and *ISUST*, which probably mean something in some language. A memory object may only be used by the process that created it unless it is specifically allocated as shared memory. Shared memory is allocated as named shared memory objects or anonymous shared memory objects. Creating a shared memory object is done with *DosAllocSharedMem*. Other processes can get access to a named shared objects with *DosGetNamedSharedMem* or unnamed shared objects with *DosGetSharedMem*. A process can give another process access to a shared segment with *DosGiveSharedMem*. Shared memory is particularly useful when using queues to send larger messages.

EXCEPTION MANAGEMENT

OS/2 is a much more stable operating system than its predecessor DOS. Part of this is due to address space protection and preemptive multitasking. Another factor is a rich exception architecture that allows applications to easily react to unforeseen errors or system problems. In addition, exception handlers can often handle many routine programming exceptions such as growing a stack when its limit is reached much more easily.

Exception handlers are functions that are called when an exception occurs in a thread. All of a program's exceptions are registered on a per-thread basis by a call to *DosSetExceptionHandler*. Execution then occurs within the process until an exception occurs. Some exceptions include *divide by zero*, *access violation*, or *guard page violation*.

Guard pages are allocated by an application and placed on the appropriate boundaries of a sequentially growing memory space. When the process attempts to access the guard page, the exception handler is invoked, which can convert the page to a normal page after allocating more memory. This is a quite elegant way to manage growing memory objects, such as stacks, without needless complication of program logic.

In addition, some *signals* are grouped under the umbrella of exception management. Signals include *kill process*, *control-c*, *control-break*, and user-defined signals. Unlike other exceptions (which are handled on a per-thread basis), signal or *asynchronous* exceptions are all sent to the main thread of an application. This generally makes these exceptions much easier to handle. In addition, only

the process that has the *signal focus* within a screen group will receive a signal exception. Signal focus is obtained by issuing a *DosSetSignalFocus* with the *flag* parameter set to *on*. Signal focus is relinquished by calling the same function with the *flag* bit set to *off*.

Processing an exception is much like drinking a good, cold beer. Once you get started, you really do not want to stop. The same is true of exception handlers. Interrupting serious exception handlers can sometimes cause situations which are unstable or undefined. For this reason, exceptions can contain *must complete* sections bounded by *DosEnterMustComplete* and *DosExitMustComplete*. Once this section is entered, further exceptions are deferred, and these exceptions must be raised later with a *DosRaiseException* call. Signal exceptions are the exception (pun intended) to this rule. They are automatically delivered to the application once the *must-complete* section is exited. *DosRaiseException* can also be used to simulate exception conditions or raise user-defined exceptions. User-defined exceptions are handled exactly like system defined exceptions.

FILES

File management within OS/2 is not remarkably different from a programmer's perspective with two exceptions. First, OS/2 allows users to choose the most appropriate file system through the *installable file system* feature. A different file system can be installed for each logical partition. This is important to the OS/2 programmer because some file systems implement *lazy write*, which delays the writing of information for performance reasons. For some applications (such as database managers or journaling), this may not be desirable. Second, OS/2 must maintain compatibility with DOS. In addition, it must build in support for the additional information needed by the Workplace Shell and long file names. This information is stored within a structure known as *extended attributes*. For this reason, when working with files in the OS/2 environment, it is important to understand that extended attributes must be preserved.

Installable File System

OS/2 comes with two alternate file systems. This is possible because of a feature called the *installable file system*. One file system is provided for compatibility with DOS. It is known as *FAT*, which stands for *file allocation table*. FAT file systems were created for early DOS, which was basically a diskette based operating system. It is good for small disks with few files per directory. *Fragmentation*, caused when many files are created and deleted leaving many small fragmented holes, is a problem. The *high performance file system (HPFS)* was created to solve these problems. HPFS uses more sophisticated data structures to store the directories. In addition, it supports longer file names and is less susceptible to fragmentation.

The installable file system feature defines the protocol for creating new file systems, and maps OS/2 APIs onto the file system. This is done dynamically, so that more than a single file system can be installed on a single computer system. In fact, different disk partitions can have different file systems!

Extended Attributes

The workplace shell requires a place to store additional information about files and directories such as icons and friendly, multi-line file names, and associations (in OS/2, .WKS files can be associated with Lotus 123(TM)). This information is stored in extended attributes. This information can be accessed through an API. The most important thing to remember about extended attributes is that they exist, and file management programs that do not take them into account will loose them, with dangerous results that can compromise system integrity. Many programs that were written for DOS can possibly loose the extended attributes, so be careful.

DYNAMIC LINK LIBRARIES

Dynamic Link Libraries, or DLLs, add incredible power to OS/2. They are basically function libraries that can be automatically or manually linked and loaded at run time. This means that the operating system and its applications can be easily extended with new libraries and services, without recompiling! There are several types of information stored within a DLL:

- **Code Segments.** These segments contain the executable program instructions. Each .obj file that is linked into a DLL can be its own code segment, or several .objs can be combined into a single code segment. When more than one instance of a program is executed, one shared copy of the executable code is loaded into memory. It can either be loaded on demand (when the DLL function is called that requires it) or when the DLL is initialized.
- **Data segments.** These segments contain information in the data that is required by functions within the DLL. This data can be shared, in which case a single copy is loaded for all process instances which call it; or private, in which case a separate copy is loaded for each instance. Like the code segments, they can either be loaded as needed or at DLL initialization time.
- **External References.** Each DLL needs information about external references. These are the functions that are required from other DLLs. This information is stored in an *import table* in the header of the DLL. It is represented in the form of the name of the DLL supplying the required function or data, and the name of the function or variable required. External references are only present for elements that are to be dynamically linked. As we mentioned previously, it is possible to link to DLLs manually, through the API defined below.

- **Offsets for Exports.** Each DLL provides information about the public data and the functions that it provides. This information is stored in an export table within the header of the DLL. It is stored in the form of the name of the function or variable and the offset into the DLL.
- **Additional information.** Within the header of the DLL, there is also some additional information. This includes the name of the DLL, the default loading instructions for the code and data segments (on demand or at initialization time), and possibly some debugging information, depending on the compile and link options.

A DLL can be loaded explicitly by the instruction `DosLoadModule` defined below. It may also be loaded implicitly through *dynamic linking*. Whenever an executable or DLL is loaded (implicitly or explicitly), it also loads all of the DLLs that are found in its export table. This is a recursive procedure, so these DLLs can also load others. All import and export references are then resolved through the magic of dynamic linking, using the information that is stored within the DLL headers.

Performance Considerations for DLLs

Dynamic linking does take time. For this reason, it is desirable to combine as many code functions into a DLL as possible, so that most of the linking can be done before run time. There are some exceptions, when using separate DLLs may be is desirable:

- When creating separate, installable features, using a DLL or EXE is almost always desirable.
- When creating separate utility libraries that may be used by different programs, using a separate DLL may prevent the loading of un-needed information.
- When startup time is crucial, some initialization code may be placed in one DLL, and other DLLs explicitly loaded later with `DosLoadModule` (possibly from another thread).

CONCLUSION

This OS/2 API is not particularly object oriented when taken alone, but it does have some object oriented characteristics. Many of the constructs support handles. We can logically view these handles as object identifiers, and the functions that we use to manipulate the objects as methods of the class implementation. Of course, the basic OO concepts of inheritance and polymorphism are not supported, but if we are purists, we can wrap them as classes easily enough, and specialize them as needed. A more common approach is to look at the system API as primitives that can be used within a method implementation.

In the next chapter, we will talk about Presentation Manager. Notice that successively higher layers within OS/2 are increasingly object-oriented. This will make a big difference when we create examples for user interface.

3

Presentation Manager

INTRODUCTION

Presentation Manager is the system that provides window management for OS/2. It has not achieved the market share of the Windows API, although most programmers love it, and few would argue against its technical superiority. PM provides the eyes and ears for OS/2. These window management functions include:

- **Window hierarchies.** Windows are objects which are instances of *window classes*. PM provides facilities that allow inheritance of parent behavior. In addition, a rich set of default *messages* is supported. These do the things you would expect out of a window manager, including *clipping* children that are drawn outside of the parent's perimeter, routing messages to child windows and controls (which are themselves child windows), and *painting* the window appropriately.
- **Message queues.** PM provides the facilities to get and dispatch messages through *message queues*. They are used to route *messages* to the appropriate window. A single message queue functions for the entire operating system. In addition, each application window has its own message queue. *Events*, such as mouse button presses or key-strokes, are delivered as messages to the appropriate window through message queues.

- **Controls library.** Dialog boxes and controls, such as *push-buttons* and *list-boxes*, are just PM windows. PM provides a set of customized controls, each of which may introduce new user defined messages.
- **Device independence.** We have learned that an important object oriented concept is encapsulation. PM insulates hardware devices from the programmer. Programs are written in a device-independent *presentation space* and then mapped onto a *device context*, which is associated with a device.
- **Graphics programming.** PM provides an entire *graphics programming interface* (GPI). It is a *vector-based API*, which means that geometric shapes are described instead of points.
- **Font management.** A difficult problem that must be solved with all window managers is font management. This involves font definition, selection, and rendering. Since characters are drawn in this GUI environment, rich font management is imperative. Furthermore, proportional fonts introduce an entirely new level of complexity. We must be able to determine the size of a string exactly, and this is different for each font. We must make it possible for each string to be rendered using a different font, color, and size.

This API is quite powerful and diverse. We cannot hope to give you comprehensive coverage. However, we can arm you with the knowledge of PM organization, facilities, and basic philosophy. This chapter will concentrate on the PM basics, focusing on the most important facilities available and giving a general description of their use. If some of the details that you seek are missing from this chapter, do not despair. Many complete programming examples are provided in the second section of this book.

PM BASICS

PM is based on windows and messages. Windows are created, deleted, and manipulated through the use of messages. Some messages produce side effects, such as resizing, maximizing, or minimizing a window. Some just provide notification of events, such as the press of a mouse button or keyboard key. It follows that the primary function of a PM program is to get messages and to send (or *dispatch*) them to windows. In fact, PM programs usually do little more than getting and dispatching messages within a loop, and reacting to these messages within a window classes implementation, called a *window procedure*. Every PM program will have some form of the following message loop:

```
while (message != WM_CLOSE) {  
    WinGetMessage(...);  
    WinDispatchMessage(...);  
}
```

This is called the main message loop. Once messages are dispatched, each message is handled on a case-by-case basis. Of course, if a message is unimportant to an application, then the default implementation can be used, or the message ignored.

PM Windows

A window is a rectangular area on the screen, though a window need not be visible. Windows have exactly one parent, except for the special *desktop* windows, which have none. The *workplace shell desktop* is an example of a desktop window. Applications reside in a window called the *main* window. By the *common user access (CUA)* convention, each application only has one main window. This window can have as many *child* windows as is necessary. When a window is created, a *parent* is specified. A child window can change its parent window if necessary. All child windows are *clipped*, which means that no part of a child window is visible outside of the parent window.

Windows have two dimensions. A third logical dimension that is also important is the *z-order*, which refers to the z axis. If a window partially or completely occupies the same area as some other window, the window with the highest z-order is displayed. Parent windows are lower in the z-order than their children. This makes sense when we look at most graphic applications. Push buttons and scroll bars are children of larger windows, and they always cover their parent window. We can make child windows transparent to overcome this effect when necessary.

Creating Windows

Windows can be created in several ways. Standard windows are created using *WinCreateStandardWindow*. A standard window has many of the common characteristics that are important to conform to CUA conventions. Standard windows have some or all of the following built in features:

- **Scroll bars** at the right side or bottom of a window. These can be used to move the physical window within a logical window that can be much larger.
- **Menu bars.** These appear at the top of a window and are used to present the user with appropriate application choices.
- **Title bars.** These contain a window title, and can be used to drag the window.
- **Size borders.** These can be used to make the window larger or smaller.
- **Minimize and maximize buttons.** These can be used to change a window to full size, normal size, or icon size.

Windows that do not need these options, or windows that have additional requirements, are created with *WinCreateWindow*. Like many entities within OS/2, windows are represented by *handles* called *window handles*. When a win-

dow is created, a window handle is returned. This handle is used to refer to the new window in messages and *Win* function parameters. Windows are destroyed via *WinDestroyWindow*.

Inheritance and the Window Procedure

Each window has an associated *window class*. A *window corresponding to a class* is said to be an instance of the class. This should begin to sound familiar. You have probably guessed that window classes are arranged in an inheritance hierarchy. This means that one window class can specialize the behavior of another. We define the behavior of a window class through its *window procedure*.

The window procedure is defined by the application, but actually runs within PM's process. Whenever a window class is created, its window procedure is specified. Whenever a message is dispatched to a window, its window procedure is called. Now, the function of a window procedure is to respond to messages. In C, a common implementation of a window procedure is the switch statement. A message is passed into the window procedure. The switch statement then processes each message on a case by case basis. When a procedure's message implementation is completed, the window procedure usually passes it on to its parent class's window procedure. The parent will process the message and pass it on, until it ultimately reaches the ultimate parent's implementation—the default message procedure. A window procedure can potentially receive hundreds of different message types. In all but the most unusual circumstances, most messages are ignored and simply passed on to the parent procedure.

The *default message procedure* completely defines the behavior of PM windows. The code to paint, resize, clip, initialize, and manage windows is all in this extremely complex and intricate procedure. We can *override* the behavior corresponding to a message by subclassing, at which time we specify a new window procedure. Our new window procedure will get the message first. We can define our own behavior, which can be just a simple modification of the default procedure, like beeping before we close a window. We can also completely change the behavior by creating our own implementation, and decide not to pass the message on to the default procedure. Of course, subclasses of our window class are free to override any behavior that we (or any of our ancestors) define.

PM's Object Model

In a sense, PM defines an object model, with the basic objects being windows. Each window can be uniquely identified by its handle. Windows have their own *attributes*, such as size and location. PM also allows us to attach instance data to each window. As usual, this data can be a single word of information, or a data structure or object. Windows are instances of classes whose behavior is defined within the window procedure. This behavior can be specialized through subclassing. The procedure is conceptually a sequence of method implementations (or *message behaviors*) which can be specialized. The major concepts that we dis-

cussed in *Chapter 1* are built into PM: encapsulated objects, message passing, instance data, classes, and inheritance.

PM's object model provides some additional interesting capabilities. In particular, the implementation of inheritance yields quite a bit of flexibility. After we process a message locally, we usually pass it to our parent class's window procedure. However, upon completion of our local implementation, *we are free to pass the message to any window procedure*. This is an extremely powerful concept. We are given the power to control the implementation of inheritance.

Handling Messages and Queues

Windows have a simple purpose. They get messages and respond to messages. These messages are delivered through *message queues*. These are not the same queues as the ones discussed in the previous chapter for interprocess communication. The system has a single message queue, and each main window also has a message queue. PM reads from the system queue and dispatches the message to the appropriate application queue. At this point, the main window procedure reads the message from the queue and dispatches the message to the appropriate window procedure. The procedure then decides how to process the message.

PM messages all have the following elements:

- **Window_handle.** The *window handle* is the receiver of the message. System-defined window handles are used to uniquely identify the window for most PM functions and messages.
- **Message.** This portion of the message defines the message type. For example, a close window message has the type *WM_CLOSE*. In PM's object model, the message type identifies the method that will be invoked.
- **Mp1 and mp2.** Often, message implementations require additional parameters. *Mp1* and *mp2*, which stand for *message parameter 1 and 2*, are additional information to the method implementation. The contents vary based on the message type. If more than two words of information are needed, one of these parameters may be a pointer to a structure.
- **Time.** The *time* portion of the message is often needed by the message implementation. For example, if triple-click is required by an application, then the default message procedure can override the implementation of
 - double-click and click for one of the mouse buttons and find the difference to determine whether a triple click is encountered.
- **Point.** Many PM messages are more powerful when an (x, y) coordinate is supplied. For example, a mouse button click when the mouse pointer is within the area of a graphical image can be used to select the image.

This structure may seem constrictive, but it is quite powerful. Even though variable parameters are fixed to two words of information, these words can be ad-

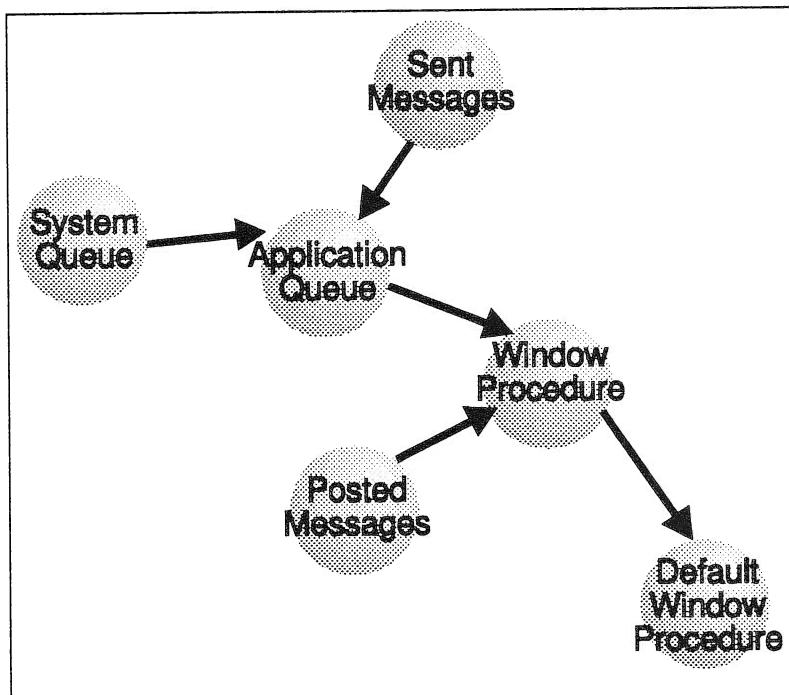


Figure 3.1. Sending and Posting PM Messages

resses to a data structure whose size is virtually unlimited. The additional context information of time and location also add to the power of a message. All of this information is packed into a fixed structure, which means that to some degree, all messages can be treated in the same way.

Messages can be delivered in the two ways shown in Figure 3.1. *Posting* messages with WinPostMsg adds the message to the target window's message queue. In this case, control will return immediately to the posting application, and the recipient will later remove the message from its queue. *Sending* a message is a direct call to the target's window procedure. The sending application will wait until that message has been processed, at which time processing can resume.

Messages are processed within a window procedure. Individual messages are usually processed within cases of a switch statement. This is what a standard window procedure might look like:

```

switch (msg)  {
    case WM_CREATE:
        /*      ...do something special for create      */
        break;

    case WM_DESTROY:
        /*      ...do something special for destroy   */
        break;
}
return(DefWindowProc(hwnd, msg, mp1, mp2));
}

```

This message is passed to a window with handle hwnd. The message, msg, and its two parameters, mp1 and mp2, come next. We switch on the message type, and then process each command. Following the protocol of passing messages to the default window procedure results in getting all of the nice default behaviors. That is all there is to it. Most of the rest of learning the Win API is learning the supporting messages and controls.

PM Messages

There are many PM messages which you will probably never need to know anything about, but there are several PM messages that are particularly important. We will introduce them here. Once again, if you do not find the implementation details that you seek, check the PM examples in the second section of this book or a PM reference. Most messages do window creation/destruction, manipulation, notification, or specialized control manipulation. Let's take a look at some of the more common messages.

Creation and Destruction

The WM_CREATE message is called whenever a window is created. This is a good place to put code that is related to initialization. The WM_CLOSE message is sent whenever a window is closed. It is handy for freeing up storage related to a window, as well as any other house keeping that must be done. The WM_DESTROY message is sent to *cause* another window to be destroyed, unlike WM_CLOSE which is generated as a result of the window closing. WM_QUIT is generated as the result of the selection of a *quit* menu choice from the user. Note that there are slight semantic differences for each of these messages.

Painting

Most of the display magic occurs within the WM_PAINT portion of the window procedures. Whenever you need to make any attribute of a window visible other than those presented by the default behaviors, you need to override this message. Keep in mind that windows can be painted more than once. Windows are painted whenever part or all of a window that was covered becomes visible, when they are created, and possibly other times as well.

When processing a paint message, a `WinBeginPaint` call is issued. This call returns a presentation space, which is the first parameter to graphics programming interface calls, which we will discuss below. After you are through painting the window, call `WinEndPaint`. The default window procedure does nothing but call these two functions.

If you put graphic calls within the `WM_PAINT` switch, then the graphics will be drawn whenever the window is painted. It may be too expensive to repaint the whole window whenever a portion is covered. You can query the invalid rectangle when you begin the paint:

```
WinBeginPaint(hwnd, NULL, &InvalidRectangle);
```

`InvalidRectangle` contains coordinates that bound the area of the screen that needs to be updated. We can then limit repainting operations to the invalid rectangle. You can also apply this performance trick when requesting a repaint. Instead of sending or posting a repaint message if only a small part of the window needs updating, a `WinInvalidateRect` call may be used, which causes only the invalid area to be repainted.

Notification

There are many different notification messages. Practically every different operation on a window causes a notification message of some sort. Resizing a window causes a `WM_SIZE`, moving causes a `WM_MOVE`, scrolling, a `WM_HSCROLL` or `WM_VSCROLL`, and so forth. In addition, there are notification messages whenever keys are pressed or the mouse moves. We will talk more about these areas later.

Controls

If the only objects in PM are windows, what about buttons, scroll bars, and other controls? These are just specialized windows. In fact, there are many different specialized windows to spice up applications. They are windows which provide additional visual characteristics and mechanisms for user interaction.

Earlier, we mentioned inheritance relationships and containment relationships. Controls are children of other windows. The parent windows are usually some variation of a standard window, or a dialog window which we will discuss below. Messages to these windows are passed to the parent window's procedure, so the appropriate action can be taken. This design is preferable, because it is much more natural to handle events like "push button pressed" within a window procedure. It would be too tedious and inefficient to create a window procedure for every control. Now, let's look at some of the different controls.

Buttons

Buttons are controls which are intended for a single atomic action. There are several different types of buttons. The simplest is a *push-button*. This button generates a message whenever it is pressed. The message can be customized,

and it can usually be used directly or indirectly to invoke a function or a method. Common uses for pushbuttons are to get help, commit or cancel some information on a window, or invoke some special action such as a recalculate a spreadsheet. *Check-boxes* are used to save a state. When the user clicks on it, the state toggles between checked and unchecked. Check-boxes are best used to choose between mutually exclusive options where the meaning of a checked box would be clear. For example, a checked box would be appropriate within a dialog that states, "Check the box next to all of the diseases or conditions that you have had." It would be inappropriate for determining the sex of a person, because the meaning of a checked state would be ambiguous. A *radio-button* would be best for determining sex, or choosing one of a number of mutually exclusive options. The radio-button gets its name from the buttons in car radios which are used to choose stations. When one is pressed, the others within its group pop up. There are mechanisms to label and group all types of buttons. We will show you more in the examples portion of this book.

Text

Text is basically divided into two areas. The first is output only. *Labels* fit this description. Labels are used to output a simple string of text. *Entry fields* are like labels, but allow users to type values into them. They can allow scrolling, so that text wider than the entry field can be typed. *Multi-line-edit (MLE)* controls are used to input more than one line of text. In fact, the default system editor is little more than a single MLE control. With both entry fields and MLE controls, we get many advanced behaviors for free, such as *clip board* support. To test this, go to any entry field within any OS/2 program. Use the mouse to highlight some text by pressing the left button and sweeping the pointer over the text. Release the left button. Press *Shift-Delete*, and the text disappears. Go to any other entry field. Press *Shift-Insert* and the text reappears. This is all default behavior, and gives us free access to the system clipboard. Of course, we can use menus to send clip board commands to the controls within our application in addition to these keyboard commands, but we have to create the menus ourselves, and use the menu choices to send messages to cut, paste, or the like.

Lists

List boxes are a single box in which many single-line strings may be added. Users can select one or more of these strings. We can then later query the list box to find out which item(s) was selected. We get several notification messages, such as selection, and can invoke methods, such as `LM_DELETEALL`, which deletes all items in the list. They also provide some useful default behaviors, such as scrolling and positioning the cursor when the first letter of an item is pressed.

Combo Boxes

Sometimes it is useful to combine the abilities of a list box, which constrains a choice to a list of items, and an entry field, which allows the user to type anything but provides no assistance. PM combines these behaviors with a *combo box*. The combo box has two areas. One allows free form typing like an entry field, and one provides a list of strings like a list box. Selecting an element from the list inserts it into the entry field.

Sliders

Scroll bars are controls at the right and bottom of a screen to allow scrolling. OS/2 also has slider controls, that can be used to input data. The bit maps (for presentation) and orientation of sliders can be configured.

Dialogs

A *dialog box* is used to encapsulate all controls for a single dialog with the user. *Modal* dialog boxes will not let the user do other tasks until the dialog box has been destroyed. *Modeless* dialogs do not have this restriction. Both types of dialog boxes are just simple windows, with some nice default behavior called *keyboard navigation*. This means that dialog boxes allow the user to change keyboard focus among its child controls with arrow and tab keys. Dialog boxes have procedures, just like other windows. They are created via WinDlgBox and destroyed via WinDismissDlg. By convention, dialog boxes usually have title bars, but not menu bars or size borders. OS/2 comes with several existing dialogs, including a *directory dialog*, which lets a user choose a file from any disk or directory, and a *font dialog*, which lets the user choose a font and point size.

Value Set

Value-sets allow the user to select options that are represented graphically. They can be viewed as a group of push-buttons; each represented by an icon or bitmap. The button bars that many applications support today are good examples of value sets.

Menus

A menu is a control that allows the user to make a choice from a variety of options. Clicking on a single option generates a WM_COMMAND message with parameters set to let the programmer identify the individual option. *Accelerator keys* can also be set, which let the user access a menu command with a single keystroke. Menu bars can be created by specifying an option, either when a standard window is created, or individually. The menu bar is divided into topics and separators. Each subtopic can be further subdivided. Clicking on a topic usually offers the sub-choices in a *drop down menu*. If any of the choices con-

tains still another level of menus, then clicking on that choice will produce a cascading menu.

Pop-up menus are specialized menus that are designed to be attached to an object. Pressing mouse button 2 (right mouse button for a right handed mouse) causes pop-up menus to come up. The workplace shell creates a pop up menu for objects, which are identified by icons.

Container

The container control is a window that contains other containers or objects. These objects can be moved between containers with a drag and drop API. Workplace Shell folders are a good example of containers. Containers can be presented with different views. Text lists, outlines, and windows of icons can all be presented.

Notebook

The notebook dialog provides a way to organize a larger area than will fit onto a single screen. Each notebook page can contain a dialog. Pages can be selected by clicking on special notebook tabs. The dialog pages need not be the same. Notebooks are handy for configuration menus. Different dialogs can be presented for color, font, and other application-dependent settings choices. To see an example, click on the right mouse button to bring up a pop-up menu for a workplace shell object. Then, open the settings. You should see a notebook control with a different dialog on each page.

Icons and Bitmaps

Like text, icons and bitmaps are for output only. Icons usually represent other windows. Icons can be shown programmatically. They can also be specified using resource files. For more about bitmaps and icons, see Petzold's text. It is listed in the bibliography in the back of this book.

Resources

Controls can be created programmatically, but sometimes it is preferable to lay out a complex dialog or window graphically. OS/2 lets you do this through a tool called the *resource compiler*. Using some graphic editor, such as the *dialog editor*, we can lay out a dialog by placing and sizing the various child controls. We can then associate key events, such as menu choices, with our *abstract events*, which we can use within our window procedure. Resources can also contain file names for icons, bit maps, and strings which may need to be translated to other languages. First the resource file is compiled to convert it to a more efficient format. Later, it is added to the program's .DLL or .EXE. The resource compiler is used to do both of these things. They can be done independently or in the same step.

Keyboard and Mouse Input

There are two major interactive user input mechanisms within Presentation Manager: the keyboard and the mouse. Much of the user keyboard and mouse input will be done for you by the PM default window procedures and the various controls, but there will be times when you need to get keyboard input yourself. Within PM, keyboard processing is done through messages. Only one window at a time will get keyboard messages. This window is said to have the *keyboard focus*. Of course, focus can be set programmatically through WinSetFocus. The user can also usually set the focus by clicking on a window or control.

When a window has the keyboard focus and a key is pressed, a WM_CHAR message is sent to that window. Within this message is the character that was pressed and some additional state information, such as whether the shift, control, or alt key is pressed. It provides information in various forms. One is a hardware scan code, which is usually ignored. The next is a virtual key code, which is used for keys that do not generate characters. Finally, the character code contains a character (if the key that is pressed creates one).

Mouse input is the other major type of user interaction. It can be used actively or passively. To use the mouse actively, you query the position through WinQueryPointerPos. A variation of this command is WinQueryMsgPos, which returns the coordinates of the mouse at the time a message is received. The other way to process mouse input is to use the mouse messages. Every time the mouse moves within a window, the window receives a WM_MOUSEMOVE message. In addition, messages are sent for each mouse button press. Both single- and double-clicks create messages. By convention, the left button is used by the application and for selection, and the right button is used for pop-up menus and drag-and-drop.

There are some more advanced mouse processing functions and messages within PM as well. One is capturing the mouse. This is done whenever a mouse action begins some logical unit of work, such as a drag of a scroll bar elevator. The programmer may want notification when the user drops the scroll bar elevator, but if the mouse moves out of the window containing the scroll bar, the message could be lost. The solution is WinSetCapture. When this is done, the mouse input is *captured*, meaning it will come to the window issuing the capture regardless of the pointer location.

GRAPHICS

Windows are a good way to present snappy user interfaces without too much work. There are times when we may want to enhance our user interface or create our own controls with graphics. The *Graphics Programming Interface* (or GPI) is dominantly a vector based graphics API. Vector based means that primitives are described geometrically, rather than describing the individual points that make up the constructs. Vector graphics have the basic advantage of being able to support many different kinds of hardware.

The presentation space is the first parameter to GPI calls. This presentation space is a data structure containing attributes related to a *device independent* surface that is related to a device context. The device can be a printer, plotter, or screen. Since we want the screen to have good performance, there is a special high performance presentation space called a cached *micro presentation space* which can only be related to a display. This type of presentation space is commonly used when processing the WM_PAINT message.

The GPI commands are divided into five basic groups, which we can call primitives. These are lines, fill patterns, text, markers, and bitmaps. Lines, fills, and markers are vector-based. Text can be vector or raster based. Bit maps are raster based.

Lines

To draw a line using the GPI, you simply move to a point and then draw a line to another point:

```
GpiMove (hps, point1);
GpiLine (hps, point2);
```

That's it. hps is a presentation space (more later); point1 and point2 are pointers to a point structure that contains the x and y coordinates of a point. The current location is just an attribute of the presentation space. The move does not do anything but change this attribute. The line draws a line from the current location to the specified point, and updates the current location. You can query the current location with GpiQueryCurrentPosition. Of course, it would be tedious to draw many of these lines one after the next. GpiPolyLine is created for this purpose. It is passed an array of points and a count. One common use for this API is to draw curves. To make the curve smoother, supply more points.

These lines are not limited to a solid-line style. Several styles are available, and can be set with GpiSetLineType. Some examples of the available line styles are long dashes, short dashes, a dash followed by two dots, and so on. The line type also is an attribute of the presentation space.

The GPI also provides some slightly more advanced functions. Rectangles can be drawn with GpiBox. These rectangles can be drawn as just an outline, or filled. (We will talk more about the filling mechanism later.) The corners can be rounded as well. Like its cousin function GpiBox, GpiEllipse can be used to draw a circle or ellipse, in outline or filled mode. More advanced arcs and ellipses can be drawn with a combination of the functions GpiSetArcParams, GpiPointArc and GpiPartialArc. Other functions are available for splines and other curves.

One question that may be asked is what coordinate system is used. Of course, pixels are supported, but pixels have problems. Supporting different hardware means translating coordinates. They do not map neatly onto real world coordinates, and just aren't friendly. If you must use pixels for some reason, it is a good idea to base most coordinates on the size of the system standard font, or start size of the window, or similar measure.

A better way to handle coordinates is through PM's *device context*. The device context is just a mechanism for graphics output. By using a context, the programmer can query the capabilities of a device through `DevQueryCaps`. This information can be used to base coordinates on appropriate real-world values.

Patterns

The next set of GPI primitives is based on patterns and fills. You can use fill patterns through the `GpiEllipse` and `GpiBox` calls defined above. You can also fill enclosed areas that you create in other ways. This is not a "flood fill." To use this fill, you specify the bounding geometry of the area that you want to fill. You can create your own pattern or use one of several predetermined ones. You can also set colors to use with or without patterns with the function `GpiSetColor`. Let's look at some of these functions.

Bounding areas can be created by drawing a series of lines bracketed by the functions `GpiBeginArea` and `GpiEndArea`. If you don't close the area, the GPI will close it for you by drawing a line between the starting point and the last point that you specified. Calling `GpiMove` also closes the area. It works like a combination of calls to `GpiEndArea`, `GpiMove`, and `GpiBeginArea`.

Actually, lines and fill patterns are actually the only two primitives we need for vector graphics. The others just build on these. Remember, the fills are needed because we need a precise way to say "Fill all of the pixels within a geometric area."

Simply setting all of the pixels within an area will not do, because we want device independence, and the coordinate systems may change. But with these primitives, we can do everything else.

Text

The GPI text primitive may be the most difficult to master, because of the many fonts that are supported. As noted previously, proportional text, which means that characters can be of different widths, also adds complexity. With this complexity we get tremendous power.

The text output commands are relatively simple. There are four. `GpiCharString` is used to output a string at the current location. Similarly, `GpiCharStringAt` is given a string and a point. The string is output at the starting position specified by the point. Both of these calls update the current position to the baseline of the end of the last character. `GpiCharStringPos` can be used for more sophisticated text management. It can be used to leave the current position at the starting point, clip the text to a rectangle, or even rotate the text output.

There are several attributes that are related to the display of text. One is color. The foreground and background colors for text can be specified. Background can be opaque or transparent.

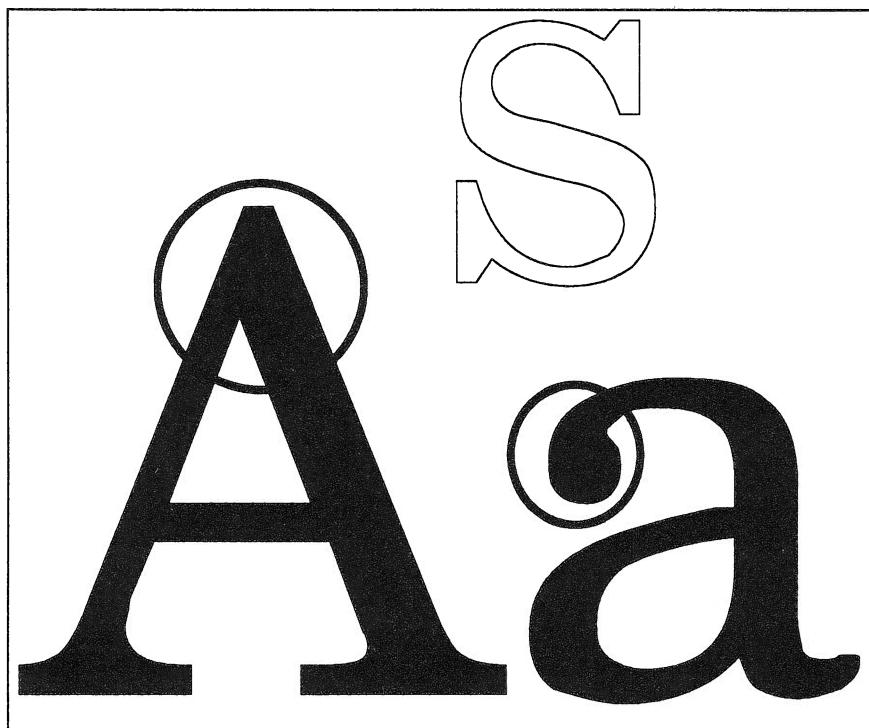


Figure 3.2. Proportional Outline Fonts

Fonts

Two different types of fonts are supported. One is the *image* font. These fonts are stored as bitmaps. *Vector fonts*, as the name implies, are described geometrically. OS/2 has Post Script fonts built in. They are known as outline fonts, because they consist of a series of lines and curves that are filled to form characters as in Figure 3.2. It shows some actual letters scaled to a larger size. Notice that the letters do not appear boxy or granular. The circled areas are perfectly shaped. You can also see the letter s is actually formed by lines and curves. These fonts can be scaled smoothly; bitmapped fonts cannot. Petzold has an excellent treatment of fonts in his book, *OS/2 Presentation Manager Programming*.

Marker Symbols

Markers are small symbols, such as cursors, which are used for navigation or emphasis. Bullets within a document and I-beam cursors are examples. They can be drawn with the call GpiMarker. Frequently, applications such as business line charts require more than one marker to be drawn. This can be done with a GpiPolyMarker call. A neat trick is to use the same array of points to create the line segments (with GpiPolyLine) and markers. Like fill patterns

and line styles, some defaults are provided, such as eight pointed stars, circles, dots, and the like.

Images

You may have noticed that we said that PM is *dominantly* a vector-based graphics system. You have seen that not all fonts are vector-based. Similarly, images can be bit maps. A bitmap or icon can be rendered with GpiImage. They can be moved efficiently with GpiBitBlt. This function just moves a block of pixels on the screen from one place to another. It can also be used to scale bit maps. Again, Petzold has an excellent treatment of this topic.

CONCLUSION

That wraps up the basic tour of Presentation Manager. We have seen the Win and Gpi interfaces. We have also explored the beginnings of an object model through PM's message passing, classes, and inheritance. Though PM provides us some basic underpinnings for an object model, we need more. We need to deal with objects on a lower level than just windows. We need to be able to define our own interfaces, beyond the *mp1* and *mp2* parameters that we are given. We need our inheritance model to be extended to inherit interfaces as well. In the later chapters, we will show you a more complete object model in SOM. We will learn more about the Workplace Shell, which uses this object model to create an object oriented user interface. We will also show you how to wrap PM windows and controls to fully integrate them into the object environment of your choosing.

4

Object-Oriented Programming using C++

INTRODUCTION

C++ has successfully combined the structured approach of C with object-oriented programming. As a result, it is an important and popular language. The use of static typing in C++ means the programs perform, and of the operating system, linker, and compiler techniques developed for C still work in C++. For example, C++ programs on OS/2 can define external entry points. They can then be compiled into binary object modules and linked with other object modules, and then placed within DLLs. In interpreted OOP languages like Smalltalk, programs are generally loaded and linked within a special language specific environment. As a result, the extent to which they can use and be used by other languages is sometimes limited. Interpreted OOP languages are evolving to address this issue, but it may help explain the initial popularity of C++ as a language for doing OOP.

Unfortunately, C++ is a complex language, both syntactically and semantically. Sometimes, C++ appears like a fractal because no matter how well you think you understand the language , you can look closer and discover a whole new level of complexity. As a result, there is a wide spectrum of expertise and understanding over which C++ programs can be written.

For example, some C++ code is simply written to get a job done. Little thought is given to software reuse or the ways that the design of a class hierarchy can influence reuse. C++ makes it easy to write these programs because it extends C so seamlessly. C programmers can use the language immediately. But

attention to software reuse in OOP requires hard thinking, and it often results in generalizing specific solutions. Good programs should handle different classes of objects where these classes are related by inheritance.

In general, OOP uses inheritance to combine and organize. Both implementation and interfaces are inherited. In the case of OS/2 PM, for example, OOP is supported for window objects. Inheritance is used to implement new classes of windows. As we learned in the previous chapter on PM, a window subclass inherits structural layout from the parent class. It also inherits a single method called the client-proc. The window subclass can define its own client-proc, which overrides the parent's client-proc. The new client-proc normally provides special handling for only a few messages, and passes all other messages on to its parent's client-proc. There are basically two method interfaces to the client-proc—a synchronous message call interface, and an asynchronous message queue interface. While this interface to PM windows is inherited, it cannot be extended. Addition of new method interfaces to PM windows is not part of the subclassing framework offered by PM.

In some ways, the C++ model for OOP is more general than PM's. For example, inheritance in C++ does allow addition of new methods. Also, multiple inheritance is supported by C++, while PM windows always have a single parent. In other ways, though, the C++ model is less general than PM. The inheritance hierarchy of C++ is statically defined, whereas the parent of a PM window can be dynamically changed at runtime. Generally, languages such as C++, which are based on static typechecking, give up a certain degree of flexibility at runtime in order to guarantee the absence of runtime type errors. Careful coding in more dynamic OOP languages can also guarantee this, while allowing programs the use of enhanced capabilities.

In this chapter, we focus on the basics of OOP in C++. We assume that you know C. We will focus on those aspects of C++ that you need to implement reusable software through subclassing. This involves a discussion of how classes are used to define object structure in C++, and how virtual functions in C++ enable the expression of generic procedures. We will illustrate the essential ideas behind the use of inheritance in C++ for defining the implementation of objects. In a final section, we will discuss some of the difficult issues resulting from support of multiple inheritance in C++.

There is much more to C++ than is illustrated here. We deliberately avoid discussing many aspects of C++ not directly related to OOP. For example, instead of discussing the many different categories of member functions in C++, we focus on virtual functions. Also, for example, we do not discuss overloading, which is the use of the same name for different functions distinguished in C++ by their argument types. This is a valuable aspect of C++ as a programming language, but it has nothing to do with subclassing or inheritance. We focus specifically on those aspects of C++ that relate directly to OOP concepts. Although our examples are simple, we hope that the perspective they offer will be useful to readers who already know C++.

C++ is a remarkably subtle and complex language whose design has evolved over a period of many years. It is still evolving. There are three books in the bibliography to help you learn more. As this is written, an ANSI standard for C++

has yet to be finalized, but we especially recommend reading [Ellis] for an appreciation of the sophistication that has gone into C++, and ultimately, its ANSI standard. Ellis provides a detailed and helpful reference manual, lending an excellent insight into the tricky design decisions influencing the present form of C++. We recommend [Lippman] as a general tutorial and guide to C++ programming.

INHERITANCE IN C++

In C++, there are two useful questions that can be asked about inheritance:

- 1 . What can be done to a C++ object?
- 2 . How is C++ object structure determined?

What can be done to a C++ object ?

Like other OOP languages, C++ objects have methods and data. C++ objects are an ingenious generalization of the concept of data structure supported by *struct* in the C language. This treatment is based on ideas that came from the SIMULA language. Objects in C++ are data structures with a fixed set of named fields. In particular, a C++ object is a data structure whose fields can hold both data and object methods called “virtual functions”. Given an object in C++, you can:

- Use the object as an argument to operations, functions, or virtual functions.

```
call_a_function(obj);
```

- Assign the object to a variable or a structure field.

```
x = obj;
```

- Access the fields of the object’s structure by name, to retrieve data,

```
x = obj.data_elem;
```

Store data,

```
obj.data_elem = x;
```

or invoke a virtual function:

```
obj.vf_elem(3);
```

So C++ objects are used in ways that are entirely familiar to C programmers. This even includes the use of virtual functions, which may simply be viewed as functions held within an object’s structure. The *definition* of virtual functions is special though, as we will see.

How is C++ Object Structure Defined ?

In C++, *classes* define the structure of objects. Here is a simple C++ program that defines and uses a simple class of C++ objects. This example uses neither subclassing nor virtual functions, but it illustrates the use of C++ classes to define the data structure of objects. In C, *typedefs* are used to introduce a name for

the organization of data described by a *struct* expression. In C++, this process is circumvented by simply allowing the use of a class name to indicate a type for an object. Here is an example C++ class definition followed by a program that uses this class:

```
#include <stdio.h>
class Object { // define a class named Object
    public:      // introduce publicly visible fields
        float weight; // declare the weight field
};
// declare a function with an argument of type Object
Object printWeight(Object o)
{
    // access and print o's weight field
    printf("o.weight = %d\n", o.weight);
    // return result
    return o;
}
main()
{
    // declare local variables of type Object
    Object o1, o2;
    // store into o1's weight field
    o1.weight = 12.34;
    // pass o1 to a function that returns an Object
    o2.weight = printWeight(o1).weight;
}
```

Class names are used to indicate object types. These types are used for variables, function arguments, and function results that are objects. This example shows how these objects and object types are used by programs. As another simple example, consider the following two very similar C++ class definitions. They again include no virtual functions or subclassing, so they are equivalent to a corresponding use of structs in C.

```
class C1
{ public:
    int    field1;
    int*   field2;
};
class C2
{ public:
    C1    field1;
    C1*   field2;
}
```

These classes will be used in many of the examples that follow. Objects of class C1 have two fields. The first, named field1, has the type int. The second, named field2, has type int*. Similarly, the first field of an object of class C2

has the type C1, whereas the second field is a pointer to an object of type C1. This example extends the first example by showing how class names can be used as types for C++ object fields.

The keyword `public` in the above class definitions indicates the scope of visibility for the associated names, and is included here for correctness of the examples that follow. The other possible scopings are “protected” and “private,” with `private` being the default. The scope of names in OOP can be an important consideration in construction of programs, and scoping facilities of C++ can be used to restrict access to object substructure in various ways. In the interest of simplicity, we will not discuss name scoping in C++. [Ellis] has a good treatment of this topic if you would like greater detail. All of our examples use “public” to provide unrestricted access to object substructure, both by code that uses an object, and by the code that implements its virtual functions in subclasses.

The following C++ procedure illustrates creation and use of C1 and C2 objects.

```
void Example1( )
{
    int x;
    C1 c1Obj;
    C2 c2Obj;

    x = 1;

    c1Obj.field1 = x; // assigns the value 1
    c1Obj.field2 = &x; // assign a pointer to the x variable

    c2Obj.field1 = c1Obj; // assigns the the C1Obj value
    c2Obj.field2 = &c1Obj; // assign a pointer to the c1Obj var.

    c2Obj.field2->field1 = 2; // sets c1Obj.field1 to 2
    c2Obj.field1.field1 = 3;

    printf("%d, %d, %d\n", x, c1Obj.field1,
c2Obj.field1.field1); // output: 1, 2, 3
}
```

When `Example1` is called, local storage for the variables `x`, `c1Obj`, and `c2Obj` is allocated on the execution stack, and the indicated assignments to this data are executed. Note that the assignment of `c1Obj` to `c2Obj.field1` copies the contents of *both* of the `c1Obj` fields into the first field of `c2Obj`.

Although this example is fairly simple, it is important to understand how C++ object fields are stored and copied in memory. This is really essential for understanding the semantics of C++ inheritance. In particular, important differences between dealing with objects and pointers to objects can be surprising without understanding the implementation of C++ objects in memory. Figure 4.1 illustrates the final state of the data created and manipulated in this example.

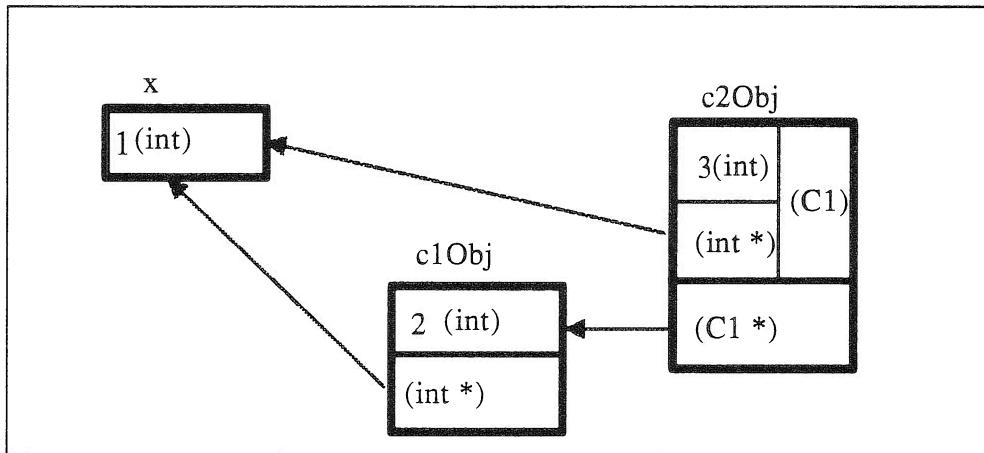


Figure 4.1. Example Object Field Layout

In Figure 4.1, the types of the various data fields are indicated in parentheses. In general, you shouldn't expect to know exactly how a compiler will lay out the different fields of a structure in memory. The important thing for a programmer to know is the different components that are present in a structure and to know how to access these components using C++ statements, as illustrated in the Example1 code. These same considerations apply when inheritance is used to define object structure.

Inheritance of Object Structure

C++ uses subclassing to define object structure. In this section, we explain subclassing by using it to reproduce the `c2Obj` structure in Figure 4.1. This simply defines a class whose objects will have the same components as `c2Obj` objects. We use inheritance to do this instead of explicitly including an object of type `C1` as the first field of the new class. The new class that illustrates this is defined below, as `C3`.

```
class C3 : public C1 // Inherit C1 structure into C3
{ public:
    C1* field2; // Add a C1* field to the inherited structure
};
```

As this simple example of C++ subclassing illustrates, inheritance is indicated by following the `C3` class name with a colon and a base class name, `C1`. The keyword “public” used before indicating `C1` as a base class for `C3` is another use of name scoping in C++. We won’t worry about scoping issues here. The overall result of this class definition is shown in Figure 4.2, which illustrates how inheritance is used to include `C1`’s structure into objects of class `C3`. Note that the internal components of `C3` are the same as those of `C2` objects.

Class `C3` is said to be a *derived* class, whose derivation uses `C1` as a *base* class. An ancestor class, in C++ terms, may be either a *direct* or *indirect base*

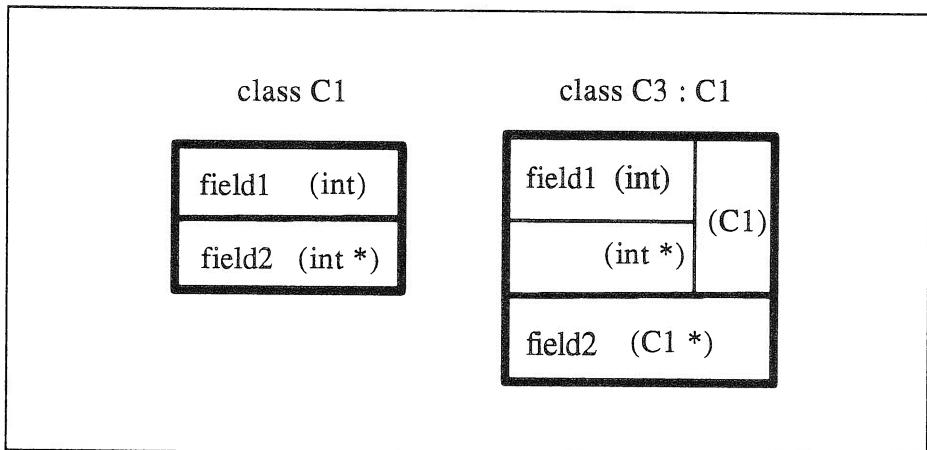


Figure 4.2. Example of Object Structure Inheritance

class. A direct base class is often called a parent class. OOP languages that allow classes to have multiple parents are called *multiple-inheritance languages*. Smalltalk is a *single inheritance language* because each class can have only one base class.

A *C3* object will have the same structure components as a *C2* object, but programmers do not refer these classes' *C1* components in the same way. In particular, *C1* does *not* have an overall name in instances of classes derived from it. Although there is an area within a *C3* object that has the structure of a *C1* object, you cannot refer to it by name.

This seems odd. How can you use an inherited object structure if it has no name? The answer is that the class name you provide lets the compiler select the desired inherited structure stored within a passed argument. Within that procedure, all structure component names that deal with this argument are automatically relative to the desired inherited structure corresponding to its declared type. This process is referred to as *argument conversion*, and it always takes place in C++ when passing objects to procedures. Understand augment conversion, and you will be able to use the same selection process for yourself whenever you need it. Consider the following example.

First, we should clarify this: an inherited structure does not have a name within objects of a derived class, but the names of the ancestor class's fields *are* still available with one exception. They may be masked by use of the same name within the derived class, or some intervening class. In the case of *C3* objects, this means that the *field1* of *C1* can be directly accessed by name. On the other hand, *field2* of *C1* is masked, because its name is used for the structure component of type *C1** that is introduced by *C3*. We can use *C3* instead of *C2* objects of Example1 to illustrate all of the above points.

```

Example2( )
{
    int x;
    C1 c1Obj;
    C3 c3Obj;

    x = 1;

    c1Obj.field1 = x; // as in Example1
    c1Obj.field2 = &x; // as in Example1

    *((C1*)(&c3Obj)) = c1Obj; // Exampled Below!
    c3Obj.field2 = &c1Obj;      // as in Example1
    // as before

    c3Obj.field2->field1 = 2; // as in Example1
    c3Obj.field1 = 3; // corresponds to c2Obj.field1.field1 = 3;

    printf("%d, %d, %d\n", x, c1Obj.field1, c3Obj.field1);
    // prints: 1, 2, 3
}

```

In Example2, the local variables `x` and `c1Obj` are like those in Example1, but we use a `C3` object instead of a `C2` object. The important difference between these two examples is how `c1Obj` is assigned to its corresponding location within `c3Obj` instead of how it was done in Example1.

In Example2, you cannot simply assign `c1Obj` to a corresponding named location within `c3Obj` as we did in Example1. The reason is that this location has no name in a `C3` object. However, we could assign the contents of the two fields separately. We could assign the first field using the statement `c3Obj.field1 = c1Obj.field1`, because the inherited `field1` of `C1` is not masked by any `C3` fields. We cannot deal with the second field so easily. For a `C3` object, the name `field2` refers to the newly introduced `C3` field, not the second field inherited from `C1`. So, we cannot simply write `c3Obj.field2 = c1Obj.field2`. Furthermore, it turns out that we can use the single assignment statement used above, which copies both fields just like we did in Example1 *because we understand augment conversion*. So we now explain the rather strange looking assignment statement shown above, which will help you understand C++ inheritance.

In C++, a pointer to an object can be *cast* to another object type when the types are related by inheritance. The value of the pointer is actually changed in order to select structures inherited from different ancestor classes. This is how the argument conversion process above is implemented for pointers. It also explains why the portions of an object's structure which are inherited from different ancestor classes can be accessed *without having to use an explicit name to select these different portions*.

In Example2, the assignment of `c1Obj` makes use of a casted pointer. At the left hand side of this assignment statement, we see `&c3Obj`. This expression denotes a pointer to `c3Obj`. The type of this pointer is `C3*` because `c3Obj` is de-

clared to be of type C3. By casting this expression to C1*, , the original pointer value is modified to select the structure in c3Obj from C1. This change is performed by executable code that is generated by the C++ compiler. Sometimes, it is not necessary to modify the pointer value, but it usually is. Only the compiler ever knows for sure. Finally, in the example assignment, the cast pointer is dereferenced to identify the inherited C1 structure as the location to receive the c1Objcopy. No structure component names are used to select the overall inherited structure from C1—only type expressions are used.

Example2's assignment of c1Obj is the very heart of C++ support for OOP. The fundamental characteristic of OOP languages is polymorphism. Interestingly enough, C++ uses this selection process based on statically declared types to support polymorphism. Almost all other OOP languages use dynamic mechanisms.

In C++, polymorphism is actually supported in two distinct ways, both of which are guided by types. Only only one of these ways is generally useful for OOP. The other is a self-consistent artifact of the language provided because it makes sense in terms of object structure. It might be considered dangerous because it is not consistent with other OOP languages. A C++ programmer needs to know the difference between these two forms of polymorphism to avoid potential problems.

The useful form of C++ polymorphism is supported by the pointer manipulation that we discussed above. C++ objects are used as procedure arguments. The C++ compiler generates whatever code it needs to modify the passed pointers. This form of polymorphism is available either through the explicit use of pointers when passing objects to procedures, or through the use of C++ reference types.

The dangerous polymorphism is based on structure copying as opposed to pointer manipulation. When an object (not a pointer) is passed to a procedure, the compiler again performs argument conversion. This assures that the argument seen by the called procedure has the declared type for this argument. Now, arguments are passed in C++ by allocating storage for them on the execution stack. The object itself is not passed. It is already sitting somewhere else in memory, so a copy of the object is made on the stack. Note that all of our simple examples so far have passed objects—not pointers to objects—as arguments!

The object copies made during argument conversion are not complete copies because some structure components must be ignored for reasons explained below. Also, the values in some of the fields that are copied may be changed, as we shall see after discussing virtual functions. As a result, important information can be lost when an object is passed this way. Specifically, when the declared argument type is more general than the actual argument object's type, we have trouble. C++ provides ways of dealing with these issues such as copy-constructors and assignment operators, but the necessary details are beyond the scope of this introduction. Their use requires a complete knowledge of the derivation and implementation of classes. But wait! We wanted to hide the implementation of an object for effective code reuse!

Whew. It is time for another example. If an object of class C3 is passed to a procedure whose argument is type C1, then only the inherited C1 structure com-

ponent will be placed on the stack. This is dangerous. If the only fields that can be used by a procedure that types its argument as a C1 object are the C1 fields that are put on the stack, what's the problem?

The answer has to do with the virtual functions that may be included in an object's structure. Classes that include virtual functions may not make use of all of their inherited object structure, but introduce new structure that is used instead. As a result, the practice of passing objects by copying inherited structure is often meaningless, even if it is type-correct. The inherited structure that is passed may never have been used to maintain the actual state of the object! To understand this issue fully, the next section considers C++ virtual functions in more detail.

Virtual Functions in Objects

The virtual function fields of an object's structure are determined by class definitions in much the same way as data fields. For example, the following definition for the class Point introduces both virtual functions and data structures by defining two data fields named x and y, and a virtual function field named radius.

```
class Point
{ public:
    float x, y;
    virtual float radius ( ) { return sqrt(x*x + y*y); }
}
```

The difference between the data and virtual functions within objects is how their contents are determined. The content of data structure components is determined by runtime assignments specified in the code, as we showed. If you know C, you have probably seen this before. However, the methods that implement the virtual functions of an object are *not* determined by runtime assignment. They are determined by the definition of the object class, and they can not be changed by assignment or any other C++ code!

An object whose virtual function is invoked is a *target object* for the virtual function call. In C++, the same process used for argument conversion is used on target objects to select the overall structure inherited from the class whose method implements the virtual function. Then, the field names of the components of this class are available to the method that implements the virtual function. This was illustrated by the above definition of radius. When the radius virtual function of a Point object is used, the method code that executes has named access to the x and y structure components of this object.

The target object argument conversion process is accessed by the keyword this. You can use this from C++ virtual functions. this always points to the portion of the target object's structure produced by argument conversion. The this keyword is useful when a pointer to the target object is needed, such as passing it as an argument to a procedure.

The above example is a complete C++ class definition. Whenever a Point object is created at runtime, C++ assures that the radius field of the object con-

tains a value that represents this virtual function. Conceptually, the content of the radius field of a Point object is the address of the method code. This is somewhat of a simplification because C++ implementations generally use an indirection to allowing grouping all the method pointers for a given class of objects into a shared virtual function table, but it works. The following example code illustrates the use of a Point object and its radius virtual function. We use pointers to objects in this example.

```
void Example3()
{
    Point *p = new Point;

    p->x = 3;
    p->y = 4;
    printf ("radius = %f\n", p->radius( ) );
    // output: radius = 5.0
}
```

Instead of *defining* a class in C++, it is also possible to *declare* a class. A declared class can allow subsequent use of its name as a type, as in the C++ declaration:

```
class Point;
```

Also, it is possible in C++ to define the overall structure of Point without defining the methods that compute its virtual functions:

```
class Point
{ public:
    float x, y;
    virtual float radius();
}
```

This last form is often the form used in #include files (often denoted .hh) for C++ programs. This gives us an object's interface without its implementation. Although this form does not provide a complete definition for objects of class Point, you can compile other files containing method definitions for the undefined virtual functions. Compiled methods for virtual functions can then be linked with compiled program code that uses these virtual functions. In this sense, the complete definition of C++ classes can be deferred until link time, even if their type structures must be known at compile-time.

Inheritance is involved in the definition of virtual functions in the following way. By default, virtual functions defined by an ancestor class are inherited by any class derived from it. On the other hand, a subclass can *override* methods by defining a new method for its execution. Thus, a new class Point3D can be defined by using Point as a base class, by introducing an additional z data structure component, and by overriding the method used to support radius:

```

class Point3D : public Point
{ public:
    float z;
    virtual float radius() { return sqrt(x*x + y*y + z*z); }
}

```

Within the method that overrides `radius`, the keyword `this` has the type `Point3D*`. This method can access the inherited `x` and `y` fields because no new structure components have been introduced that mask the inherited ones.

When objects of either class `Point` or `Point3D` are created at runtime, their virtual function components are automatically initialized by the C++ implementation. These portions of an object's structure are determined solely by the object's class and can't be changed.

Based on this, it is now possible to more fully understand the problem with passing objects to procedures in C++. Look at two different generic procedures, each of which invokes the `radius` virtual function of an object :

```

void Example4(Point *p)
{ p->radius( ); }

void Example5(Point p)
{ p.radius( ); }

```

If an object of class `Point3D` is passed to `Example4`, this will be done by passing its address. As explained above, this memory address will be modified by argument conversion so the value, `p`, received and used by `Example4` will access the portion of the argument's structure inherited from `Point`. This structure includes the `radius` virtual function component, and will be a pointer to the method that uses `z`, as appropriate for `Point3D` objects. Clearly, the object manipulated by the code of `Example4` is actually the object that is passed to it, and the result of invoking `radius` on this object will be appropriate for this object. This is the essence of OOP.

In contrast, if an object of class `Point3D` is passed to `Example5`, this will be done by structure copying. Argument conversion will build a new object on the stack for use by `Example5`. This object will not include `z`, because the expected type is `Point`, not `Point3D`. Furthermore, the `radius` portion of this object's structure will be loaded with the `radius` implementation used by `Point` objects, not `Point3D` objects. Trouble! In this case, the object seen by the `Example5` procedure will be only remotely related to the object that is passed to it. True, everything is type-correct, and there will be no runtime errors, but virtual functions invoked on the received object will be supported by different method code, and will generally return different results than those invoked on the original object. It is hard to imagine that this behavior is useful.

This concludes an explanation of how inheritance is used in C++. To summarize, inheritance of structure when subclassing (along with introduction of new fields) is used to determine the components of different classes of objects. Executable code determines the content of data fields, and inheritance of method implementation is used (along with overriding when this is desired) to deter-

mine the content of virtual function fields. Types (i.e., class names) are used by the C++ compiler to guide argument conversion. This supports polymorphism in C++ by guaranteeing that the arguments received by procedures always have the expected structure type.

GENERIC PROCEDURES IN C++

Now that we understand inheritance in C++, and the corresponding impact on object structure, we can use C++ to write *generic* code. OOP is valuable for many reasons. An essential characteristic of OOP is that it enables definition of generic functions. The more generic, the better the reuse. The ability define type-safe generic code in OOP overcomes an important limitation otherwise faced by programming languages. Put simply, generic code works for different types of arguments. In C++, generic code is written by using the virtual function fields of argument objects.

To illustrate generic functions, consider the idea of filtering lists of objects according to some criteria. Because points have already been introduced, we will consider filtering lists of points. It would seem useful if we could somehow parametrize the filtering process in terms of a selection criterion—call this criterion a predicate on points. This predicate will return 1 or 0, meaning pass or fail. Our desire is to achieve two things when filtering lists of points. First, whatever selection criterion is used for choosing points, it should be sensitive to the semantics of different classes of points. In other words, our predicates should be generic, and use the virtual functions provided by points. Second, we want to be able to characterize a wide variety of tests as predicates on points and indicate any one of these as an argument passed to a single, generic filter function.

The solution we develop will illustrate two kinds of generic behavior. One will handle different classes of points, and the other will handle different classes of predicates on points. The second kind of generic behavior may be interesting to experienced C++ programmers, since it shows how functions can be usefully grouped into different classes that provide similar abstract behavior. Many of the techniques developed in functional programming can be adapted to C++ using this approach.

As we already know, our objects will be instances of any class derived, directly or indirectly, from the class, Point. In the case of our running example, this includes objects of class Point3D. Let us begin by repeating and augmenting the Point classes somewhat. This will allow us to introduce the concept of constructors.

```
/*
 * define classes of test objects: Point and Point3D
 */
typedef class Point
{ public:
    Point(float x_, float y_) { x = x_; y = y_; };
    float x, y;
```

```

virtual float radius() { return sqrt(x*x + y*y); };
virtual void show()
{
    printf("2dpoint (%f,%f) w. radius = %f",
           x,y,radius());
}
} *Pointp;

```

The reason for using a `typedef` here is to define the name `Pointp`, which will be used to type pointers to `Point` objects. This pointer type will be used for all points, which is why the class definition for `Point3D` appearing below doesn't also include a `typedef`.

Notice the *constructor*. The purpose of a constructor is to combine the creation of a new C++ object with its initialization. As you can see, constructors can take arguments. The semantics of constructors is fairly involved because they can invoke virtual functions of the object being constructed before it is fully initialized. C++ has done a good job of dealing with this complex issue, but we won't discuss it here. A good rule of thumb is to avoid use of virtual functions in constructors.

Constructors are not virtual functions; they are not inherited. Constructors are defined within the structure of C++ class definitions for convenience. The scoping rules of C++ then provide named access to all instance data—even private data. We will see how constructors are invoked later. In `Point`, the constructor code simply loads the `x` and `y` instance data of a new `Point` object. The `show` method introduced by `Point` is used to allow points to describe themselves.

```

class Point3D : public Point
{ public:
    Point3D(float x_, float y_, float z_) : Point(x_,y_)
    { z = z_; }
    float z;
    virtual float radius(){ return sqrt(x*x + y*y + z*z); };
    virtual void show()
    {
        printf("3dpoint (%f,%f,%f) w. radius = %f",
               x,y,z,radius());
    };
};

```

Here we see *chained constructors* for the first time. When a new `Point3D` object is created, it is convenient to invoke the `Point` constructor with the `Point3D` constructor in order to initialize all portions object's structure. `Point`'s constructor is executed *before* `Point3D`'s. Of course, here, we could simply assign all three point coordinates within the `Point3D` constructor code, but that's because we know the instance data and the constructor code for `Point`. Remember encapsulation! In general, when subclassing, you don't always know the constructor code for ancestor classes. Nor does a subclass always

```

have access to inherited instance variables because they may be private within
the ancestor.
/*
 * Define Point predicate classes.
 * All PointPreds have a description and test a point.
 */
typedef class PointPred /* PointPred is an abstract class */
{ public:
    PointPred(char *descr_) // a PointPred constructor
    { descr = descr_; }
    char *descr;           // a string to describe the test
    int virtual test(Pointp p) = 0; // the test
} *PointPredp;

```

Here we see an *abstract class*. An abstract class is a class that it is not intended to be instantiated. Instead, it introduces a structure that will be inherited by all of its subclasses. The reason PointPred cannot be instantiated is that the test virtual function that it introduces is given a null implementation. In C++ terms, test is a “pure” virtual function in PointPred.

Abstract classes are useful when different possible implementations of a virtual function by different subclasses of the abstract class make sense, yet there is no general implementation of this virtual function. Because all uses of these subclasses will involve common object structure , it is important to reflect this so C++ typechecking will always allow its use. Note that even though PointPred is not intended be instantiated, it still has a constructor to initialize the description component that it introduces. This will be called when instances of its subclasses are created. The next two subclasses of PointPred are the ones that provide actual implementations for test.

```

/* point predicates that perform a single argument test */
class PointPredfloat1 : public PointPred
{ public:
    PointPredfloat1( // a PointPredfloat1 constructor
        char *descr_,
        int (*fcn_)(Pointp, float),
        float arg_) : PointPred(descr_)
    { fcn = fcn_; arg = arg_; }

    int virtual test(Pointp p) // override inherited test
    { return (*fcn)(p, arg); }

    int (*fcn)(Pointp, float); // hold the test function
    float arg;                // hold the test argument
};

```

The above class provides an implementation for the test virtual function inherited from PointPred. The test, in this case, will be performed by a function that requires a single floating point number in addition to the point that is being tested. When a PointPredfloat1 is constructed, its instance data is used to re-

cord both the function and the floating point argument that it requires. This technique (called *currying* in functional programming) is what allows us to represent arbitrary point tests in the abstract class `PointPred` by using a single virtual function whose single parameter is a point.

The easiest way to understand the argument types for the `PointPredfloat1` constructor is to look first at the declaration of the `fcn` data field (the second to last introduced field). This declares `fcn` to be a pointer to a function that accepts a `Pointp` and a `float` as arguments, and returns an `int`. Notice how the implementation of `test` simply applies the stored `fcn` to its arguments: `p`, and the stored `arg`.

The `PointPredfloat1` class shows both virtual function structure components, which cannot take assignments, and structure components that hold pointers to normal functions, which can take assignments as in the `PointPredfloat1` constructor. Here are the class of point predicates that accommodates two arguments.

```
/*
 * a class of 2 argument point predicates
 */
class PointPredfloat2 : public PointPred
{ public:
    PointPredfloat2(char* descr_,
                    int (*fcn_)(Pointp,float,float),
                    float arg1_,
                    float arg2_) : PointPred(descr_)
    { fcn = fcn_; arg1 = arg1_; arg2 = arg2_; }

    float arg1,arg2;
    int (*fcn)(Pointp,float,float);
    int virtual test(Pointp p)
    { return (*fcn)(p,arg1,arg2); }
};
```

The `PointPredfloat2` class contains tests that involve two floating point numbers. Can you tell which of the above classes each of the following generic functions belongs to?

```
/*
 * A few samples of point predicates
 */
int radiusLessThan(Pointp p, float r)
{ return (p->radius() < r); }

int radiusMoreThan(Pointp p, float r)
{ return (p->radius() > r); }

int radiusBetween(Pointp p, float r1, float r2)
{ return ((p->radius() > r1) && (p->radius() < r2)); }
```

To express a generic filter for points, it is useful to first define the lists of points that will be filtered. C++ includes *templates*, which are declared to the compiler by parametrizing a function or class definition with respect to type names. The following illustrates a definition of lists of points (pointed to by data of type `List<Point*>`) that uses a class template.

```
/*
 * define a list of points
 */
template<class ElementType>
class List
{
public:
    List(ElementType e,List<ElementType>*p_) // constructor
    { elem = e_; next = p; }

    ElementType elem; // introduced data fields
    List<ElementType> *next;
};

typedef List<Point> *Listp;
```

Finally, we can define the final function of this example: a generic filter function that accepts as arguments a list of points, and a point predicate from any non-abstract class descendant from `PointPred` (two of which were defined above). To produce output, it shows the point being tested, runs the test, and prints whether the test was passed or failed.

```
/*
 * A Generic Filter Function
 */
void filter(Listp list, PointPredp pred)
{
    while (list) // filter points on the list
    {
        list->elem->show(); // show the point being tested
        printf("- %s %s\n", // test point and print the result
               (pred->test(list->p))? "passed " : "failed ",
               pred->descr);
        list = list->next; // handle remaining points
    }
}
```

The following program constructs and uses objects of the above classes to show use of the above generic filter function.

```

main() /* demo */
{
    // make different kinds of points
    Pointp
        p1 = new Point( 1.0,2.0 ),
        p2 = new Point3D( 3.0, 4.0, 1.0 );

    // make a list of the points
    Listp
        list = new List(p1,new List(p2,0));
    // make different kinds of point predicates
    PointPredp
        pr1 = new PointPredfloat1(
            "radius less than 3",
            radiusLessThan, 3.0 ),
        pr2 = new PointPredfloat1(
            "radius more than 3",
            radiusMoreThan,3.0 ),
        pr3 = new PointPredfloat2(
            "radius between 2 and 3",
            radiusBetween, 2.0, 3.0 );
    // filter the list of different kinds of points
    // using the different kinds of point predicates
    filter( list, pr1 );
    filter( list, pr2 );
    filter( list, pr3 );
}

```

The output produced is as follows:

```

2dpoint(1.000000,2.000000) w. radius = 2.236068
    - passed r less than 3
3dpoint(3.000000,4.000000,1.000000) w. radius = 5.099020
    - failed r less than 3
2dpoint(1.000000,2.000000) w. radius = 2.236068
    - failed r more than 3 3dpoint(3.000000,4.000000,1.000000)
w. radius = 5.099020
    - passed r more than 3
2dpoint(1.000000,2.000000) w. radius = 2.236068
    - passed r between 2 and 3
3dpoint(3.000000,4.000000,1.000000) w. radius = 5.099020
    - failed r between 2 and 3

```

The value of generic code is that it works on so many different classes of objects in so many different ways. By combining generic behavior as in this example, effective reuse is achieved. In effect, the filter function we have defined is MxN different functions, where M is the number of different classes of point predicates, and N is the number of different classes of points. Generic code is highly reusable code. Generic code is supported by OOP because OOP allows the expression of general solutions according to the common structure of objects, while

virtual functions provide specialized behavior appropriate to the actual class of any object to which generic code is applied.

MULTIPLE INHERITANCE

Multiple inheritance allows creating new classes of objects that inherit from multiple parent classes. The parents may not be directly related via inheritance. This is useful because generic code of widely different purpose and utility may then be defined for use on unrelated classes. When two classes are combined using multiple inheritance, the new class can be used as arguments of either base class type. Multiple inheritance can provides the promise of even greater sharing of code.

Unfortunately, there is no adequate semantic model for multiple inheritance. There are a variety of different possible approaches to multiple inheritance, and all have drawbacks. Of course, one option for an OOP system is to simply not support multiple inheritance. But this has drawbacks, too. In any case, an important result of the lack of a completely acceptable model for multiple inheritance is the fact that the C++ model is not perfect either. In surprising ways, it can be very difficult to understand the correct use of multiple inheritance, or even single inheritance within an overall context of multiple inheritance in C++.

C++ is especially complex in this regard because it actually provides and integrates two different models of multiple inheritance. The first of these, the default model, is based on tree inheritance, while the second model, available in C++ by using “virtual base classes”, is based on graph inheritance[Snyder]. Argument conversion in C++ is more efficiently implemented and more easily understood in the case of tree inheritance, which is likely the reason that tree inheritance is the default model for C++. On the other hand, for OOP languages in which polymorphism is not based on argument conversion (in other words, all other OOP languages), graph inheritance seems more natural. The OMG CORBA specification for use of objects is based on the graph inheritance model, and, by itself, does not support use of objects implemented using general tree inheritance.

It seems unlikely that C++ would ever have introduced tree inheritance if graph inheritance had been more natural to its support for polymorphism. This is because even though argument conversion is very efficient for C++ objects implemented using tree inheritance, the user of objects implemented with tree inheritance may be required to explicitly guide the argument conversion process. This can be confusing to an object’s user without knowledge of the class hierarchy, and this is a consideration that is unnecessary in graph inheritance. Furthermore, because C++ hides the details of argument conversion from programmers as much as possible, novice C++ programmers can easily design classes with the graph inheritance model in mind and then be surprised by the requirements that are imposed on an object’s user by tree inheritance.

Just as with single inheritance, discussing multiple inheritance in C++ relies on understanding the structural components of objects whose classes are defined by subclassing. Often, the tree and graph inheritance models coincide—that is,

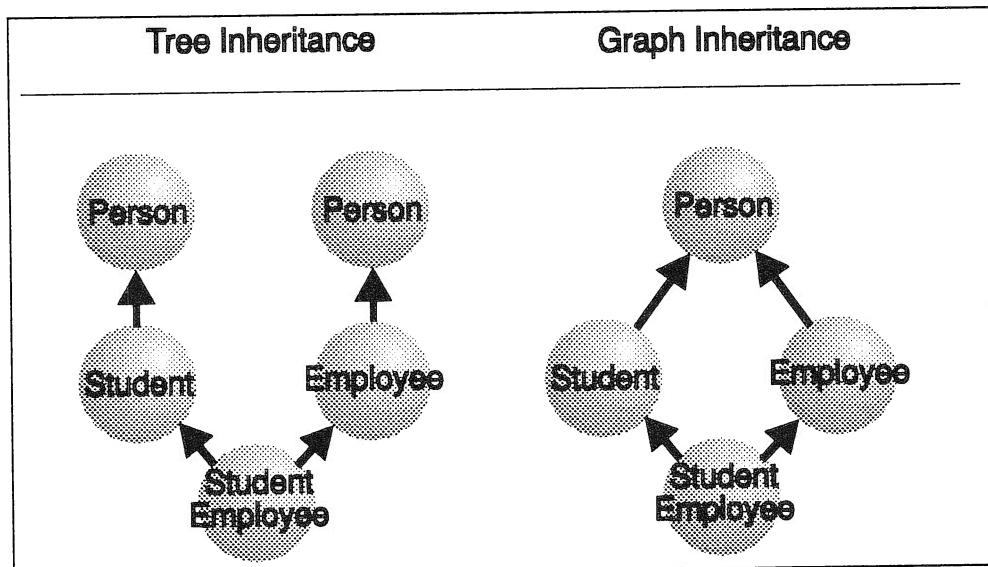


Figure 4.3 Tree vs. Graph Inheritance

there is no difference in the structural content of objects as a result of using one or the other approach to inheritance. The difference appears when two parents have some common ancestor. Of course, this can only happen when multiple inheritance is allowed.

For tree inheritance, an object inherits a separate copy of an ancestor class's object structure for each way that the ancestor class appears in the derivation of the object's class. Multiple appearances mean multiple object structures. For graph inheritance, there is only a single copy of any given ancestor class's structure, no matter how many ways the ancestor appears. Furthermore, these two different approaches can actually be mixed in C++.

For example, here is a simple class `StudentEmployee` whose derivation includes the ancestor class `Person` in two different ways (via both `Student`, and `Employee`).

```

class Person
{ public: char* person_name;
  ...
};

class Student : public Person
{ public: char* school_name;
  ...
};
  
```

```

class Employee : public Person
{ public: char* employer_name;
  ...
};

class StudentEmployee : public Student, public Employee
{ ...
};

```

The origin of the terms tree versus graph inheritance becomes clearer when you consider Figure 4.3. The left graph of Figure 4.3 illustrates the actual semantics of the above C++ class declarations. The right graph represents a second C++ class derivation which is given below, after first discussing tree inheritance. With tree inheritance, an object of class `StudentEmployee` will have two different `person_name`s—one located within the `Person` structure inherited from the `Student` class, and one located within the structure inherited from the `Employee` class. The following C++ code is therefore ambiguous, and in fact illegal.

Example6()

```

StudentEmployee *se = new StudentEmployee;
p->person_name = "scott"; // ambiguous
}

```

To remove the ambiguity, we must decide *which* `person_name` we want to use—the one that all `Employees` have, or the one that all `Students` have. Let us continue this example, and assume that we want to assign to both `person_name`s. Can you guess how this is done? The answer is to use type casting to select the desired portions of the `StudentEmployee` object, as follows:

Example6()

```

StudentEmployee *se = new StudentEmployee;
((Student*)se)->person_name = "scott";
((Employee*)se)->person_name = "scott";
}

```

Likewise, using argument conversion, we would be able to pass `se` to a procedure expecting a pointer to a `Student`, or a procedure expecting a pointer to an `Employee`. In either case, the procedure will be able to access an appropriate `person_name`. But, what if we want to pass `se` to a procedure expecting a pointer to a `Person`? This also would be ambiguous, since there are two distinct `Person` structures within a student employee. So, again the user of the object will need to explicitly cast `se` in one way or the other, as in Example6. Among the implications of this example is that programs that use an object in C++ must in general know the complete derivation of its class.

Now, if the class designer that defines `StudentEmployee` using multiple inheritance is also responsible for defining the classes `Student` and `Employee`, the way that `Student` and `Employee` inherit from `Person` could be reconsidered.

ered. In particular, the following class derivations for these classes could be used instead:

```
class Student : public virtual Person
{ public:
    char* school_name;
    ...
};

class Employee : public virtual Person
{ public:
    char* employer_name;
    ...
};
```

Given the above class declarations for `Student` and `Employee` on the right in Figure 4.3, there will be only one copy of the object structure inherited from `Person` in `Student`/`Employee` instances. The ambiguities identified above no longer exist. The `virtual` keyword used here has no relationship to the same keyword used when declaring virtual functions. The use of the word in this context is as a name modifier, to allow naming a special class guaranteed to contribute exactly one copy of its object structure to an instance of any class from which it is directly or indirectly derived, no matter how many ways it appears in the derivation. This consideration is entirely independent of the question of how many times (if any) this same inherited object structure also occurs as a result of using the given class as a “non-virtual” base class in ancestor derivations — i.e., *without* use of the `virtual` keyword. Thus, a given ancestor class can contribute its introduced data structure $n+1$ times for instances of a subclass — once for each different way in which the class is used as (regular) base class, plus once for all of the different ways it occurs as a virtual base class.

Note that the changed class derivations above are not even multiple inheritance derivations. A C++ programmer should be concerned with the possibility of multiple inheritance and the issue of graph vs. tree inheritance, even if the designer does not personally define any multiple inheritance classes. The question then arises as to how a C++ class designer should decide whether to use the default base class, or the corresponding virtual base class. As far as we can see, there is no way that a C++ class designer can know what the right decision is, unless they are responsible for the entire class hierarchy. In this example, some other class can always use `Person` as a regular “non-virtual” base class. Using `virtual Person` in `Student` and `Employee` derivations doesn’t guarantee that other classes derived from either of these will not introduce an ambiguity with respect to `person_name`. The best that can be done is to avoid actually causing an ambiguity problem for classes that inherit from both `Student` and `Employee`.

Why not simply always use virtual base classes? After all, other OOP languages work this way. At first glance, there seem very few reasons for ever wanting to use the default, non-virtual inheritance (i.e., tree inheritance). Graph inheritance is the most useful model because it reflects the unifying existence of

shared information as opposed to the ambiguity of replicated data. Yet this issue remains cloudy in C++ for a number of reasons.

One important factor bearing on this issue is that C++ constructors cannot be chained to indirect virtual base classes. This means that when a given class inherits directly from a virtual base class, and the constructor for this class invokes a chained constructor on the virtual base class, only for objects of this class will the chained constructor ever be executed. This is trouble, because the virtual base class may chain constructor calls higher on. If the virtual base class constructor is not called with its arguments, no higher constructors will be called with their arguments either. A class implementer would know that chained initializers to virtual ancestor classes would rarely execute, but, at the same time, would always have to chain constructor code to initialize *all* the virtual base classes from which it is derived, directly or indirectly.

What does all of this mean? Simply this: a class implementer *must always know and understand the derivation of all the classes above it in the C++ class hierarchy*. Imagine how difficult this could make things if a C++ class hierarchy needs to be modified. Modification of a class by deriving it from a virtual base class instead of the non-virtual base would require all descendant classes to be recoded!

Small wonder that there are few examples of virtual base classes in most C++ class hierarchies. But, then, it also seems that most C++ class hierarchies are primarily single inheritance with a few carefully designed exceptions, created with a global view of the whole hierarchy. We believe that multiple inheritance in C++ is quite limited. SOM does a much better job in this area. For SOM, merging of multiple-inheritance with metaclass technology produces rich opportunities for modular and extendable solutions.

There are many other difficult issues addressed by C++ in its support for multiple inheritance. Which method implements a given virtual function when there are multiple inheritance paths from the object's class to a virtual base class? C++ introduces a well-designed rule called *dominance* to chose a method in this case, and issues an error whenever there is the possibility of ambiguity.

Complications aside, the essential idea is that inheritance in C++ is inheritance of structure. Multiple inheritance doesn't change this, but dealing with the resulting structure is not always obvious. It is important to understand that this problem is inherent to multiple inheritance itself. Every OOP system must come to grips with these issues and deal with them in some consistent way if multiple inheritance is to be supported. But as we have indicated, only C++ supports tree inheritance. Also, most OOP systems don't try to support two different multiple inheritance models at the same time.

INTERFACING C++ OBJECTS WITH OTHER DOMAINS

An essential problem faced by C++ and other OOP languages concerns how to interface with other domains, which may or may not be object-oriented, but are not built on the same object model. For example, on OS/2 there is the desire to interface with PM, which has a very different style of support for OOP than

C++. The simplest interface is one based on procedure calls, and an accepted solution at this level involves the use of *callbacks*. The idea of a callback is that some service such as an event manager provides for registration of a procedure and some of the arguments that will be passed to this procedure when it is called. Then, at some later time, the registered procedure will be called, using the registered arguments, plus additional arguments relaying information provided by the service.

In C++, a programmer that registers a callback procedure often provides as a callback argument the address of a C++ object. Then, when the callback procedure is invoked and receives control, it can access whatever methods or instance data of this object are required to respond appropriately to the callback. But in C++, a variety of different mechanisms for name scoping are used to limit access to portions of an object. To prevent these mechanisms from restricting the activities of a callback procedure, a C++ programmer can use what are called *static member functions* to implement callbacks. Like constructors, these are not elements of object structure, but are simply procedures that are declared with the scope of a class definition, and which are therefore given full access to the names of the class's object's structure components. The following code illustrates a C++ class definition for Widget that includes a static member function that could be used as a callback. The point of this example is that the other code handling events may not be C++ code, or may not have been compiled with the same C++ compiler as the code that implements Widget. Therefore, the code that handles events may not be able to directly invoke methods on Widget. This is the problem that a callback solves.

```
class Widget
{
    /* Define a Callback Procedure */
    static void changeWidget(
        Widget *theWidget,
        EventData *theEventData)
    {
        ...
        theWidget->handleEvent(theEventData);
        ...
    }
    virtual void handleEvent(EventData *theEventData);
}
```

CONCLUSION

This chapter discussed how C++ supports OOP. The use of subclassing aids in determining object structure. Inheritance of methods for virtual functions and the use of types to guide the selection process supports polymorphism between related types. This enables the development of generic C++ code for object use. There is much to C++ that has not been discussed here. Nevertheless, you now have the essentials necessary for understanding the C++ examples and code that come later.

5

Object-Oriented Programming Using SOM

INTRODUCTION

IBM's System Object Model, SOM, provides the basic object support for OS/2. It supports the definition and use of language-neutral object-oriented systems. Although languages like C++ and Smalltalk also support OO systems, SOM does so in a way that is independent of programming languages. It does so by defining an application programming interface (API) to SOM objects that is based on simple procedure calls.

In a way, SOM is similar to PM. Both are based on runtime data structures and procedure calls uniformly accessible to various programming languages. But, where PM focuses on the special case of windows, SOM provides an exceptionally powerful and general-purpose object model. Furthermore, SOM accommodates different languages through *compile time bindings*, allowing programmers to encounter this functionality in terms they already understand.

SOM's objective is not to replace existing programming languages. It is designed to allow applications written in different programming languages to use the same binary OOP class libraries. SOM also allows class implementors to extend and replace these libraries without affecting existing application binaries. On OS/2, a SOM class library is distributed as a dynamic link library (DLL). It can be replaced without requiring the applications that use its classes to change at all. They don't even have to be recompiled, as long as the new library does not require changes in the applications' source code.

Binary software packaging technology is now fairly common in the case of procedure libraries. Most modern operating systems provide support for shared libraries, which may be used from different languages and which may be enhanced and replaced without recompiling client code. SOM is unique because it extends this capability to support OO class libraries. This is essential if system-provided OO application frameworks are to be used by software vendors, because end-users do not usually have access to source code for recompilation when new releases of a system library are installed. The benefits of OOP for software development are widely accepted, but SOM is the only system-level framework that allows binary packaging of language-neutral class libraries.

Language neutrality and support for backwards binary compatibility are two reasons why SOM was chosen for building the OS/2 Workplace Shell class framework. The result is that OS/2 application developers can use OOP for developing applications based on Workplace Shell classes, knowing that major enhancements and modifications of the Workplace Shell will not break their applications.

SOM need not be used to implement all of the objects and classes important to a class library. Many objects created and used by an application may be strictly internal. They may not need to be accessed by multiple languages or applications. Perhaps these internal objects will be SOM objects in those cases where a SOM class library provides useful functionality, or perhaps not. Only classes whose evolving implementations need backwards binary compatibility across application and language boundaries need to be SOM classes. No matter. Since the exposed interfaces are SOM, you get the benefits. This is OO encapsulation at its best.

At the most fundamental level, SOM has a runtime kernel and a set of compiler utilities that support language bindings. Four primitive SOM objects are created by the runtime. They provide the major portion of the SOM API. You can see that SOM itself derives benefits offered by binary class libraries. New releases of the SOM kernel DLL can provide increased functionality while still supporting existing application binaries. For example, version 2 of SOM supports the OMG CORBA (Common Object Request Broker Architecture) standard and it includes a variety of features not available from the original version of SOM. Differences include multiple inheritance classes and support for distributed objects, yet the original single inheritance SOM DLL used by OS/2 can be replaced with the version 2 SOM DLL without affecting the Workplace Shell or any other applications that depend on SOM!

The SOMobjects Toolkit available from IBM includes the SOM kernel, compiler utilities supporting language bindings, and a number of OOP frameworks built from SOM classes, including a metaclass framework, a collection class framework, a persistence framework, a replicated object framework, a distributed object framework, and an emitter framework. Class frameworks allow application programmers to inherit useful capabilities by subclassing. They can specialize a small number of specific methods in order to tailor the framework to the particular application at hand. The SOMobjects Toolkit frameworks are reviewed later, with some interesting examples.

INTRODUCTION TO SOM TERMINOLOGY

A *SOM Object* is a run-time entity with a specific set of *methods* and *instance variables*. The object's methods are used to request behavior from the object. The instance variables encode the object's state. They can support method execution. When a method is invoked on an object, that object is said to be the *receiver* or *target* of the method call.

An object's *implementation* determines the procedures that execute its methods. Such procedures are called *method procedures*. Implementation determines the types and overall structural layout of the object's instance variables. SOM does not care which languages are used to define method procedures and instance variables. This is because the method procedures and instance variables that implement an object are encapsulated. Instead, an object's user is given access to the object through an *interface* that hides these details.

An *object type* indicates how an object can be used, as opposed to how the object is implemented. The type represents a specific interface through which an object's methods may be invoked. By declaring an object type for a procedure argument or a local variable, a programmer chooses the interface that will be used when accessing the corresponding object. CORBA's Interface Definition Language (IDL) is used to define SOM classes, and the SOM language bindings allow programmers to use these interface names when typing variables.

In SOM, as in most approaches to OOP, classes define the implementation of objects. Every SOM object is an instance of some specific SOM class. Because a class determines the available methods and instance data, all of the instances of a given class have the same interface, and thus the same type. The process of class definition in SOM starts with an IDL definition of the interface, and the name of this interface is then given to the class as well. As a result, a class name in SOM is always a type name, even though the things that are named are different—a type is an object interface, a class is an object implementation. Because the first step in defining a SOM class is to specify the interface, an IDL interface definition in SOM is often referred to as a *class declaration*.

Subclassing derives new classes by inheriting, extending and modifying the implementations of a classes. The classes from which a new class is derived are called its *parent classes* (direct base classes in C++). Through subclassing, a derived class inherits all of its parent's methods and instance variables. Furthermore, by default, a subclass inherits the parents' implementation for these methods. As usual, subclasses can override inherited methods by defining new method procedures, and can *introduce* new methods and instance variables. If a class results from several generations of successive class derivation, that class supports all of its *ancestors'* methods. If different parents have methods of the same name that execute differently, SOM provides ways for resolving ambiguities. SOM's multiple inheritance model uses graph inheritance, which we explained in the C++ chapter.

The relationship between classes and types in SOM can be expressed as follows. Objects have a class, and program variables have a type. A variable is correctly typed if (1) the object is an instance of the class named like the type, , or any class derived from the class named like the type. We often use the

phrase *an object of type X*, which means that the object is either an instance of the class X, or an instance of some class derived from X.

In SOM, classes are themselves objects. This means that classes have their own methods and interfaces, and are implemented by other classes. For this reason, we often use the term *class object* when we refer to an object that is a class, and talk of *class methods* or *class variables* in order to distinguish between the methods and instance variables of this object and those that it defines for its instances.

A class whose instances are classes is called a *metaclass*, so in SOM, the methods of a class are defined by a metaclass. A class's methods include those that implement inheritance from the class's parents. As a result, if you define metaclasses, you can actually modify the default semantics of inheritance provided by SOM. To support this, SOM provides a unique facility called *derived metaclasses*. Let's go into this in more detail.

SOM CLASSES

The SOM kernel DLL contains two primitive classes that are the basis for all subsequent classes:

- SOMObject, the root ancestor class for all SOM classes, and
- SOMClass, the common ancestor class of all SOM metaclasses.

Also, the SOM kernel includes a third class, SOMClassMgr, that maintains a SOM class registry. This registry allows dynamically loading and unloading DLLs that define SOM classes.

All SOM Objects are an instance of a SOM class, and all SOM classes are ultimately derived by subclassing from SOMObject. Thus, all SOM objects, meaning all proper objects, proper classes, and metaclasses, can execute the methods introduced by SOMObject. In a similar manner, all SOM classes must be an instance of a SOM metaclass, and all SOM metaclasses are ultimately derived from SOMClass. This means all classes can execute the methods introduced by SOMClass. Because they are classes, the three primitive classes listed above are all instances of the metaclass SOMClass.

When SOM starts up, the above classes and a class manager object are created and initialized with a bootstrap procedure. The relationships between the resulting four runtime SOM objects are shown in Figure 5.1. Two different kinds of relationships are shown. The inheritance relationship between class objects should be familiar. However, the instance relationship between a class object and its instances is new, at least in comparison with C++.

The existance of class objects in SOM results in a flexibility and power not available to C++ programmers. The extent of this difference will be demonstrated later. For now, it important to understand that the major difference between SOM and OOP systems without class objects is the instance relationship. This provides an additional dimension of design when defining classes, and, for SOM programmers, can be of equal importance to the inheritance relationship.

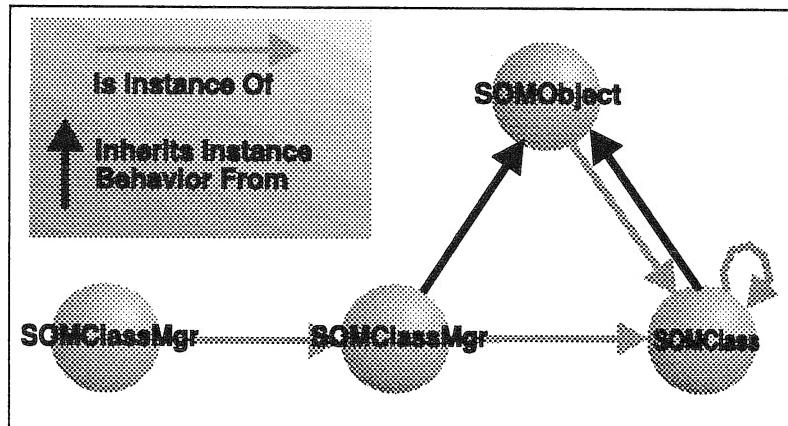


Figure 5.1. SOM Relationships

within class designs. Figure 5.2 provides a more elaborate example of some possible SOM objects and their relationships.

USING SOM CLASSES

SOM classes are designed to be language neutral. A given class of SOM objects may be implemented in one programming language and subclassed or used by another. To achieve this kind of language neutrality, the interface for a class of objects must be defined separately from its implementation. The purpose of this section is to briefly introduce the IDL language used to define SOM object interfaces, and then to illustrate how usage bindings for SOM classes allow a program to create and use SOM objects through their defined interfaces. We'll provide a bit more detail concerning IDL when discussing the definition of SOM classes in a following section. You can refer to the SOMobjects Toolkit Users Guide for more information if you need it.

Object Interface Definitions

IDL is a formal language defined by the CORBA standard of the OMG (Object Management Group). IDL's purpose is to define object interfaces. Just as subclassing in SOM is used to derive a new class from existing classes, subtyping in IDL is used to derive new object types from existing object types. At first, we can think of an IDL object type as a set of method signatures. Subtyping in IDL then appears as a union operation, in which the method signatures from each parent interface are combined into a new interface that supports all the inherited methods.

Like SOM, the CORBA standard was developed to support language neutrality. As a result, a notable property of IDL interfaces is that they contain only methods. Object instance variables are not available through IDL interfaces.

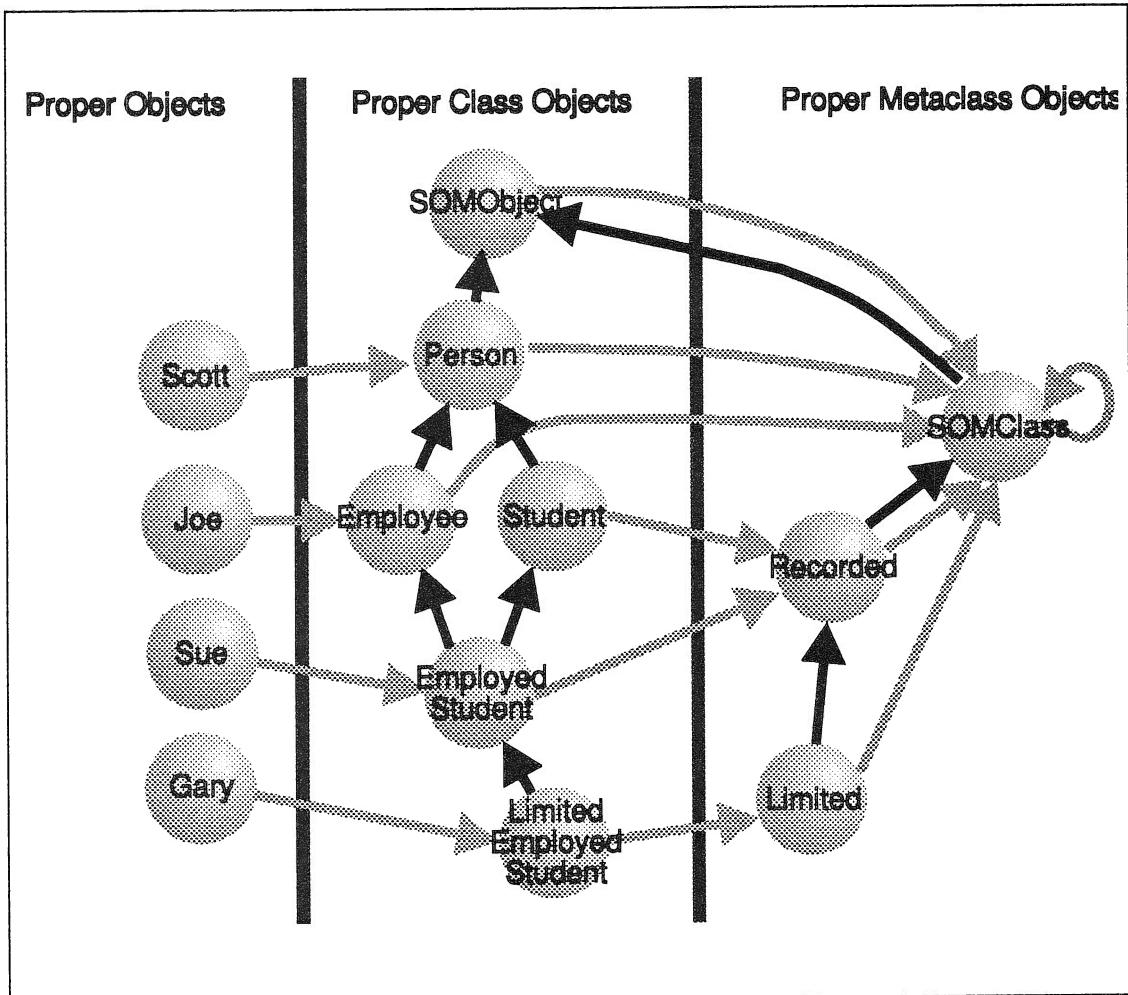


Figure 5.2. Other Possible Relationships

SOM does provide an API through which instance data within objects may be directly accessed. This is an important aspect of SOM discussed in a following section concerned with class implementation. Here, we focus on the aspects of SOM made available to their users through IDL interfaces. The following is an example interface definition expressed in IDL:

```

// include parent interface definition
#include <somobj.idl>
// NamedObject inherits from SOMObject
interface NamedObject : SOMObject
{
    // introduce _get_name and _set_name methods
    attribute string name();
    // introduce a method for comparing NamedObjects
    boolean compareName(in NamedObject obj);
};

```

The above interface declaration defines the `NamedObject` type, which inherits all of the methods in the `SOMObject` interface. `NamedObject` introduces three new methods: `_get_name`, `_set_name`, and `compareName`. An IDL method declaration has target, argument, and return types. There are two implicit arguments. `SomSelf` represents the target object, and its type is the same as the target interface. The other implicit argument is an *environment*. As defined by CORBA, this simple data structure is not an object. It is used to return status information to a calling process. It is useful for handling error conditions in a distributed environment. SOM provides routines for accessing or creating new environments when programmers do want to explicitly concern themselves with environment structures. We won't say much more about it, but the SOM documentation can tell you more.

IDL *attributes* provide a shorthand syntax for introducing *get* and *set* methods. For example, the `_get_name` method applied to an object of type `NamedObject` takes as its single argument an environment (as explained in the above paragraph), and returns a string, which is the type of the attribute. The `_set_name` method takes two arguments: an environment and a string, and it returns nothing. IDL semantics say absolutely nothing about how these two methods are implemented, or what behavior they are intended to perform. For example, the result returned from a `_get_name` invocation need not be the same name as that indicated in the most recent `_set_name` invocation. If this seems strange, think of a clock object with a time attribute. IDL is simply a language for defining object types, which are basically sets of method signatures. IDL is by design not concerned with aspects related to implementation.

Objects are passed through IDL method interfaces *by-reference*. For example, the arguments used when invoking the `compareName` method on a target object would be an environment and a pointer to an object of type `NamedObject`. In SOM, object references are implemented as pointers, or memory addresses. A framework built on top of SOM called Distributed SOM or DSOM, uses *proxy* objects to extend the SOM model. The proxy objects are just objects that stand for some other object. They mirror the interface of their associated object. Proxy objects handle distribution and multiple address spaces. DSOM and other class frameworks are included in IBM's SOMobjects Toolkit.

Creating and Using SOM Objects

As an example of SOM object use, let us assume that a SOM class implementor has implemented the class `NamedObject`, after first defining the interface to its objects using IDL exactly as illustrated above. Furthermore, the class implementor has given us two files representing this implementation (`nmdobj.dll`, and the corresponding static import library, `nmdobj.lib`), and also has given us the above IDL specification (located in file `nmdobj.idl`). Assuming we have access to the SOMobjects Toolkit, what are the steps we would go through to create and use objects of class `NamedObject` on OS/2?

- Choose a “client” programming language.
- Use the SOM compiler to generate client language usage bindings for `NamedObject` from its IDL specification. For C, this is an `.h` file.
- Write the client program, including the generated usage bindings for `NamedObject`.
- Compile the client program.
- Link it with the static import libraries for both the SOM runtime and the `NamedObject` class (the, `somtk.lib` and `nmdobj.dll`).
- Run the program, making sure that the SOM kernel DLL and the `NamedObject` DLL are in the OS/2 LIBPATH.

The client programming language is chosen without concern for the language used to implement the class. In general, SOM users never know the language used to implement SOM classes, just as the users of a procedure library do not know what language was used to implement the procedures.

Although most any language can use the SOM API, it is quite involved and difficult. The main practical concern when choosing a client programming language is availability of an emitter for usage bindings that hide the details of the SOM API, providing instead a simple language-specific interface. In the examples given here, we use IBM CSet/2 C++ as the client programming language. C and C++ usage bindings are provided by the SOMobjects Toolkit. Bindings for other languages may be provided by various compiler vendors. Also, the SOM Toolkit provides an object-oriented emitter framework for construction of new language bindings, so we should see more languages supported in the near future. Here are some further details concerning the above steps:

First, the SOM compiler is used to generate C++ usage bindings for objects of type `NamedObject`. The input to the SOM compiler is an IDL file, `nmdobj.idl`. The output is determined by the *emitter* that is chosen. The emitter for C++ usage bindings is named `xh`. Also, the SOM compiler uses an environment variable `SMINCLUDE` to search for `.idl` files included by the `.idl` files that it processes. Because `nmdobj.idl` contains an include statement for `somobj.idl`, it is necessary that `SMINCLUDE` be set so this file can be found. To generate C++usage bindings for `NamedObject`, the following commands can be used:

```
set SMINCLUDE=.;%SOMBASE%\include
sc -sxh nmdobj.idl
```

The output from the above commands will be a file named nmdobj.xh. This can be used by a client program, as illustrated in the following example:

```
// filename: example.cpp - written by a SOM user
#include nmdobj.xh
#include <stdio.h>
main()
{
    NamedObject *no1 = new NamedObject;
    NamedObject *no2 = new NamedObject;
    environment *ev = somGetGlobalEnvironment();
    no1->_set_name(ev, "Name1");
    no2->_set_name(ev, "Name2");
    boolean testresult = no1->compareName(ev, no2);
    printf("compareName on %s with %s returns %d\n",
           no1->_get_name(ev),
           no2->_get_name(ev),
           testresult);
}
```

When compiling a SOM client program, it is necessary to access the usage bindings for all ancestors of the classes used. When linking, it is necessary to access the import libraries for all these classes. These statements can compile and link:

```
set INCLUDE=%SOMBASE%\include;.;%INCLUDE%
set LIB=%SOMBASE%\lib;.;%LIB%
icc example.cpp somtk.lib nmdobj.lib
```

The output from the above commands will be, example.exe. It can be executed if the necessary DLL's are in the OS/2 LIBPATH. If, at some future date, the implementor of the NamedObject class gives us a new implementation DLL, the original nmdobj.dll file can be replaced with the new version, and the example.exe file will still be valid. There is no need for recompilation.

Of course, without knowing more about the implementation of NamedObject objects (in particular, the intended semantics of their methods), we have no idea what the above program will print. For this reason, a class implementor should always provide comments in an IDL file. These comments are important not only to guide users of a given class's objects, but also to assist other class implementors when they override the inherited implementation for methods.

This chapter focuses on the static interface to SOM objects supported by CORBA IDL. In addition, SOM supports name-lookup access to methods, and a generic *apply* method interface appropriate for interpretive applications. For complete information on facilities provided by the SOM usage bindings, including further details on the use of SOM objects, check out the *SOMobjects Toolkit User's Guide*.

IMPLEMENTING SOM CLASSES — THE BASICS

This section focuses on the basics of implementing SOM classes when language bindings are available. A new SOM class is declared by using IDL to define the type of its objects. Then language bindings can then be created by choosing an appropriate emitter. In addition to interface information, you can use IDL to provide implementation information that guides SOM emitters in the creation of an *implementation binding* file for the new class, and an *implementation template* file.

The implementation binding file is not intended to be modified. Of course most C .h files are treated in the same way. It contains code that handles the details of the SOM API that create and register the new SOM class object. This is all done automatically, based on the class declaration. The implementation template file contains templates for method procedures for newly-introduced methods, or for overridden methods. These are filled in by the class implementor to achieve the desired semantics. The implementation template emitters for C and C++ work incrementally. They never change existing code in a previously generated method procedure template, but they can be used to add new method procedures to a template and to change the signatures on existing method procedures as a result of IDL modifications.

The implementation information included within the IDL file is only intended to be seen by SOM compiler tools, so it is guarded with an IDL #ifdef. For example, the following IDL specification shows the approach by adding to the nmdobj.idl specification.

```
#include <somobj.idl>
interface NamedObject : SOMObject {
    attribute string name;
    boolean compareName(in NamedObject obj);
#ifdef __SOMIDL__
    implementation {
        releaseorder: _get_name, _set_name, compareName;
        somInit: override;
    };
#endif
};
```

The additional information in the implementation section consists of an ordering for the new methods that are introduced by the interface, and an override for the method somInit, which is inherited from SOMObject. The method ordering is needed because SOM's implementation of object interfaces is based on an offset that depends on the ordering of methods. The resulting interface then supports a style of method resolution called offset resolution which is almost as fast as C++. When new methods are added to an object interface, they are always placed at the end of the release order so binaries that use the previous interface still work. SOM also supports name-lookup method resolution, which is not dependant on object types or release order. This can be useful. We will talk about it in the next chapter.

The override modifier for `somInit` means a new method procedure will be used for executing this inherited method. As a result of this information, a template for the new method procedure is provided by the template emitter. The implementation bindings emitter also introduces an instance variable that corresponds to the `name` attribute, and automatically produces corresponding `_get_name` and `_set_name` methods for within the class implementation bindings. Modifiers are available for changing this default in various ways. For example, templates for attribute methods can be placed in the implementation template file if desired. The template file in this case will contain two method procedures--one for `somInit`, and one for the `compareName` method. The implementation bindings will include the `_get_name` and `_set_name` method procedures in addition to the rest of the code necessary for runtime creation of the class `NamedObject`.

To summarize, the overall process of implementing a SOM class on OS2 involves the following steps:

- SOM IDL is used to provide language-neutral implementation information as well as interface information.
- An implementation language is chosen.
- Usage bindings, implementation bindings, and an implementation template are generated using emitters for the chosen implementation language. Usage bindings are generated because the implementation template and implementation bindings use portions of the SOM API that are provided by objects, and because they may use objects of the class being implemented.
- The implementation template file is edited by the class implementor to complete the definition of all the necessary method procedures.

The implementation bindings and the completed template are compiled and linked into a DLL, and an import library is created.

The implementation language is chosen without concern for the language used to implement ancestors of the class. In general, a class designer doesn't even know what languages were used to implement ancestors. We will use CSet/2 C++, since C++ is supported by SOMobjects Toolkit language bindings. The relation between template, implementation, and usage binding files is as follows: the template file includes the implementation binding file, and the implementation binding file includes the usage bindings.

We have already seen how usage bindings for C++ are generated—the `xh` emitter is used. The other emitters used to implement the above class in C++ are the `xih` emitter for the implementation bindings, and the `xc` emitter for the implementation template. For example, using the same `SMINCLUDE` environment variable as before, the following invocation of the SOM compiler would produce the usage binding, implementation binding and implementation template files:

```
sc -sxh;xih;xc nmdobj.idl
```

The entire content of the resulting implementation template file (`nmdobj.cpp`) is as follows:

```
/*
 * Generated by the SOM Compiler and Emitter Framework.
 */
#define NamedObject_Class_Source
#include <nmdobj.xih>
SOM_Scope boolean SOMLINK compareName(
    NamedObject *somSelf,
    Environment *ev,
    NamedObject* obj) {
    NamedObjectData *somThis = NamedObjectGetData(somSelf);
    NamedObjectMethodDebug("NamedObject", "compareName");
    /* Return statement to be customized: */
    { boolean retVal; return (retVal); }
}
SOM_Scope void SOMLINK somInit(NamedObject *somSelf)
{
    NamedObjectData *somThis = NamedObjectGetData(somSelf);
    NamedObjectMethodDebug("NamedObject", "somInit");
    NamedObject_parent_SOMObject_somInit(somSelf);
}
```

Two simple text macros used by the implementation templates are `SOM_SCOPE` and `SOMLINK`. These are used to simplify linking. System linkage is used since it provides a single protocol for argument passing that compilers for different languages like.

The initial `#define` statement (at the beginning of the implementation template file) is provided because much of the implementation binding file content is guarded with a corresponding `#ifdef`. This allows other source files to include the implementation bindings without needing multiple definitions for the procedures and data structures.

Other source files might want to include these implementation bindings because the unguarded portion of the implementation bindings defines the instance data structure introduced by `NamedObject`. This structure can be used with an untyped SOM API provided for direct access to instance data structures within SOM objects. This provides capabilities similar to the “friend” concept in C++.

To illustrate how implementation bindings define introduced data structures, an excerpt from the `NamedObject` implementation bindings is now included. Note that the `NamedObjectGetData` macro defined here is used in the template method procedures to acquire access to the instance data introduced by `NamedObject`.

```
/*
 * nmdobj.xih — Generated by the SOM Compiler.
 */
/*
```

```

* — Instance Data Structure
*/
typedef struct {
    string name;
} NamedObjectData;
/*
 * — GetData Macro
 */
#define NamedObjectGetData(somSelf) \
    ((NamedObjectData *) \
        SOM_DataResolve(((SOMObject *)((void *)somSelf)), \
        NamedObjectCClassData.instanceDataToken))
...

```

This excerpt begins with the introduced instance data structure for `NamedObject`. This is the structure that will be used to hold the `NamedObject` instance data—in this case, to store `name` attribute data. In addition, the excerpt shows a macro that uses the untyped SOM API for direct access to the instance data introduced by `NamedObject`. The macro casts the result of `SOM_DataResolve` to a pointer to a structure of type `NamedObjectData`, making this pointer useful to the C++ code within the method procedure templates.

The method procedure templates for `NamedObject` illustrate a fundamental difference between the way C++ and SOM support support polymorphism. As described earlier, C++ modifies object pointers so the instance data pointed to by `this` within C++ method code can be directly accessed by method procedures. But this is the only pointer available to C++ methods. Access to the overall C++ object is lost. This doesn't happen in SOM. Instead, the object pointer in SOM is not changed when the method procedure is invoked. The *entire object* remains accessible. When access to instance data is necessary, the `GetData` macro is used to locate and point to the appropriate structure. By convention within implementation templates, SOM uses the name `somSelf` to reference an object, and the name `somThis` to reference instance data.

Note that the method procedure templates only arrange for direct access to the instance data structure introduced by `NamedObject`. There are two important reasons. First, the instance data introduced by ancestors of `NamedObject` will not be useful to the `NamedObject` method procedures because different implementation languages may have been used. For example, we have chosen to implement `NamedObject` using C++, but there is no guarantee that ancestor classes are implemented in the same language. Inherited instance data might be Smalltalk object references. Second, and more fundamentally, providing automatic access to inherited instance data could result in a binary dependency between the `NameObject` class implementation and those of its ancestor classes.

The implementation templates produced by SOM are designed to support subclassing from binary class libraries. They do not automatically provide direct access to inherited instance data, but SOM allows you to create binary dependencies between classes when you want to. This can be useful to provide an effi-

cient coupling between different class implementations. To do this, you simply include the necessary ancestors' implementation binding files and use the corresponding GetData macros. This often makes sense within the restricted context of a class library. Of course, when this is done, it becomes necessary to recompile class implementations when dependencies change.

Two additional aspects of the implementation templates are the automatic generation of debug macro invocations and the automatic inclusion of "parent method" calls for inherited methods. The purpose of a parent method call in SOM is to allow the implementation of an overridden method to invoke the method procedure of the parent.

The debug macro used by C and C++ implementation templates is defined in the corresponding implementation bindings, and tests whether the global variable `SOM_TraceLevel` is non-zero to determine whether or not to print a message on entry to a method procedure. To turn tracing on, simply assign a non-zero value. To avoid the overhead of testing this variable once your class implementation is debugged, you can either delete the debug statements or insert the following after the include of the implementation bindings:

```
#undef NamedObjectMethodDebug
#define NamedObjectMethodDebug
```

The SOM API supports access to the method procedure used by any given class to support any given instance method. `somClassResolve` supports this capability. Parent method calls are given additional special support by the C and C++ implementation bindings. This is because overriding is often used with the desire of incrementally extending a method. Implementation bindings therefore define parent method calls for each parent that supports an overridden method, and the implementation template for the overridden method is generated so that it invokes each of these. You can keep the parent call if you want, or delete it.

Parent method call macros are given unique names. Uniqueness is guaranteed even when the implementations of multiple classes are contained within a single compilation unit. This is done by using the name of the class being implemented as well as the name of the parent class in the macro name. This is necessary because IDL allows the definition of modules that contain multiple interfaces. When modules are defined, a single implementation binding file and implementation template file are generated for all the corresponding classes. This provides an easy and safe way of implementing C++ friends, since each class implementation then has access to the others' introduced instance data structure.

To complete the implementation process, you fill out the method procedure templates with C++ code. For example:

```
/*
 * Generated by the SOM Compiler and Emitter Framework.
 */
#define NamedObject_Class_Source
#include <nmdobj.xih>
#include <string.h>
```

```

SOM_Scope boolean  SOMLINK compareName(
    NamedObject *somSelf,
    Environment *ev,
    NamedObject* obj) {
    NamedObjectData *somThis = NamedObjectGetData(somSelf);
    NamedObjectMethodDebug("NamedObject", "compareName");
    return !strcmp(somThis->name,
                    obj->_get_name(somGetGlobalEnvironment()));
}
SOM_Scope void  SOMLINK somInit(NamedObject *somSelf)
{
    NamedObjectData *somThis = NamedObjectGetData(somSelf);
    NamedObjectMethodDebug("NamedObject", "somInit");
    somSelf->_set_name(somGetGlobalEnvironment(),
                        "unknown");
    NamedObject_parent_SOMObject_somInit(somSelf);
}

```

When a new object is created, SOM invokes the `somInit` method on it. Class implementors that introduce instance data can override this method to place this data in some default state. If this is done, parent method calls should be used (as here) to allow ancestor classes to also initialize their introduced instance data.

Finally, the completed template is compiled and linked, and a static import library is created which defines the exported symbols of the DLL.

Programs can use a SOM class library (i.e., a DLL containing SOM class implementation binaries) in one of two ways:

- 1 . The program may employ SOM usage bindings to instantiate the class, create class instances, and invoke their methods. The resulting client program then contains static references to implementation binding procedures that create and register the class. When this occurs, OS/2 will automatically resolve those references when the program is loaded, by also loading the appropriate class DLL.
- 2 . SOM also provides interfaces for dynamic loading and use of SOM classes, which allows for runtime discovery and use of new SOM classes. The Workplace Shell takes advantage of these facilities to allow installing new classes of objects onto the desktop.

Because the provider of a class library cannot predict which of these ways classes will be used, SOM class libraries should be built so that either usage is possible. The first case above only requires the class library to export the entry points needed by the SOM bindings, whereas the second case also requires the library to provide an initialization function that creates and registers the classes it contains. This initialization function can then be called by SOM when the class is dynamically loaded, to create and register all of the classes in the DLL.

The SOM Compiler provides an OS/2 "def" emitter to produce a file containing the necessary exported symbols. The following command can be used to create a .def file for the NamedObject class:

```
sc -sdef nmdobj.idl
```

Once this has been done, the corresponding static import library is created as follows:

```
implib /noi nmdobj.lib nmdobj.def
```

The first filename (nmdobj.lib) specifies the desired name for the new import library and should always have a suffix of ".lib". The second filename (nmdobj.def) specifies the exported symbols to include in the import library.

As explained above, a DLL initialization function should be provided by a SOM class library to support dynamic loading of new classes. Here is a C++ initialization function for a NamedObject DLL. In general, a DLL will include multiple class definitions, and the initialization function would then invoke the creation procedures for multiple classes. (Implementation bindings for class <className> always define the class creation procedure, <className>NewClass.)

```
#include <nmdobj.h>
SOMEXTERN void SOMLINK SOMInitModule (
    long majorVersion,
    long minorVersion,
    string className)
{
    NamedObjectNewClass (
        NamedObject_MajorVersion,
        NamedObject_MinorVersion) ;
}
```

The source code for the initialization function can be added to one of the implementation files for the classes in the library, or you can put it in a separate file and compile it independently. The final step is creation of the DLL, as follows:

```
set LIB=%SOMBASE%\lib;%LIB%
link386 /noi /packd /packc /align:16 /exepack nmdobj.obj
initfunc.obj, nmdobj.dll,,os2386 somtk, nmdobj.def
```

If your classes make use of classes in other class libraries, which is the general case, you should include the names of their import libraries immediately after somtk (before the next comma).

This completes an initial description of SOM, showing how programmers both use and implement SOM objects. In the next chapter, we introduce important aspects of the the SOM API to set the stage for the Workplace Shell programming examples that will come later.

6

Using the SOM API

INTRODUCTION

SOM provides all the essential features of OOP. It supports language neutrality and upwards compatibility for binary executables. It makes these features available to users through an IDL interface. Using IDL and the SOM language bindings, you can easily translate the C++ point filter example into a corresponding collection of SOM classes, but there is more to SOM than this.

OOP languages like C++ provide a fixed syntax that allows programmers to define and create objects and invoke methods on them. In contrast, the SOM kernel provides an instruction set designed for supporting OOP capabilities such as method inheritance and method resolution. This instruction set, the SOM API, consists of OOP methods that can be overridden by a SOM programmer. The result is a flexibility that can be influenced by SOM class designers.

Look at SOM as a class library that can be extended. The instruction set is the SOM API. The SOM kernel provides initial classes that consists of the SOMObject, SOMClass, SOMClassMgr, and SOMClassMgrObject classes, but by subclassing, a you can seamlessly extend this library with additional classes used to support the SOM API in different ways. For example, the methods of programmer-defined metaclasses can be used instead of SOMClass methods to build new class objects and initialize instance method tables.

There are many ways to use the instruction set provided by SOM. You can use the SOM API directly, to dynamically create new class objects at runtime and to create and use instances of these classes. You can also use the SOM com-

piler to do the work for you. It takes IDL class declarations and hides use of the SOM API within language-specific bindings which provide others with a well-defined IDL interface through which the resulting objects may be used. Finally, an OOP language compiler might produce SOM instructions directly, avoiding the need for language bindings. For example, a C++ compiler might map C++ programs to the SOM instruction set, using SOM classes to implement C++ objects and using IDL to represent the resulting object interfaces in a language-neutral fashion. If this is done, C++ objects become useful to programs written in other languages as well. At the time of this writing, two different C++ compiler vendors are developing such *DirectToSOM* C++ compilers.

All of these levels let you directly use the SOM API. For example, metaclass designers currently use IDL to declare a new metaclass, use language bindings to automate much of its implementation, and then make direct use of the SOM API from within metaclass method procedures.

Within an overall OOP context, SOM provides powerful system-level capabilities not available from other object models. These capabilities are often used by SOM programmers that define metaclasses. Important central concepts used for this purpose include method resolution, method inheritance, explicit metaclasses, and derived metaclasses. This chapter provides a brief introduction to these concepts and describes their overall interaction.

METHOD RESOLUTION

Method resolution is the step of determining which method procedure to execute in response to a method invocation on an object. For example, consider this scenario:

- Class "Dog" introduces a method "bark," and provides a method procedure for its execution.
- A subclass of "Dog," called "BigDog," overrides "bark." providing a different method procedure for implementing "bark".
- A client program creates an instance of either "Dog" or "BigDog" depending on some runtime criteria and invokes method "bark" on that instance.

Method resolution in this case is the process of determining, at run time, which method procedure to execute in response to the method invocation. This may be either the method procedure for "bark" defined by "Dog," or the method procedure for "bark" defined by "BigDog", depending on the class of the object to which the method is applied.

SOM allows you considerable flexibility in deciding how SOM performs method resolution. In particular, SOM supports three mechanisms for method resolution. In order of increased flexibility and decreased performance, these are offset resolution, name-lookup resolution, and dispatch-function resolution. You can influence the default behavior exhibited by each of these in various ways. For instance, the behavior of offset resolution could be affected by defining new

metaclasses whose classes load their instance method tables in some special way. Also, the behavior of dispatch-function resolution is specially designed to allow runtime decisions concerning the implementation of methods—even when these methods are invoked using offset resolution.

Offset Resolution

SOM's C and C++ language bindings use offset resolution as the default because it is the fastest, nearly as fast as an ordinary procedure call. Offset resolution in SOM is roughly equivalent to the way that C++ resolves virtual functions. Although offset resolution is the fastest technique for method resolution, it is also the most constrained. Using offset resolution involves these constraints:

- The name and signature of the method to be invoked must be known at compile time
- The name of the class that introduces the method must be known at compile time, although not necessarily by the programmer—usage bindings can handle these details
- The method to be invoked must be part of the introducing class's IDL interface definition

The first step is to obtain a method token from a global data structure associated with the introducing class. This data structure is called the `ClassData` structure, and it includes a method token for each method introduced by the class. For example, here is a simple IDL class declaration, and the resulting `ClassData` structure:

```
interface X : Y, Z {
    long foo(float arg1, boolean arg2);
    attribute boolean healthy;
    implementation {
        releaseorder: f1, _get_healthy, _set_healthy;
    };
};

// A C++ definition for the XClassData structure
extern "C" struct XClassDataStructure {
    SOMClass      *classObject;
    somMToken     foo;
    somMToken     _get_healthy;
    somMToken     _set_healthy;
} XClassData;
```

The first field of a `ClassData` structure is always a pointer to the corresponding class object, and the remaining fields (which are determined by the `releaseorder` specified in the IDL implementation section) are method tokens.

A method token is used as an “index” into a target object’s method table, to access the appropriate method procedure. The SOM API function `somResolve` performs this indexing process. An object and a method token are passed to it,

and it returns the address of the correct method procedure. It is known at compile time which class introduces the method. It is also known where in that class's ClassData structure the method's token is stored. This means offset resolution is quite efficient. Furthermore, method tokens are now implemented as thunks, which means that they can be called directly in order to invoke methods. On OS/2, method thunks require as few as three assembly language instructions for accessing an object's method table and calling the correct method code!

An object's method table contains pointers to the procedures that implement the methods supported by an object. This table is constructed by the object's class and is shared among the class instances. The method table built by a class is referred to as the class's *instance method table*. This is useful terminology, since a class is itself an object with an associated method table which is used to support method calls on the class object. An object's method table is thus the instance method table of its class.

Here is a C++ example of using the SOM API directly to invoke the method `foo`, introduced by the class `X`, above, on an object of type `X` (i.e., an instance of `X`, or any class derived from `X`).

```
#include <X.xh>
void OffsetExample(X *x)
{
    ((somTD_X_foo) (somResolve(x,XClassData.foo)))
        (x, 12.34, TRUE);
}
```

The name `somTD_X_foo` is defined by the `X` usage bindings. It is the type of method procedures that implement the `foo` method. Casting the result of `somResolve` to this type allows the compiler to correctly call the method procedure. On OS/2, this type *must* indicate system linkage, because that is how all SOM method procedures expect to be called. If you know that method tokens are implemented using thunks, the above code could be replaced with the following code. Current OS/2 usage bindings do this, using a similar inline expansion for method calls:

```
include <X.xh>
void OffsetExample(X *x)
{
    ((somTD_X_foo) (XClassData.foo)) (x, 12.34, TRUE);
}
```

Offset resolution determines the method procedure used to implement a method for some specific class of objects—not necessarily the class of the object to which the method is being applied, but often some ancestor of the object's class. This capability is not provided directly by current usage bindings, but is available through use of the `somClassResolve` function provided by the SOM API, as illustrated here:

```

include <X.xh>
void OffsetExample(X *x)
{
    ((somTD_X_foo)(somClassResolve(_X, XClassData.foo)))
        (x, 12.34, TRUE);
}

```

The difference between somResolve and somClassResolve is that somResolve uses an object's method table for resolution, and somClassResolve uses a class's instance method table. In the above example, the macro _X is defined by usage bindings as a pointer to the X class object. No matter what x's class, the method procedure defined for X will be used to execute the foo method.

Name-Lookup Resolution

Offset method resolution is normally sufficient, but in some cases the more flexible name-lookup resolution can be useful. It allows creation of generic code in ways that would be impossible if restricted to offset resolution. Name-lookup resolution is similar to the techniques employed by Objective-C and Smalltalk. It is slower than offset resolution, roughly two to three times as slow as an ordinary procedure call, but it is more flexible. In particular, name-lookup resolution can be used when:

- The name of the method to be invoked is not known until run time
- The name of the class introducing the method isn't known until run time

Name-lookup method resolution is accessed in SOM through the method somLookupMethod which is available on all SOM classes. To use name-lookup resolution for an object, invoke somLookupMethod on the class of the object that has the method to be applied. Use SomGetClass on an object to get the class object. somLookupMethod performs a name-based search of data structures maintained by the target class and its ancestors in order to find a method's method token and then uses offset resolution to select the appropriate method procedure from the class's instance method table.

Offset and name-lookup resolution achieve the same net effect. They both ultimately select a method procedure stored within an instance method table. They just achieve it via different search mechanisms. Offset resolution is faster, because it is based on using a method token stored in a statically known location. Name-lookup resolution is more flexible. The SOM API provides a simple procedure call for performing name-lookup resolution, somResolveByName. It is implemented by invoking somLookupMethod on the class of an indicated object. Use of this procedure is illustrated below.

Name-lookup resolution is appropriate when a programmer knows that an object will respond to a method of some given name but does not know enough about the type of the object to use offset method resolution. How can this happen? It normally happens when a programmer wants to write some code using

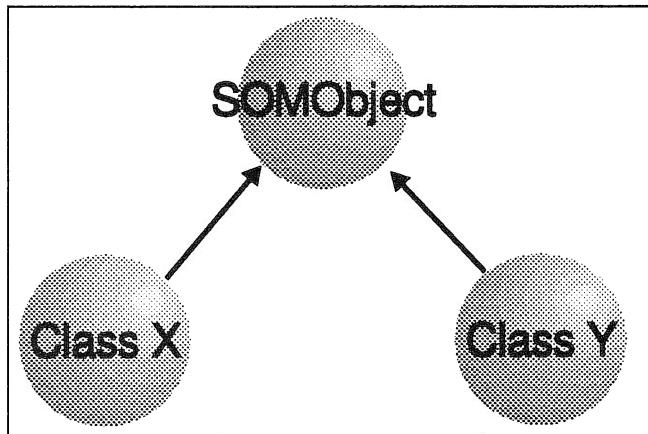


Figure 6.1. Hierarchy for Text Example

methods of the same name and signature that are applicable to different classes of objects, and yet these classes have no common ancestor that introduces the method. This can easily occur in single inheritance systems, and can also happen in multiple-inheritance systems when class hierarchies designed by different people are brought together for clients' use.

If multiple inheritance is available, it is always possible to create a common class ancestor into which methods of this kind can be migrated. Such a refactoring often implements a good generalization that unifies common features of two previously unrelated class hierarchies. Note, however, that this step is most practical when class hierarchies can be *refactored* without requiring the redefinition or recompilation of subclasses or applications that use offset-resolution to invoke these methods. SOM allows this.

But refactoring requires redefining the classes that originally introduced the common methods so they can instead inherit the method from the new unifying class. A client programmer who simply wants to create an application may not control the implementations of the classes. Although there are ways around this problem in SOM, using name-lookup method resolution in this situation seems the best approach for programmers who simply want to make use of available classes rather than defining new ones.

For example, assume the existence of two different SOM classes. Class X and ClassY have only one common ancestor in SOMObject. They introduce two different methods, both named reduce. Each of these methods accepts a string as an argument and returns a long:

```

// IDL
#include <somobj.idl>
ClassX : SOMObject { long reduce(in string arg);  };
ClassY : SOMObject { long reduce(in string arg);  };
  
```

Figure 6.1 illustrates the class hierarchy for this example.

The following is a C++ function. It uses name-lookup resolution to invoke the `reduce` method on its argument, which is either of `ClassX` or `ClassY`. This code includes `ClassX`'s usage bindings because these contain a `typedef` for the `reduce` method procedure.

```
// use classX's method proc typedef
#include <classX.xh>
// This procedure can be invoked on a classX or
// classY target
// It generalizes these class's reduce methods to
// implement
// a single, generic function.
long generic_reduce(SOMObject *target, string arg)
{
    somTD_classX_reduce reduceProc =
        (somTD_classX_reduce)
            somResolveByName(target, "reduce");
    return reduceProc(target, arg);
}
```

Dispatch Resolution

While it is more flexible than offset resolution, name-lookup resolution still requires the signature of a method to be known at the time the program is written. In contrast, any method on any SOM object can be called through *dispatch resolution*. Dispatch resolution is the slowest but most flexible of the three method resolution techniques. It is accessed through the method `somDispatch` which is available on all SOM objects.

The method to be invoked, its target, and its expected arguments need to be known at the time an invocation of `somDispatch` is written. You can write an interactive OOP interpreter calling arbitrary methods using dispatch resolution. Theoretically, it would be possible to construct any possible SOM object system through interactive use of such a calculator. The following example illustrates the basic idea by using dispatch resolution to provide a calculator program.

```
// A Simple OOP Calculator for SOM Objects - interp.cpp
// To create executable, do:
// icc interp.cpp %SOMBASE%\lib\somtk.lib
#include <somcls.xh>
#include <somcm.xh>
#include <stdio.h>
#include <stdarg.h>
//
// ** REGISTERS **
//
//      <reg_spec> := <char><int>
//      e.g., s0, i3, ...
//
```

```
struct reg_specification {    char bank;      int num;      }
result, reg;
//
// register banks
//
SOMObject*      o[10]; /* objects */
somId          i[10]; /* somIds */
long            l[10]; /* integers */
string          s[10]; /* strings */
boolean         b[10]; /* booleans */
// a macro to load a register specification from a command
#define reduce_reg(r) \
{ r.bank = *(cmd_ptr); sscanf(cmd_ptr+1,"%d",&(r.num)); }
//
// **COMMANDS**
//
// <command>     := <reg_spec> = <literal> | <method_call>
// <method_call>  := <method_name>(<reg_spec>*)
//
//       e.g., i3 = SOMClass
//              o1 = somGetClass(o0)
//                     also, see usage example below
//
va_list          args;
char             cmd, *cmd_ptr;
boolean          cmd_is_call;
void             assign_literal(), call_method();
boolean          get_cmd(), push_args();
// define a macro to shift a new token
#define      get_token(s1,s2) cmd_ptr = strtok(s1,s2)
//
// ** MAIN INTERPRETER LOOP **
//
main() {
    args = (va_list) SOMMalloc(1000);
    o[0] = somEnvironmentNew();
    somPrintf("- o0 = "); o[0]->somPrintSelf();
    while (1)
        if (get_cmd())
            if (cmd_is_call) call_method();
            else assign_literal();
        else somPrintf("bad command syntax, try
                        again\n");
}
boolean get_cmd()
{
    somPrintf(":  "); fgets(cmd,100,stdin);
    cmd_is_call = (boolean) (int) strchr(cmd,'(');
    get_token(cmd, " "); /* lhs of assignment */
    reduce_reg(result); /* determine result register */
```

```

get_token(0," "); /* ignore the "=" */
get_token(0," (\n");/* rhs of assignment */
return (int) cmd_ptr;
}

void call_method() // on entry, cmd_ptr points to rhs of a
method call assignment
{
    va_list arg_ptr = args;
    somId mid = somIdFromString(cmd_ptr);
    get_token(0,",");
    // get target object register spec
    reduce_reg(reg);
    SOMObject *receiver = o[reg.num];
    // put target object address on va_list
    va_arg(arg_ptr, SOMObject *) = receiver;
    // put remaining args on va_list
    if (push_args())
        // dispatch with result into result.bank
        switch (result.bank) {
            case 'o':
                receiver->SOMObject_somDispatch(
                    (somToken*)&(o[result.num]),mid,args);
                somPrintf("- o%d = ",result.num);
                o[result.num]->somPrintSelf();
                break;
            case 's':
                receiver->SOMObject_somDispatch(
                    (somToken*)&(s[result.num]),mid,args);
                somPrintf("- s%d = string<%s>\n",
                    result.num,s[result.num]);
                break;
            case 'l':
                receiver->SOMObject_somDispatch(
                    (somToken*)&(l[result.num]),mid,args);
                somPrintf("- l%d = long <%d>\n",
                    result.num,l[result.num]);
                break;
            case 'b':
                receiver->SOMObject_somDispatch(
                    (somToken*)&(b[result.num]),mid,args);
                somPrintf("- b%d = bool<%d>\n",
                    result.num,b[result.num]);
                break;
            case 'v':
                receiver->SOMObject_somDispatch(
                    (somToken*)0,mid,args);
                break;
        }
}

```

```

default:
    somPrintf("method result restricted to"
              " o(bject),s(tring),l(ong),b(ool)"
              " and v(oid) register banks\n");
}
}

boolean push_args()
{
    va_list arg_ptr = args+4;
    get_token(0,",")\n";
    while (cmd_ptr) {
        reduce_reg(reg);
        switch (reg.bank) {
            case 'i':va_arg(arg_ptr, char **) = i[reg.num];
                        break;
            case 'o':va_arg(arg_ptr, SOMObject *) = o[reg.num];
                        break;
            case 's':va_arg(arg_ptr, char *) = s[reg.num];
                        break;
            case 'l':va_arg(arg_ptr, long) = l[reg.num];
                        break;
            case 'b':va_arg(arg_ptr, boolean) = b[reg.num];
                        break;
            default: printf(
                        "arguments restricted to i, s, o, l,"
                        "and b register banks\n");
                        return 0;
        }
        get_token(0,",")\n";
    }
    return 1;
}

void assign_literal()
// At entry, cmd_ptr points to rhs of assignment
{
    switch (result.bank) {
        case 'i':
            i[result.num] = somIdFromString(cmd_ptr);
            somPrintf("- %d = somId <%s>\n",
                      result.num, somStringFromId(i[result.num]));
            break;
        case 's':
            s[result.num]=(char*)SOMMalloc(strlen(cmd_ptr)+1);
            strcpy(s[result.num],cmd_ptr);
            somPrintf("- %d = string<%s>\n",
                      result.num,s[result.num]);
            break;
    }
}

```

```

        case 'l':
            sscanf(cmd_ptr,"%d",&(l[result.num]));
            somPrintf("- l%d = long <%d>\n",
                      result.num, l[result.num]);
            break;
        case 'b':
            sscanf(cmd_ptr,"%d",&(b[result.num]));
            somPrintf("- b%d = boolean <%d>\n",
                      result.num, b[result.num]);
            break;
        default:
            printf("literal assignment restricted to "
                  "i, s, l, and b register banks\n");
    }
}
*****
```

Example Use:

At startup register o0 references the class manager
 Info messages are preceded by "-",
 and the command prompt is ":"
 All commands are register assignments, and
 all method calls contain only registers as arguments.

```
%interp
- o0 = {A SOMClassMgr instance at address 20059598}
: i0 = somFindClass
- i0 = somId <somFindClass>
: b0 = somRespondsTo(o0,i0)
- b0 = boolean <1>
: i0 = SOMObject
- i0 = somId < SOMObject >
: 10 = 0
- 10 = long <0>
: o1 = somFindClass(o0,i0,10,10)
- o1 = {A SOMClass instance at address 200567A8}
: s0 = somGetName(o1)
- s0 = string < SOMObject >
: 11 = somGetNumMethods(o1)
- 11 = long <20>
*****
```

Because dispatch-function resolution is provided by the object method `somDispatch`, you can override this method. This way, a class implementor has a great deal of freedom in determining how methods invoked via `somDispatch` are ultimately resolved. Furthermore, a class designer can actually arrange that all methods invoked on an object, even those invoked using offset resolution, are ultimately routed through `somDispatch`. This is accomplished by placing *redispatch stubs* in the instance method table.

A redispatch stub is a method procedure that accepts a specific set of arguments, puts these on a `va_list`, and then invokes `somDispatch`. By putting redispatch stubs for methods in its instance method table, a class can guarantee that all method invocations, whether by offset, name-lookup, or dispatch reso-

lution, will be handled by `somDispatch`. In practice, a metaclass cooperation framework should be used to automate and localize the code that handles redispach stubs. This is necessary to avoid interference between different metaclasses. Simply by subclassing from a special metaclass, a wide variety of different codes can cooperate in the execution of `somDispatch` without interfering with each other.

Currently, redispach stubs are thunks generated at runtime, based on information registered earlier by the class method `somAddStaticMethod`. Also, code that places redispach stubs in a method table is most naturally associated with the implementation of the class method, `somInitMIClass`. This method implements the inheritance aspect of subclassing. To pursue this, let's look at how subclassing is done in SOM.

Subclassing and Inheritance

Instance methods and instance variables are inherited from the parents of a class. For example, the parent of class "Dog" might be the class "Animal". Hence, the instance methods and variables introduced by "Animal" (such as methods for breathing and eating, or a variable for storing an animal's weight) would also apply to instances of "Dog." As a result, any given dog instance would be able to breathe and eat, and would have a weight. Implementation bindings hide the details associated with the creation of new subclasses in SOM, but it can be useful to understand the overall procedure that is followed when using the SOM API directly.

- 1 . First, a metaclass is chosen. This determines the implementation of the new class object.
- 2 . Next, the method `somNew` is invoked on the metaclass. This creates the new class object.
- 3 . Next, the method `somInitMIClass` is used on the newly-created class object. This method informs the new class object of its parents, the number of new methods that are introduced by the new class, and the size of the instance data structure that is introduced by the class. Based on this information, `somInitMIClass` creates an initial instance method table for the class, copies into it the appropriate portions of the parents' instance method tables to perform inheritance, and determines the storage layout for instances of the new class. Remember that the procedure that implements `somInitMIClass` is determined by the metaclass.
- 4 . Finally, the methods `somAddStaticMethod` and `somOverrideSMethod` are invoked on a class object to add methods newly introduced by the class, and to override inherited methods. The procedures that execute `somAddStaticMethod` and `somOverrideSMethod` methods are also determined by the chosen metaclass.

In practice, metaclasses that override their parent's `somInitMIClass` defer initial setup of method tables and object storage layout to their parent. This means that the method procedure provided for this purpose by `SOMClass` executes first, after which its results may be modified. For this reason, it is important to understand the default inheritance provided by `SOMClass`. A primary issue here involves a well-known ambiguity associated with the use of multiple-inheritance.

When OOP is restricted to single inheritance, there is never any question of which method procedure should be inherited. With only a single parent, there can only be one choice. But when multiple parents are allowed, it is possible that a class will inherit the same method or instance variable from multiple parents. Because SOM is based on graph inheritance, a SOM subclass inherits exactly one implementation of the method or instance variable whenever this happens. The implementation of an instance variable is simply the storage location within an object where it is stored, and the content of instance variables is determined on a per-object basis, by the methods that execute on the object. There is no problem with ambiguity here due to the use of graph inheritance. But the implementation of a method is determined on a per-class basis, by the content of an entry in its instance method table. It is possible for ambiguity to arise with respect to the appropriate method table entry. The following example addresses the question of which method procedure pointer is stored in the instance method table of a new subclass by `SOMClass` when there are multiple alternatives.

Consider the situation in Figure 6.2. Class `W` defines a method `foo`, implemented by method procedure `proc1`. Class `W` has two subclasses, `X`, and `Y`. Class `Y` overrides its inherited implementation of `foo` with procedure `proc2`, but `X` does not override `foo`. Finally, class `Z` is derived by subclassing from both `X` and `Y`. The IDL interface definition for `Z` lists its parents as `X` and `Y`, in this order. As a result, the list of parents passed to the `somInitMIClass` method invoked on `Z` includes `X` and `Y`, again in this order. Here is the corresponding IDL:

```
interface W : SOMObject { void foo(); };
interface X : W { ... };
interface Y : W { implementation { foo: override; } };
interface Z : X, Y { ... };
```

Now, the question is this: which implementation of method `foo` does class `Z` inherit—procedure `proc1` defined by `W` and inherited by `X`, or procedure `proc2`, defined by class `Y`? The answer is determined by the fact that the method procedure defined by `SOMClass` for `somInitMIClass` uses a *left-path precedence rule*. When a method is inherited from multiple parents, the method procedure that is inherited is the one used by the leftmost parent from which the method is inherited. This approach can be viewed as giving priority to the implementations defined by parent classes that are listed first in the IDL class declaration.

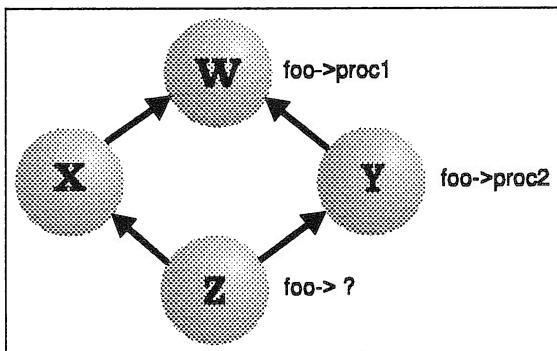


Figure 6.2. Multiple Inheritance Problem

In Figure 6.2, then, class Z inherits its implementation of method `foo` from class X, which is the implementation inherited from W—procedure `proc1` rather than the implementation defined by class Y. If the Z class implementor decides that the implementation for `foo` that is inherited by default is not appropriate, then there are many ways in which this default may be changed. Two possible approaches include: (1) the inherited method can be specially selected using an `override` modifier in Z's IDL implementation section: "`foo: override, select = X`" (this is a special case of the `override` modifier that avoids the need for defining a new method procedure), or (2) an explicit metaclass might be introduced for Z that loads Z's instance method table in whatever way is desired. The second alternative should not be used lightly, but it is a possibility.

The example of Figure 6.2 shows an ambiguity for the inherited implementation of a *single* method of class Z—namely, the method `foo` introduced by W. In other words, every method in SOM is always associated with the class that first introduces it. Two different classes might introduce different methods that have the same name—but these would be considered distinctly different methods by SOM, independently of whether they have the same signature. When an ambiguity results from giving different methods the same name, we consider the method to be *overloaded*.

Figure 6.3 illustrates the kind of ambiguity that can arise in connection with overloaded methods. In this example, class X introduces a method `bar` of type T1, and class Y introduces a different method named `bar` of type T2. Class Z is derived by multiple inheritance from X and Y, so the question here concerns which method `bar` is available on instances of class Z. In SOM, both `bar` methods would be available on a Z instance. Method tables in SOM are organized into groups corresponding to the classes that introduce methods. But this leads to the question of how a SOM user would request one method or the other. The answer is that for offset-resolution, the method token used for resolution identifies the desired method since this token comes from the introducing class's `ClassData` structure. For name-lookup resolution and dispatch resolution, a *method descriptor* can be used. This is an identifier of the form `<className>::<methodName>`.

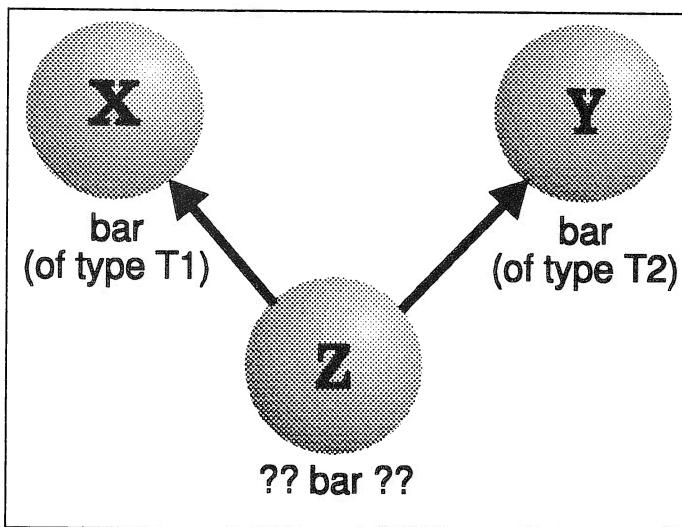


Figure 6.3. Overloaded Methods

However, an interface of such classes as Z in this example cannot be defined using IDL. The CORBA definition of IDL forbids the definition of interfaces in which methods are overloaded as illustrated here. As a result, although classes like Z can be constructed and their instances can be used through the SOM API, these classes cannot be implemented using the SOM language bindings. In other words, this kind of ambiguity is not an issue when using SOM objects whose classes are declared using IDL. We want to focus on classes implemented using language bindings, so issues related to method overloading are not discussed further here.

Explicit Metaclasses

A metaclass is a class whose instances are class objects. Its introduced instance methods and instance variables are therefore the methods and variables of class objects. For this reason, a metaclass is often said to define *class methods*. For example, the metaclass of "Animal" might be the class "M_Animal," which determines the methods that can be invoked on the class "Animal." Examples of class methods involved in subclassing were given in the previous section. These methods are part of the SOM API, because they are introduced by the primitive SOM kernel classes. Also, a metaclass can introduce new class methods and variables.

The distinctions between parent class and metaclass are summarized in Figure 6.4. In this figure, proper objects are illustrated using spheres with no outline, proper classes are shown using solid outlines, and metaclasses are shown using broken outlines. Figure 6.4 shows that a general class, "C", has both parent classes and a single metaclass. The parent class of "C" provide the inherited instance methods that individual instances (objects "O_i") of class "C" can perform. Methods that an instance "O_i" responds to always include:

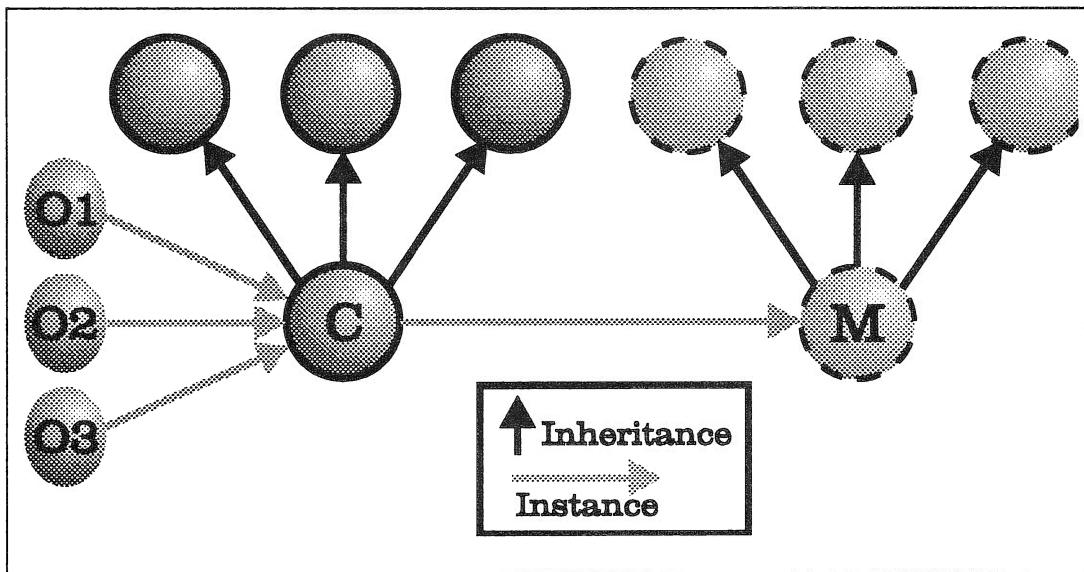


Figure 6.4. Parent Class and Metaclass

- Initializing itself (`somInit`)
- Performing dispatch-function resolution on itself (`somDispatch`),
- Printing its instance variables (`somPrintSelf`)
- Returning its size (`somGetSize`)

These methods are originally introduced by the class `SOMObject`, the root ancestor of all SOM objects.

Just as `C` defines the methods that its instances can perform, the metaclass `M` in Figure 6.4 defines the methods that its instance `C` can perform. `SOMObject` is the parent of `SOMClass`, and all metaclasses are ultimately derived from `SOMClass`. This means the class methods supported by metaclass `M` include those mentioned above, available on all objects. The methods supported by "`M`" also include those that allow "`C`" to :

- Inherit its parents' instance methods and instance variables (`somInitMIClass`)
- Tell its own name (`somGetName`)
- Create new instances (`somNew`)
- Tell how many instance methods it supports (`somGetNumMethods`).

These methods are originally introduced by the class `SOMClass`. Additional methods and instance variables introduced by `M` might allow `C` to count how many instances it creates. As shown in Figure 6.4, each metaclass has its own inheritance hierarchy through its parent classes that is independent of its instances' inheritance hierarchies.

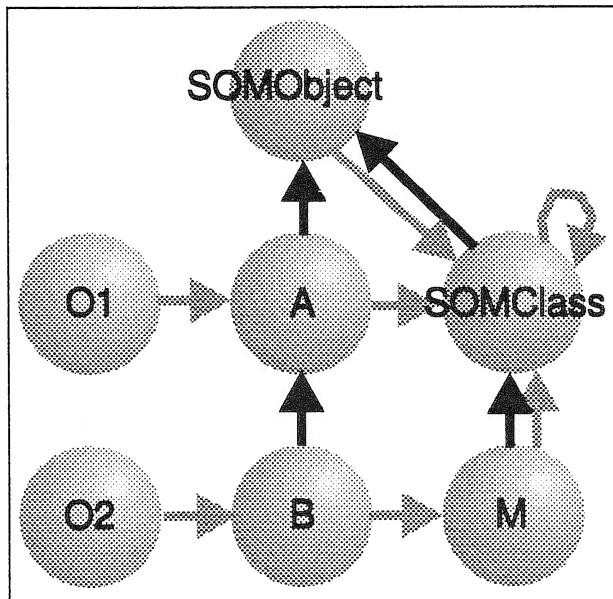


Figure 6.5. Explicit Metaclass Example

To summarize, there is a distinct difference between the notions of parent class and metaclass. Both are related to the fact that a class defines the methods and variables of its instances. The parent provides inherited methods that the class's instances can perform. The metaclass provides class methods that the class itself can perform because the class is an instance of the metaclass.

Because parents and a metaclass are fundamental to all classes in SOM, the implementation section of an IDL class declaration allows indication of a metaclass. When a metaclass is indicated, the class is said to have an *explicit metaclass*. But this does not necessarily mean that the explicit metaclass will actually be used as the class of the declared class. Instead, the actual metaclass may either be the explicit metaclass or some other metaclass derived from the explicit metaclass—perhaps derived dynamically by the SOM runtime.

Leaving derived metaclasses aside for the moment, let's look at a simple example to illustrate the use of explicit metaclasses in IDL. Figure 6.5 shows a sequence of classes defined by subclassing, as would result from the following IDL:

```

module Figure5 {
    interface M : SOMClass { ... };
    interface A : SOMObject { ... };
    interface B : A { ... { implementation { metaclass = M; } };
};
```

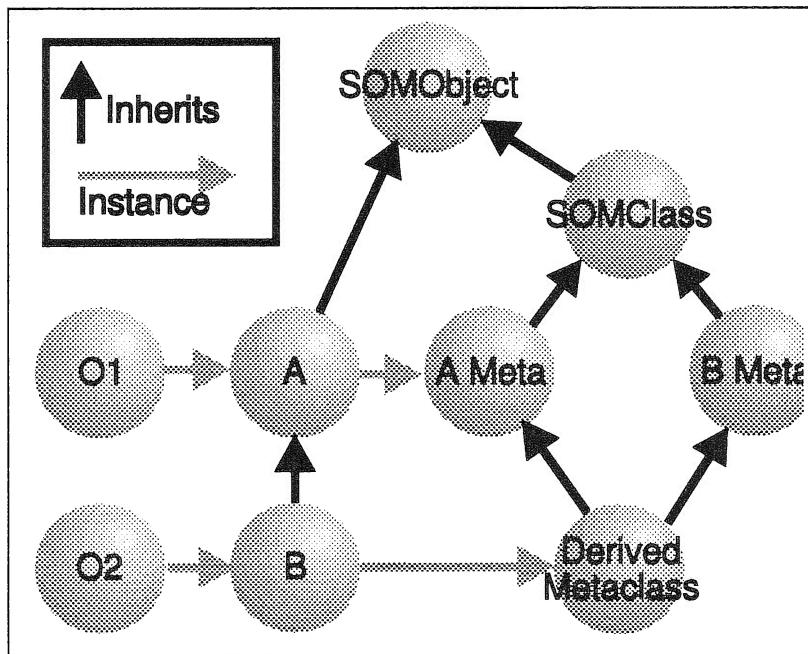


Figure 6.6. Another Metaclass Example

In this example, both SOMObject and class A are instances of SOMClass, whereas class B is an instance of metaclass M, which inherits from SOMClass.

Derived Metaclasses

Four factors are central to use of metaclasses in SOM:

- Classes in SOM are objects, and have their own classes called metaclasses.
- Programmers can define and name new metaclasses, using subclassing.
- A programmer can indicate an explicit metaclass when declaring a new class.
- Finally, the above three capabilities are not allowed to interfere with the fundamental rule of OOP, polymorphism. If method resolution works for the class, then it should work for any subclass.

Surprisingly, SOM is currently the only OOP system that can make this final guarantee while also allowing programmers to explicitly define and use arbitrary metaclasses. This is possible because SOM automatically determines an appropriate metaclass that supports this guarantee, dynamically deriving new metaclasses by subclassing when this is necessary. Consider the situation in Figure 6.6, arising from the following IDL.

```

module figure6 {
    // Declare two arbitrary metaclasses
    interface AMeta : SOMClass { void bar(); };
    interface BMeta: SOMClass { ... };
    // Use AMeta and BMeta while subclassing
    interface A : SOMObject {
        void foo();
        implementation {
            metaclass = AMeta;
        };
    };

    interface B : A {
        implementation {
            metaclass = BMeta;
        };
    };
}

```

Here, class A is an instance of metaclass AMeta, which supports a class method named bar. Now, assume that the method foo introduced by A and inherited by B uses the expression somSelf->somGetClass()->bar(). Then, when foo is invoked on an instance O1 of A, this in turn invokes bar on class A itself.

Now consider what happens when foo is invoked on O2, an instance of the class B. If BMeta were used as the actual class of B, then an invocation of foo on O2 would fail, because metaclass BMeta does not support the bar method on its instances. There is only one way that BMeta could support this specific method—by having AMeta as ancestor. Such is not the case here.

Allowing an invocation of foo on O2 to fail would violate the fundamental principle of OOP. As a result, BMeta cannot be allowed to be the actual class of B. That is, BMeta is not compatible with the requirements placed on B by the fundamental principle of OOP referred to above. This situation is referred to as *metaclass incompatibility*.

SOM will not construct class hierarchies with metaclass incompatibilities. Instead, SOM automatically builds derived metaclasses when this is necessary. The result of doing this is shown in Figure 6.6, where SOM has automatically built the metaclass DerivedMetaclass, and used this as the class of B. This ensures that the invocation of method foo on instances of class B will not fail, and also ensures that, as desired by the class implementor, the class methods provided by BMeta will be available on class B.

There are three important aspects of SOM's approach to derived metaclasses:

- First, the creation of derived metaclasses is integrated with programmer-specified metaclasses. If a programmer-specified metaclass already supports all the class methods and variables needed by a new class, then the programmer-specified metaclass will be used as is.
- Second, if SOM must derive a different metaclass than the one explicitly indicated by the programmer, then this derived metaclass directly inherits from the explicitly indicated metaclass as its first parent. As a result, the method procedures defined by the specified metaclass take precedence over other possibilities.
- Finally, the class methods defined by a derived metaclass use parent method calls to invoke the appropriate initialization methods of its parents. This ensures that class variables are correctly initialized.

As another example of the automatic derivation of metaclasses, consider the following multiple inheritance example in Figure 6.7, resulting from the following IDL.

```
module Figure7 {
    interface Ameta : SOMClass { ... };
    interface BMeta : SOMClass { ... };
    interface A : SOMObject { ...
        implementation {
            metaclass = Ameta;
        };
    };
    interface B : SOMObject { ...
        implementation {
            metaclass = BMeta;
        };
    };
    interface C : A,B { } ;
}
```

In Figure 6.7, class *C* does not have an explicit metaclass declaration in its SOM IDL, yet its parents do. As a result, class *C* requires a derived metaclass. (If you still have trouble following the reasoning behind derived metaclasses, ask yourself the following question: What class should *C* be an instance of? After a bit of reflection, you will conclude that if SOM did not build the derived metaclass you would have to do so yourself.)

SOM encourages the definition and explicit use of named metaclasses. With named metaclasses, programmers can not only affect the behavior of class instances by choosing the parents of classes, but also the classes themselves by choosing metaclasses.

At the same time, SOM is unique because it relieves programmers of the responsibility for avoiding metaclass incompatibility when defining a new class. At first glance, this might seem to be merely a useful convenience. In fact, it is essential, because SOM is predicated on binary compatibility.

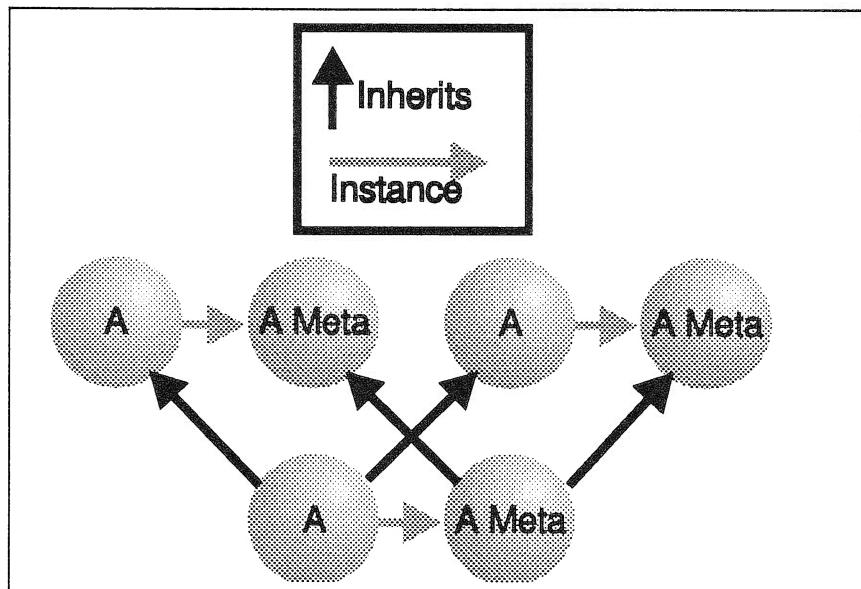


Figure 6.7. Automatic Derivation of Metaclasses

You might know the metaclasses of all ancestor classes of a new subclass, and as a result, be able to explicitly derive an appropriate metaclass for the new class. Nevertheless, SOM must guarantee that this new class will still execute and perform correctly when any of its ancestor class's implementations are changed. And changes may include specifying different parent classes and different metaclasses. Derived metaclasses allow SOM to make this guarantee. A SOM programmer need never worry about the problem of metaclass incompatibility; SOM does this for the programmer. Explicit metaclasses can be used declaratively to "add in" whatever behavior is desired for a new class. SOM automatically handles anything else that is needed to support correct inheritance of class behavior.

Chapter 24 provides a more detailed look at derived metaclasses, and introduces a metaclass framework designed to prevent interference between different metaclasses. You will find many applications for the techniques that are illustrated there.

CONCLUSION

This completes an introduction to the most important aspects of the overall SOM API used by language bindings and advanced SOM programmers. The following chapter provides a contrasting view of the SOM API by comprehensively enumerating its various components.

7

A Guided Tour through the SOM API

INTRODUCTION

The SOM API is the instruction set used by SOM language bindings, DirectTo-SOM (DTS) compilers, and SOM programmers to implement OOP capabilities. Although the language bindings (and certainly DTS compilers) hide the use of this API from programmers, it is important for SOM programmers to have a general feel for how “things are done” in SOM and to know how to find their way around the files of an installed SOM system. In this chapter, we will locate and examine the appropriate top-level files, and talk about what we see. We will not discuss obsolete methods that are still supported for backwards binary compatibility, but are of limited utility. SOM users should refer to the SOMobjects Toolkit Users Guide for more details, but the comments offered here will provide a general orientation.

This chapter provides an overview of the SOM API as it appeared in the general availability release of SOM version 2.0. Future SOM releases will almost certainly shuffle some of these files around, to provide more beneficial or intuitive packaging, and will introduce new methods that support enhanced capabilities, but the general shape of things should remain as described here. When identifying files, we assume SOM was installed in the directory, \SOMObjects.

NON-OBJECT INCLUDE FILES

The fundamental basis for SOM as a software system is its implementation language, C. As a result, its types are built out of C types. A small number of files contain the basic type definitions. Within the current SOM installation, these files are accumulated into an overall include file for C named som.h. For C++ there is the file som.xh. Generally, som.xh includes .h files where non-object types are involved which differ from som.h, because it provides forward declarations of SOMObject, SOMClass, and SOMClassMgr as C++ classes. This is done to support the C++ language bindings, which view all SOM classes as C++ classes. Because we want to stay as close to the implementation as possible in this chapter, we'll look at the C include files.

The four important files are somtypes.h, somcdev.h, somapi.h, and (with the advent of OMG and CORBA) somcorba.h. Only somtypes.h and somapi.h are essential for understanding how SOM works. All these files are in \SOMObjects\include. From somtypes.h, we learn that there is a structure for method tables in SOM, and the first word of any SOM object points to a method table with this structure:

```
/* - Public Object Instance Structure */
struct somMethodTabStruct;
typedef struct SOMAny_struct {
    struct somMethodTabStruct *mtab;
    integer1 body[1];
} SOMAny;
```

We also learn that SOMObject, SOMClass, and SOMClassMgr are objects.

From somapi.h, we learn the signatures of all the SOM API functions — that portion of SOM's API that is provided by C functions rather than SOM object method calls. These functions include

- somEnvironmentNew

This brings the four kernel objects into existence: SOMObject, SOMClass, SOMClassMgr, and SOMClassMgrObject. When this happens, the external data locations SOM API provides for classes (their ClassData and CClassData structures) are loaded with various values, and the three class objects are registered with the class manager (the forth object), thereby creating a valid initial SOM environment. The three class objects are then accessible either through their external data structures (the ClassData structures) or through the class manager. The class manager itself is returned as the result of the somEnvironmentNew function call and is then available from the external location, SOMClassMgrObject).

The following functions support somIds:

- somCheckId
- somRegisterId
- somIdFromString
- somStringFromId
- somCompareIds
- somTotalRegIds
- somSetExpectedIds
- somUniqueKey
- somBeginPersistentIds
- somEndPersistentIds

To make name comparison fast in SOM, somIds are used to map all identifiers having the same (case insensitive) spelling to a single number (actually, the address of a structure whose layout is not made public by the SOM API). This number is a somId. Given a traditional null-terminated C character string, for example, one can get back the somId for this string. Likewise, given a somId, one can get back the first string that was given this Id.

The following functions provide support for method resolution.

- somResolve
- somClassResolve
- somParentNumResolve
- somResolveByName

The first three are efficient and are based on method tokens, and the fourth is based on the name of the method. Resolution is always done with respect to some method table. The possibilities are: use an object's method table to resolve a method that is presumably going to be applied to the object (somResolve and somResolveByName); use a class's instance method table (somClassResolve); or use a parent method table list that was returned by a call to somGetPClsMtabs (somParentNumResolve).

There is an additional resolution function.

- somDataResolve

This is the data resolution function that is used to return a pointer to any class's introduced instance data in a object. Of course, to use the result that is returned, you need to know what is the structure of that class's introduced data. In SOM version 2.0, the SOM API provides no way for you to know this (you must be the class implementor, or somehow have access to the implementation bindings for the class). Future versions of the SOM API may provide a binary interface to data supported by SOM IDL.

Another interesting function is

- somIsObject.

This function tests whether an address is the address of a SOM object. This is designed to be failsafe in the sense that it will never generate a runtime error—perhaps as a result of accessing unmapped virtual memory. If true is returned, then its argument points to a valid SOM object of some specific class — an object on which methods may be invoked; otherwise it doesn't.

Next, the SOM API also provides definitions of the data structures that are passed to various methods. For example,

- `Typedefs` for constructing the structures needed to tell SOM enough about the signature of a method to allow SOM to automatically generate a redispach stub for the method should one be needed. This section of `somapi.h` is not for novices. Only the SOM Toolkit language bindings and DTS compilers need to know this stuff. These structures are used when invoking `somAddStaticMethod` on class objects.
- `somApply`.

This is a generic method procedure application function. The information passed to includes an object, and a method procedure (the result of doing method resolution). Generally, it is used as follows: Redispach stubs are placed in instance method tables to allow arbitrary code to be included in the act of dispatching a SOM method call. A redispach stub's sole purpose in life is to invoke `somDispatch`, to which metaclasses can tie arbitrary code. And `somDispatch` invokes `somApply`. `somApply` uses the method data registered via `somAddStaticMethod` call to dynamically create the appropriate stack frame for invoking the indicated method procedure on the indicated object. See chapter 24 on metaclass programming for further information.

Another API function supported by a complex assortment of data structures is

- `somBuildClass`.

This function completely automates and encapsulates the construction of a new SOM class. No SOM programmer would ever find this easy to use. Furthermore, the data structures passed to it have evolved through a number of successive layout versions. Luckily, SOM programmers aren't supposed to use it. But language bindings and DTS compilers do use it, and this is important. For example, in the GA release of SOM 2.0, this is the only access to SOM-derived metaclasses.

Finally, the SOM API provides various pointers to functions that can be loaded by users to select support for particular operating system level primitives. These can be viewed as

- *Portability functions* include system level, repeatable character output, and memory allocation primitives.

The remainder of the SOM API is the methods that can be applied to objects in general, to classes in general, and to the class manager object in particular. For this information, we go to the IDL for these classes, also found within \SOMObjects\include.

OBJECT METHODS

- void somFree();

In \SOMObjects\include\somobj.idl, the first method we see is somFree. This deletes and object. So the first thing we find out about an object is that we can make it go away. If only it were this simple! One can safely invoke somFree only on an object whose memory was allocated by the class of the object. This is because somFree calls the class-specific deallocation routine. But SOM objects do not need to be created in memory allocated by their class — this is just one of the possibilities, a convenient default used by the SOMobjects Toolkit C and C++ language bindings. Normally, class designers should not assume that somFree will always be called. For example, users of the C bindings generally call somFree, but users of the C++ bindings may not.

Next there are two methods used for initializing and uninitializing objects.

- void somInit();
- void somUninit();

The purpose of somUninit is to free resources taken up by an object before the object itself is freed, and it should always be called, one way or another, before an object is deleted. For example, users of the Toolkit C bindings will generally use somFree on objects constructed with the bindings, and somFree always calls somUninit before invoking the class's memory deallocation method. The C++ bindings make sure somUninit is called when C++ delete is used.

The purpose of somInit is to place the instance data of an object into some useful, default state. Chapter 23 on "Initializers and Destructors in SOM" is germane to somInit and somUninit.

The next methods are used to provide a SOM programmer with runtime information about an object.

- SOMClass somGetClass();
- string somGetClassName();
- long somGetSize();
- boolean somIsA(in SOMClass aClassObj);
- boolean somIsInstanceOf(in SOMClass
 aClassObj);
- boolean somRespondsTo(in somId mId);

The last three of these methods are useful for checking the derivation of an object's class, or determining whether a particular method can be invoked on an object. With just `somGetClass` and the methods available on classes described in the next section, the rest of these methods are not really necessary. It is possible that they shouldn't be methods at all. On the other hand, they allow special handling of class-related functionality when performed through an object. DSOM makes use of this in order to provide "invisible" dynamic proxies for SOM remote SOM objects.

The following method is used to allow class designers to tie into the method dispatch process:

```
■ boolean somDispatch(
    out somToken retValue,
    in somId methodId,
    in va_list ap);
```

If there were a single most important method in the SOM API, this could be it. Users don't normally call this method directly, although this is certainly possible, as illustrated by the OOP calculator example in the previous chapter. The reason this method is so important is that it provides the basis for interpretive, dynamic execution of method calls. By placing redispach stubs in its instance method table, a class can guarantee that all methods resolved through this method table will end up in `somDispatch`, and by overriding `somDispatch`, the class can provide special behavior for all, or a selected subset of the methods to which an object responds. See Chapter 24 on "Metaclass Programming in SOM" for an introduction to the capabilities this method enables.

```
■ boolean somClassDispatch(
    in SOMClass clsObj,
    out somToken retValue,
    in somId methodId,
    in va_list ap);
```

The purpose of `somClassDispatch` is to provide a casted dispatch. In other words, the object on which this method is invoked is treated as if it were a instance of the argument class, `clsObj`. There is no special interaction of this method with redispach stubs. Redispach stubs don't call this method.

The following methods provide general support for allowing an object to describe itself.

```
■ SOMObject somPrintSelf();
■ void somDumpSelf(in long level);
■ void somDumpSelfInt(in long level);
```

These all have slightly different purposes, however. `somPrintSelf` is intended to print a short, specific description of the object. A class designer should override this method but should probably not make any parent-method calls. This is because the class of an object normally knows best what are the most

specific, identifying features of the object. The next method, `somDumpSelf`, is intended to be invoked by a user to print a complete, detailed listing of all the instance variables of the object. But, `somDumpSelf` should *not* be overridden by a class implementor. Instead, the method `somDumpSelfInt` (for intermediate) should be overridden by each class, to print its introduced instance variables. What `somDumpSelf` does is to invoke `somDumpSelfInt` for each ancestor of the object's class. This protocol provides a simple example of a situation in which a class implementor is given the opportunity of contributing to a more global context, by overriding an inherited method (in this case, `somDumpSelfInt`) that is specially invoked within a larger context. This kind of arrangement is unique to OOP, and is very powerful when used at a system level. However, it relies on each class implementor to "do his part," and there is normally no way of guaranteeing that this occurs. As a result, the usefulness of the output from `somDumpSelf` may vary, depending on how well the class implementors have fulfilled their responsibilities for supporting it.

So, if you implement a SOM class, you should generally override `somPrintSelf` and `somDumpSelfInt`. You should also, of course, override `somInit` whenever you introduce instance variables that should be initialized before the object is used by "clients." Finally, you should override `somUninit` when your introduced instance variables include pointers to dynamically allocated storage that should be freed before the object is freed. These four methods are especially important parts of the overall functioning of a SOM system.

CLASS METHODS

In `\SOMObjects\include\somclz.idl`, we find the methods that are available on all SOM classes. For example

- `string somAllocate(in long size);`
- `void somDeallocate(in string memptr);`

Classes create objects, and to support this, they can allocate and deallocate memory. By default, these methods just use the replacable memory allocation/deallocation functions provided by the SOM API, but a clever metaclass programmer can override these, or any of the other class methods described below. In general, whenever one considers overriding class methods, one must be very careful. Read the chapter on "Programming Metaclasses in SOM" for more information on this topic.

- `SOMObject somNew();`
- `SOMObject somNewNoInit();`
- `SOMObject somRenew(in void *obj);`
- `SOMObject somRenewNoInit(in void *obj);`
- `SOMObject somRenewNoZero(in void *obj);`
- `SOMObject somRenewNoInitNoZero(in void *obj);`

The above methods all provide various combinations of activities appropriate to the creation of a *new* object. New is the common word in all of their method names. For example, it may or may not be necessary to allocate new memory to hold a new SOM object. In the case of DTS compilers, for example, the object may be placed on the execution stack instead of in heap storage. This provides the first, top level partitioning for these methods. The methods with the word New in them allocate storage, and the methods with the word Renew in them don't. The remaining distinctions between these methods concern whether or not object memory is zeroed out, and whether the default initialization method somInit is executed. Read the chapter on *Initializers and Destructors in SOM* to learn more about these methods and how they should be treated by class designers.

You may wonder about the behavior of a new object created by a class. The following methods are used to determine the answer to that question. Normally, somBuildClass is the function that makes these calls in the process of building a class, so class programmers using the Toolkit language bindings or a DTS compiler do not need to be concerned with these methods. On the other hand, metaclass programmers need to at least understand how these methods fit into the general scheme of things.

```

void somInitMIClass(
    in long inherit_vars,
    in string className,
    in SOMClassSequence parentClasses,
    in long dataSize,
    in long dataAlignment,
    in long maxStaticMethods,
    in long majorVersion,
    in long minorVersion);

somMToken somAddStaticMethod(
    in somId methodId,
    in somId methodDescriptor,
    in somMethodPtr method,
    in somMethodPtr redispachStub,
    in somMethodPtr applyStub);

void somAddDynamicMethod(
    in somId methodId,
    in somId methodDescriptor,
    in somMethodPtr method,
    in somMethodPtr applyStub);

void somOverrideSMethod(
    in somId methodId,
    in somMethodPtr method);

```

As indicated above, these methods are the ones provided by the SOM API for determining the behavior of objects. The first of these, `somInitMIClass`, does inheritance. This is the method used to tell a class who its parents are, so the class can create an initial instance method table. Then it copies to this new method table the instance method table contents of its parent classes. Once `somInitMIClass` has been used to do these things, new methods introduced by the class are added using `somAddStaticMethod` and `somAddDynamicMethod`. and, where the class overrides inherited methods, `somOverrideSMethod` is used to replace the initial content of the method table with a pointer to a new method procedure.

Static methods are those that have an entry in the method table, as a result of having been *declared* using IDL. In contrast, *dynamic methods*, by their very nature, are not known of statically; they are not resolved through a method table. They do provide the ability for a class to dynamically provide new instance methods that are not declared in IDL. There are a number of aspects about these new methods that restrict their applicability to special situations. First, because dynamic methods have no method table entries, they must be invoked dynamically, using their name via `somDispatch`, for example. Second, methods generally take arguments, and you have to ask yourself how a programmer would know what these arguments are, since dynamic methods don't have declarations. Only in very carefully arranged situations, supported by special documentation, does it seem that dynamic methods would be useful to a SOM class designer.

Such situations might occur when adapting multiple class libraries into an application, but system-level extension by subclassing is also possible through use of `somSubstituteClass` (described with the class manager methods), which effectively allows adding new static methods to an existing class without replacing its DLL.

There is another use for `somAddDynamicMethod` in older SOM version 1.0 code, however, a "hackers" trick. In version 1.0 of SOM, the `somOverrideSMethod` was only useful for overriding inherited methods, but, luckily, `somAddDynamicMethod` behaves as an override when a static method of the same name is already known to the class (inherited or newly introduced). Thus, `somAddDynamicMethod` was the "preferred" way for a metaclass programmer to change a class's instance method table content for non-inherited methods (it is the only way to write code that does this which runs on both version 1.0 and 2.0). The requirement to "override" non-inherited methods is not a usual circumstance, but it does arise in advanced SOM programming (especially in the absence of support provided by the metaclass cooperation framework discussed in "Metaclass Programming in SOM").

The following method interacts with the class manager object.

```
void somClassReady();
```

It is clear that creating a useful class is an incremental process that takes place over a period of time that includes a number of method calls. The `somClassReady` method call provides a way of saying when a class is ready to be used by clients to create new objects and for subclassing.

The remaining class methods are secondary to the primary function classes, object creation. These methods provide information generally of interest to code that deals with class objects—for example, code written by metaclass programmers. Understanding the model behind these methods is useful.

- `somMethodProc *somGetRdStub(in somId methodId);`
- `void somOverrideMtab();`

A redispach stub is a tiny piece of code designed to replace the procedure pointer for a specific method in a method table, so that when the method is executed on an object, the redispach stub gains control. When it gains control, the redispach stub asks the object to execute `somDispatch` for the given `methodId`. The result can vary based on dynamic runtime values, such as the state of class variables, depending on how SOM Metaclass programmers have coded `somDispatch`. Read the chapter on “Metaclass Programming in SOM” for more information on this topic. The method `somOverrideMTab` is used to place redispach stubs in a class’s instance method table. It places redispach stubs in every method’s entry but one. Can you guess which one? The entry that is left unchanged belongs to `somDispatch`. After all, placing redispach stubs in a class’s instance method table routes all method calls through `somDispatch`, so that entry better not have a redispach stub in it, or a loop in method dispatch would be created. Users of the metaclass cooperation framework discussed in Chapter 24 subclass from `SOMMCooperativeRedispached` if this method is used.

- `somClassDataStructure *somGetClassData();`
- `void somSetClassData(in somClassDataStructure cds);`

In SOM, there are two ways to gain access to a class once it has been registered as being ready for use (i.e., after `somClassReady` has been invoked on the class). One way is to ask the class manager object for a pointer to the class. The other way is to access a specific external data structure associated with the class. This data structure is called the `ClassData` structure, and classes in SOM generally define and provide external access to this structure. This structure is the basis for offset method resolution, and the first word of the structure points to the corresponding class object. The two methods provide for registration and access to the address of this structure.

- `long somGetInstancePartSize();`
- `long somGetInstanceSize();`

Each ancestor of a class contributes a certain size to the overall size of an object, as necessary to hold the instance variables introduced by the class. The method `somGetInstancePartSize` returns the size of the instance data introduced by a class; the method `somGetInstanceSize` returns the total size of a class instance.

```

■ somMToken somGetMethodToken(in somId methodId);
■ somDToken somGetInstanceToken();
■ somDToken somGetMemberToken(
    in long memberOffset,
    in somDToken instanceToken);
■ boolean somGetMethodData(
    in somId method,
    out somMethodData md);

```

SOM classes encapsulate the layout of method procedure groups within method tables and the layout of instance variable groups in objects. This is done by providing “tokens” for identifying a specific method or instance variable. Thus, for example a method token identifies a specific method, introduced by a specific class.

When a method token is resolved against an object’s method table using any of the method resolution functions, the result is the method procedure that supports the identified method on that class of object. Likewise, a data token identifies a specific instance variable, introduced by a specific class. When a data token is resolved against a particular object, the result is the address of the instance variable within the object.

The method `somGetMethodToken` can be applied to any class that supports a given method (i.e., to the class that initially introduces the method, and to all classes descendant from the introducing class). The method token for the indicated method is returned if the class supports the method, otherwise a NULL is returned. When the method `somGetInstanceToken` is invoked on a class, the result is a data token for the instance data introduced by that class. The method `somGetMemberToken` is used to produce a data token for introduced data at a given offset from the beginning of the class’s data. The method `somGetMethodData` loads a structure whose address is passed by its caller with information about the indicated method — this includes the method’s method token, for example. (The method returns a FALSE in the case that class doesn’t support the method.) The `somMethodData` structure is defined in `\SOMObjects\include\somapi.h`:

```

typedef struct somMethodDataStruct {
    somId id;
    uinteger4 type; /* 0=static, 1=dynamic */
    somId descriptor; /* for use with IR interfaces */
    somMToken mToken; /* NULL for dynamic methods */
    somMethodPtr method; /* via resolution context */
    somSharedMethodData *shared;
} somMethodData, *somMethodDataPtr;

```

The method component of the `somMethodData` structure returned by `somGetMethodData` contains the content of the class’s instance method table for the identified method. The shared area contains other method information not currently made public. The shared area should not be changed by a SOM programmer.

The following methods are all basically convenience methods for accessing method pointers from a class, given a method id. They use somGetMethodData, and then act accordingly. The method somLookupMethod seems representative of the overall concept represented by these methods. It returns a method pointer for the indicated method, or NULL if the target class doesn't support the method.

- somMethodPtr somLookupMethod(in somId methodId);
- boolean somFindMethod(in somId methodId,
 out somMethodPtr m);
- boolean somFindMethodOk(in somId methodId,
 out somMethodPtr m);
- somMethodPtr somFindSMethod(in somId methodId);
- somMethodPtr somFindSMethodOk(in somId methodId);

The following methods support a model in which each class numbers the methods it supports arbitrarily, starting with 0. In general, different classes will give the same method a different index. You can use these methods to iterate through the methods supported by a class without knowing any of the methods' names.

- long somGetMethodIndex(in somId id);
- long somGetNumMethods();
- long somGetNumStaticMethods();
- boolean somGetNthMethodData(in long n,
 out somMethodData md);
- somId somGetNthMethodInfo(in long n,
 in somId *descriptor);

The last of the above methods deals with method descriptors. In SOM version 2.0, these have been integrated with use of the CORBA specified Interface Repository supported by instances of the SOM Repository class. Descriptors are identifiers that are used to access information about methods from an Interface Repository.

- somId somGetMethodDescriptor(in somId methodId);
- boolean somSetMethodDescriptor(
 in somId methodId,
 in somId descriptor);

Finally, the following methods all provide different kinds of information about a class. Their names should be self-explanatory.

```
■ string somGetName();
■ SOMClassSequence somGetParents();
■ somMethodTabs somGetPClsMtabs();
■ void somGetVersionNumbers (
    out long majorVersion,
    out long minorVersion);
■ boolean somCheckVersion(
    in long majorVersion,
    in long minorVersion);
■ boolean somDescendedFrom(in SOMClass
    aClassObj);
■ boolean somSupportsMethod(in somId mId);
■ readonly attribute somOffsets somInstanceDataOffsets;
```

The method `somGetPClsMtabs` deserves special comment. The value returned by this method call provides a binary interface to information that would otherwise be fairly expensive to compute at runtime. It is needed to support operations that occur frequently. Initially, in SOM 1.0, the result of this method call was simply used to support parent method calls — the result of the method call is stored in the `parentMTab` component of a class's `CClassData` external data structure. Thus, the name of the method. But, the initial SOM 1.0 functionality for this data structure (that of supporting parent method calls to a single parent class) has been generalized in SOM 2.0 and may continue to evolve in future SOM versions. In SOM 2.0, it is most appropriate to view the result of this method call simply as a pointer to an opaque structure, statically accessed through a class's `CClassData` structure. Future versions of SOM may open up this API to some extent, providing SOM users with the ability to statically access certain pre-computed information otherwise available only through method calls.

CLASS MANAGER METHODS

This final section completes our tour of the SOM API by discussing the methods that are available on the SOM class manager object, an object created by `somEnvironmentNew`. As mentioned earlier, `somEnvironmentNew` places a pointer to this object in the external data local, `SOMClassMgrObject`.

The SOM class manager has two different kind of clients: those that simply come to the class manager for access to a given class (presumably for either subclassing or instance creation), and those that want to influence or be involved in the management of classes at some level (if only to register the existence of a class). The first category of use basically includes the following two methods.

```

■ SOMClass somFindClass(
    in somId classId,
    in long majorVersion,
    in long minorVersion);
■ SOMClass somFindClsInFile(
    in somId classId,
    in long majorVersion,
    in long minorVersion,
    in string file);

```

These methods return a pointer to a class object. In the process, it may be necessary to locate a class implementation within a DLL and load its containing file. On the other hand, the requested class may already have been registered, possibly as a result of having been found by some earlier call. There are a number of ways to influence how `somFindClass` will search for a DLL when the class is not yet registered. The following method, for example, is designed to be overridden by a subclass, following which `somMergeInto` may be used to replace the class manager object with an instance of the subclass.

```

string somLocateClassFile(in somId classId,
                           in long majorVersion,
                           in long minorVersion);

```

The default implementation will look up the class name in the Interface Repository, if one is available, to determine the DLL or C file name. If this information is not available, the class name itself is returned as the file name. Subclasses may use version number info to assist in deriving the file name.

```

■ void somMergeInto(in SOMObject targetObj);

```

This method transfers the `SOMClassMgr` registry information from the receiver to `targetObj`. It is required to be an instance of `SOMClassMgr` or one of its subclasses. At the completion of this operation, the `<targetObj>` should be able to function as a replacement for the receiver. At the end of the operation the receiver object is freed. Subclasses that override this method should similarly transfer their sections of the object and pass this method to their parent as the final step. If the receiving object is the distinguished instance pointed to from the global variable `SOMClassMgrObject`, `SOMClassMgrObject` is then reassigned to point to `targetObj`.

Another way that client code can influence the management of classes is provided by the following method:

```

■ long somSubstituteClass(
    in string origClassName,
    in string newClassName);

```

SOM may be viewed as an evolving, runtime environment that maps names to class objects. In this environment, there are two ways of accessing a class object given its name. If the name is known statically, a programmer may be able to link to an external data location statically associated with the class. Otherwise, one of the `somFindClass` methods can be used. The `somSubstituteClass` method is used to change the mapping of names to classes. Inheritance is used to guarantee that the new class is always a subclass of the original class. This method is intended for use at an overall system level, and provides the ability to enhance previously built DLL's containing classes whose names are used by and built into client binaries by subclassing followed by substitution.

Finally, the following method is provided to enable management of DLL files that contain multiple class implementations.

- `SOMClassArray somGetRelatedClasses(in SOMClass classObj);`

This method returns an array of class objects that were all registered during the dynamic loading of a DLL file. These classes define an *affinity group*. Any class is a member of at most one affinity group. The affinity group returned by this call is the one containing the class identified by `classObj`. The first element in the array is the class that caused the group to be loaded, or the special value -1, which means that the `SOMClassMgr` is currently in the process of unregistering and deleting the affinity group. Only `SOMClassMgr` subclasses would ever see this value. The remainder of the array, elements one through `n`, consists of pointers to class objects ordered in reverse chronological order to the way they were originally registered. If the supplied class was not dynamically loaded, it is not a member of any affinity group, and `NULL` is returned.

The remaining methods available on the SOM class manager provide lower-level functionality associated with the management of classes.

- `void somRegisterClass(in SOMClass classObj);`
- `long somUnregisterClass(in SOMClass classObj);`
- `long somUnloadClassFile(in SOMClass classObj);`
- `string somGetInitFunction();`
- `attribute Repository somInterfaceRepository;`
- `readonly attribute sequence<SOMClass> somRegisteredClasses;`

The two attributes above are the Interface Repository used by the class manager, and the currently registered classes.

CONCLUSION

This concludes a top-level but comprehensive view of the SOM API, those functions and methods provided by the SOM kernel. This abstract machine provides an API for supporting traditional models of OOP. This default machine may be dynamically enhanced and adapted in many ways to provide special functionality. As this chapter hopefully demonstrates, SOM is not so monstrous or imposing that it takes too very long to understand its overall behavior. Further details concerning advanced use of the SOM API are provided in the closing chapters.

8

The Workplace Shell

INTRODUCTION

The *Workplace Shell* (WPS) combines the power of SOM and Presentation Manager to provide a true object-oriented user interface. The WPS is easily the most visible element of OS/2. Applications, files, directories, printers, and other system entities are represented as icons, and they can be directly manipulated. Programmers that use the Workplace Shell can get some nice behaviors with minimal effort. Programs that exploit the WPS will fit much more nicely into OS/2 than those that do not, and the programs that do will almost definitely sell better and be more popular with the users. With all of this in mind, you can see that WPS programming is an important skill for the OS/2 programmer.

Most of the OS/2 interfaces we have talked about so far are APIs, or application programming interfaces where libraries of functions are grouped together and stored in a DLL. Usage of these APIs involves simple function calls. The WPS is a SOM object-oriented *framework*, meaning a collection of related classes. The framework is used by creating instances of these classes, specializing them through subclassing, and invoking methods to access the behaviors that we want.

WPS: AN OO USER INTERFACE

The WPS is an OO user interface. Simply put, an OO user interface is one that is broken down into logical objects. These objects can encapsulate behaviors and data. Typically, we access the behaviors through pop up menus and direct manipulation. To delete an object, drag it to the shredder and drop it. To print an object, drop it on the printer. To edit a file, drop it on an editor program icon.

OO user interfaces tend to model the world much more closely than previous user interfaces. We have desk tops for your work area, folders instead of directories, and the like. We also have objects on the screen that can be used to configure the system, such as the mouse object. Apple has achieved significant success with OO user interfaces.

Characteristics of OO User Interfaces

OO user interfaces are based on objects rather than applications. Think about the Workplace Shell. If you are using it right, you think first in terms of your data. You have documents, spread sheet files, and program objects. We print, delete, and edit objects by directly manipulating them. We move them between directories by dragging from one folder to the next.

With the Workplace Shell, we can create instances of classes. The available classes are represented in the templates folder. To create a folder, drag the folder template to the desktop. Since the templates are specialized, their default behavior is to change the drag/drop operation to create an instance of the appropriate type. So WPS objects have additional data, such as type. Their associated object type, or class, is stored within the file's extended attribute. OS/2's designers included extended attributes as a mechanism to hold more information than DOS could accommodate.

Objects are directly manipulated wherever possible. Direct manipulation helps to extend our metaphor of objects. When we can pick up and drop objects, many actions become more intuitive. Drag-and-drop is typically difficult to program, so we build most drag-and-drop functions into our user interface framework, and subclass to get the behavior for free. Workplace shell objects support methods that notify us of various drag-and-drop events.

That's really most of what you need to know about OO user interfaces. The interface is based on objects which are represented by icons. These objects are instances of classes, which are represented as templates. The objects represent data and behaviors. The data is stored in the object's data file and extended attributes. The behaviors are accessed via direct manipulation where possible.

The WPS uses the notebook and container controls extensively. These are complex controls, but you will not need to know much about them. The code that needs them is already written for you. It is encapsulated and exposed through SOM. This really begins to show the power of OO technology. By subclassing the various WPS components, you get extensive complex behaviors for free. You need only specialize various behaviors, and tell the WPS how to access various areas of your application.

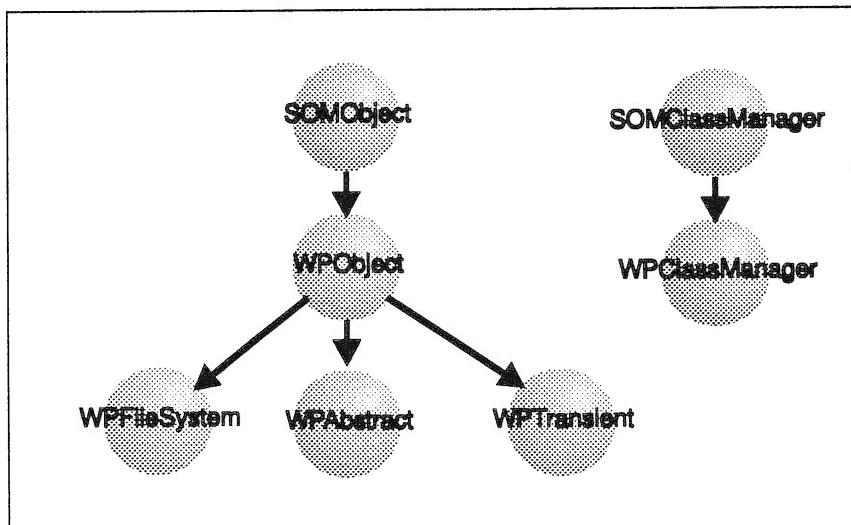


Figure 8.1. Major Workplace Shell Classes

WRITING WPS PROGRAMS

We have already discussed the look and feel of the user interface. Programming the WPS involves enabling various elements of the user interface. WPS objects are instances of classes, so we must create any templates for any special data objects that our program will manipulate. We know that the objects will have behaviors. Some will be accessed via direct manipulation, so we must override the appropriate methods, such as print and delete, for the classes that our template will create. We know that we will introduce new behaviors, so we must use WPS methods to add to the appropriate pop up menus for our application objects. We know that our application will have instance data, so we must use the WPS methods to create a notebook control that can be used to manipulate our instance data. We also know that WPS uses extended attributes, so we must be sure that r application handles extended attributes appropriately.

Wow. That looks like a lot of work. Let's take a closer look at the SOM classes that make up the WPS.

Workplace Shell Classes

From a programmer's perspective, the Workplace Shell is nothing more than a collection of SOM classes, and instances of those classes. Programming the WPS involves subclassing from the library of classes. Understand the various SOM classes that make up the WPS, and you understand the WPS.

Much of the WPS code within the operating system is dedicated to using the notebook and container controls. The container control is used to implement the folder, and is used extensively for drag-and-drop support. The notebook control

is used to set the instance data within an object. Since using these controls tends to be complex and problematic, we want to let the WPS do as much of that work for us as possible. We do not want to have to recode the portion of our application that manages these controls. Thanks to OOP, we really don't have to worry about these controls too much. The WPS classes will do much of it for us. You should be seeing the power of OOP working for you in a big way. By staying with the basic WPS classes, you will save yourself a lot of work.

WPS classes are derived from the basic SOM classes. The main WPS class is appropriately enough called *WPObject*, which is derived directly from *SOMObject*. WPS also has its own class manager, called *WPClassManager*, which is derived directly from *SOMClassManager*. We will concentrate on *WPObject*, shown in Figure 8.1. There are three important subclasses of *WPObject*. *WPFileSystem* contains all objects which are file system files or directories. *WPAbstract* is a class that contains abstract objects which stand for other objects within the operating system or computer system (e.g. printer, mouse, or color palette).

WPObject

The nice thing about the Workplace Shell class is that you have the ultimate reference in the .SC files. You can use these files to find out exactly which methods are supported, and the associated parameters. We will gloss over the highlights here.

Like *SOMObject*, *WPObject* is the highest level Workplace Shell object. This class is an abstract class. It defines the interfaces that make Workplace Shell programming go. You will find the methods and instance data that is required for all Workplace Shell objects.

Some of the methods of *WPObject* are related to the processing of the notebook control. The notebook control is ideal for setting up the instance data for an object. The notebook does a good job of organizing the different types found in instance data. *wpAddObjectGeneralPage*, *wpAddSettingsPage*, and *wpAddObjectWindowPage* all help to mask the end user from the complexities of the notebook control. Similarly, *wpCnrInsertObject*, *wpCnrRemoveObject*, and *wpCnrSetEmphasis* all help to manage the container control. There is also support for dragging and dropping Workplace Shell objects. *wpDraggedOverObject*, *wpDragOver*, *wpDrop*, and *wpDroppedOnObject* are all methods for testing, notifying, and performing various drag-and-drop functions. There are also some simple methods for general use, such as *wpClose* to close an object, and a whole range of methods to copy an object or create one from scratch. Again, you can find a list of all of the available methods within the .SC file.

WPFileSystem

The *WPFileSystem* class is shown in Figure 8.2. Objects that map directly onto files or directories are instances of some subclass of *WPFileSystem*. There are two subclasses of interest. *WPFolder* is a WPS folder, and is implemented with the container control. The folder objects map onto directories in the file system.

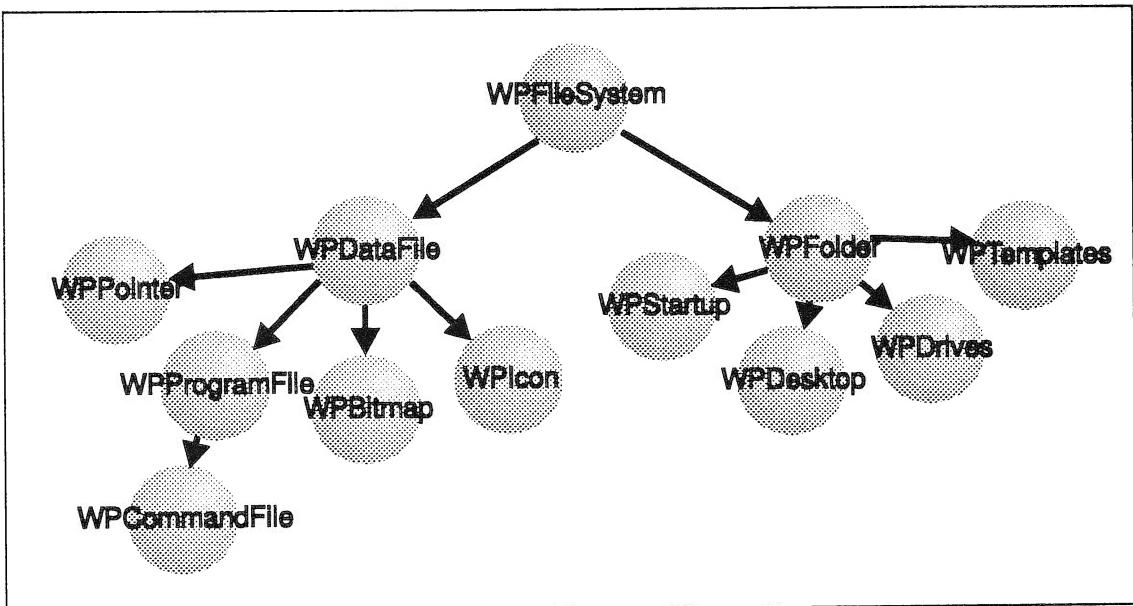


Figure 8.2. Workplace Shell File System Objects

As such, they can be nested. The *WPDataFile* class represents any object that maps directly onto an OS/2 file system file. Some files only contain data, such as icons or bitmaps. All executable files map onto the program class.

There are several interesting methods within the *WPFileSystem* class. *wpSetRealName* and *wpSetType* are used to access the WPS type and name of the file, which is not the same as the extension and name of the associated file. You can similarly set and query the attributes through *wpSetAttr* and *wpQueryAttr*. There are some methods to provide information about the associated file. You can get the last access or write through *wpQueryLastAccess* or *wpQueryLastWrite*, get the size of the file or extended attributes through *wpQuerySize* or *wpQueryEASize*. Several methods and flags can be used to refresh the object to check to make sure that the file system object is "in sync." There are methods to rename the file, as well as methods to put up a confirmation message when the file is renamed. There are also some methods which are used to help manage the notebook pages, as with the *WPAbstract* class. There are also methods to add items to the pop up menu.

The *WPDataFile* is a special class because of the way in which files are managed within the WPS. When you create a WPS application, you will probably use this class to help manage your program's data. You may not have to specifically override the methods of the class, but you will need to understand them. You will associate your program's data with a type. This will define to OS/2 how the data file is to be interpreted. For example, when you open this file, you may want to open the system editor, or you may want to open part or all of your application. If your program has associated print requirements, you will need to override one or all of the print methods that follow here:

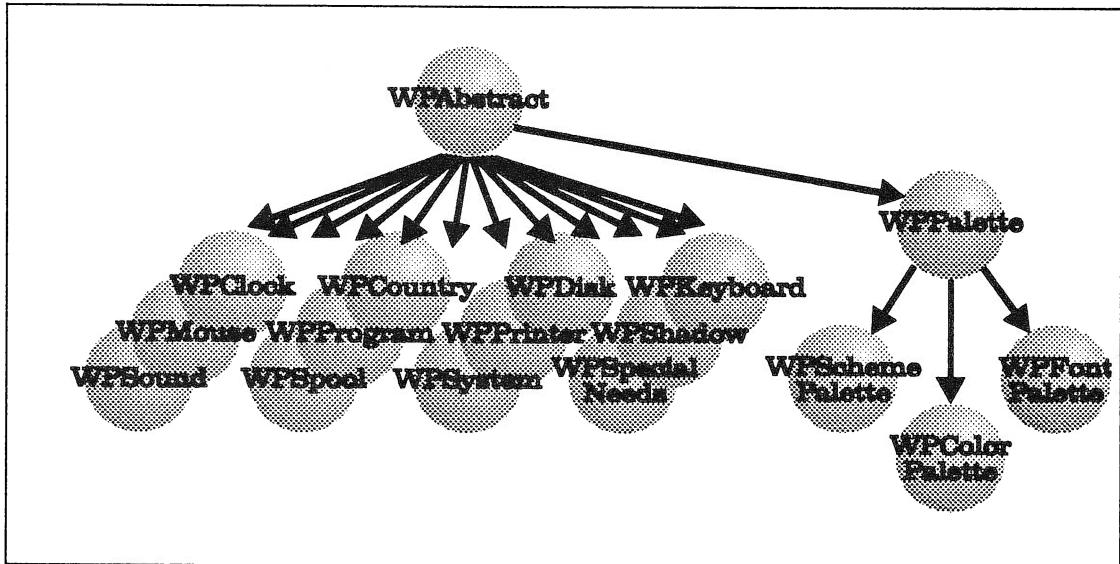


Figure 8.3. Workplace Shell Abstract Classes

`wpPrintMetaFile`, `wpPrintPifFile`, `wpPrintPrinterSpecificFile`, or `wpPrintUnknownFile`). The `WPProgram` class is associated with executable .EXE files or REXX .cmd files. Like the `WPDataFile` class above, many of the methods for this class relate to which types of files that the program files will use, if any. You can set this type with `wpSetAssociationType`, for example.

WPAbstract

The `WPAbstract` class shown in Figure 8.3 is used to create objects that we can use to manipulate things like printers or keyboards. This class illustrates a concept known as the *proxy object*. We would rather deal with objects than other constructs because they are familiar, and they allow you to apply the OOP concepts that you have learned from this book. No problem. The method implementations of the abstract class deal with the real world objects. We only see the class interface, we are effectively insulated from the real world. Our object-oriented programs need only deal with familiar objects. We effectively reuse the classes that have knowledge about the real world.

`WPAbstract` is an unfortunate name. *Abstract class* has a clear meaning to object-oriented programmers. An abstract class is one that defines an interface, but no implementation. Such a class does nothing, but subclasses of the abstract class will all have a consistent interface. If the name was given because the class is abstract, then it is not descriptive enough. A better name for this class would be *ProxyClass* (because the class stands in for other things), but as long as you understand the concepts, you can guard against misuse.

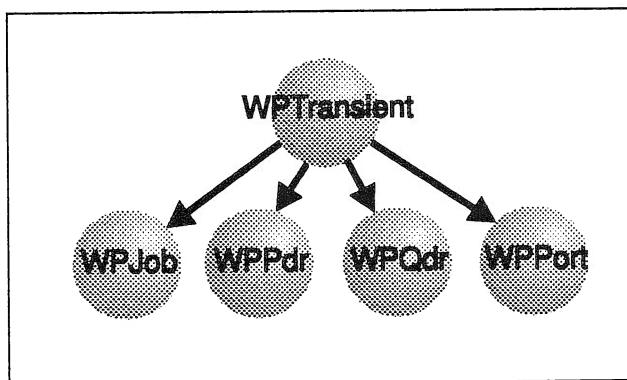


Figure 8.4. WPS Transient Objects

The `WPAbstract` class has many subclasses. Most are related to configuration. `WPCountry` and `WPKeyboard` are used to help the OS/2 user configure the WPS to support other languages and nationalities. `WPPrinter`, `WPDisk`, and `WPMouse` are used to let the user configure and directly manipulate the corresponding computer components. In addition, the `WPAbstract` class deals with *palettes*. These objects let you customize the OS/2 desktop with your choice of color and font. There are many other `WPAbstract` classes listed, many of which you can see for yourself in the System folder on the WPS desktop.

Surprisingly enough, there are no new methods for the `WPAbstract` class. When you look at the .SC file, you will find that many methods were simply overridden. We get the specialization that we need for `WPAbstract` objects, but we can program to the same interface as `WPObject`. This is polymorphism at its best.

`WPMouse` is a good example of a subclass of `WPAbstract`. We will see some new methods for mouse customization. There are only three methods, and all deal with setting up the notebook control for the mouse object. `wpAddMouseMappingsPage`, `wpAddMouseTimingsPage`, and `wpAddMouseTypePage` all help to define the notebook pages for the `wpMouse` instance data. The other objects in `wpAbstract` are specialized in a similar manner.

The various palettes make configuration of system wide and application fonts and colors easy.

WPTransient

The `WPTransient` classes in Figure 8.4 do not really stand alone. They are not *persistent* (saved on fixed media); they are only used to save some state for another program. A good example of a transient class is a print job, or `WPJob`. Objects of this class are only jobs in the print queue, and they will go away after they are printed. To say this another way, elements of this class are not persistent. Like `wpAbstract`, `wpTransient` introduces no new methods—it simply specializes what is already there. So now you know that once you learn

`wpObject`, you already know all of the methods in both `wpTransient` and `wpAbstract`!

IDL VS OIDL

Unfortunately, the WPS was developed just before IDL was available. At that time, SOM was implemented with *OIDL: Object Interface Definition Language*. OIDL has a friendlier syntax, but it is not as flexible. The new SOM compiler can be used in the OIDL mode by adding the keyword *CallStyle*. To force IDL mode, add the phrase “`CallStyle = IDL`”. IDL will work fine with the WPS, but you will have to create .IDL files out of the existing .SC files for the classes you need. Conversely, to force OIDL (which is necessary for WPS programming if you do not want to create the new IDL files), add the phrase “`CallStyle = OIDL`.” Hopefully, IBM will create some IDL files for the WPS soon. We will now give a brief OIDL introduction.

Declaring a Class with OIDL is similar to IDL. You must give the class a name, specify your parent class, provide method signature, and define your instance data. Remember, with OIDL there is no multiple inheritance, and there are no attributes. You will only be able to specify a single parent, and your instance variables will not be IDL attributes, but most other things work the same. Declaring the class looks like this:

```
class: wpObject;
```

There are some optional pieces to the class statement, such as major and minor versions and file stems that may be appended, but the class name alone is sufficient. The file stem is the name of the file prefix that contains the headers, class definition, and code. The major and minor versions are used to determine compatibility. For example, if you deleted a method, obviously your class would not be compatible with code that needed this method, so you would change the major version number. Next, you specify the parent class:

```
parent class: SOMObject;
```

After your parent class, you can specify your method signatures. The syntax for OIDL is a little less verbose. A method definition looks like this:

```
methods:  
    returnType methodName(type p1, type p2...);
```

Instance data is handled in a similar way, but the instance data for WPS is all accessed via methods, so you should not need it. Release order is the same for IDL and OIDL. Now that you know the differences, you can make OIDL and IDL talk. We hope that soon IBM will release IDL versions of the WPS classes, but this information will help you get started if our book appears before this happens.

IDs and Setup Strings

IDs are the unique identifiers that WPS uses to refer to any WPS object. The default value is the full path to the object, but you can specify your own ID through the use of the setup string. These strings specify the instance data for a WPS object. You can access the setup string via the method call `wpsWinSetObject`. Each class within a hierarchy defines its own setup string. When the setup strings are scanned, each class scans its own string and then passes the rest to its parent. Some examples of setup strings are `ICONFILE` and `OBJECTID`.

LEVELS OF COMPLIANCE

Of course, it is our hope that, armed with the awesome knowledge that we have provided, you will storm into the real world and crank out WPS applications by the truckload. This attitude is not completely practical. WPS applications require a lot of work. Sometimes, it just is not possible to do a full-blown WPS application. There are some intermediate steps that will make your programs behave better with OS/2. These steps are taken from a course entitled *WPS Programming* by IBM.

Compatible

The goal here is for the application to *work* under the WPS. You need to preserve EAs and build an associtable into your application. You also need to write the type EAs to your datafile, so that it will automatically launch the right executable when you double click on it. This does not require much code at all.

Aware

After you have read this book, you should be able to do this step easily. The goal is to be able to print through drag-and-drop. All that you need to do is override the `wpPrint` method of the `WPDataFile` class. No problem. You already have to have the print code for your application. To do this right, all you need to do is move it to a SOM class, or call it from a SOM class.

Integrated

Of course, the goal here is to develop your entire application using SOM. You can use the container control liberally, so that the user can drag-and-drop objects as well as data files. You can use the techniques in this book to wrap PM controls. The sky is the limit!

Installation

With a little extra work in installation code, your application installation program can create a program object. This can point to your actual .EXE. Automat-

ic creation of the program object just removes one more step for the user's installation process, and is definitely recommended.

RELATED TOPICS

There are a few areas that you can improve that will make your life as a WPS programmer dramatically easier. These are a few that we recommend:

Special Controls

The WPS makes extensive use of the container control, the pop up menu, and the notebook control. The container is used to manage files within a folder. Pop up menus are attached to each WPS object, and they are accessed by pressing the right mouse button. The notebook control is used to store the settings for WPS objects. It pays to brush up on these three controls if you are going to be doing extensive WPS programming.

Extended Attributes

OS/2 needs more information than its DOS counterpart for each file. This information is known as *extended attributes* or EAs. Examples are icons and long file names. The WPS uses extended attributes extensively. EAs are like environment variables, and are all stored in a file separate from the file they represent. They may be simple text strings, such as the long file name. They may also be binary, such as icons. The WPS associates a type and an association table, so that WPS will know how to treat the file. If the file has a type and an association table, you can launch the program associated with the data file by simply double clicking on the datafile. EAs can be a pain, though. If you use DOS programs that do not properly preserve these EAs, it is easy to lose information that can significantly damage the WPS, so be careful.

CONCLUSION

If you have learned anything by now, you know that WPS objects are just SOM objects. As such, you really have your own private set of documentation: the class definition files! These files tell you exactly what the WPS hierarchy looks like, what the methods are, what types these methods have for return values and parameters, and everything else about the class definition that you will need to know. Of course, the details of the implementation are well masked, but we have learned that is not necessarily a bad thing. The next section of the book will go on to introduce some programming examples. We'll show you extensive SOM, PM, GPI, EA...whew. I start throwing out acronyms and get a little carried away. You are armed with the supporting theory. Let's put it into practice!

Section II: Samples

9

SOM Examples

INTRODUCTION

So you've been asked by your manager to bring the latest and greatest in software technology to your software shop: Object Oriented Programming. You ask yourself, "why do I need OOP? What is it that OOP can offer me that I didn't already have with my procedural programming techniques?"

Those are good questions to ask. As an OS/2 PM C programmer, you probably think there isn't anything worth learning in the programming world. You probably can do just about anything you'll ever need to do with procedural programming, but OOP can help you simplify and effectively hide some of the ugly sides of PM and OS/2 programming. OOP can help you work smarter and more efficiently, and ultimately can help you present a more powerful, flexible, and intuitive application to your customer, that end user.

In the examples that follow we'll be looking at how you can use IBM's SOM to implement classes and class libraries that will enhance your productivity and help you to master some of the concepts needed to become a good OOP programmer.

USING THE SYSTEM OBJECT MODEL, SOM

We begin using SOM with a simple example. Through this example we will illustrate some basic SOM concepts:

- Defining the interface definition using IDL
- The implementation file and implementation bindings
- The client program and usage bindings

We will also become familiar with some of the tools we will use in our test programs:

- **sc.** The SOM Compiler with its emitters
- **icc.** The IBM CSet/2 compiler
- **link386.** The IBM CSet/2 linker

EXAMPLE ONE

The first example is a “bare bones” SOM class with one method. You need three files:

- **x1.idl.** Define your class using the Interface Definition Language (IDL).
- **ex1.c.** Implement the method of your class using the C language.
- **runapp.c.** The starting point of your application. the main{} routine.

Defining Your Interface

The first file you need to write when creating a SOM Class is the IDL file. The IDL file defines the key components of the class: the classname, the parent class, the methods, and the data or attributes. OO software technology has been compared by many to integrated circuit (IC) chips or Legos. IC chips of a variety of functionality and content can be plugged into a larger circuit by plugging the external pins into the circuitry. The user doesn't need to know anything about the inside of the chip, but only needs to understand the behaviour of the chip at the pin level. In a similar metaphor Legos of a variety of shapes, sizes, and function fit together because they have a surface that connects with the surface of an adjoining lego. Think of the IDL file as the surface of your lego that you can plug into a bigger structure of legos.

EX1.IDL

```
#include <somobj.idl> // include definition of your parent
interface ExampleOne: SOMObject // ExampleOne, a subclass of
// SOMObject
```

```
{
    void xSayHi(); // my very first method
};
```

The name of your class is defined using the keyword *interface*. In `ex1.idl`, we have named our class `ExampleOne`. Your class must have one or more parent classes that are listed after your class name and a colon. `SOMObject` is the class from which all classes descend. Each of your parent's IDL files must be included before this interface line so the SOM compiler will recognize the class-name when it occurs. In our case, include `<somobj.idl>` must occur before the interface line which informs us that `ExampleOne` is a subclass of `SOMObject`.

The body of the interface definition is enclosed in {} curly braces and is terminated with a ; semi-colon:

```
* interface ExampleOne: SOMObject
{
    ...
};
```

In our example, there is only one line in the body of the interface definition:

```
void xSayHi();
```

This is a method declaration. It looks a lot like a prototype for a C function. The return type here is `void`, but could be any of a number of primitive types or objects that are defined by CORBA. The method `xSayHi` has no arguments in the IDL, but if you look at the implementation file (`ex1.c`) you will see that the C function takes two arguments, `somSelf` and `ev`. We'll discuss these parameters as we look at the implementation file.

THE IMPLEMENTATION FILE

The implementation file is where you actually provide the functionality that you have advertised in your IDL definition. This is the answer to "Where's the beef?" Currently, SOM allows you to implement your methods using the C or C++ languages. Clients who want to use your class will not need to know what language you used in your implementation file. Their only concern is your outer IDL surface.

EX1.C

```
#define ExampleOne_Class_Source
#include <ex1.ih>

SOM_Scope void SOMLINK xSayHi(ExampleOne somSelf, Environment *ev)
{
```

```
    printf("→ I'm an OO SOM programmer!!\n\n");
}
```

Two lines occur at the beginning of all implementation files. In our example they look like:

```
#define ExampleOne_Class_Source
#include <ex1.ih>
```

The first is of the form:

```
#define <yourClassName>_Class_Source
```

This an essential definition needed to help the compiler resolve the many macros that the SOM compiler uses in managing the internal method tables. If you ever change the name of your class, be sure that the name of the class defined on the interface line of your IDL matches the class name in this definition, otherwise you will get very strange compiler errors and warnings.

The second line includes your implementation header file. If you are using the C language you should have put an .ih in your SMEMIT environment variable and included the .ih file here in your implementation file. C++ users would replace xih with ih. This .ih (or .xih) file is known as the *bindings* file and must be included in all implementation files.

The SOM emitter framework

At the outset, I mentioned three files that you would need to create, but they ended with .c or .idl, so where does the .ih file come from, or in the case of the client program coming up, where will the .h file come from? These files are generated for you automatically by the SOM compiler. They will be output by, or in SOM compiler terminology emitted by, the sc program because of the instructions you gave in your SMEMIT environment variable.

The SOM compiler is capable of emitting a variety of types of target files all using your .idl file as the source for the variety of targets. In this example we set SMEMIT to h;ih; which says "We would like just two files to be emitted, thank you." Later we'll make use of a few other of the emitter options that are provided. For our ExampleOne class we need to know about four files that all share the same filename stem, ex1.

- **ex1.idl**. Your definition file. You write it all by yourself.
- **ex1.c**. Your implementation file. You write it, but "sc" will help you.
- **ex1.ih**. Your C bindings or implementation header file. It is emitted for you by "sc" and must be included near the start of your implementation file.
- **ex1.h**. The usage header file. It is emitted for you by "sc" and must be included by all users or clients of your class.

After the first two required lines, we implement our method. Our method declaration looks a lot like a C program, but based on the IDL file, you might have hoped to see us use a much simpler method declaration like:

```
void xSayHi()
```

But this would be too easy for us sophisticated OO programmers. Instead, the simplest of method declarations looks like:

```
SOM_Scope void SOMLINK xSayHi(ExampleOne somSelf, Environment *ev)
```

Macros

SOM_Scope, SOMLINK, somSelf, and ev...they're everywhere. Every SOM method makes use of these two macros and two parameters. Of the four, the only one that will concern us is somSelf. SOM_Scope and SOMLINK are two macros that must appear in your implementation file with every SOM method. These are used to assist in compiling your implementations with platform independence. Also, all methods of a particular class will have somSelf and ev as the first two parameters. The first is a pointer to the instance of the class that will be the receiver of the method action. The second is a pointer to a structure of type Environment. It is used for exception handling among other things. This second parameter is new to the second release of SOM and has been added to comply with the CORBA standards being set by the OMG.

It sounds like a lot of overhead to add every time you create a method, but in Example Two in this chapter, you'll see how the SOM compiler provides an emitter that can be used to assist you in setting up your methods. In this example, I've forced you to key in all the overhead, so that when I introduce the new emitter you'll appreciate the SOM compiler that much more!

In our examples we will usually get a value for ev in one of two ways. One, as in test.c we will use the return value of the macro somGetGlobalEnvironment(); Environment *ev = somGetGlobalEnvironment(). Two, if you are making a method call in another SOM method, you can simply use the value of ev that was passed to your method. Don't worry for now about the contents of ev, just trust your global ev or the ev you get from your client and pass it along.

```
SOM_Scope void SOMLINK yourMethod(ClassName *somSelf, Environment
*ev)
{
    Class2 classObj;
    _anyMethod(classObj, ev);
}
```

One other note about the second Environment parameter. SOM release 1.0 used an interface definition language called OIDL (Object Interface Definition Language) which predates the CORBA guidance for using Environment *ev. In order to support backward compatibility, SOM release 2.0 supports both call-

ing conventions; not all method calls will contain the Environment parameter (including SOMObject methods). Our examples will all use the new calling styles, which is what the SOM compiler assumes by default.

THE CLIENT PROGRAM

The example program, which uses our new class, is called a *client* program. This implies, of course, that our SOM classes can be thought of *servers*.

TEST1.C

```
#include <ex1.h>
int main(int argc, char *argv[])
{
    Environment *ev=somGetGlobalEnvironment();
    ExampleOne obj=ExampleOneNew(); /* create instance */
    _xSayHi(obj, ev); /* invoke method on instance obj */
    _somFree(obj);
    return (0);
}
```

test1.c includes the .h file corresponding to our class. test1.c creates an instance of the class ExampleOne, it calls our method xSayHi using the newly created object and the omnipresent ev parameter, and then does a somFree to free up the memory allocated during ExampleOneNew(). Clients are not restricted to main{} programs. Every method call constitutes a client/server relationship. The caller is the client and the implementer is the server, so any method could act as both a server and a client.

Invoking SOM methods.

The invocation of our method is done by adding an underscore to the beginning of the name of the method. Other than that, it looks like an invocation of a simple C procedure. The first parameter is the pointer to the object on which the method can act. The second is our friend the ev parameter, and after that you include any other parameters required by the method (in this example no additional parameters are necessary). To know what parameters to use in addition to the somSelf and ev, you can examine the class's IDL file. There you will find everything you need to know about a method except for the underscore, somSelf, and ev. The method's name, with its set of required parameters, is sometimes referred to as the method's *signature*.

BUILDING AND RUNNING YOUR PROGRAM

To build your test program so you can run it, we will use three programs:

- **sc.** The SOM compiler.
- **icc.** IBM's CSet/2 compiler.
- **link386** The 32-bit linker also included with IBM's CSet/2 compiler.

Environment Variables

To use the SOM compiler, you need to have a few environment variables set up:

```
SMEMIT=ih;h; /* separate multiple output opts with ";" */  
SMINCLUDE=<dir1>;<searchdir2>;...
```

The SOM compiler can emit a variety of information. When we set SMEMIT to ih;h; we are telling the SOM compiler to emit the C bindings for the implementation file (.ih) and the usage program or client (.h). Later we will add c to our options, and the SOM compiler will emit C code for us.

You can also control the types of emitted files by using the *sc* command option *-s*. For example, to emit simply the usage bindings (the .h file) invoke the SOM compiler as follows:

```
sc -sh <filestem of IDL file>
```

SMINCLUDE is set to a search path similar to the *INCLUDE* environment variable for your C compiler. It will help the SOM compiler find all the IDL files.

You can initialize these two environment variables from a command line using the *SET* command or you can place the following lines in your config.sys file

```
SET SMEMIT=ih;h;  
SET SMINCLUDE=...
```

SOM compiler, sc

Then, to run the SOM compiler type:

```
sc ex1
```

C Compiler, icc

To compile your implementation file and client program use your standard C compiler. In our case we use the CSet/2 compiler from IBM:

```
icc -Ti -c ex1.c  
icc -Ti -c test1.c
```

```
/* for a description of the icc compiler options type icc ?
 */
```

Linker, link386

Then we link our newly created objects using the **LINK386** command:

```
link386 /PM:VIO /NOI /NOL /CO test1.obj ex1.obj, test1.exe,
,somtk;
```

/PM:VIO indicates that I want this to run in a PM window even though it is not a true PM program.

/NOI means No Ignore Case. Most C language links use this option because of its case sensitivity. */NOL* means No Logo. The linker will not output its standard spiel telling us about the linker we just invoked. Very optional. */CO* means keep debug info. This will increase the size of your binaries almost two-fold, but I keep it in until the last minute to ease troubleshooting.

Run!

Now run your program, *test1.exe*.

```
D:\EXAMPLE\SRC>test1
-> I'm an OO SOM programmer!!
```

You did it! Now we are on our way into the world of OOP programming on OS/2 using SOM.

10

SOM Attributes and Overrides

INTRODUCTION

In the first example, we introduced a method called SayHi, and in our client we simply called the method. In this next example we are going to look at attributes and then we will look at how to override a method that is introduced in our base class.

ATTRIBUTES

Attributes or Instance Variables

Attributes are just like instance variables. They are both merely data that are a part of each instance of the class. In SOM, the difference between the two, attribute and instance variable, is that we use attributes for variables that you would like to be directly accessible by your clients. Client variables are usually introduced in a supporting role and can often be accessed indirectly. Attributes and instance variables must be one of the IDL primitive types. When you declare an attribute, for example:

```
attribute long mylong;
```

the SOM compiler automatically generates two methods for accessing that variable: `_set_mylong`, and `_get_mylong`. If you would like to prevent a client from changing the value of the attribute, you can declare it as `readonly`:

```
readonly attribute long mylong;
```

In this case the SOM compiler will only generate the `_get_mylong` method for a client to use.

Modifying attributes in your implementation section

There are other variations on how you can control your attributes and these are done in the *implementation section*. The implementation section is where you would declare your non-attribute instance variables. This section is nested within the interface section and is also enclosed in curly braces and terminated by a semi-colon:

```
interface ExampleTwo: SOMObject
{
    //      The external or public side of your interface.
    //      ... methods, attributes, etc. are introduced here.

#ifndef __SOMIDL__
implementation
{
    // the non-public side of your interface
    // adds nothing to the external interface
    // but aids in your implementation of the
    // interface advertised above.
    releaseorder: _get_mylong, _set_mylong;

    long x, y, w, h;

    mylong: noget, noset;
};

#endif
};
```

Above is an example of how to declare instance variables `x`, `y`, `w`, and `h`. I might put these variables in an implementation section, because I choose not to allow the getting and setting of the individual instance variables; instead, I might provide a couple of methods in the interface section (before the implementation section) like:

```
RECT GetArea();
void SetArea(RECT rectangle);
```

which would allow a user to get at the variables all at once.

Modifying attributes

In the above example, the *mylong* attribute is also modified. Earlier, a long attribute, called *mylong*, was introduced. If you want to blindly store the long variable and return it as clients invoke the set and get methods, you don't need to modify the attribute in the implementation section at all. However, if you want to do something extra during the invocation of the get/set methods you must ask the SOM compiler to let you implement the set/get methods yourself in your implementation source code. If you opt for the blind option, SOM will implement the set/get methods in your header files, and you will never see these methods in your implementation source.

Some variations on the data modification are as follows:

```
mylong: noget;    /* you implement the _get_ method yourself */
mylong: noset;   /* you implement the _set_ method yourself */
mylong: noset,noget; /* you implement the _sets and _gets */
mylong: nodata;  /* you implement the _set_ and
                   _get_methods and your instances don't
                   allocate space for the variable. */
```

When using the *nodata* modifier you must remember that you can't access *mylong* using the *sommThis->mylong* or *_mylong* techniques.

Identify your IDL compiler.

These instructions are largely for the IDL compiler, and this implementation section is very IDL-compiler specific. This implementation section must be enclosed in an *#ifdef / #endif* pair to identify the IDL compiler you are using. In our case:

```
#ifdef __SOMIDL__ // two _ before and after the SOMIDL
#endif
```

releaseorder:

The *releaseorder:* statement is one you will get used to right away if you want to get rid of SOM compiler warnings. The *releaseorder* should simply list all the methods, including methods that are implicitly introduced when you declare an attribute. This listing fixes the order in which methods are stored in a method table that SOM uses for method resolution. By using this, you can prevent some of the impact on your clients when you add or delete methods. Without this line, your adding or deleting a method would force all your subclasses and clients to recompile with your new headers. But if your *releaseorder* had been previously declared as:

```
releaseorder: a, b, c;
```

and you later removed method "b" but kept it in the releaseorder line, your clients would not have to recompile.

In this next example, we will use attributes and the implementation section. In this example, you will need the following files:

```
ex2.idl
ex2.c
test2.c
makefile
```

EX2.IDL

```
#include <somobj.idl> // get definition of parent

interface ExampleTwo: SOMObject
{
    attribute string msg; /* an attribute!! */
    attribute long mylong;
/* methods will be generated to get/set these
attribute values */

    void xSayHi();

#ifndef __SOMIDL__
implementation
{
    ## Class modifiers
releaseorder:
    _get_msg, _set_msg,
    _get_mylong, _set_mylong,
    xSayHi;

    ## Method Modifiers
somInit: override;
};

#endif

};
```

EX2.C

```
#define ExampleTwo_Class_Source
#include <ex2.ih>

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

SOM_Scope void SOMLINK xSayHi(ExampleTwo *somSelf,
```

```

        Environment *ev)
{
    ExampleTwoData *somThis = ExampleTwoGetData(somSelf);

    printf("-> xSayHi begin\n");
    printf(
"->   the attribute 'msg' => %s<=\n", __get_msg(somSelf, ev));
    printf("->   the attribute 'mylong' => %ld <=\n",
           _mylong);
}

/*
 * override 'somInit' which I inherit from SOMObject
 */
SOM_Scope void  SOMLINK somInit(ExampleTwo *somSelf)
{
    ExampleTwoData *somThis = ExampleTwoGetData(somSelf);
    Environment *ev = somGetGlobalEnvironment();

    ExampleTwoMethodDebug("ExampleTwo", "somInit");

    printf("--somInit: Initializing an instance of the
           ExampleTwo Class...\n");

/* give the 'msg' parameter a default value in somInit */
/* the next two assignments could be made in a number of
 * ways: */
/*          _msg = "default string"; */
/*          somThis->msg = "default string"; */
/*          __set_mylong(somSelf, ev, 33L); */
/*          somThis->mylong = 33L; */
__set_msg(somSelf, ev, "default string");
_mylong = 33L;

ExampleTwo_parent_SOMObject_somInit(somSelf);
}

```

TEST2.C

```

#include <ex2.h>

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[])
{
    ExampleTwo *obj;
    Environment *ev=somGetGlobalEnvironment();

```

```

/* create an instance of class ExampleTwo
 * call the xSayHi method once to see initial values
 * and again to see how our modifications worked. */

obj = ExampleTwoNew();
_xSayHi(obj, ev);
__set_msg(obj, ev, "Pssst, this is a msg");
__set_mylong(obj, ev, 4444L);
_xSayHi (obj, ev);
_somFree (obj);

return (0);
}

```

MAKEFILE — for SOM example two

```

.c.obj:
    icc -MS -Q -Ti+ -C $<

test2.exe: test2.obj ex2.obj
    link386 /CO /PM:VIO test2.obj ex2.obj,test2.exe,nul,somtk;

ex2.ih: ex2.idl
    sc ex2

ex2.obj: ex2.ih

test2.obj: test2.c ex2.ih

```

The C emitter option

After you create your IDL file, use the SOM emitter to create an implementation template for you by using the SOM compiler's C emitter. Update your SME-MIT environment variable to the following:

```
SMEMIT=ih;h;c;
```

or simply emit the C template using an **sc** option:

```
sc-sc ex2
```

This will speed up the creation of your C file (implementation file). After emitting using the C emitter, you will have the following implementation file to begin with:

```

/*
 * This file was generated by the SOM Compiler
 * Generated using:
 *     SOM Emitter emitctm: somc/smmain.c

```

```
*/  
  
#define ExampleTwo_Class_Source  
#include <ex2.ih>  
  
SOM_Scope void SOMLINK xSayHi(ExampleTwo somSelf,  
                           Environment *ev)  
{  
    ExampleTwoData *somThis = ExampleTwoGetData(somSelf);  
    ExampleTwoMethodDebug("ExampleTwo", "xSayHi");  
  
}  
  
SOM_Scope void SOMLINK somInit(ExampleTwo somSelf)  
{  
    ExampleTwoData *somThis = ExampleTwoGetData(somSelf);  
    ExampleTwoMethodDebug("ExampleTwo", "somInit");  
  
    ExampleTwo_parent_SOMObject_somInit(somSelf);  
}
```

You can then proceed to quickly fill in your methods and continue as before. This feature will save you time and help you to avoid a lot of headaches. You will notice that lines are deleted with ...GetData (if commented out or if the instance data is not used) or ...MethodDebug on them. This is done merely to save space in my source listings. The MethodDebug line is useful when you are debugging your program, and can be used by setting the global variable SOM_TraceLevel to a nonzero value. When SOM_TraceLevel is nonzero, the <classname> MethodDebug method will print out a message as your methods are entered.

Run Example Two

The output of the above example should look like the following (run this by entering test2 at the command line):

```
D:\example\two> test2  
--somInit: Initializing an instance of the ExampleTwo Class...  
-> xSayHi begin  
->   the attribute 'msg' => default string <=  
->   the attribute 'mylong' => 33 <=  
-> xSayHi begin  
->   the attribute 'msg' => Pssst, this is a msg <=  
->   the attribute 'mylong' => 4444 <=
```

In the above example attributes to store a string pointer and a long value are used. The SayHi method is used to indicate the values of those attributes. In addition, something new was done here.

THE OVERRIDE

In the ex2.idl file above, I have the statement:

```
somInit: override;
```

This is how to override methods in SOM IDL. This line is included in the implementation section, because this doesn't add anything to the public interface. This method was already introduced elsewhere—in the SOMObject class in this example. *somInit* and *somUninit* are methods introduced by SOMObject that you can override if you want to do initialization when your object is instantiated (*somInit*) or cleanup when you are *somFree*'ing your object (*somUninit*).

```
myObj = <classname>New(); /* results in calling of
                           * somInit */
_somFree(myObj);      /* results in the calling of
                           * somUninit */
```

In the previous example, the *SayHi* method printed information that was hard-coded into the *SayHi* method itself. In this example *somInit* is used to set the values of the two attributes, and when *SayHi* is called the values of those attributes are obtained and printed. I also update the attribute values from the client program (*test2.c*) and then call *SayHi* again to print out the attributes.

In the *somInit* method, some of the variations on how you can set the values of the attributes are shown (in comments). *somInit* is able to use these variations because it is in the implementation file itself which has included the *ex2.ih* implementation header file. This gives the implementor the ability to get at the object's data using the ...*GetData* macro:

```
ExampleTwoData *somThis = ExampleTwoGetData(somSelf);
```

Once this is called, *somThis->* or the underscore prefix can be used to get at my object's data. The client program, *test2.c*, was not able to use all these variations. It could only set and get the attribute variables through the *__set_<attribute>* *__get_<attribute>* methods because it included the *.h* file and not the *.ih* file.

Now we have been introduced to the basic elements needed to create object oriented programs. In the next two chapters we will look at two advanced concepts that will give us more power and ability: *Metaclasses* and *Multiple Inheritance*.

11

The SOM Metaclass

INTRODUCTION

In our first examples, we created an instance of a class which was called an object. A typical application can create several of these “instance objects” during the running of an application. Each object has its own data that is independent of all the other instance objects. What do you do if you want to have a variable or attribute be shared by all the objects of a class? Perhaps a single variable is applicable to all the instances of an entire class, but is also useful information to objects or clients NOT of the class?

METACLASS

SOM has a feature called the “Metaclass”. Every SOM class is itself an object. Each of the instance objects is created by the Metaclass which acts as a kind of cookie cutter as it creates new objects. For example:

```
Proletariat pro1, pro2, pro3;  
pro1 = ProletariatNew();  
pro2 = ProletariatNew();  
pro3 = ProletariatNew();
```

Above we borrow from Karl Marx's object-oriented analysis of classes and we have a class called the *Proletariat*. After instantiating one class object ...

```
pro1 = ProletariatNew();
```

we then have a metaclass object called *Proletariat* and we have an instance object called *pro1*. After three calls to *ProletariatNew*, we have three instance objects and just one metaclass object.

Implicit and Explicit Metaclasses

A metaclass exists *implicitly* after you create your first instance of a class. So far we have never used metaclasses *explicitly*, but in the next two examples, we will. The above object creation could have been done with the explicit metaclass of the class *Proletariat* in the following way:

```
/* proletariatClass is a pointer to the explicit
 * metaclass */
SOMClass proletariatClass; /* declared as an object
                           * of type SOMClass */
Proletariat pro1, pro2, pro3; /* three as before */
proletariatClass = ProletariatNewClass(0,0);
                           /* explicit metaclass */
pro1 = _somNew(proletariatClass);
                           /* now use metaclass to */
pro2 = _somNew(proletariatClass);
                           /* create instance object */
pro3 = _somNew(proletariatClass);
```

SOM also provides a macro that allows you to use the metaclass class object by simply prefixing an underscore to the name of the class. The previous example could thus look like this:

```
Proletariat pro1, pro2, pro3;
ProletariatNewClass(0,0);
pro1 = _somNew(_Proletariat);
pro2 = _somNew(_Proletariat);
pro3 = _somNew(_Proletariat);
```

Uses for Metaclasses

OK, so what have we gained now that we know about the existence of metaclasses, and now that we know how to access them explicitly? So far it looks like the programming model just got a little more complex! What's in it for lazy programmers?

Explicit metaclasses are useful as *constructors* of instance objects or for maintaining information about all the instances of a particular class. Of course, newer SOM programmers will have the Constructor framework available, and

metaclasses are not necessary for this purpose, but the constructor example is still instructive. Any older SOM code is still likely to have them.

A CONSTRUCTOR EXAMPLE

A constructor is an object initialization routine. Up to now when you initialize your object you have been limited to the method `somInit` which gets invoked when you call the macro `<myclassname>New()`. However, this does not give you the ability to specify initialization parameters, and so you have been forced to do your initialization by setting the attributes or calling initialization-type methods after the class is instantiated. In this example, we will use metaclasses to help us with object initialization.

The Speaker Class

The metaclass is defined within the same .IDL file that is used to define the class instance objects, so there are two interface definitions. The first definition is the Metaclass which we call `M_Speaker`. `M_Speaker` is a subclass of `SOMClass`. In the `M_Speaker` interface definition we introduce only one method, `SpeakerCreate`, which takes a string parameter and returns a `Speaker` object.

SPEAKER.IDL

```
#include <somobj.idl> // the parent class definition
#include <somcls.idl> // the parent metaclass class def

interface Speaker;

interface M_Speaker: SOMClass
{
    Speaker SpeakerCreate(in string message);
/* SpeakerCreate is used here as our constructor returning
instances of the Speaker class */

#ifndef __SOMIDL__
implementation
{
    releaseorder:
    SpeakerCreate;
};
#endif
};

interface Speaker: SOMObject
{
    void speakToMe();
    attribute string whatToSay;
```

```
#ifdef __SOMIDL__
implementation
{
    //# Class Modifiers
    metaclass = M_Speaker; //# Speaker's metaclass ID
    releaseorder:
    speakToMe,
    _get_whatToSay, _set_whatToSay;

};

#endif
/* in later examples we will use the data modifier noset
whenever we have an attribute of type string. A class should
manage its own memory allocation of its string attributes. We
get away with it in this example, but this is not quite the
right way to do a string attribute. */
};
```

SPEAKER.C

```
/* We implement both the metaclass methods and the class
methods in this source file. In this example there are one of
each. */

#define Speaker_Class_Source
#include <speaker.ih>

SOM_Scope void SOMLINK speakToMe(Speaker somSelf,
Environment *ev)
{
    SpeakerData *somThis = SpeakerGetData(somSelf);
    SpeakerMethodDebug("Speaker", "speakToMe");

    printf("%s\n", __get_whatToSay(somSelf, ev));
}

SOM_Scope Speaker SOMLINK SpeakerCreate(M_Speaker somSelf,
                                         Environment *ev, string message)
{
/*      M_SpeakerData *somThis = M_SpeakerGetData(somSelf); */
    Speaker returnObj = _somNew(somSelf);
    M_SpeakerMethodDebug("M_Speaker", "SpeakerCreate");

    __set_whatToSay(returnObj, ev, message);
    return returnObj;
}
```

MAIN.C

```
#include <speaker.h>

int main(int argc, char *argv[])
{
    Speaker a, b, c;
    Environment *ev = somGetGlobalEnvironment();

    SpeakerNewClass(0,0);

    a = _SpeakerCreate(_Speaker, ev, "I am A");
    b = _SpeakerCreate(_Speaker, ev, "I am B");
    c = _SpeakerCreate(_Speaker, ev, "I am C");

    _speakToMe(a, ev);
    __set_whatToSay(a, ev,"I say whatever you ask me to
                           say");
    _speakToMe(a, ev);
    _speakToMe(b, ev);
    _speakToMe(c, ev);

    _somFree(a);
    _somFree(b);
    _somFree(c);

    return(0);
}
```

MAKEFILE

```
CC = icc
TARGET = hi.exe

.c.obj:
    icc -Q -Ti+ -c $<

hi.exe: speaker.obj main.obj
        link386 /CO /PM:VIO /NOI /NOL speaker.obj
main.obj,hi.exe,nul,somtk;

speaker.obj: speaker.c speaker.ih

speaker.ih: speaker.idl
            sc speaker

main.obj: main.c speaker.ih
```

RUNNING EXAMPLE ONE

The output for the above program looks like: (type “hi” to execute)

```
d:\ hi
I am A
I say whatever you ask me to say
I am B
I am C
```

EXAMPLE: OBJECT ADMINISTRATION

In the second example, we will use the constructor we introduced in example one. We will use two metaclass attributes to keep track of some information on the usage of our class and its objects. The first attribute, numberObjs, keeps a count of the total objects that have been instantiated. The second attribute, numberTimesSpokeAll, is used to keep track of how many times the method speakToMe is invoked by all classes. Compare this with the class attribute numberTimesSpoke, which is an attribute of each instance object and is used to keep track of the number of times a particular object is asked to speak.

SPEAKER.IDL

```
#include <somobj.idl> // the parent class definition
#include <somcls.idl> // the parent metaclass class
definition

interface Speaker;

interface M_Speaker: SOMClass
{
    Speaker SpeakerCreate(in string message);
/* the next two variable are Metaclass attributes and act as
global variables which are accessible to all instances of the
Speaker class and to all clients of Speaker class objects. */

    readonly attribute long numberObjs;
    readonly attribute long numberTimesSpokeAll;

#ifndef __SOMIDL__
implementation
{
    releaseorder:
    SpeakerCreate,
    _get_numberObjs,
    _get_numberTimesSpokeAll;
```

```

};

#endif
};

interface Speaker: SOMObject
{
    void speakToMe();
    attribute string whatToSay;
    readonly attribute long numberTimesSpoke;

#ifndef __SOMIDL__
implementation
{
    /* Class Modifiers
    metaclass = M_Speaker; //# identify Speaker's metaclass
    releaseorder:
    speakToMe,
    _get_whatToSay, _set_whatToSay,
    _get_numberTimesSpoke;
};
#endif
};

```

SPEAKER.C

```

#define Speaker_Class_Source
#include <speaker.ih>

SOM_Scope void SOMLINK speakToMe(Speaker somSelf,
Environment *ev)
{
    SpeakerData *somThis = SpeakerGetData(somSelf);
    M_SpeakerData *somThat = M_SpeakerGetData(_Speaker);
/* somThis AND somThat?? See explanation below of how we use
the GetData macro here to get instance data for both the class
instance and the metaclass itself in this fashion. */

    SpeakerMethodDebug("Speaker", "speakToMe");
    printf("%s\n", __get_whatToSay(somSelf, ev));
    somThis->numberTimesSpoke++;
    somThat->numberTimesAll++;

}

SOM_Scope Speaker SOMLINK SpeakerCreate(M_Speaker somSelf,
Environment *ev,
                                         string message)
{
    M_SpeakerData *somThis =

```

```

        M_SpeakerGetData(somSelf);
Speaker returnObj = _somNew(somSelf);
M_SpeakerMethodDebug("M_Speaker", "SpeakerCreate");

__set_whatToSay(returnObj, ev, message);
_numberObjs++;
return returnObj;
}

```

MAIN.C

```

#include <speaker.h>

int main(int argc, char *argv[])
{
    Speaker a, b, c;
    Environment *ev = somGetGlobalEnvironment();

    M_Speaker clsObj = SpeakerNewClass(0, 0);

    a = _SpeakerCreate(clsObj, ev, "I am A");
    b = _SpeakerCreate(_Speaker, ev, "I am B");
    c = _SpeakerCreate(_Speaker, ev, "I am C");

    _speakToMe(a, ev);
    __set_whatToSay(a, ev,
                    "I say whatever you ask me to say");
    _speakToMe(a, ev);
    _speakToMe(b, ev);
    _speakToMe(c, ev);

    printf(" we now have %ld total objects\n",
           __get_numberObjs(_Speaker, ev));
    printf(" total times spoken by all objects was %ld
           times\n",
           __get_numberTimesSpokeAll(_Speaker, ev));
    printf(" A spoke %ld times, B spoke %ld times and C spoke
           %ld times\n",
           __get_numberTimesSpoke(a, ev),
           __get_numberTimesSpoke(b, ev),
           __get_numberTimesSpoke(c, ev));

    _somFree(a);
    _somFree(b);
    _somFree(c);

    return(0);
}

```

RUNNING IT

The output for this example looks like the following (type hi again to execute):

```
d:\ hi
I am A
I say whatever you ask me to say
I am B
I am C
we now have 3 total objects
total times spoken by all objects was 4 times
A spoke 2 times, B spoke 1 times and C spoke 1 times
```

ACCESSING OTHER DATA

Here, we will show you how to access data outside of your own class. This example is pretty straightforward, except for the method `speakToMe`.

```
SOM_Scope void SOMLINK speakToMe(Speaker somSelf,
                                  Environment *ev)
{
    SpeakerData *somThis = SpeakerGetData(somSelf);
    M_SpeakerData *somThat = M_SpeakerGetData(_Speaker);

    SpeakerMethodDebug("Speaker", "speakToMe");
    printf("%s\n", __get_whatToSay(somSelf, ev));
    somThis->numberTimesSpoke++;
    somThat->numberTimesSpokeAll++;
}
```

In this method, information that is part of the instance's attributes must be updated. This is something we often do with either

```
somThis->numberTimesSpoke++;
```

or

```
_numberTimesSpoke++;
```

Accessing metaclass attributes

Here we access metaclass attributes from an instance method. We also want to update the metaclass attribute, `numberTimesSpokeAll`, from an instance method. But how does the `speakToMe` method get at the attributes of the metaclass? `somSelf` is not a subclassed object of `SOMClass` or `M_Speaker`.

<classname>GetData macro

The solution is this: we know that the `somSelf` is an object of the Speaker class, so we know the Speaker metaclass has been instantiated. Therefore we can get at the metaclass's attributes through the `M_SpeakerGetData` macro. This macro is available since both the class and the metaclass were defined in the same IDL and the `<speaker.ih>` implementation header file which gives me that macro (`GetData`) for both the class and the metaclass has been included.

This is an important point. Direct access to an object's data is possible only when the implementation source file has included the .ih file for that object. The `GetData` macro is not available to classes that only include the .h usage bindings header. You must be careful when including an .ih header file. Including a second .ih file for another class should only be done for tightly coupled classes, such as those packaged in the same library. A class that can access another classes data is sometimes referred to as a *friend*. In our examples, metaclasses and classes are always friends.

So we store the Metaclass data in `somThat` (`somThis` having already been taken) and we then simply point to it (directly access it without the aid of a method):

```
somThat->numberTimesSpokeAll++;
```

In this case we can't use the underscore shorthand because the underscore becomes *somThis* rather than our desired *somThat*.

Exercise 1.

As you probably noticed, I keep track of objects as they are created, but I don't decrement my count as they are deleted. What would I add to this program to keep the count up to date?

Exercise 2.

Modify the data in the implementation section of `speaker.idl` so that the `string` attribute is of type `noset`. Then implement the `__set__...` method in your implementation file. The C emitter will create a stub for you.

12

SOM Multiple Inheritance

INTRODUCTION

Our examples up to this point have all used single inheritance; each class has been derived from a single parent. Sometimes you may want your class to inherit the behavior of two or more base classes and yet these two classes have separate unique behavior and are not derived from each other. With *multiple inheritance*, your class can have more than a single base class.

Uses for Multiple Inheritance

In this example, we will explore possible uses for multiple inheritance. In our last example, we created a `Speaker` class which was capable of speaking out when a client program would call its `speakToMe` method. The mode of speaking was always simple text being sent to the screen. However, this mode of “speaking” is not always going to satisfy our needs. In the *SOM Toolkit User’s Guide*, multiple inheritance is introduced using an example that shows how an object can multiply inherit from classes that give the object the ability to output three different forms: screen, printer, or disk. Their output class, known as the `Hello` class is derived from a `Disk` class and a `Printer` class as in Figure 12.1. The screen output ability is built into the `Hello` class itself and is not derived from a higher class, although `Hello` could have had a `Screen` class as well.

Our example is going to use a similar class hierarchy in Figure 12.2, but we will explore the usefulness of being able to output in a variety of formats. Our

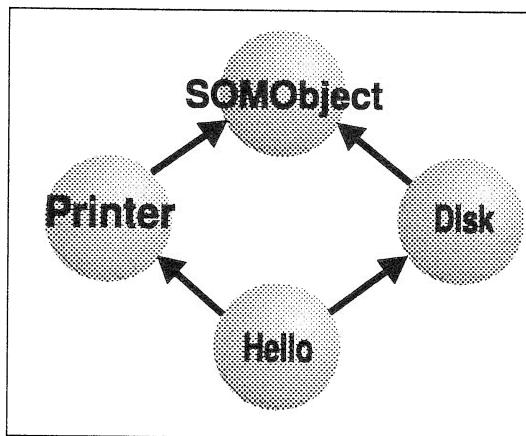


Figure 12.1. Multiple Inheritance

`Speaker` class currently prints as ASCII. This is not too snazzy, however, and in our world of fancy printers and Graphical User Interfaces, we would like the `Speaker` class to be able to output more than a mere stream of text that shows up in some default font on your screen or to wherever you direct it.

For our fancy output option, we have chosen a format known as PostScript, a product of Adobe Systems, Inc. It would be easy to expand this example to incorporate other formats by inheriting from other output objects. PostScript is actually a programming language itself; a device-independent page description language. The point of this example is not to teach you PostScript, but to show you another possible use for multiple inheritance. Depending on how many base classes `Speaker` is derived from, you could have the potential for output to possibly several output formats.

A Multiple Inheritance Class Declaration

Declaring a class using multiple inheritance is easy. In single inheritance we introduce a class named `A` with the statement

```
interface A: BaseClassName
```

With multiple inheritance, the class declaration looks similar. For example, a Class, `A`, derived from three base classes looks like the following:

```
interface A: BaseClass1, BaseClass2, BaseClass3
```

Any methods and/or attributes introduced by the base classes will be inherited by an instance of `A`.

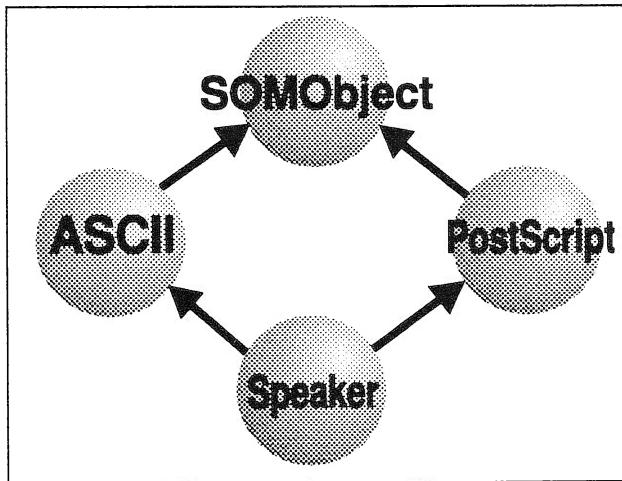


Figure 12.2. SPEAKER with PostScript

Enumerations

In this example, we also make use of an enumeration type, which is defined as follows:

```
enum languageSpoken{ascii, postscript};
```

This declaration of an enumeration is similar to the C *enum* type. ASCII is assigned the value of 0, postscript is given the value of 1, and other elements added to the enumeration, separated by commas, would be given consecutive integer assignments. These values can then be used by prefixing the name of the class to the front of the enumeration identifiers. For example, our Speaker class introduces the enumeration given above and the members of the enumeration are accessed with a switch statement. An attribute is declared using the enumeration type defined above:

```
attribute languageSpoken formatOut;
```

and in the client program

```
switch(__get_formatOut(somSelf, ev)) {
    case Speaker_ascii:
        /* do it the basic ASCII way */
        printf("....");
        /* or possibly call an AsciiOut class method */
        <asciiOutputMethod>(somSelf, ev, ...);
    case Speaker_postscript:
        /* do it the fancy PostScript way */
        <postscriptOutputMethod>(somSelf, ev, ...);
}
```

EXAMPLE. SPEAKING IN TONGUES

This example shows us how we can create a `Speaker` class, which is essentially bilingual. With other classes of output objects we could create many different varieties of bilingual and multilingual output classes. We could mix and match the output formats like Postscript, GML, etc. or GUI types such as PM and X Windows; or device options such as screen, printer, or file. Since objects of class `Speaker` are inherited from `Postscript`, we create a new file for each instance of a `Speaker` object. Output that is done using a `Speaker` object will be sent to its individual postscript file if the `formatOut` attribute is set to `Speaker_postscript`. The postscript file can then be printed on any printer with a PostScript interpreter.

ascout.idl

```
#include <somobj.idl>
interface AsciiSpeak: SOMObject
{
    readonly attribute somToken asciiFilePtr;

    void outputAsciiBuffer(in string ascBuffer);

#ifndef __SOMIDL__
implementation
{
    releaseorder:
        _get_asciiFilePtr,
        outputAsciiBuffer;

    override: somInit;
    override: somUninit;

};
#endif
};
```

ascout.c

```
#define AsciiSpeak_Class_Source
#include <ascout.ih>

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

SOM_Scope void  SOMLINK outputAsciiBuffer(AsciiSpeak somSelf,
                                         Environment *ev, string ascBuffer)
{
    AsciiSpeakData *somThis = AsciiSpeakGetData(somSelf);
```

```

FILE *fp = (FILE *)_asciiFilePtr;
AsciiSpeakMethodDebug("AsciiSpeak", "outputAsciiBuffer");

    fprintf(fp, "%s\n", ascBuffer);
}

SOM_Scope void  SOMLINK somInit(AsciiSpeak somSelf)
{
    AsciiSpeakData *somThis = AsciiSpeakGetData(somSelf);
    FILE *fp;
    string filename;

    static char tmpStr[256];
    static short defaultFontSize=20, counter=0;

    AsciiSpeakMethodDebug("AsciiSpeak", "somInit");

/*-----*/
/* open file that we will use to store this ascii output */
/*-----*/
    sprintf(tmpStr, "ascii%d", counter++);
    filename = (string)malloc(strlen(tmpStr)+1);
    strcpy(filename, tmpStr);
    fp = fopen(filename, "w");
    _asciiFilePtr = (void *)fp;

    AsciiSpeak_parent_SOMObject_somInit(somSelf);
}

SOM_Scope void  SOMLINK somUninit(AsciiSpeak somSelf)
{
    AsciiSpeakData *somThis = AsciiSpeakGetData(somSelf);
    FILE *fp = (FILE *)_asciiFilePtr;
    AsciiSpeakMethodDebug("AsciiSpeak", "somUninit");

    fclose(fp);

    AsciiSpeak_parent_SOMObject_somUninit(somSelf);
}

```

postscr.idl

```

#include <somobj.idl>

interface PostScriptSpeak: SOMObject
{
    attribute string currentFont; /* the font and
                                   * style in a string */
    attribute long  currentHPos;
    attribute long  currentVPos;

```

```

        attribute long   fontSize;
readonly attribute somToken psFilePtr;

void outputPostScriptBuffer(in string psBuffer);
// take a buffer of text and output using PostScript
format.

#ifndef __SOMIDL__
implementation
{
    releaseorder:
        _get_currentFont, _set_currentFont,
        _get_currentHPos, _set_currentHPos,
        _get_currentVPos, _set_currentVPos,
        _get_fontSize, _set_fontSize,
        _get_psFilePtr,
        outputPostScriptBuffer;

    //## instance variable
    long lineIncrement;

    override: somInit;
    override: somUninit;

    currentFont: noset; /* we need to take special
                           * action on this attr. */
    fontSize: noset; /* and this one too. */
};

#endif
};

```

postscr.c

```

#define PostScriptSpeak_Class_Source
#include <postscr.ih>

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/*
 *Method from the IDL attribute statement:
 *"attribute string currentFont"
 */

/*
 * the font and style in a string
 */

```

```

SOM_Scope void  SOMLINK __set_currentFont (
                                         PostScriptSpeak somSelf,
                                         Environment *ev, string currentFont)
{
    PostScriptSpeakData *somThis =
        PostScriptSpeakGetData(somSelf);
    FILE *fp = (FILE *)_psFilePtr;

    PostScriptSpeakMethodDebug("PostScriptSpeak",
                               "__set_currentFont");

    free(_currentFont);
    _currentFont = (string)malloc(strlen(currentFont)+1);
    strcpy(_currentFont, currentFont);
    fprintf(fp, "/%s choosefont\n", __get_currentFont(somSelf,
                                                       ev));
}

/*
 * the font's "scale"
 */

SOM_Scope void  SOMLINK __set_fontSize(PostScriptSpeak somSelf,
                                       Environment *ev, long fontSize)
{
    PostScriptSpeakData *somThis =
        PostScriptSpeakGetData(somSelf);
    FILE *fp = (FILE *)_psFilePtr;
    long oldLineIncrement = _lineIncrement;

    PostScriptSpeakMethodDebug("PostScriptSpeak", "__set_fontSize");

    _fontSize = fontSize;
    _lineIncrement = fontSize+4; /* update this instance
                                * variable */

    fprintf(fp, "/fontSize %ld def\n", fontSize);
    fprintf(fp, "/lineIncrement %ld def\n", _lineIncrement);

/*
The next two lines correct the lineIncrement from the previous
'newline' directive which was based on the old fontSizeThis
will also act as a correction to the placement given in
'somInit for vertpos which is based on a default fontSize.
This second type of correction is needed if you are setting
the fontSize before your first newline.
*/
    fprintf(fp, "/vertpos vertpos %d sub def\n",
            _lineIncrement - oldLineIncrement);
    fprintf(fp, "horzpos vertpos moveto\n");
}

```

```

        return;
    }

/*
 *Method from the IDL attribute statement:
 *"attribute string currentFont"
 */

/*
 * take buffer of text; output using PostScript.
 */

SOM_Scope void  SOMLINK outputPostScriptBuffer(PostScriptSpeak
somSelf,
                                              Environment *ev, string psBuffer)
{
    PostScriptSpeakData *somThis =
        PostScriptSpeakGetData(somSelf);
    FILE *fp = (_FILE *)_psFilePtr;

PostScriptSpeakMethodDebug("PostScriptSpeak", "outputPostScript
Buffer");

    fprintf(fp, "(%s) show newline\n", psBuffer);
}

SOM_Scope void  SOMLINK somInit(PostScriptSpeak somSelf)
{
    PostScriptSpeakData *somThis =
        PostScriptSpeakGetData(somSelf);
    Environment *ev = somGetGlobalEnvironment();
    string filename;
    FILE *fp;
    static char tmpStr[256];

    static short defaultFontSize=20, counter=0;

PostScriptSpeakMethodDebug("PostScriptSpeak", "somInit");

/* _____*/
/* open file that we will use to store this output */
/* _____*/
sprintf(tmpStr, "ps%d", counter++);
filename = (string)malloc(strlen(tmpStr)+1);
strcpy(filename,tmpStr);
fp = fopen(filename, "w");
_psFilePtr = (void *)fp;

/*_____*/
/* set the default font and allocate a string buffer */

```

```

/* to store the name */  

/*—————*/  

_currentFont = (string)malloc(strlen("Times-Roman") + 1);  

strcpy(_currentFont, "Times-Roman");  

_lineIncrement = defaultFontSize + 4;  

fprintf(fp, "%% —— Some useful procedures  

————\n");  

fprintf(fp, "/inch {72 mul} def\n");  

fprintf(fp, "/fontSize %d def\n", defaultFontSize);  

fprintf(fp, "/lineIncrement %d def\n", _lineIncrement);  

fprintf(fp, "/vertpos 10 inch fontSize sub def\n");  

fprintf(fp, "/horzpos 1 inch def\n");  

fprintf(fp, "\n");  

fprintf(fp, "/choosefont {findfont fontSize scalefont  

    setfont} def\n");  

fprintf(fp, "\n");  

fprintf(fp, "/newline\n");  

fprintf(fp, " {/vertpos vertpos lineIncrement sub def %%  

decrease vertpos\n");  

fprintf(fp, " horzpos vertpos moveto } def\n");  

fprintf(fp, "\n");  

fprintf(fp, "%% —— Beginning of your PS program  

————\n\n");  

fprintf(fp, "%% —— The next few lines are some  

    default initializations —\n\n");  

fprintf(fp, "%% — if the client program set these values  

    for font and initial position \n");  

fprintf(fp, "%% — then you should remove these lines  

    which will only be redundant —\n");  

fprintf(fp, "horzpos vertpos moveto\n");  

fprintf(fp, "/$s choosefont\n", _currentFont);  

fprintf(fp, "%% —— the rest is the result of the  

    client program. —\n\n");  

free(filename);  

PostScriptSpeak_parent_SOMObject_somInit(somSelf);  

}  

SOM_Scope void SOMLINK somUninit(PostScriptSpeak somSelf)  

{  

    PostScriptSpeakData *somThis =  

        PostScriptSpeakGetData(somSelf);  

    FILE *fp = (FILE *)_psFilePtr;  

    PostScriptSpeakMethodDebug("PostScriptSpeak",  

        "somUninit");  

    fprintf(fp, "showpage");  

    fclose(fp);
}

```

```
    PostScriptSpeak_parent_SOMObject_somUninit(somSelf);
}
```

speaker.idl

```
#include <somcls.idl> // the parent metaclass class

#include <postscr.idl> // the parent classes
#include <ascout.idl>

interface Speaker;

interface M_Speaker: SOMClass
{
    Speaker SpeakerCreate(in long speakLanguage);
    readonly attribute long numberObjs;
    readonly attribute long numberTimesSpokeAll;

#ifndef __SOMIDL__
implementation
{
    releaseorder:
        SpeakerCreate,
        _get_numberObjs,
        _get_numberTimesSpokeAll;
};

#endif
};

interface Speaker: PostScriptSpeak, AsciiSpeak
{
    void speakOut(in string whatToSay);
    void setFontAndSize(in string font, in long fontSize);

    enum languageSpoken{ascii, postscript};
    attribute languageSpoken formatOut;

    readonly attribute long numberTimesSpoke;

#ifndef __SOMIDL__
implementation
{
    /* Class Modifiers
    metaclass = M_Speaker; // identify Speaker's metaclass
    releaseorder:
        speakOut, setFontAndSize,
        _get_formatOut, _set_formatOut,
        _get_numberTimesSpoke;

    override: somInit;
```

```
};  
#endif  
};
```

speaker.c

```
#define Speaker_Class_Source  
#include <speaker.ih>  
  
#include <stdio.h>  
  
SOM_Scope void SOMLINK speakOut(Speaker somSelf,  
                                Environment *ev, string whatToSay)  
{  
    SpeakerData *somThis = SpeakerGetData(somSelf);  
    M_SpeakerData *somThat = M_SpeakerGetData(_Speaker);  
  
    SpeakerMethodDebug("Speaker", "speakOut");  
    switch(__get_formatOut(somSelf, ev)) {  
        case Speaker_ascii:  
            _outputAsciiBuffer(somSelf, ev,  
                               whatToSay);  
            break;  
        case Speaker_postscript:  
            _outputPostScriptBuffer(somSelf, ev,  
                                   whatToSay);  
            break;  
        default:  
            fprintf(stderr, "ERROR: formatOut=%ld  
                    not supported\n",  
                    (long) __get_formatOut(somSelf, ev));  
            break;  
    }  
    somThis->numberTimesSpoke++;  
    somThat->numberTimesSpokeAll++;  
  
}  
  
SOM_Scope void SOMLINK setFontAndSize(Speaker somSelf,  
                                      Environment *ev,  
                                      string font, long  
                                      fontSize)  
{  
    SpeakerData *somThis = SpeakerGetData(somSelf);  
    SpeakerMethodDebug("Speaker", "setFontAndSize");  
    switch(__get_formatOut(somSelf, ev)) {  
        case Speaker_ascii:  
            /* do nothing. A flat  
               ascii file will not
```

```

use this info */

break;

case Speaker_postscript:
    __set_fontSize(somSelf, ev, fontSize);
__set_currentFont(somSelf, ev, font);
    break;

default:
    printf("ERROR: formatOut=%ld not supported\n",
(long)__get_formatOut(somSelf, ev));
    break;
}
}

SOM_Scope void  SOMLINK somInit(Speaker somSelf)
{
SpeakerData *somThis = SpeakerGetData(somSelf);
SpeakerMethodDebug("Speaker", "somInit");

Speaker_parent_PostScriptSpeak_somInit(somSelf);
Speaker_parent_AsciiSpeak_somInit(somSelf);
}

SOM_Scope Speaker  SOMLINK SpeakerCreate(M_Speaker somSelf
                                         Environment *ev, long speakLanguage)
{
M_SpeakerData *somThis = M_SpeakerGetData(somSelf);
Speaker returnObj = _somNew(somSelf);
M_SpeakerMethodDebug("M_Speaker", "SpeakerCreate");

__set_formatOut(returnObj, ev, speakLanguage);
_numberObjs++;
return returnObj;
}
}

```

main.c

```

Environment *ev = somGetGlobalEnvironment();

M_Speaker clsObj = SpeakerNewClass(0,0);

a = _SpeakerCreate(clsObj, ev, Speaker_ascii);
b = _SpeakerCreate(_Speaker, ev, Speaker_ascii);
c = _SpeakerCreate(_Speaker, ev, Speaker_ascii);

/* speak out in the two ways that we know how to speak */
/* which are enumerated as ascii and postscript */

for(speakOut = Speaker_ascii; speakOut < Speaker_ascii+2;
    speakOut++) {

    /* in the header to the section, print out a banner */
    /* using a percent sign which is interpreted as a comment by
    the postscript interpreter */

    printf("\n%%%%%% language = %s %%%%%%\n\n",
(speakOut==Speaker_ascii?"ASCII":"POSTSCRIPT"));

    /* now let's all the speak the same language or format; a,
    b, and c are all multi-lingual by virtue of their multiple
    inheritance */

    __set_formatOut(a, ev, speakOut);
    __set_formatOut(b, ev, speakOut);
    __set_formatOut(c, ev, speakOut);

    _speakOut(a, ev, "Object A speaks out in default
                      font");
    _setFontAndSize(a, ev, "Helvetica-Oblique", 50L);
    _speakOut(a, ev, "Object A can talk big!");
    _setFontAndSize(a, ev, "Times-Roman", 5L);
    _speakOut(a, ev, "or it can talk quite small!");
    _speakOut(b, ev, "B Speaks out"); /* now object b

                                         speaks out */
    _speakOut(c, ev, "C Speaks out"); /* and object c
                                         speaks as well */
}

_somFree(a);
_somFree(b);
_somFree(c);

return(0);
}

```

makefile

```
#####
###  MAKEFILE for multiple inheritance
###  multiple inherit from ascii and postscript
###  formatter
#####
CC = icc
HDRS = speaker.ih postscr.ih ascout.ih
OBJS = main.obj speaker.obj postscr.obj ascout.obj
TARGET = multiout.exe
SOMIR = $(SOMIR);b5.ir

.SUFFIXES : .ih .sc .lib .def .dll .idl

.c.obj:
    icc -Q -Ti+ -c $<

.idl.ih:
    sc -u $<

multiout.exe: postscr.obj ascout.obj speaker.obj main.obj
    link386 /CO /PM:VIO /NOI /NOL
$(OBJS),multiout.exe,nul,somtk;

somir: $(HDRS)
    sc -usir *.idl

hdrs: $(HDRS)

speaker.obj: speaker.c speaker.ih

speaker.ih: speaker.idl

postscr.obj: postscr.c postscr.ih

postscr.ih: postscr.idl

ascout.obj: ascout.c ascout.ih

ascout.ih: ascout.idl

main.obj: main.c speaker.ih
```

the postscript file output

```
% ____ Some useful procedures _____
/inch {72 mul} def
/fontSize 20 def
/lineIncrement 24 def
/vertpos 10 inch fontSize sub def
```

```
/horzpos 1 inch def  
  
/choosefont {findfont fontSize scalefont setfont} def  
  
/newline  
{/vertpos vertpos lineIncrement sub def % decrease vertpos  
horzpos vertpos moveto } def  
  
% ----- Beginning of your PS program -----  
  
% ----- The next few lines are some default initializations -----  
  
% - if the client program sets these values for font and  
initial position  
% - then you should remove these lines which will only be  
redundant --  
horzpos vertpos moveto  
/Times-Roman choosefont  
% ----- the rest is the result of the client program. --  
  
(Object A speaks out in default font) show newline  
/fontSize 50 def  
/lineIncrement 54 def  
/vertpos vertpos 30 sub def  
horzpos vertpos moveto  
/Helvetica-Oblique choosefont  
(Object A can talk big!) show newline  
/fontSize 5 def  
/lineIncrement 9 def  
/vertpos vertpos -45 sub def  
horzpos vertpos moveto  
/Times-Roman choosefont  
(or it can talk quite small!) show newline  
showpage
```

Try printing it and see what it looks like! These last few chapters have introduced you to some SOM concepts. Now, let's put them together to produce an object-oriented PM framework.

13

Taming PM with OOP

INTRODUCTION

When I began to program with the C Language, I fell in love with the simplicity, the conciseness, and the power that I had with C. I studied my *Kernighan and Ritchie* from the basic one line “hello world” through many of the elements of the C language. It seemed to me that every example was short and elegant, yet powerful. So much was packed into each line that many times after analyzing a short piece of code I would say to myself “Wow, that’s great!” (My vocabulary was a bit limited.) After writing applications in C for a while, I felt that there wasn’t much I couldn’t do with my knowledge of C. But then came the windowed user-interface environments and my life got so much more complicated.

My first real window environment was Presentation Manager running on OS/2 1.1. I got a new assignment and I had to learn PM. I got my Petzold book on *Programming the Presentation Manager* and I was off and running. But as many of you know, PM, like X Windows and MS Windows, is complicated stuff. My “Hello World” starter programs had exploded from my one line C program: `main(){printf("hello world");}` to a 50-line program with all new types, all new APIs and an overwhelming set of new macros.

I looked at Petzold’s “WELCOME” set of programs, and I thought to myself, “Man! This doesn’t even look like C to me!” I used to be able to look up most of the functions that I saw in *K&R*, or in an ANSI reference, and the types used in declarations used to be the familiar `int`, `char`, `float` variety. Now I was dealing with functions that they referred to as APIs for Application Programming

Interfaces, and none of the types were basic C types: int and char had been replaced by LONG, SHORT, CHAR, HWND, HPS, HDC, and many more. And what happened to the printf's and scanf's that I had used for so long?

PM gave me a lot of power that I didn't have in my full screen world, and after a while the APIs and macros began to grow on me. I also began to see how their window environment had attempted to give me an OO model for creating applications. The notions of encapsulation of data, methods organized or grouped around the types of objects on which they would be acting, and classes and subclasses were evident. The set of APIs also gave me some powerful functions in a line or two that would have taken me many lines before, giving me the added OOP feature of code reuse.

I have now been programming with OO languages for some time. I have some ideas about how I can improve my life as a PM programmer through the use of OOP in general, and through the use of SOM in particular. In the next chapters I will look at parts of the design of PM itself and try to take advantage of the OOP ideas that are already available in PM.

In the next two examples, we'll begin to try to encapsulate the PM programming paradigm within an easy-to-use SOM framework. By framework, I mean a set of classes, or a class library that will make programming in the PM world much easier. We'll begin with the basics of PM application programming and continue to add more useful examples that will help simplify the use of PM graphics output. We'll also look at how we can integrate the existing set of PM controls within our framework.

As our framework grows, you will see how we will at first only nibble around the edges of the immense PM programming model. With each example we will attempt to add more objects, encapsulating more and more of the PM functionality into easy-to-use chunks, which will further simplify the use of PM and OS/2.

IS PM OO?

Before we begin our PM examples, we'll do what all good OOP programmers do before they begin to do OOP programming: We get down on our knees and say a prayer? No, not now, that's later. What we do is we sit back and look at the job before us. We think about how we can organize PM's huge and complicated programming model. First, we describe the big picture of what the PM programming model consists of — what are the basic parts and the basic functionality? We can then examine how PM has already done some grouping and encapsulating of common function and data, and make use of its design to help build our OOP framework and examples. Let's spend a minute with an overview of the Presentation Manager.

WINDOWS AND MESSAGES

That's the big picture with PM, windows and messages. The most basic part of a PM application is the Window. Let's think of these windows as objects right from

the start. The methods that act on these objects in PM are called messages. At first we will focus on windows and messages as we begin to put together our framework. Other objects that we will create will be designed to encapsulate various parts of PM's or OS/2's functionality, like graphics, fonts, semaphores, and file I/O.

The PM Window, the Desktop Window, and Main Windows

A window is a rectangular area on a computer display. Applications use this window to receive input and to display output. OS/2 creates the *desktop window* when your PM session starts. The desktop window, also referred to as the *workplace*, is the display's background and also the ultimate ancestor of all other windows displayed in a PM application. All windows that are displayed in PM have a *parent*, except for the desktop window (Note: the desktop-object window is the only other window without a parent, but this object is non-visible), and they are arranged in a hierarchy of parent-child relationships that form a family tree or view tree with the desktop window as the root. Windows that are direct children of the desktop, are referred to as *main*, *top-level*, or *primary* windows. Every PM application has at least one main window.

Parent-child

Parent-child relationships determine how a window will appear on your display when it is drawn. Windows that have the same parent are called *siblings*; for example, all main windows are siblings. All windows are arranged, or ranked, in a three-dimensional order known as the *Z-order*. Windows that appear on the screen on top of another window are said to have a higher Z-order ranking, while the desktop always maintains the lowest position.

Rules of parent-child relationships

- All windows have one parent except the desktop and desktop-object windows, which have none. The parent must be identified at creation time, but a parent can be changed after it has been created.
- The coordinates used to create a child window are given relative to the position of the parent.
- No part of a child window ever appears outside of its parent. When a window is minimized, hidden, moved, or destroyed, all of its children are also minimized, hidden, moved, or destroyed.
- Siblings are above their parent in Z-order ranking.

Owner-owned

Windows can also have an owner. Owner-owned relationships are used in directing messages, sometimes referred to as *message-routing*. Windows usually in-

form their owners of significant events which occur in the owned window. For example, a push-button will notify its owner when it is pressed.

Rules of owner-owned relationships

- Owner and owned must be created from the same thread, thereby sharing the same message queue.
- Owner-owned and parent-child are independent relationships. A window's owner doesn't need to be its parent, but it often is. A window can own a sibling.

PM, YOUR TRAFFIC COP

You, as a user of the Presentation Manager, direct input to the various windows by using the keyboard and mouse. Keyboard input is sent to the window with the *input focus*, and mouse input is usually directed to the window currently under the mouse pointer. When I say sent or directed, I am referring to PM and how it directs input to its windows. Figure 13.1 shows us the big picture.

Your application (app) is one of the many apps to which PM is sending messages. Your app is sharing resources with these other apps and PM acts like a traffic cop ensuring that messages are routed and resources are being shared in an organized fashion. PM communicates with all the apps by way of messages.

PM Messages

The PM message is information used to communicate a request for action, a request for information, or simply a notification of an event that has occurred. The message is initiated by one of three sources:

- User
- Application
- System

A user initiates messages with the keyboard, or mouse, or some other external device, or by indirectly manipulating menus, push-buttons, etc. (once again manipulating with the mouse and keyboard). Application-initiated messages are sent from one window to another window, usually through the use of window APIs. The system sends messages based on user activities such as resizing a window, changing the system colors, or setting up timers.

The message comes to the application's message loop in the form of a set of parameters contained in a structure of type QMSG, which is defined as follows:

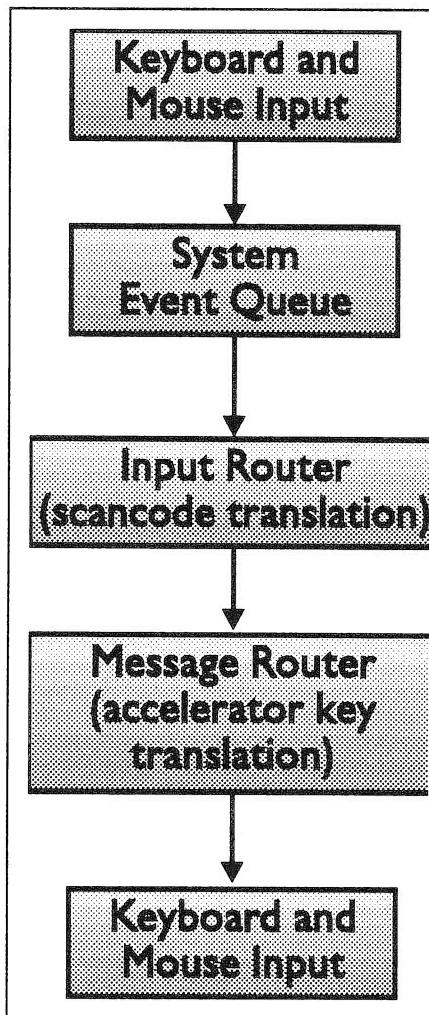


Figure 13.1. Messages in PM

```

typedef struct _QMSG {
    HWND      hwnd;
    ULONG     msg;
    MPARAM    mp1;
    MPARAM    mp2;
    ULONG     time;
    POINTL   pt1;
} QMSG;
  
```

The message is then dispatched to the window procedure for that window identified by the window handle in the HWND argument. When a window is created, it is identified as being a new window of a particular class. In OO terminology, when you create a window you are creating an object which is an instance of a

particular window class. These classes are either public window classes that are packaged with PM like WC_BUTTON, WC_ENTRYFIELD, etc., or are classes which you register using the WinRegisterClass API.

Each window class has a window procedure associated with it. When you want to send a message to a window, it is received by that window's window procedure. That procedure is of the form:

```
MRESULT windowProcedure(HWND hwnd,  
                      MSG msg,  
                      MPARAM mp1, MPARAM mp2);
```

From an OO perspective, it is as if our window objects all support just one method — its window procedure. From the PM references, you can learn how to send messages to the various window classes' window procedures. PM also provides a large set of higher level APIs that allow you to initiate action or request information without actually setting up a standard PM message of the form (hwnd, msg, mp1, mp2), for example, WinSetWindowPos, WinQueryWindow, WinSetWindowText, etc. In our examples, we will give a variety of methods that a user can use instead of a catch-all window procedure.

PM classes?

Now that we have a feel for some of the basic PM concepts, we can begin to think objectively (so to speak) about what our classes and objects will do. We have looked at PM's rules for managing windows. Most of us have noticed, as we began our PM programming, that the names of almost all procedures that act on a window begin with the *Win* prefix, and have as their first argument a window handle of type HWND. So, I'm sure it will come as no surprise to you that the first objects we deal with will be of a class patterned after the PM window called *Window*. The procedures with the *Win* prefix will form the basis for many of the methods that will act on the objects of class *Window* and its subclasses, and the PM window handle will be the key piece of instance data for a *Window* object.

However, there are other good candidates for classes in our framework other than windows. In later chapters, we'll look at graphics, for example. With PM graphics, we have a set of commonly used APIs that share a prefix, *Gpi* in this case, that almost always have as their first argument a presentation space handle (a variable of type HPS). The set of procedures with the *Gpi* prefix will form the basis for the methods that will act on a class I will call *Graphic* which uses a presentation space as its key piece of instance data.

As we go through the examples we will find that there are other classes that we can create to encapsulate other chunks of PM's functionality, or other entities within PM that we can manipulate like objects. We will look at windows, graphics, controls, fonts, colors, and semaphores, among others.

PMAPP EXAMPLE

The first example in this PM section is the PMApp program. We are going to attempt to return to the days of minimal lines of coding. I wanted a one line PM introduction program for you, but I settled for two. You may think that I'm just playing games with you because I am still programming as much as Petzold in his WELCOME world. I've even had to code more because of the added IDL definitions and methods. But as our example develops, I think you'll see how we are able improve our lot through the use of SOM.

As we begin this PM section of examples, let's first think about the basic PM program. A PM application can be thought of as having three basic parts: initialization, execution, and cleanup.

The initialization of a PM program is typically of the form:

```

HAB hab;
HMQ hmq;
HWND hwndFrame, hwndClient;
static ULONG flFrameFlags = FCF_TITLEBAR      |
                           FCF_SYSMENU     |
                           FCF_SIZEBORDER  |
                           FCF_MINMAX      |
                           FCF_SHELLPOSITION |
                           FCF_TASKLIST;

QMSG qmsg;

hab = WinInitialize(0);
hmq = WinCreateMsgQueue (hab, 0);
WinRegisterClass(
    hab,           /* anchor block handle */
    "yourClassName", /* Client Class name */
    winClientProc, /* EMClientProc window
                     procedure for class */
    CS_SIZEREDRAW, /* Class style */
    4L);           /* Extra bytes for data */
hwndFrame = WinCreateStdWindow(
    HWND_DESKTOP,
    WS_VISIBLE,
    &flFrameFlags,
    "yourClassName",
    NULL,
    0L,
    (HMODULE)0,
    0L,      /* resource ID */
    &hwndClient);

```

The cleanup usually looks like the following:

```

WinDestroyWindow(hwndFrame);
WinDestroyMsgQueue(hmq);
WinTerminate (hab);

```

The initialization and cleanup portions match nicely with the SOM methods `somInit` and `somUninit`, which are inherited from `SOMObject`, the class from which all SOM classes descend.

The execution part doesn't fit neatly into a `SOMObject` method, so we have to think a little more about this. The PM application is driven by a message loop most often in the form:

```

while(WinGetMsg(hab, &qmsg, NULLHANDLE, 0L, 0L))
    WinDispatchMsg(hab, &qmsg);

```

The `WinDispatchMsg` then results in the invocation of your application client procedure which looks something like:

```

MRESULT EXPENTRY vwClientProc(HWND hwnd, ULONG msg,
                               MPARAM mp1, MPARAM mp2)
{
    HPS hps;
    RECTL rcl;

    switch(msg) {
        case WM_PAINT:
            hps = WinBeginPaint(hwnd, NULLHANDLE, &rcl);
            WinDrawText(hps, (LONG)13, "good morning", &rcl,
                        0L, 0L,
                        DT_ERASERECT | DT_CENTER | DT_VCENTER |
                        DT_TEXTATTRS);
            WinEndPaint(hps);
            return FALSE;
        ...
    }
    return WinDefWindowProc(hwnd, msg, mp1, mp2);
}

```

There is a certain object-oriented quality to this PM model. The window class that you register during initialization is a subclass of a default PM Window class, and your `ClientProc` is essentially overriding the default PM Window procedure, `WinDefWindowProc`. Since you want to preserve most of the default behaviour of the default window class, you call your parent's method `WinDefWindowProc` for most messages. You process only one message in this example.

The first example is intended to just get our feet wet in PM. Wait for PM Example Two to put our knowledge of windows and messages to work for us. In this example, we are creating a simple one-class hierarchy derived from `SOMObject`. We will place initialization and cleanup code into `somInit` and `somUninit`, and we will also tuck away our message loop (`WinGetMsg` ... `Win-`

DispatchMsg..) into somInit. By doing this we are passing off control to PM for the duration of our app right when we instantiate PMApp for the first time. In fact, somInit won't ever end until you are all done, so we had better call our parent_somInit before we get ourselves into this loop.

The first example is not very exciting, but it will help us to begin building a set of classes that will help us manage the PM model. In Example One, we will examine the following topics:

SOM

1. Initializing with somInit.
2. Cleanup using somUninit.

PM

1. PM Initialization
2. PM Message Loop
3. PM Shutdown

The files needed in this example are:

pmapp.idl
pmapp.c
runpm.c

PMAPP.IDL

```
#include <somobj.idl> // get definition of your parent

typedef somToken HWND; // get rid of the SOM
                      // compiler warnings
typedef somToken HAB;
typedef somToken HMQ;

interface PMApp: SOMObject
{
    attribute HWND hwndFrame; /* make these
                               readonly eventually */
    attribute HWND hwndClient;
    attribute HAB hab;
    attribute HMQ hmq;
    attribute string className;
```

```

#ifndef __SOMIDL__
implementation
{
    // Class Modifiers
    releaseorder:
        _get_hwndFrame, _set(hwndFrame,
        _get_hwndClient, _set(hwndClient,
        _get_hab, _set_hab,
        _get_hmq, _set_hmq;

    passthru C_h_before =
        "#define INCL_WIN"
        "#include <os2.h>"
    ;

    //# Method Modifiers
    somInit: override;
    somUninit: override;
};

#endif

};

```

PMAPP.C

```

#define PMApp_Class_Source
#include <pmapp.ih>

FNWP vwClientProc;
/*
 *  override the method 'somInit' which is inherited from
 *  SOMObject
 */
SOM_Scope void  SOMLINK somInit(PMApp somSelf)
{
    PMAppData *somThis = PMAppGetData(somSelf);
    Environment *ev = somGetGlobalEnvironment();
    short i;
    static ULONG flFrameFlags = FCF_TITLEBAR | FCF_SYSMENU |
                                FCF_SIZEBORDER | FCF_MINMAX |
                                FCF_SHELLPOSITION |
                                FCF_TASKLIST;
    HWND hwndTitleBar;
    QMSG qmsg;
    static BOOL firsttimein=TRUE;

    _className = "Good Morning SOM";

    if(firsttimein) {
        _hab = WinInitialize(0);
        _hmq = WinCreateMsgQueue (_hab, 0);

```

```

        WinRegisterClass(
            _hab,           /* anchor block handle */
            _className,     /* Client Window Class name */
            vwClientProc,  /* window procedure for class */
            CS_SIZEREDRAW, /* Class style */
            4L);           /* Extra bytes for data */

        firsttimein=FALSE;
    }
    _hwndFrame = WinCreateStdWindow(
        HWND_DESKTOP,
        WS_VISIBLE,
        &flFrameFlags,
        _className,
        NULL,
        0L,
        (HMODULE) 0,
        0L,      /* resource ID */
        &_hwndClient);
    if((hwndTitleBar =
        WinWindowFromID (_hwndFrame, FID_TITLEBAR)) != 0L)
        WinSetWindowText (hwndTitleBar, _className);

    PMApp_parent_SOMObject_somInit(somSelf);

    while(WinGetMsg(_hab, &qmsg, NULLHANDLE, 0L, 0L))
        WinDispatchMsg(_hab, &qmsg);
}

SOM_Scope void SOMLINK somUninit(PMApp somSelf)
{
    PMAppData *somThis = PMAppGetData(somSelf);

    WinDestroyWindow(_hwndFrame);
    WinDestroyMsgQueue(_hmq);
    WinTerminate (_hab);
    PMApp_parent_SOMObject_somUninit(somSelf);
}

MRESULT EXPENTRY vwClientProc(HWND hwnd, ULONG msg,
                           MPARAM mp1, MPARAM mp2)
{
    HPS hps;
    RECTL rcl;

    switch(msg) {
        case WM_PAINT:
            hps = WinBeginPaint(hwnd, NULLHANDLE, &rcl);

```

```

        WinDrawText(hps, (LONG)11, "SOM is cool", &rcl,
                     0L, 0L, DT_ERASERECT | DT_CENTER |
                     DT_VCENTER | DT_TEXTATTRS);
        WinEndPaint(hps);
        return FALSE;
    }
    return WinDefWindowProc(hwnd, msg, mp1, mp2);
}

```

RUNPM.C

```

#include <pmapp.h>
void main(int argc, char *argv[])
{
    PMApp obj = PMAppNew();
    _somFree(obj);
}

```

MAKEFILE — for PMApp number one

```

LIB = $(LIB)
INCLUDE = .;$(INCLUDE)
SMEMIT = ih;h;c;

install: runpm.exe

run:
    runpm

.c.obj:
    icc -Q -Ti+ -c $<
# note the new link option for PM: /PM:PM
# use /PM:VIO for non-PM apps and /PM:PM for PM apps
runpm.exe: pmapp.obj runpm.obj
    link386 /NOL /NOI /CO /PM:PM runpm.obj pmapp.obj,
              runpm.exe,nul,somtk;

pmapp.ih: pmapp.idl
    sc pmapp

pmapp.obj: pmapp.ih

runpm.obj: runpm.c

```

Ok, so we haven't accomplished much in example one. But we have gotten up and running in a PMApp, and we have made it very easy for the user of our class to get a PM application on the screen.. Let's move quickly on to the second PM example to see how we can begin making our basic application more object-oriented.

Questions

1. I thought I could give you a one line main{} procedure to run this, but I got a strange side effect when I did. Explain why the following client program doesn't work quite right. Try it out to see the side effect.

```
main()
{
    _somFree(PMAppNew());
}
```

14

PMApp Divided Into Classes

INTRODUCTION

The only interesting work being done in our first program occurs in a PM Client procedure and not in a SOM method. As mentioned in example one, the PM programmer writes a Client procedure then passes the message to the `WinDefWindowProc()` in a way that resembles the way an overriding method might work. This is because overriding methods often call their parent(s) before or after they perform their own work. This client procedure, however, is not a typical method you will see in OO systems. This one does a lot more than a typical method will do in OOP. Our client procedure does everything from key handling to mouse handling, focus management to size negotiation, painting/drawing to button command messages, and initialization to destruction. In OOP, a method is normally a little more focused in its mission.

In Example Two, we will begin by creating a class hierarchy. First, PMApp will be separated into three classes: Window, Client, and PMApp. Window will be the class that can handle attributes and behaviour common to all windows. Client will be the class that an application programmer can override to create his application window. The Client class will take our catch-all `ClientProc` of the last example and begin converting its multi-functionality into a series of SOM methods. We are also going to begin making a distinction between the framework class and the application class. The framework must be usable by many PM developers, and the applications are specific programs of one of those

PM developers, using the framework. We will break our previous example into a class hierarchy of three classes. PMApp is the application class.

The hierarchy for Example Two is slightly more complicated. YourApp inherits from Client which inherits from Window. The source for these classes follows:

The Window Class

The Window class introduces methods that can be performed on all windows. I have not given a client user much ability, as yet, to do output. The WM_PAINT case of our client procedure has been hardcoded to print out some hokey message. The only way we have to perform output currently is by setting the text. The _set_text() routine has been programmed so that it updates the title bar, if the window has a frame with a title bar for its parent. Note that the no-data option is used to modify my attribute text. This is done to avoid having SOM's default _get_/_set_ methods store a simple string. In this case, I want to implement my own _set_/_get_ methods and to let PM store that string.

WINDOW.IDL

```
#include <somobj.idl> // get parent definition
#include <somcls.idl> // parent of the metaclass

typedef somToken HWND; // get rid of the SOM
// compiler warnings
typedef somToken HAB;
typedef somToken HMQ;
typedef somToken HPS;
typedef somToken RECTL;

interface Window: SOMObject // Window is a subclass
    // of SOMObject
{
    attribute string text;
    attribute long x;
    attribute long y;
    attribute long w;
    attribute long h;
    attribute HWND hwndFrame;
    attribute HWND hwndWindow;
    attribute HWND hwndParent;
    attribute HWND hwndOwner;
    attribute HAB hab;
    attribute HMQ hmq;

    short UniqueID();
    Window WindowFromID(in short ID);

    void ProcessEvents();
}
```

```

void SetSize(in long x, in long y,
             in long w, in long h);

#endif __SOMIDL__
implementation {

    //# Class Modifiers
    releaseorder:
        _get_text, _set_text,
        _get_x, _set_x,
        _get_y, _set_y,
        _get_w, _set_w,
        _get_h, _set_h,
        _get_hwndFrame, _set(hwndFrame,
        _get_hwndWindow, _set(hwndWindow,
        _get_hwndParent, _set(hwndParent,
        _get_hwndOwner, _set(hwndOwner,
        _get_hab, _set_hab,
        _get_hmq, _set_hmq,
        UniqueID, WindowFromID, ProcessEvents, SetSize;

    //# Method Modifiers
    somInit: override;
    somUninit: override;

    passthru C_h_before =
        "#define INCL_WIN"
        "#include <os2.h>"
    ;

    passthru C_h_after =
        " extern Window gBaseWindow;""
    ;

    //# Data Modifiers
    text:           nodata;

};

#endif /* __SOMIDL__ */
};

```

WINDOW.C

```

#define Window_Class_Source
#include <window.ih>

#include <stdio.h>

Window gBaseWindow;

```

```

/*
 * Method from the IDL attribute statement:
 * "attribute string text"
 */
SOM_Scope string SOMLINK _get_text(Window somSelf,
                                    Environment *ev)
{
    WindowData *somThis = WindowGetData(somSelf);
    string textStr;
    long textLen;

    if(_hwndWindow) {
        textLen =
            WinQueryWindowTextLength(_hwndWindow)+1;
        textStr = (string)malloc(
            WinQueryWindowTextLength(_hwndWindow)+1);
        WinQueryWindowText (_hwndWindow, textLen,
                            textStr);
    }

    return textStr;
}

/*
 * Method from the IDL attribute statement:
 * "attribute string text"
 */
SOM_Scope void SOMLINK _set_text(Window somSelf,
                                  Environment *ev, string text)
{
    WindowData *somThis = WindowGetData(somSelf);
    if(_hwndWindow) {
        WinSetWindowText (_hwndWindow, text);
    }
    if(somSelf==gBaseWindow && _hwndFrame)
        WinSetWindowText (_hwndFrame, text);
}

/*
 * A primitive unique id generator
 */
SOM_Scope short SOMLINK UniqueID(Window somSelf, Environment
*ev)
{
    static short id=0;
    return (++id);
}

/*
 * this method will return the Window pointer of the
 * SOMobject identified by ID if somSelf is the parent of

```

```

    * that window
 */
SOM_Scope Window  SOMLINK WindowFromID(Window somSelf,
                                         Environment *ev,
                                         short ID)
{
    WindowData *somThis = WindowGetData(somSelf);
    Window w = NULL;

    WindowMethodDebug ("Window", "WindowFromID");

    if (_hwndWindow) {
        HWND hwndChild;
        hwndChild = WinWindowFromID (_hwndWindow, ID);
        w = (Window )WinQueryWindowPtr(hwndChild, QWL_USER);
    }
    return w;
}

SOM_Scope void  SOMLINK ProcessEvents(Window somSelf,
                                         Environment *ev)
{
    WindowData *somThis = WindowGetData(somSelf);
    QMSG qmsg;
    while(WinGetMsg (_hab, &qmsg, NULLHANDLE, 0L, 0L))
        WinDispatchMsg (_hab, &qmsg);
}

SOM_Scope void  SOMLINK SetSize(Window somSelf,
                                 Environment *ev,
                                 long x, long y,
                                 long w, long h)
{
    WindowData *somThis = WindowGetData(somSelf);

    if (_hwndWindow) {
        fprintf(stderr, "Set Size: hwndWindow=%lx %ld,%ld
%ldx%ld\n", _hwndWindow, x, y, w, h);
        WinSetWindowPos (_hwndWindow,
                         HWND_TOP,
                         x, y, w, h,
                         SWP_MOVE | SWP_SIZE |
                         SWP_ZORDER);
    }
    _x = x;
    _y = y;
    _w = w;
    _h = h;

    fprintf(stderr, "Set Size: end\n");
}

```

```

SOM_Scope void  SOMLINK somInit(Window somSelf)
{
    WindowData *somThis = WindowGetData(somSelf);
    Environment *ev = somGetGlobalEnvironment();
    static BOOL firsttimein=TRUE;

    if(firsttimein) {
        _hab = WinInitialize(0);
        _hmq = WinCreateMsgQueue (_hab, 0);
        gBaseWindow = somSelf;
        firsttimein=FALSE;
    }
    _x = _y = _w = _h = 0L;

    Window_parent_SOMObject_somInit(somSelf);
}

SOM_Scope void  SOMLINK somUninit(Window somSelf)
{
    WindowData *somThis = WindowGetData(somSelf);

    WinDestroyWindow(_hwndFrame);
    WinDestroyMsgQueue(_hmq);
    WinTerminate (_hab);

    Window_parent_SOMObject_somUninit(somSelf);
}

```

The Client Class

As you can see, in the last example we had a hidden `ClientProc` that acted as our only real method. We now have a few methods that a user can use to interact with the graphical user interface.

<u>previous</u>	<u>is now accessed subclassing this method:</u>
WinClientProc	Button1Click(in short x, in short y)

Of course, this brand new method only gives us access to the PM message type `WM_BUTTON1CLICKED`, so with this design we will have many methods to override if we want to do something with many of the available PM messages.

`Button1Click` is one of many event handler methods that will be introduced. There is a separate method for each event type (sometimes two or three) in these examples, because my list of methods is fairly short and manageable. As your client class grows to handle more types of messages, you may find it easier to have fewer *handler* methods that specialize in a special type of message. For example, all mouse events could be sent to the `MouseHandler()` method, or keyboard events could be sent to a method like `KeyboardHandler()`. Many class libraries in the industry also turn the messages and

events themselves into objects. See IBM's SOM Toolkit, or IBMClass (packaged with IBM's C++ compiler) for an example of an Event hierarchy.

client.idl

```
#include <window.idl> // get definition of your parent

interface Graphic;

interface Client: Window // your classname and parent(s)
{
//# Event Handler Methods
    void Button1Click(in short x, in short y); /* x,y when
                                                 * pressed */

#ifndef __SOMIDL__
implementation
{
    //## Class Modifiers
    releaseorder:
        Button1Click;

    //## Method Modifiers
    somInit: override;
    somUninit: override;
};

#endif
};
```

client.c

```
#define Client_Class_Source
#include <client.ih>

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

FNWP vwClientProc;

/* I inherit somInit from SOMObject */
SOM_Scope void  SOMLINK somInit(Client somSelf)
{
/*   ClientData *somThis = ClientGetData(somSelf); */
    Environment *ev = somGetGlobalEnvironment();
    short i;
    static ULONG flFrameFlags = FCF_TITLEBAR      |
                                FCF_SYSMENU     |
```

```

        FCF_SIZEBORDER |  

        FCF_MINMAX |  

        FCF_SHELLPOSITION |  

        FCF_TASKLIST;  

        HWND hwndTitleBar,  

        hwndFrame,  

        hwndClient;  

        HAB hab;  

        static BOOL firsttimein=TRUE;  

        string className="Client App";  

        Client_parent_Window_somInit(somSelf);  

        hab = __get_hab(somSelf, ev);  

        if(firsttimein) {  

            WinRegisterClass(  

                hab,      /* anchor block handle */  

                className, /* Window Client name */  

                vwClientProc, /* Client procedure */  

                CS_SIZEREDRAW, /* Class style */  

                4L);          /* Extra bytes for data like  

                                HPS */  

            firsttimein = FALSE;  

        }  

        hwndFrame = WinCreateStdWindow(  

            HWND_DESKTOP,  

            WS_VISIBLE,  

            &flFrameFlags,  

            className,  

            NULL,  

            0L,  

            (HMODULE)0,  

            0L,      /* resource ID */  

            &hwndClient);  

        if((hwndTitleBar = WinWindowFromID(hwndFrame,  

            FID_TITLEBAR)) != 0L)  

            WinSetWindowText(hwndTitleBar, className);  

        WinSetWindowPtr(hwndClient, 0L, somSelf);  

        __set_hwndWindow(somSelf, ev, hwndClient);  

        __set_hwndFrame(somSelf, ev, hwndFrame);  

    }

```

```

SOM_Scope void  SOMLINK somUninit(Client somSelf)
{
/*   ClientData *somThis = ClientGetData(somSelf); */
    Environment *ev = somGetGlobalEnvironment();
    Client_parent_Window_somUninit(somSelf);
}

/*
 * x,y when pressed
 */
SOM_Scope void  SOMLINK Button1Click(Client somSelf,
                                      Environment *ev, short x, short y)
{ }

/*
 * x,y when clicked
 */
MRESULT EXPENTRY vwClientProc(HWND hwnd, ULONG msg,
                               MPARAM mp1, MPARAM mp2)
{
    HPS hps;
    RECTL rcl;
    char key;
    short vkey, sFlag, kbState;
    Environment *ev = somGetGlobalEnvironment();
    SOMObject somSelf;
    short x = MOUSEMSG(&msg)->x;
    short y = MOUSEMSG(&msg)->y;
    short w, h;
    SHORT src;
    Window somWin;
    SHORT winID;
    HWND winHwnd;
    static BOOL button1motion=FALSE;

    if(hwnd)
        somSelf = WinQueryWindowPtr(hwnd, 0);

    switch(msg) {
        case WM_BUTTON1CLICK:
            _Button1Click(somSelf, ev, x, y);
            break;
        case WM_PAINT:
            if (somSelf) {
                hps = WinBeginPaint(hwnd,
                                    NULLHANDLE, &rcl);
                WinDrawText(hps, 39,
                            "SOM PM Objects! Click Mouse Button One!",
                            &rcl, 0L, 0L,
                            DT_ERASERECT | DT_CENTER |
                            DT_VCENTER | DT_TEXTATTRS);
            }
    }
}

```

```

                WinEndPaint(hps);
                return FALSE;
            }
            break;
        }
        return WinDefWindowProc(hwnd, msg, mp1, mp2);
    }
}

```

The App1 Class

Next is the application. This is the view that the user of our framework will have as he writes an application. This is a simple class to design and implement. This App1 class will give us objects that keep track of the number of mouse clicks which occur in the application client area. As mouse button 1 is clicked we update the string attribute of the App1 object.

app1.idl

```

#include <client.idl> // get definition of your parent

interface App1: Client // App1 is a subclass of Client
{
    attribute short timesClicked;

#ifndef __SOMIDL__
implementation
{
    //## Class Modifiers
    releaseorder:
        _get_timesClicked, _set_timesClicked;

    //## Method Modifiers
    Button1Click: override;

    somInit: override;
};
#endif
};

```

app1.c

```

#define App1_Class_Source

#include <app1.ih>
#include <stdio.h>

static char appString[30];

```

```

SOM_Scope void  SOMLINK Button1Click(App1 somSelf,
                                      Environment *ev,
                                      short x, short y)
{
    App1Data *somThis = App1GetData(somSelf);
    _timesClicked++;
    sprintf(appString,"times clicked=%d", _timesClicked);
    __set_text(somSelf, ev, appString);

}

SOM_Scope void  SOMLINK somInit(App1 somSelf)
{
    App1Data *somThis = App1GetData(somSelf);
    Environment *ev = somGetGlobalEnvironment();

    App1_parent_Client_somInit(somSelf);
    _timesClicked = 0;
    sprintf(appString,"Click Mouse Button 1. Current times
clicked is %d", _timesClicked);
    __set_text(somSelf, ev, appString);
}

```

The client program is still a bare bones launcher which merely instantiates an App1 object and then calls the process loop. All the work is then handled by the App1 class.

test3.c

```

#include <app1.h>
void main(int argc, char *argv[])
{
    App1 *obj = App1New();
    _ProcessEvents(obj, somGetGlobalEnvironment());
    _somFree(obj);
}

```

makefile

```

LIB = $(LIB)
INCLUDE = .;$(INCLUDE)
SMEMIT = ih;h;c;
SRCS = window.c client.c app1.c test2.c
OBJS = $(SRCS:.c=.obj)

install: test2.exe

.SUFFIXES : .ih .lib .def .dll .idl

.c.obj:

```

```
icc -Q -Ti+ -c $<  
.idl.ih:  
    sc $*  
  
test2.exe: window.obj client.obj app1.obj test2.obj  
    link386 /NOL /NOI /CO /PM:PM test2.obj client.obj  
    window.obj app1.obj, test2.exe, nul, somtk;  
  
window.ih: window.idl  
window.obj: window.c window.ih  
  
client.ih: client.idl  
client.obj: client.c client.ih  
  
app1.obj: app1.c app1.ih  
app1.ih: app1.idl  
  
test2.obj: test2.c
```

On our way

Now we have the start of a class hierarchy that wraps a lot of PM's functionality and gives the user an easier programming model. We will expand on this basic window model when we spend some more time wrapping controls in a later chapter, but first, let's look at some more PM basics. The next example explores some of PM's rich graphics APIs.

15

Graphics: Boxes on your Client

THE PM GRAPHICS INTERFACE

In our first PM examples, we used a handy little PM routine, WinDrawText, to put text on the screen. There are a limited number of Win-prefixed functions that allow you to do output, but the true PM programmer doesn't use Win... methods to put output on the screen. No sirree! The real PM programmer uses APIs with the Gpi prefix!

The OS/2 PM Graphics Programming Interface (GPI) is a set of over 200 functions that allows a user to do everything from drawing lines, polygons, arcs, patterns, regions, and markers, to doing fancy output with colors, fonts, and bitmaps. The GPI gives you power to do more things than most of you will ever care about, and it's easy to become overwhelmed with the sophistication of the interface. Most of you might want to change the color to something more interesting than your default colors, but you don't want to do it if it means setting up logical color tables and using 1 of 256-cubed RGB options. Or some of you may want to vary the font a little, but you will look in vain for the call that looks like `GpiSetFont ("TimesRoman12", Italic)`; It's not out there! Instead you will have to learn to deal with imposing `FONTMETRICS` and `FATTR` structures, and query the available fonts and create your own logical font using the routines `GpiQueryFonts` and `GpiCreateLogicalFont`.

So let's start at the top. You say you want to just draw a line or maybe a box, and you would like to display text. We will create a class called `Graphic`. This class's objects will store the attributes and will provide the methods that we

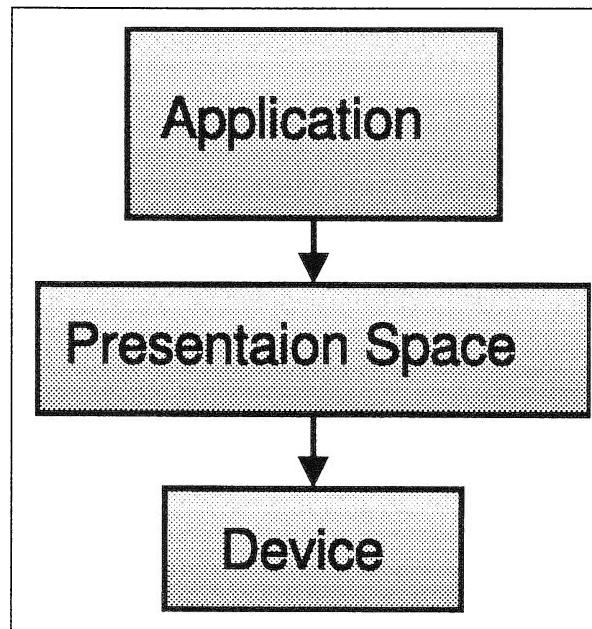


Figure 15.1. Presentation Spaces

need to use the GPI. As we create this class, we need to know the key elements of Gpi... programming.

The Presentation Space

The first and most important concept you need to know about is the presentation space. The presentation space is the first parameter of most of your Gpi calls. It is a large data structure that contains the relevant information needed to perform your graphics output. Everything from the current color and current font to the line width and current position of your graphics drawings. It also represents your drawing area. You work with the presentation space, and the presentation space must then be linked or *associated* with a device context to which your output will be displayed or stored. Figure 15.1 shows us the role of a presentation space.

Presentation Spaces

Presentation spaces come in 3 types: *standard micro*, *cached micro*, and *normal*. By far the most common presentation spaces used are the micro varieties, and between those two the most commonly used is the cached micro. The normal presentation space is needed if you want to use the same presentation space to send your output to multiple devices, like your display and printer. The normal PS is also needed for the complex-segment and retained-graphics drawings.

In our next example, we will use the easiest of our options, the cached micro presentation space. OK, so I'm not as macho as I led you to believe in my opening comments about using `Gpi` instead of the wimpy `Win` calls. Cached micro will serve our purposes just fine. These presentation spaces are easy to use because the window manager does some of the dirty work for us. For example, you don't have to associate your presentation space with a device context, the window manager associates it with your display screen. You also don't have to disassociate your `PS` from the window device context, you just release the presentation space and the window system does that for you.

This type of presentation space is given to you temporarily by the window system. You acquire it with a simple request, you do your output and then you quickly give it back by releasing it. The cache space is then available to be used by other applications. These cached micro presentation spaces yield the best system performance, because they are not permanently allocated to your application. Their usage is a bit cumbersome, however, because you must repeatedly initialize your presentation space to your desired attributes with every new `WM_PAINT` message or every time you acquire a new one. And you can't just get a cached micro presentation space handle and hang on to it indefinitely; the window system doesn't allow that.

Our `Graphic` class will assist you by hiding the presentation space acquisition and the repeated initialization. We will have some class attributes and methods that have counterparts in the `GPI`, but we will not attempt to wrap the entire `GPI` class.

In this example, we will use the three classes of the previous example, and add to it our `Graphics` class, as in figure 15.2.

Why do I attach the `graphic` object to the `Client` class and not the `Window` class itself? Don't all windows need to use graphics to perform output? We put the `Graphic` object at the `Client` level, because at that level, a PM Programmer may need to resort to performing his own graphics. Many of the windows that you use will be prepackaged windows provided by PM. These prepackaged windows, called PM Controls, will take care of their own graphics.

This example is a rudimentary wire box drawing program. The application subclasses the methods it needs from the `Client` class to do its box drawing. It also subclasses the `CharPress` method to allow users to fill the drawn wire box by typing the letter 'F' or 'f'.

The files you will need for this example are:

<code>window.idl</code>	The Window class definition
<code>window.c</code>	The Window class implementation
<code>client.idl</code>	The Client class definition
<code>client.c</code>	The Client class implementation
<code>graphic.idl</code>	The Graphic class definition
<code>graphic.c</code>	The Graphic class implementation
<code>pmapp.idl</code>	The PMApp class definition
<code>pmapp.c</code>	The PMApp class implementation
<code>makefile</code>	

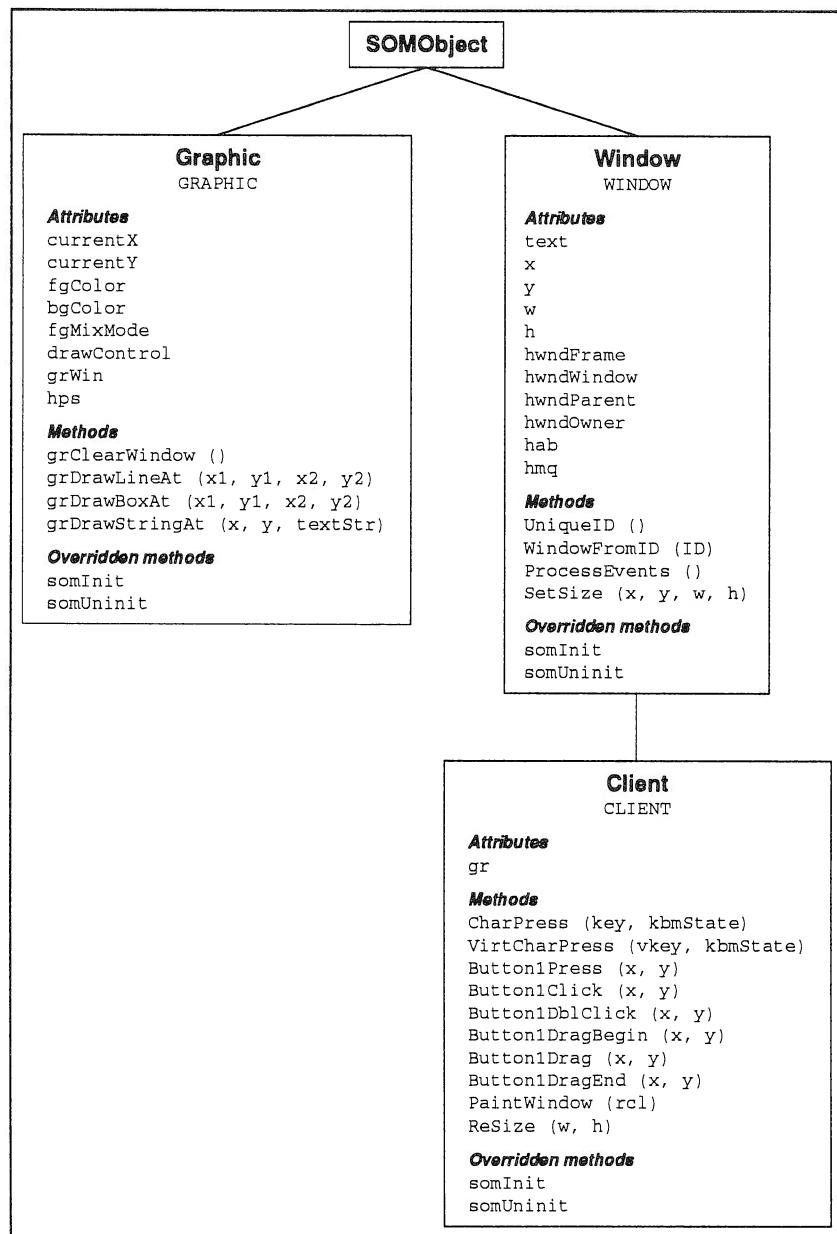


Figure 15.2. Graphics Hierarchy

WINDOW.IDL and **WINDOW.C** are the same as in the last example.

/* see previous example for a source listing */

Client

We have expanded the `Client` class that we introduced in the previous example. Our `Client` class now supports several handler methods that our application can use.

```
CharPress, VirtCharPress,  
Button1Press, Button1Click, Button1DblClick,  
Button1DragBegin, Button1Drag, Button1DragEnd,  
PaintWindow, ReSize;
```

If you look at the source code for `client.c`, you will notice that the bodies for all these *handler* methods are empty. This technique allows us to implement the notion of *virtual* methods, which are methods designed to be overridden by subclasses. The client implementation will dutifully call each of these virtual methods when the appropriate PM action takes place, but it is up to the subclasses of the `Client` class to do something with this method.

We also introduce an attribute, `gr`, which is an object of the class `Graphic`. Every client thus has a `graphics` object associated with it, and applications that subclass from `Client` automatically have a `Graphic` object they can use to do graphical output.

CLIENT.IDL

```
#include <window.idl> // get definition of your parent  
  
interface Graphic;  
  
interface Client: Window // your classname and parent(s)  
{  
    attribute Graphic gr;  
  
    //# Event Handler Methods  
    void CharPress(in char key, in short kbmState);  
    void VirtCharPress(in short vkey, in short kbmState);  
  
    void Button1Press(in short x, in short y);  
    void Button1Click(in short x, in short y);  
    void Button1DblClick(in short x, in short y);  
    void Button1DragBegin(in short x, in short y);  
    void Button1Drag(in short x, in short y);  
    void Button1DragEnd(in short x, in short y);  
  
    void PaintWindow(in RECTL *rcl);  
    void ReSize(in short w, in short h);  
  
#ifdef __SOMIDL__  
implementation  
{
```

```

//# Class Modifiers
releaseorder:
    _get_gr, _set_gr,
    CharPress, VirtCharPress,
    Button1Press, Button1Click, Button1DblClick,
    Button1DragBegin, Button1Drag, Button1DragEnd,
    PaintWindow, ReSize;

//# Method Modifiers
somInit: override;
somUninit: override;
};

#endif

};

```

CLIENT.C

```

#define Client_Class_Source
#include <client.ih>

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <graphic.h>

FNWP vwClientProc;

/* I inherit somInit from SOMObject */
SOM_Scope void SOMLINK somInit(Client somSelf)
{
    ClientData *somThis = ClientGetData(somSelf);
    Environment *ev = somGetGlobalEnvironment();
    short i;
    static ULONG flFrameFlags = FCF_TITLEBAR | FCF_SYSMENU |
                                FCF_SIZEBORDER | FCF_MINMAX |
                                FCF_SHELLPOSITION | FCF_TASKLIST;
    HWND hwndTitleBar, hwndFrame, hwndClient;
    HAB hab;
    static BOOL firsttimein=TRUE;
    string className="Client App";

    Client_parent_Window_somInit(somSelf);

    hab = __get_hab(somSelf, ev);
    _gr = GraphicNew();

    if(firsttimein) {

```

```
WinRegisterClass(
    hab,      /* anchor block handle */
    className, /* Window Client Class name */
    vwClientProc, /* Client procedure for class */
    CS_SIZEREDRAW, /* Class style */
    4L);          /* Extra bytes for app data */
    firsttimein = FALSE;
}
hwndFrame = WinCreateStdWindow(
    HWND_DESKTOP,
    WS_VISIBLE,
    &f1FrameFlags,
    className,
    NULL,
    0L,
    (HMODULE)0,
    0L,      /* resource ID */
    &hwndClient);
if ((hwndTitleBar = WinWindowFromID (hwndFrame,
FID_TITLEBAR)) != 0L)
    WinSetWindowText (hwndTitleBar, className);

    WinSetWindowPtr(hwndClient, 0L, somSelf);

/* the graphic object needs to know its associated window */
    __set_grWin(__gr, ev, somSelf);

    __set_hwndWindow(somSelf, ev, hwndClient);
    __set_hwndFrame(somSelf, ev, hwndFrame);
}

SOM_Scope void  SOMLINK somUninit(Client somSelf)
{
    ClientData *somThis = ClientGetData(somSelf);
    Environment *ev = somGetGlobalEnvironment();
    Client_parent_Window_somUninit(somSelf);
}

SOM_Scope void  SOMLINK CharPress(Client somSelf, Environment
*ev,
        char key,
        short kbmState)
{ }

SOM_Scope void  SOMLINK VirtCharPress(Client somSelf,
Environment *ev,
        short vkey, short kbmState)
{ }
```

```
/*
 * x,y when pressed
 */
SOM_Scope void SOMLINK Button1Press(Client somSelf,
                                      Environment *ev,
                                      short x, short y)
{ }

/*
 * x,y when pressed
 */
SOM_Scope void SOMLINK Button1Click(Client somSelf,
                                      Environment *ev,
                                      short x, short y)
{ }

/*
 * x,y when pressed
 */
SOM_Scope void SOMLINK Button1DblClick(Client somSelf,
                                         Environment *ev,
                                         short x, short y)
{ }

SOM_Scope void SOMLINK Button1DragBegin(Client somSelf,
                                         Environment *ev,
                                         short x, short y)
{ }

SOM_Scope void SOMLINK Button1Drag(Client somSelf,
                                    Environment *ev,
                                    short x, short y)
{ }

SOM_Scope void SOMLINK Button1DragEnd(Client somSelf,
                                       Environment *ev,
                                       short x, short y)
{ }

SOM_Scope void SOMLINK PaintWindow(Client somSelf,
                                    Environment *ev,
                                    RECTL* rcl)
{ }

SOM_Scope void SOMLINK ReSize(Client somSelf,
                             Environment *ev,
                             short w, short h)
{ }

MRESULT EXPENTRY vwClientProc(HWND hwnd, ULONG msg,
```

```
MPARAM mp1, MPARAM mp2)

{

    HPS hps;
    RECTL rcl;
    char key;
    short vkey, sFlag, kbmState;
    Environment *ev = somGetGlobalEnvironment();
    SOMObject somSelf;
    short x = MOUSEMSG(&msg)->x;
    short y = MOUSEMSG(&msg)->y;
    short w, h;
    SHORT src;
    Window somWin;
    SHORT winID;
    HWND winHwnd;
    static BOOL button1motion=FALSE;

    if(hwnd)
        somSelf = WinQueryWindowPtr(hwnd, 0);

    switch(msg) {
        case WM_CHAR:
            sFlag = CHARMSG(&msg)->fs;
            if(sFlag & KC_KEYUP)
                return FALSE;
            if(sFlag & KC_CHAR) {
                key = CHARMSG(&msg)->chr;
                _CharPress(somSelf, ev, key, 0);
            } else if(sFlag & KC_VIRTUALKEY) {
                vkey = CHARMSG(&msg)->vkey;
                _VirtCharPress(somSelf, ev, vkey, 0);
            }
            break;
        case WM_BUTTON1DOWN:
            _Button1Press(somSelf, ev, x, y);
            break;
        case WM_BUTTON1CLICK:
            _Button1Click(somSelf, ev, x, y);
            break;
        case WM_BUTTON1DBLCLK:
            _Button1DblClick(somSelf, ev, x, y);
            break;
        case WM_MOUSEMOVE:
            if(button1motion) {
                x = MOUSEMSG(&msg)->x;
                y = MOUSEMSG(&msg)->y;
                _Button1Drag(somSelf, ev, x, y);
            }
            break;
        case WM_BUTTON1MOTIONSTART:
            button1motion=TRUE;
```

```

        _Button1DragBegin(somSelf, ev, x, y);
        fprintf(stderr, "WM_BUTTON1MOTIONSTART\n");
        break;
    case WM_BUTTON1MOTIONEND:
        button1motion=FALSE;
        _Button1DragEnd(somSelf, ev, x, y);
        fprintf(stderr, "WM_BUTTON1MOTIONEND\n");
        break;
    case WM_SIZE:
        if(somSelf) {
            w = SHORT1FROMMP(mp2);
            h = SHORT2FROMMP(mp2);
            __set_w(somSelf, ev, w);
            __set_h(somSelf, ev, h);
            _ReSize(somSelf, ev, w, h);
        }
        break;
    case WM_PAINT:
        if (somSelf) {
            hps = WinBeginPaint(hwnd, NULLHANDLE, &rcl);
            _PaintWindow(somSelf, ev, &rcl);
            WinEndPaint(hps);
            return FALSE;
        }
        break;
    }
    return WinDefWindowProc(hwnd, msg, mp1, mp2);
}

```

The Graphic Class

Next we introduce the `Graphic` class. Note how we call `InitGraphic` and `QuitGraphic` before and after every graphics method.

```

void InitGraphic(GraphicData *somThis)
{
    Environment *ev = somGetGlobalEnvironment();
    POINTL pt;
    HWND hwndGraphic = __get_hwndWindow(_grWin, ev);

    pt.x = _currentX;
    pt.y = _currentY;
    _hps = WinGetPS(hwndGraphic);
    GpiMove(_hps, &pt);
    GpiSetColor(_hps, _fgColor);
    GpiSetBackColor(_hps, _bgColor);
    GpiSetMix (_hps, _fgMixMode);
}

void QuitGraphic(GraphicData *somThis)

```

```

{
    WinReleasePS(_hps);
}

```

This is how we can hide some of the bothersome initialization and administration that a programmer needs to do when using the PM GPI set of calls with the cached micro presentation spaces. A user can set attributes and know that they will stay constant until he changes them.

Fonts, like graphics in general, also must be initialized each time a new presentation space is acquired. A similar initialization scheme could work for initializing fonts to a current font, which the user could set up at one time and then use until the user initiates the change. Petzold's book, *Programming the Presentation Manager*, contains a good example on how such a font initialization routine might look in his coverage of fonts. The utilities he introduces are called *EzfQueryFonts* and *EzfCreateLogFont*.

The attributes introduced in the graphics object are patterned after the PM attributes that can be used to modify graphics in PM.

`currentX, currentY, fgColor, bgColor, fgMixMode, drawControl,`

The methods we introduce in `Graphic` are also patterned roughly after the Gpi calls that perform similar functionality.

`grClearWindow, grDrawLineAt, grDrawBoxAt, grDrawStringAt`

This set of methods could be much larger to support all of PM's graphics functionality. We can add more as we need them.

GRAPHIC.IDL

```

#include <somobj.idl> // get definition of your parent

typedef somToken HPS; // get rid of the SOM compiler warnings
typedef somToken HWND;
typedef somToken RECTL;
typedef somToken POINTL;

interface Window;

interface Graphic: SOMObject// name and parent(s)
{
    attribute long currentX;
    attribute long currentY;
    attribute unsigned long fgColor;
    attribute unsigned long bgColor;
    attribute long fgMixMode;
        // use FM_ constants from pmgpi.h
    attribute long drawControl;
        // use DRO_FILL, DRO_OUTLINE, or DRO_OUTLINEFILL
}

```

```

        attribute Window grWin;
                // the window for this graphic
        attribute HPS      hps;    // the HPS

//# Graphics Methods
void grClearWindow();
        // clears current window using current background color

void grDrawLineAt(in short x1, in short y1,
                  in short x2, in short y2);
        // draws line from (x1,y1) to (x2,y2)

void grDrawBoxAt  (in short x1, in short y1,
                    in short x2, in short y2);
        // draws a box from (x1,y1) to (x2,y2)

void grDrawStringAt (in short x, in short y,
                     in string textStr);
        // draws textStr at point (x,y)

#endif __SOMIDL__
implementation
{
    releaseorder:
        _get_currentX, _set_currentX,
        _get_currentY, _set_currentY,
        _get_fgColor, _set_fgColor,
        _get_bgColor, _set_bgColor,
        _get_fgMixMode, _set_fgMixMode,
        _get_drawControl, _set_drawControl,
        _get_grWin, _set_grWin,
        _get_hps, _set_hps,
    grClearWindow, grDrawLineAt, grDrawBoxAt, grDrawStringAt;

    passthru C_h_before =
        "#define INCL_GPI"
        "#define INCL_WIN"
        "#include <window.h>"
;
// Note: you must include this before #include <window.h>

//# Method Modifiers
somInit: override;
somUninit: override;
};

#endif

};

```

GRAPHIC.C

```
#define Graphic_Class_Source

#include <graphic.ih>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void InitGraphic(GraphicData *somThis);
void QuitGraphic(GraphicData *somThis);

/*
 * grClearWindow clears current window using current
 * background color
 */
SOM_Scope void SOMLINK grClearWindow(Graphic somSelf,
Environment *ev)
{
    GraphicData *somThis = GraphicGetData(somSelf);
    POINTL upRightPt;
    RECTL rcl;
    HWND hwndGraphic = __get_hwndWindow(_grWin, ev);
    GraphicMethodDebug("Graphic", "grClearWindow");

    _currentX = 0;
    _currentY = 0;
    InitGraphic(somThis);
    WinQueryWindowRect(hwndGraphic, &rcl);
    upRightPt.x = rcl.xRight;
    upRightPt.y = rcl.yTop;

    /* the box drawing on next 2 lines will have the effect of
filling with our background */
    GpiSetColor(_hps, _bgColor);
    GpiBox(_hps, DRO_FILL, &upRightPt, 0L, 0L);

    QuitGraphic(somThis);
}

/*
 * grDrawLineAt draws line from (x1,y1) to (x2,y2)
 */
SOM_Scope void SOMLINK grDrawLineAt(Graphic somSelf,
Environment *ev,
short x1,
short y1,
short x2,
short y2)
{
    GraphicData *somThis = GraphicGetData(somSelf);
```

```

POINTL upRightPt;
GraphicMethodDebug("Graphic", "grDrawLineAt");

_currentX = x1;
_currentY = y1;
InitGraphic(somThis);
upRightPt.x=(long)x2;
upRightPt.y =(long)y2;

GpiLine(_hps, &upRightPt);

QuitGraphic(somThis);

}

/*
 * grDrawBoxAt draws a box from (x1,y1) to (x2,y2)
 */
SOM_Scope void SOMLINK grDrawBoxAt(Graphic somSelf,
                                    Environment *ev,
                                    short x1, short y1,
                                    short x2, short y2)
{
    GraphicData *somThis = GraphicGetData(somSelf);
    POINTL upRightPt;

    GraphicMethodDebug("Graphic", "grDrawBoxAt");

    _currentX = x1;
    _currentY = y1;
    InitGraphic(somThis);
    upRightPt.x=(long)x2;
    upRightPt.y =(long)y2;
    GpiBox(_hps, _drawControl, &upRightPt, 0L, 0L);
    QuitGraphic(somThis);
}

/*
 * grDrawStringAt draws textStr at point (x,y)
 */
SOM_Scope void SOMLINK grDrawStringAt(Graphic somSelf,
                                       Environment *ev, short x, short y, string textStr)
{
    GraphicData *somThis = GraphicGetData(somSelf);
    GraphicMethodDebug("Graphic", "grDrawStringAt");
    _currentX = x;
    _currentY = y;
    InitGraphic(somThis);
    GpiCharString(_hps, strlen(textStr), textStr);
    QuitGraphic(somThis);
}

```

```

}

SOM_Scope void  SOMLINK somInit(Graphic somSelf)
{
    GraphicData *somThis = GraphicGetData(somSelf);
    GraphicMethodDebug("Graphic", "somInit");
    _currentX = 0;
    _currentY = 0;
    _bgColor = CLR_BLACK;
    _fgColor = CLR_YELLOW;
    _fgMixMode = FM_DEFAULT;
    _drawControl = DRO_OUTLINE;

    Graphic_parent_SOMObject_somInit(somSelf);
}

SOM_Scope void  SOMLINK somUninit(Graphic somSelf)
{
    GraphicData *somThis = GraphicGetData(somSelf);
    GraphicMethodDebug("Graphic", "somUninit");

    Graphic_parent_SOMObject_somUninit(somSelf);
}

void InitGraphic(GraphicData *somThis)
{
    Environment *ev = somGetGlobalEnvironment();
    POINTL pt;
    HWND hwndGraphic = __get_hwndWindow(_grWin, ev);

    pt.x = _currentX;
    pt.y = _currentY;
    _hps = WinGetPS(hwndGraphic);
    GpiMove(_hps, &pt);
    GpiSetColor(_hps, _fgColor);
    GpiSetBackColor(_hps, _bgColor);
    GpiSetMix (_hps, _fgMixMode);
}

void QuitGraphic(GraphicData *somThis)
{
    WinReleasePS(_hps);
}

```

boxdraw.idl

```

#include <client.idl> // get def of your parent

interface Boxdraw: Client // Boxdraw is a subclass
    // of Client
{

```

```

        attribute short anchorX;
        attribute short anchorY;
        attribute short lastX;
        attribute short lastY;
        attribute boolean wireExists;

#ifndef __SOMIDL__
implementation
{
    /* Class Modifiers
    releaseorder:
        _get_anchorX, _set_anchorX,
        _get_anchorY, _set_anchorY,
        _get_lastX, _set_lastX,
        _get_lastY, _set_lastY,
        _get_wireExists, _set_wireExists;

    /* Method Modifiers
    CharPress: override;
    Button1Click: override;
    Button1DragBegin: override;
    Button1Drag: override;
    PaintWindow: override;

    somInit: override;
};

#endif
};

```

boxdraw.c

```

#define Boxdraw_Class_Source

#include <graphic.h> /* this picks up pmgpi.h (window.h
                           won't) */
#include <boxdraw.ih>

#include <stdio.h>

SOM_Scope void SOMLINK CharPress(Boxdraw somSelf,
                                  Environment *ev,
                                  char key, short kbmState)
{
    BoxdrawData *somThis = BoxdrawGetData(somSelf);
    Graphic gr = __get_gr(somSelf, ev);
    long saveDrawMode;

    if(key=='f' || key =='F') {
        saveDrawMode = __get_drawControl(gr, ev);
        __set_drawControl(gr, ev, DRO_FILL);
    }
}

```

```
    _grDrawBoxAt(gr, ev, _anchorX, _anchorY, _lastX,
    _lastY); _set_drawControl(gr, ev, saveDrawMode);
}
}

SOM_Scope void SOMLINK Button1Click(Boxdraw somSelf,
Environment *ev,
            short x, short y)
{
    BoxdrawData *somThis = BoxdrawGetData(somSelf);
    Graphic gr = __get_gr(somSelf, ev);

    if(_wireExists)
        _grDrawBoxAt(gr, ev, _anchorX, _anchorY, _lastX,
    _lastY);
    _wireExists = FALSE;
}

SOM_Scope void SOMLINK Button1DragBegin(Boxdraw somSelf,
Environment *ev,
            short x, short y)
{
    BoxdrawData *somThis = BoxdrawGetData(somSelf);
    Graphic gr = __get_gr(somSelf, ev);

    if(_wireExists)
        _grDrawBoxAt(gr, ev, _anchorX, _anchorY, _lastX,
    _lastY);

    _lastX = x;
    _lastY = y;
    __set_currentX(gr, ev, (long)x);
    __set_currentY(gr, ev, (long)y);
    _anchorX = x;
    _anchorY = y;
}

SOM_Scope void SOMLINK Button1Drag(Boxdraw somSelf,
Environment *ev,
            short x, short y)
{
    BoxdrawData *somThis = BoxdrawGetData(somSelf);
    Graphic gr = __get_gr(somSelf, ev);

/* erase the old line */
    if(_anchorX!=_lastX || _anchorY!=_lastY) {
        _grDrawBoxAt(gr, ev, _anchorX, _anchorY, _lastX,
    _lastY);
        _wireExists = TRUE;
    }
}
```

```

        _lastX = x;
        _lastY = y;

        /* and draw the new one */
        _grDrawBoxAt(gr, ev, (long)_anchorX, (long)_anchorY,
                     (long)_lastX, (long)_lastY);

    }

SOM_Scope void SOMLINK PaintWindow(Boxdraw somSelf,
Environment *ev,
                    RECTL* rcl)
{
    BoxdrawData *somThis = BoxdrawGetData(somSelf);
    Graphic gr = __get_gr(somSelf, ev);
    long saveMixMode = __get_fgMixMode(gr, ev);

    __set_fgMixMode(gr, ev, FM_DEFAULT);
    __grClearWindow(gr, ev);
    __set_fgMixMode(gr, ev, saveMixMode);
    __grDrawBoxAt(gr, ev, rcl->xLeft, rcl->yBottom,
                  rcl->xRight, rcl->yTop);
    if(_wireExists)
        __grDrawBoxAt(gr, ev, _anchorX, _anchorY, _lastX,
                      _lastY);
        __grDrawStringAt(gr, ev, 10, 30,
                         "Draw a wire rectangle by dragging mouse button
1");
        __grDrawStringAt(gr, ev, 10, 10, "Press F to fill wire
rectangle");
    }

SOM_Scope void SOMLINK somInit(Boxdraw somSelf)
{
    BoxdrawData *somThis = BoxdrawGetData(somSelf);
    Environment *ev;

    Boxdraw_parent_Client_somInit(somSelf);
/* set the foreground mixmode */
    __set_fgMixMode(__get_gr(somSelf, ev), ev, FM_INVERT);
    _wireExists = FALSE;
}

```

test3.c

```

#include <boxdraw.h>
void main(int argc, char *argv[])
{
    Boxdraw obj = BoxdrawNew();

```

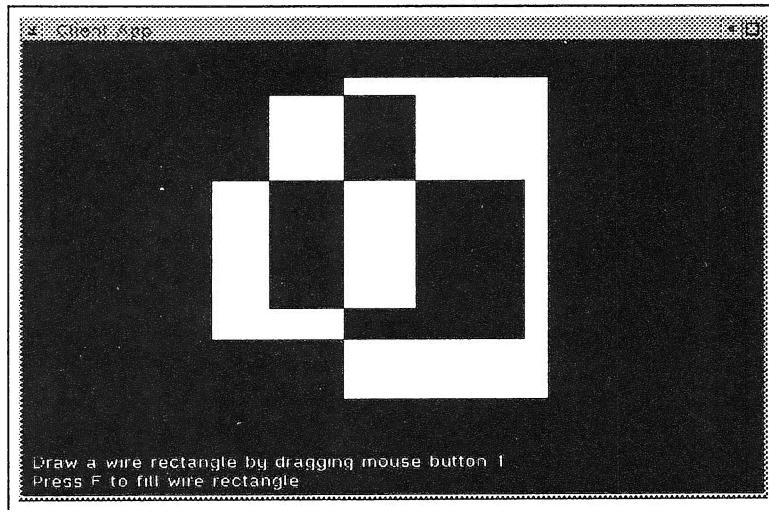


Figure 15.3. Graphics Application

```
_ProcessEvents(obj, somGetGlobalEnvironment());
_somFree(obj);
}
```

Figure 15.3 shows what your application will look like.

So now we know how to create a PM application, and output into our window using graphics. What else is there to learn? Well, so far we have been working at a fairly low level. You will find that you don't always have to start from scratch when you create an application. PM provides many ready-to-use classes of windows that you can put to use. In the next chapter, we will look at those classes known as PM controls which give you some valuable functionality.

16

Wrapping PM Controls

INTRODUCTION

In the next three examples, we are going to spend a lot of time doing what we call *wrapping*. In particular, we will be wrapping PM Controls, but the concepts that we will learn can be applied to other classes of functionality like the file system, semaphore management, or managing interprocess communication. PM controls are nice predefined classes of functionality associated with the user interface. They are easy to form classes from because PM has already created classes for us.

When taking existing functionality and forming objects, the OO designer faces decisions such as what goes into the object, and what variations on that object are grounds for creating an additional object. For example, should we create a single push-button class patterned after PM's WC_BUTTON class and the BS_PUSHBUTTON style, or should we have a hierarchy of buttons subclassed from a generic button class. Our hierarchy could have subclasses for radio-buttons, check-boxes, and push-buttons, etc. To phrase the same question in PM terminology, does setting different style bits result in the definition of new classes, or a change in data of a single class of object?

The answer to this question is not simple. The most object-oriented solution would be to opt for more classes, because the object-oriented purist doesn't like to have many decisions made in his or her code. We do not need constructions that branch out in eight different directions based on the eight different styles of a particular class if we simply create eight different subclasses. On the other

hand, the more practical lay programmers out there prefer fewer classes, because of issues mostly related to manageability. A very pure OO class library could easily present a user with well over a thousand classes, which can be a little overwhelming to the lay programmer. On the other hand, a class library that encapsulates the identical functionality but is not quite as pure could be less than 200—not quite so threatening. So as we continue through these examples, keep in mind that they will tend to adopt the “fewer-classes-are-easier-to-manage” approach, though it is not necessarily the best approach to take. The ultimate decision is up to you.

PM CONTROLS

So far, we have played with the basic PM Client window and we've done some graphics using PM's graphics programming interface. We now have all we need to put together a sophisticated application. But before we jump into bigger and better applications, let's have a look at some of what PM gives us for free.

As you look at applications on your OS/2 PM machine, you see a lot of standard pieces. Push-buttons, scrollbars, listboxes, entry fields, and the like are common to many PM applications. A PM programmer doesn't have to program these pieces every time he or she creates an application. PM provides the programmer a set of predesigned windows to use for building applications. These windows are called controls. They help an application developer control the interaction between a user and the application in a standard fashion. Besides referring to these windows as Controls, PM also refers to each specialized window as a Class which of course sounds rather object oriented. Think of each of the PM Controls as a class of window subclassed from the basic PM window. The prepackaged window classes that PM provides are identified with WC_... constants which are used at window creation time using WinCreateWindow:

```
WC_BUTTON  
WC_ENTRYFIELD  
WC_MLE  
WC_COMBOBOX  
WC_LISTBOX  
WC_NOTEBOOK  
WC_CONTAINER  
WC_STATIC  
...
```

These classes have been preregistered and are publicly available to the PM programmer. Your client application, on the other hand, is privately registered with the WinRegisterClass call and is available only until the application terminates.

In the next few examples we are going to put these PM controls to use as SOM objects. Taking a piece of existing functionality, in this case PM controls, and making an object that embodies that functionality is referred to informally

as *wrapping*. First we will wrap a PM push-button. We will use two examples to demonstrate some techniques for wrapping a PM push-button using SOM. After that, we will have a third example which uses our newly learned techniques to quickly wrap more controls like the static text field, the entry field, the list box, and the combo box.

In the first example we create the PushButton class, and we put it to work for us in our client application. We will see that our first example is extremely limited, and in example two, we explore many of the techniques provided by SOM to wrap this control more completely and efficiently.

The first two examples will use the same class hierarchy shown in Figure 16.1. In the first example for wrapped controls, I'm going to make a first pass at wrapping a simple PM push-button. This example does the job and could serve as a prototype for wrapping your controls, but it is an extremely weak and limited one. Hopefully, we can learn a lot about SOM as we evolve from Example One to Example Two.

Window and Client class

The Window class has not changed since our last example, so I don't need to describe it here. The client is still much the same as it used to be. It is still the basic window that our applications will subclass (App1 in this example), and it still provides some abstract methods that App1 can override in order to work with the events it needs. Many of the abstract methods I used in the graphics example focus on the methods that will be used in this example. This example introduces the first of what could be many abstract methods based on higher level messages sent to the client window by the controls owned by the client. In the previous example, the client was responding to low-level events since we had no PM Controls. These higher-level messages are usually contained in a PM message of the following two types.

WM_COMMAND and WM_CONTROL

WM_COMMAND messages are usually generated by push-button presses, Menu Item selections, or accelerator key executions. These messages are intended to cause a specific action to be executed ... now.

WM_CONTROL messages are more general high-level messages sent by controls to their owners. These events range from informational messages to command type messages. Anything from letting the owner know its child has taken or lost focus, or that someone is adding text to an entry field, to clicks and selections occurring in buttons, listboxes, or notebooks.

Other high level messages that are generated by controls include:

- WM_VSCROLL
- WM_HSCROLL
- WM_INITMENU
- WM_MENUSELECT
- WM_MENUEND

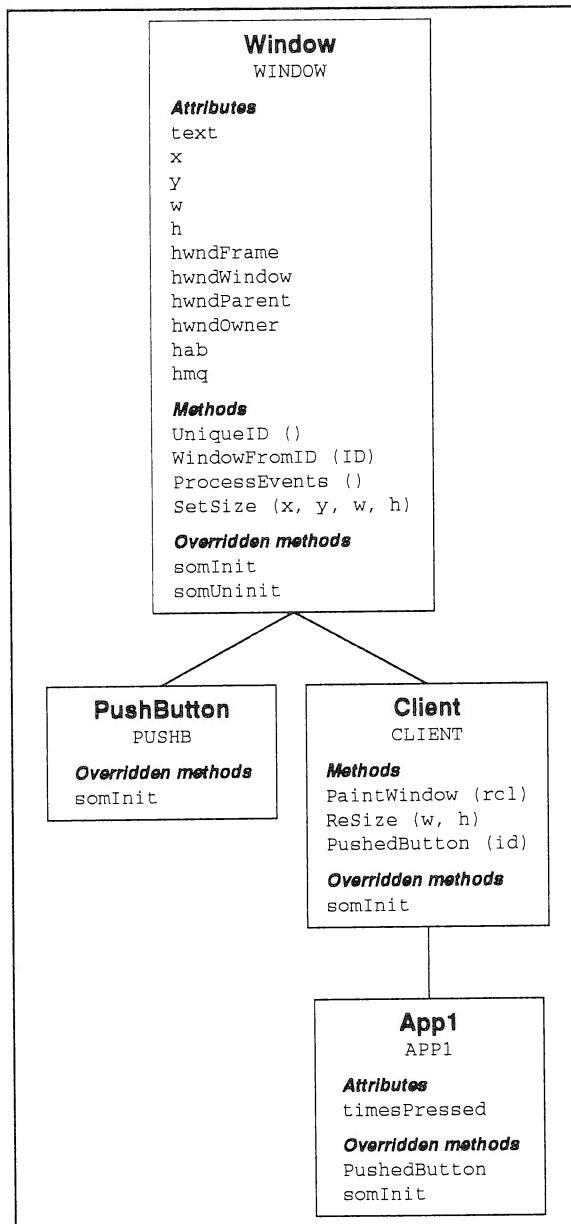


Figure 16.1. Class Hierarchy for Example

- WM_DRAWITEM
- WM_MEASUREITEM
- WM_CONTROLPOINTER

WINDOW.C and WINDOW.IDL are the same as in the last example

see previous example for listing

CLIENT.IDL

```
#include <window.idl> // get definition of your parent

interface Client: Window // your classname and your parent(s)
{
    void PaintWindow(in RECTL *rcl);
    void ReSize(in short w, in short h);

    //## Control Events. Command (WM_COMMAND) and Control Notifications
    (WM_CONTROL)
    void PushedButton(in short id);

#endifdef __SOMIDL__
implementation
{
    //## Class Modifiers
    releaseorder:
        PaintWindow, ReSize, PushedButton;

    //## Method Modifiers
    somInit: override;
};

#endif
```

CLIENT.C

```

HWND hwndTitleBar, hwndFrame, hwndClient;
HAB hab;
static BOOL firsttimein=TRUE;
string className = "controlsApp";

Client_parent_Window_somInit(somSelf);
hab = __get_hab(somSelf, ev);

if(firsttimein) {
    WinRegisterClass(
        hab, /* anchor block handle */
        className, /* Window Client Class name */
        vwClientProc, /* Client procedure for class */
        CS_SIZEREDRAW, /* Class style */
        4L); /* Extra bytes for data */
    firsttimein = FALSE;
}
hwndFrame = WinCreateStdWindow(
    HWND_DESKTOP,
    WS_VISIBLE,
    &flFrameFlags,
    className,
    NULL,
    0L,
    (HMODULE)0,
    0L, /* resource ID */
    &hwndClient);
if((hwndTitleBar = WinWindowFromID (hwndFrame, FID_TITLEBAR)) != 0L)
    WinSetWindowText (hwndTitleBar, className);
WinSetWindowPtr(hwndClient, 0L, somSelf);

__set_hwndWindow(somSelf, ev, hwndClient);
__set_hwndFrame (somSelf, ev, hwndFrame);

__set_hwndParent(gBaseWindow, ev, hwndFrame);
__set_hwndOwner (gBaseWindow, ev, hwndFrame);
__set_hwndWindow(gBaseWindow, ev, hwndClient);
}

MRESULT EXPENTRY vwClientProc(HWND hwnd, ULONG msg,
                             MPARAM mp1, MPARAM mp2)
{
    HPS hps;
    RECTL rcl;
    char key;
    Environment *ev = somGetGlobalEnvironment();
    Window somWin;
    SOMObject somSelf;
    SHORT vkey, sFlag, kbmState;
    SHORT src, winID, x,y, w, h;
    HWND winHwnd;

    if(hwnd)
        somSelf = WinQueryWindowPtr(hwnd, 0);
}

```

```

switch(msg) {
    case WM_SIZE:
        if(somSelf) {
            w = SHORT1FROMMP(mp2);
            h = SHORT2FROMMP(mp2);
            _ReSize(somSelf, ev, w, h);
        }
        break;
#endif NOGRAPHICS
    case WM_ERASEBACKGROUND:
        return (MRESULT)TRUE;
#endif
    case WM_PAINT:
        if (somSelf) {
            hps = WinBeginPaint(hwnd, NULLHANDLE, &rcl);
            _PaintWindow(somSelf, ev, &rcl);
            WinEndPaint(hps);
            return FALSE;
        }
        break;
    case WM_COMMAND:
/* source of WM_COMMAND is saved in src, should be -_PUSHBUTTON */
        src = SHORT1FROMMP(mp2);
        winID = SHORT1FROMMP(mp1);
        winHwnd = WinWindowFromID(hwnd, winID);
        somWin = WinQueryWindowPtr(winHwnd, 0L);
        fprintf(stderr,
                "WM_COMMAND received somWin=%p winID=%d %lx %lx\n",
                somWin, winID, mp1, mp2);
        switch(src) {
            case CMDSRC_PUSHBUTTON:
                _PushedButton(somSelf, ev, winID);
                break;

            default:
                fprintf(stderr,
                        " pushb.c ERROR: shouldn't have executed
                        pbSubProc\n");
                break;
        }
        return 0;
    }
    return WinDefWindowProc(hwnd, msg, mp1, mp2);
}

/* The following are stub methods (Virtual) designed to be
   overridden by applications */

SOM_Scope void SOMLINK PaintWindow(Client somSelf,
                                    Environment *ev,
                                    RECTL* rcl)
{ }

SOM_Scope void SOMLINK ReSize(Client somSelf,
                            Environment *ev,
                            short w, short h)

```

```

{ }

SOM_Scope void  SOMLINK PushedButton(Client somSelf,
                                      Environment *ev,
                                      short id)
{
}

```

PushButton class

In this example we create a push-button class. As a PM programmer, to create a PM PushButton control, a user must call `WinCreateWindow()` using the `WC_BUTTON` class and the `BS_PUSHBUTTON` style bit set. Where is the correct place to put this? “Well”, I thought to myself, “this is the initialization of a window and so it belongs in `somInit`, of course!” By putting it there, I have made it easy to create a PushButton—I just call the SOM macro `PushButtonNew()`, and *voila*, I have a push-button! However, the problem with this approach results from the fact that the SOM macro, `PushButtonNew()` does not allow for any parameters to help the PushButton class create the button, so the PushButton class has to make some important assumptions for us: Who is the parent and who is the owner? What are the initial dimensions and location? Should the class be visible? Should the class use any special style bits?

In our first example we take care of the parent/owner problem by using a global variable, `gBaseWindow`, which is set to correspond to our Main client window. As long as all PushButtons are simply children of the Main client window, we will be okay. All windows have been set to have dimensions 0 X 0 located at 0, 0. This will force all users to do a `SetSize` on all windows after they are instantiated. Also all windows are visible in these examples. This assumption is okay for small, unsophisticated applications like this one but a heavy-duty, sophisticated application will want to be able to dictate the visibility of its controls at any time.

PUSHB.IDL

```

#include <window.idl> // get definition of your parent

interface PushButton: Window      // your classname and your parent(s)
{
    #ifdef __SOMIDL__
    implementation
    {
        //# Method Modifiers
        somInit: override;
    };
    #endif
};

```

PUSHB.C

```

#define PushButton_Class_Source
#include <pushb.ih>

#include <stdio.h>

SOM_Scope void SOMLINK somInit(PushButton somSelf)
{
    /* PushButtonData *somThis = PushButtonGetData(somSelf); */

    HWND hwndParent, hwndOwner, hwnd;
    Environment *ev = somGetGlobalEnvironment();
    USHORT idPushButton = 1;
    ERRORID errorid;
    USHORT i;

    /* use the base window as the parent and owner this time */

    hwndParent = __get_hwndWindow(gBaseWindow, ev);
    hwndOwner = __get_hwndWindow(gBaseWindow, ev);

    __set(hwndParent, somSelf, ev);
    __set(hwndOwner, somSelf, ev);

    hwnd = WinCreateWindow(
        hwndParent,
        WC_BUTTON,           /* window class */
        "I'm a Push Button", /* window text */
        WS_VISIBLE | BS_PUSHBUTTON, /* window style */
        0L, 0L, 0L, 0L,       /* position and size */
        hwndOwner,           /* owner window */
        HWND_BOTTOM,         /* placement */
        idPushButton,         /* child window id */
        NULL,                /* Control data */
        NULL);               /* pres. params */

    WinSetWindowPtr(hwnd, 0L, somSelf);
    __set(hwnd, somSelf, ev, hwnd);
    __set(hwndFrame, somSelf, ev, NULLHANDLE);
    fprintf(stderr, "push button created. hwnd=%lx\n", hwnd);
    PushButton_parent_Window_somInit(somSelf);
}

```

App1 class.

As in the previous examples, this application selectively overrides the abstract methods it is interested in using—in this case, PushedButton. App1 creates a PushButton object, sizes it in its somInit method, and does its work in the PushedButton method. We take advantage of the fact that our _set_text method on a client will update the title bar if the client has a frame and a title bar. This allows us to set the text of the title bar as well as in the PushButton object.

app1.idl

```
#include <client.idl> // get definition of your parent

interface PushButton;

interface App1: Client // App1 is a subclass of Client
{
    attribute short timesPressed;

#ifndef __SOMIDL__
implementation
{
    /* Class Modifiers
    releaseorder:
        _get_timesPressed, _set_timesPressed;

    /* Method Modifiers

    PushedButton: override;

    somInit: override;
#endif Instance vars /* noone outside of app1 needs to access pb */
    PushButton pb;
};

#endif
};
```

app1.c

```
#define App1_Class_Source
#include <app1.ih>

#include <stdio.h>

#include <pushb.h>

SOM_Scope void SOMLINK somInit(App1 somSelf)
{
    App1Data *somThis = App1GetData(somSelf);
    PushButton pb;
    Environment *ev;

    App1_parent_Client_somInit(somSelf);

    _pb = PushButtonNew();
    _SetSize(_pb, ev, 10L, 10L, 240L, 60L);
}

SOM_Scope void SOMLINK PushedButton(App1 somSelf, Environment *ev,
                                     short id)
{
    App1Data *somThis = App1GetData(somSelf);
    char strBuf[20];

    _timesPressed++;
```

```

        sprintf(strBuf, "%d", _timesPressed);
        __set_text(_pb, ev, strBuf);
        sprintf(strBuf, "Pressed %d times", _timesPressed);
        __set_text(somSelf, ev, strBuf);
    }

```

wrap1.c

```

#include <appl.h>
void main(int argc, char *argv[])
{
    App1 obj = ApplNew();
    _ProcessEvents(obj, somGetGlobalEnvironment());
    _somFree(obj);
}

```

makefile

```

LIB = $(LIB)
INCLUDE = .;$(INCLUDE)
SMEMIT = ih;h;c;
SRCS = window.c client.c pushb.c appl.c wrap1.c
OBJS = $(SRCS:.c=.obj)
install: wrap1.exe

.SUFFIXES : .ih .sc .lib .def .dll .idl

run:
    wrap1

.c.obj:
    icc -Q -Ti+ -c $<

.idl.ih:
    sc $*

wrap1.exe: $(OBJS)
    link386 /NOL /NOI /CO /PM:PM $(OBJS), wrap1.exe, nul,somtk;

window.ih: window.idl
window.obj: window.ih

client.obj: client.ih
client.ih: client.idl

pushb.obj: pushb.ih
pushb.ih: pushb.idl

appl.obj: appl.ih
appl.ih: appl.idl

wrap1.obj: wrap1.c

```

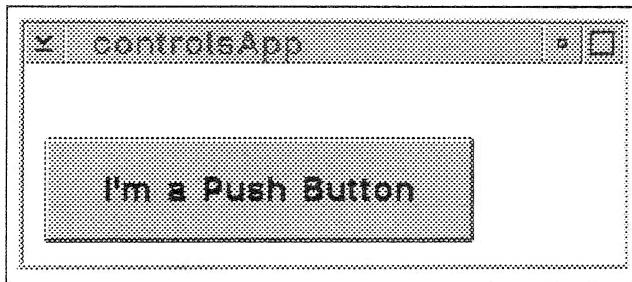


Figure 16.2. Output for WRAP1.EXE

Run It

To run this example type wrap1 at the OS/2 command line

```
d:\example> wrap1
```

and the application appears as in Figure 16.2.

This doesn't do much except count the number of times you press the push-button. You may be thinking, "Hey weren't we just counting button1clicks a few examples ago?" Let's move on to wrapped controls Example 2.

17

A Piano of PushButtons

INTRODUCTION

In this example, the classes and their hierarchy will remain the same, but we will begin to correct many of the weaknesses of our first example. We will also refine some flawed elements of our architecture introduced in earlier examples but left unrefined until now.

We will no longer instantiate our controls using the limited `<classname> New();` macro as in `PushButtonNew();`. We will improve our initialization procedures, and we will manage our data better. The key improvements are made possible through the use of metaclasses and the special SOM function identified by the SOM keyword `classinit` in your IDL.

THE WINDOW CLASS

Our Window class has finally grown up after not changing during the past few examples. The biggest change is due to our use of the metaclass.

An Explicit Metaclass

In this example, we introduce our metaclass, `M_Window`. The `M_Window` metaclass is a class that will be instantiated before the first object in our entire Window hierarchy is ever instantiated. We can always assume so, because all of our

applications and controls are subclassed from Window or one of its derived subclasses. This allows us to make use of the macro, _Window, to call metaclass methods introduced by the M_Window metaclass. We know the M_Window metaclass instance is one object that will be only instantiated once and always at least once. This knowledge leads us to see this metaclass as a valuable place to put things like our PM application initialization and message queue initialization. The M_Window metaclass will also be the place where a user can query information. We will also store our anchor block and message queue handles here.

classinit

So where in this metaclass is the best place for us to put these initializations? Is there a method that will only get called one time regardless of how many objects are instantiated and regardless of how many levels of derived subclasses are introduced? The prefix *m_* is a common prefix for metaclass methods. It is identified in the IDL by the key word *functionprefix* in the metaclass definition. This prefix helps us and the compiler keep metaclass method names and instance method names straight. For example, somInit could be overridden for both the metaclass and the class, causing the somInit method to appear twice in your implementation file. How about in the metaclass's somInit, (*m_somInit*)? No. *m_somInit* is not a bad choice, but will get called one time for each new derived class that is instantiated. The answer is a special function identified in the IDL by the key word, *classinit*:

```
classinit: windowInit;
```

This line identifies a routine that will get executed one time when the first instance of the class is instantiated. It will only be called once, no matter how many instances of Window and its derived classes are created. This helps us get rid of the inelegant code we commonly have to insert so that certain things only get executed first time through.

The following construction ...

```
static boolean firsttimethrough=TRUE;
if(firsttimethrough) { /* check every time. Possibly 100s of
times!! */
    firsttimethrough=FALSE;
    /* do one-time-only stuff here */
}
```

can now be replaced with

```
/* do one-time-only stuff here */
```

We also use this *classinit* identifier in client.idl and do our client registration, WinRegisterClass, in a classInit routine called clientInit.

No global variables??

We have removed our need for a global variable like `h`. We have made this `Window` object an attribute of the `Window` metaclass. At any point in the application, we know we can get the handle for this object by using the following call:

```
Window base;
base = __get_baseWindow(_Window, ev);
```

Our constructor is here!

The `Window` metaclass has also introduced a method that can be used as a constructor for all of our derived application and control windows.

```
Window CreateWindow(in Window parent,
                    in Window owner,
                    in long style,
                    in string initText,
                    in short x, in short y,
                    in short w, in short h,
                    in somToken userStruct);
```

This method is, of course, patterned after the PM API used to create PM windows —`WinCreateWindow`. Each window class must implement this method so that a user

can better control the initialization of its windows. We will no longer be using `somInit` to initialize windows, as our updated hierarchy shows in Figure 17.1.

`ProcessEvents` methods have been moved to the `Window` metaclass. This method applies to all windows in a process on the same thread, and it makes more sense to place this method into the `Window` metaclass.

In the `Window` metaclass, we also have five attributes: `baseWindow`, `hmq`, `hab`, `screenWidth`, and `screenHeight`. These attributes have all been modified in the IDL by using the type `nodata`. These attributes are one time only attributes that should only have one location in memory in which they are stored. There is no need for subclasses of the `Window` metaclass to also take up space for these attributes.

Window

Also in `window.idl`, alongside the `M_Window` interface definition, is the `Window` class itself. We have added methods to change colors. This is done in a common way among all PM controls, through the use of Presentation Parameters. The PM controls use system colors by default to paint themselves. The colors are defined in the header file, `<pmwin.h>`, using the prefix `SYSCLR_`. If a user wants to color his or her windows differently, he or she can do so using appropriate presentation parameters also defined in `<pmwin.h>` with `PP_` prefixes as in `PP_FOREGROUNDCOLOR` and `PP_BACKGROUNDCOLOR`. PM follows a rule that goes roughly as follows:

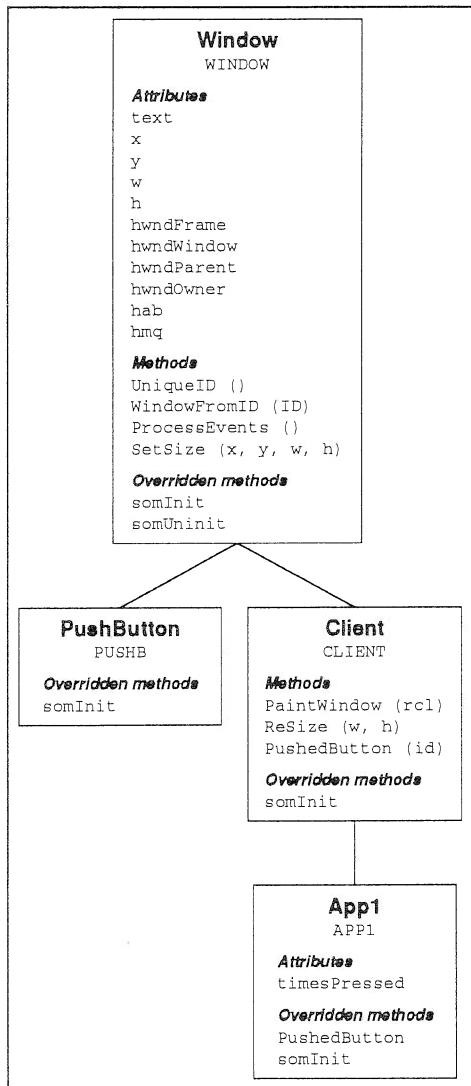


Figure 17.1. Updated Hierarchy

A PM window will color itself using its presentation parameters if set programmatically or by dragging from the color palette. When no preferences are given, PM will trace through the parents to the HWND_DESKTOP to see if any ancestors have their PP_... parameters set. In this way, children will inherit the presentation parameters of their parents. If no window in the view hierarchy has its presentation parameters set, then PM will use the system default colors appropriate for that window. These colors are defined in the *PM Programming Reference, Volume III*. You will find the default parameters used for colors and fonts, and the presentation parameters a user can use to alter these defaults.

We have also introduced a method `SetFocus`, which allows us to set the focus to any wrapped PM window that will take focus. Most windows can take focus. There are some exceptions, like the Static Text control, which will not take the focus but will pass the `SetFocus` instructions to its nearest next sibling, or, if none exists, to its owner.

window.idl

```
#ifndef window_idl
#define window_idl

#include <somobj.idl> // get definition of your parent
#include <somcls.idl> // parent of the metaclass

typedef somToken HWND; // get rid of the SOM compiler
warnings
typedef somToken HAB;
typedef somToken HMQ;
typedef somToken HPS;
typedef somToken RECTL;
typedef somToken BYTE;

interface Window;

interface M_Window: SOMClass
{
    //# metaclass attributes
    readonly attribute Window baseWindow;
    readonly attribute HAB hab;
    readonly attribute HMQ hmq;
    readonly attribute long screenWidth;
    readonly attribute long screenHeight;

    //# metaclass methods
    Window CreateWindow(in Window parent,
                        in Window owner,
                        in long style,
                        in string initText,
                        in short x, in short y,
                        in short w, in short h);
    void ProcessEvents();

#ifndef __SOMIDL__
    implementation {
        functionprefix=m_; /* helps us identify
                           metaclass methods */
        filestem = window;
        classinit: windowInit;
        releaseorder:
            _set_baseWindow, _get_baseWindow,

```

```
    __get_hab, __get_hmq,
    __get_screenWidth, __get_screenHeight;
    CreateWindow, ProcessEvents;

//# Data Modifiers
baseWindow:      nodata;
hab:            nodata;
hmq:            nodata;
screenWidth:
nodata;
screenHeight:    nodata;

};

#endif
};

interface Window: SOMObject // Window is a subclass
                    // of SOMObject
{
    attribute string text;
    attribute long x;
    attribute long y;
    attribute long w;
    attribute long h;
    attribute HWND hwndFrame;
    attribute HWND hwndWindow;
    attribute HWND hwndParent;
    attribute HWND hwndOwner;
    attribute long userData;

    short UniqueID();
    Window WindowFromID(in short ID);

    void SetSize(in long x, in long y, in long w, in long h);
    void SetFocus();
        // sets focus to the Window on which this method acts.

    void SetColorFG( in BYTE r, in BYTE g, in BYTE b);
    void SetColorBG( in BYTE r, in BYTE g, in BYTE b);

#endif __SOMIDL__
implementation {

//# Class Modifiers
filestem = window;
metaclass = M_Window;
releaseorder:
    __get_text, __set_text,
    __get_x, __set_x,
```

```

        _get_y, _set_y,
        _get_w, _set_w,
        _get_h, _set_h,
        _get_hwndFrame, _set(hwndFrame,
        _get_hwndWindow, _set(hwndWindow,
        _get_hwndParent, _set(hwndParent,
        _get_hwndOwner, _set(hwndOwner,
        _get_userData, _set(userData,
        UniqueID, WindowFromID,
        SetSize, SetFocus, SetColorFG, SetColorBG;

//# Method Modifiers
somInit: override;
somUninit: override;

passthru C_h_before =
    "#define INCL_WIN"
    "#include <os2.h>"
;

//# Data Modifiers
text:          nodata;

};

#endif /* __SOMIDL__ */
};

#endif /* #ifndef window_idl */

```

window.c

```

#define Window_Class_Source
#define INCL_GPI
#include <window.ih>

#include <stdio.h>

Window baseWindow;
HMQ      hmq;
HAB      hab;

/*
 * Method from the IDL attribute statement:
 * "attribute string text"
 */
SOM_Scope string SOMLINK _get_text(Window somSelf,
Environment *ev)
{
    WindowData *somThis = WindowGetData(somSelf);
    string textStr;
    long   textLen;

```

```
    if (_hwndWindow) {
        textLen = WinQueryWindowTextLength(_hwndWindow)+1;
        textStr =
(string)malloc(WinQueryWindowTextLength(_hwndWindow)+1);
        WinQueryWindowText (_hwndWindow, textLen, textStr);
    }

    return textStr;
}

/*
 * Method from the IDL attribute statement:
 * "attribute string text"
 */
SOM_Scope void  SOMLINK _set_text(Window somSelf, Environment
*ev,
                                string text)
{
    WindowData *somThis = WindowGetData(somSelf);
    if (_hwndWindow) {
        WinSetWindowText (_hwndWindow, text);
    }
    if (somSelf==baseWindow && _hwndFrame)
        WinSetWindowText (_hwndFrame, text);
}

/*
 * A primitive unique-id generator
 */
SOM_Scope short  SOMLINK UniqueID(Window somSelf,
                                      Environment *ev)
{
    static short id=1;
    return (++id);
}

/*
 * WindowFromID will return the pointer to the Window
 * SOMobject
 * identified by ID if somSelf is the parent of that window
 */
SOM_Scope Window  SOMLINK WindowFromID(Window somSelf,
                                         Environment *ev,
                                         short ID)
{
    WindowData *somThis = WindowGetData(somSelf);
    Window w = NULL;

    WindowMethodDebug ("Window", "WindowFromID");
```

```

        if(_hwndWindow) {
            HWND hWndChild;
            hWndChild = WinWindowFromID(_hwndWindow, ID);
            w = (Window )WinQueryWindowPtr(hWndChild, QWL_USER);
        }
        return w;
    }

SOM_Scope void SOMLINK SetSize(Window somSelf,
                                Environment *ev,
                                long x, long y, long w, long
h)
{
    WindowData *somThis = WindowGetData(somSelf);

    if(_hwndWindow) {
        WinSetWindowPos(_hwndWindow,
                        HWND_TOP,
                        x, y, w, h,
                        SWP_MOVE | SWP_SIZE | SWP_ZORDER);
    } else
        fprintf(stderr, "Set Size hwndWindow was NULL\n");
    _x = x;
    _y = y;
    _w = w;
    _h = h;
}

/*
 * SetFocus sets focus to the Window on which this method
acts.
 */
SOM_Scope void SOMLINK SetFocus(Window somSelf,
                                Environment *ev)
{
    WindowData *somThis = WindowGetData(somSelf);
    HWND hwnd = __get(hwndWindow(somSelf, ev));
    WindowMethodDebug("Window", "SetFocus");
    if(hwnd)
        WinSetFocus(HWND_DESKTOP, hwnd);
    return;
}

SOM_Scope void SOMLINK SetColorFG(Window somSelf,
                                  Environment *ev,
                                  BYTE r, BYTE g, BYTE b)
{
    WindowData *somThis = WindowGetData(somSelf);
    WindowMethodDebug("Window", "SetColorFG");

    if(_hwndWindow) {

```

```
RGB2          rgb;

rgb.bRed      = r;
rgb.bGreen    = g;
rgb.bBlue     = b;
rgb.fcOptions = 0;
WinSetPresParam(_hwndWindow, PP_FOREGROUNDCOLOR,
(ULONG)sizeof(RGB2), (PVOID)&rgb);
}

}

SOM_Scope void  SOMLINK SetColorBG(Window somSelf,
                                    Environment *ev,
                                    BYTE r, BYTE g,
                                    BYTE b)
{
    WindowData *somThis = WindowGetData(somSelf);
    WindowMethodDebug("Window", "SetColorBG");

    if(_hwndWindow) {
        RGB2          rgb;

        rgb.bRed      = r;
        rgb.bGreen    = g;
        rgb.bBlue     = b;
        rgb.fcOptions = 0;
        WinSetPresParam(_hwndWindow, PP_BACKGROUNDCOLOR,
                        (ULONG)sizeof(RGB2), (PVOID)&rgb);
    }
}

SOM_Scope void  SOMLINK somInit(Window somSelf)
{
    WindowData *somThis = WindowGetData(somSelf);
    Environment *ev = somGetGlobalEnvironment();

    static BOOL firsttimein=TRUE;

    if(firsttimein) {
        baseWindow = somSelf;
        __set_hwndWindow(baseWindow, ev, HWND_DESKTOP);
        __set_hwndFrame (baseWindow, ev, NULLHANDLE);
        __set_hwndParent (baseWindow, ev, NULLHANDLE);
        __set_hwndOwner (baseWindow, ev, NULLHANDLE);
        firsttimein=FALSE;
    }

    _x = _y = _w = _h = 0L;
```

```

        Window_parent_SOMObject_somInit(somSelf);
    }

SOM_Scope void SOMLINK somUninit(Window somSelf)
{
    WindowData *somThis = WindowGetData(somSelf);

    if(somSelf==baseWindow) {
        WinDestroyWindow(_hwndFrame);
        WinDestroyMsgQueue(hmq);
        WinTerminate (hab);
    }

    Window_parent_SOMObject_somUninit(somSelf);
}

/*
 * Method from the IDL attribute statement:
 * "readonly attribute Window baseWindow"
 */

/*
 * Metaclass methods all have m_ added to the front because
 * of our prefix instructions in the IDL.
 */

SOM_Scope Window SOMLINK m__get_baseWindow(M_Window somSelf,
                                             Environment *ev)
{
    /* M_WindowData *somThis = M_WindowGetData(somSelf); */
    M_WindowMethodDebug("M_Window", "m__get_baseWindow");

    return baseWindow;
}

/*
 * Method from the IDL attribute statement:
 * "readonly attribute HAB hab"
 */
SOM_Scope HAB SOMLINK m__get_hab(M_Window somSelf,
                                   Environment *ev)
{
    /* M_WindowData *somThis = M_WindowGetData(somSelf); */
    M_WindowMethodDebug("M_Window", "m__get_hab");

    /* Return statement to be customized: */
    return hab;
}

```



```

SOM_Scope void SOMLINK m_ProcessEvents(M_Window somSelf,
                                       Environment *ev)
{
    /*      M_WindowData *somThis = M_WindowGetData(somSelf); */
    QMSG qmsg;
    M_WindowMethodDebug("M_Window", "m_ProcessEvents");

    while(WinGetMsg(hab, &qmsg, NULLHANDLE, 0L, 0L))
        WinDispatchMsg(hab, &qmsg);
}

/*
 * m_windowInit will be executed first,
 * before all instance methods and class methods
 * that are introduced at the Window level or by its
 * subclasses. m_windowInit was introduced in the
 * IDL using 'classinit'.
 */
void SOMLINK m_windowInit(SOMClass *cls)
{
    hab = WinInitialize(0);
    hmq = WinCreateMsgQueue (hab, 0);
}

```

Client

Client hasn't changed much since the last example. The parameter passed to the abstract method for PushedButton did change. In the previous example, the ID of the control, which was the source of the button press, was passed. Now we are passing the SOM object associated with this PM ID. The PM control ID is yet another abstraction that can be hidden from the user in our OO design. The programmer should know how to handle objects—we won't force him or her to learn how to translate IDs into objects. We will communicate with objects as much as possible—within reason.

We also brought back our CharPress method so our app can work with something other than the PushedButton method. This method is identical to its appearance in a previous example.

client.idl

```

#include <window.idl> // get definition of your parent

interface Client: Window // your classname and your parent(s)
{
    void PaintWindow(in RECTL *rcl);
    void ReSize(in short w, in short h);

    // Event Handler Methods

```

```

        void CharPress(in char key, in short kbmState);

//# Control Events. Based on PM messages WM_COMMAND and
//# WM_CONTROL Command (WM_COMMAND) and Control Notifications
//# (WM_CONTROL)
void PushedButton(in Window sourceWindow);
// this method gets called when a user presses down and up
// on a pushbutton. The sourceWindow parameter indicates
// the particular SOMObject that was pressed.

#ifndef __SOMIDL__
implementation
{
    //# Class Modifiers
    releaseorder:
        PaintWindow, ReSize, CharPress, PushedButton;
    classinit: clientInit;

    //# Method Modifiers
    somInit: override;
};

#endif

};

```

client.c

```

#define Client_Class_Source
#include <client.ih>

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define NOGRAPHICS      /* */

FNWP vwClientProc;
string className = "clientApp";

static char strBuf[100];      /* used by many routines to print
msgs */

/* I inherit somInit from SOMObject */
SOM_Scope void  SOMLINK somInit(Client somSelf)
{
/*     ClientData *somThis = ClientGetData(somSelf); */

    Environment *ev = somGetGlobalEnvironment();
    short i;

```

```

static ULONG flFrameFlags = FCF_TITLEBAR |
                           FCF_SYSMENU |
                           FCF_SIZEBORDER |
                           FCF_MINMAX |
                           FCF_SHELLPOSITION |
                           FCF_TASKLIST;

HWND hwndTitleBar,
hwndFrame,
hwndClient;

Client_parent_Window_somInit(somSelf);

hwndFrame = WinCreateStdWindow(
    HWND_DESKTOP,
    WS_VISIBLE,
    &flFrameFlags,
    className,
    "Wrapped Control",
    0L,
    (HMODULE)0,
    0L, /* resource ID */
    &hwndClient);
if((hwndTitleBar = WinWindowFromID (hwndFrame,
FID_TITLEBAR)) != 0L)
    WinSetWindowText (hwndTitleBar,
                      "Wrapped Controls");
WinSetWindowPtr(hwndClient, 0L, somSelf);

__set_hwndWindow(somSelf, ev, hwndClient);
__set_hwndFrame (somSelf, ev, hwndFrame);

__set_hwndParent (somSelf, ev, hwndFrame);
__set_hwndOwner (somSelf, ev, hwndFrame);
}

MRESULT EXPENTRY vwClientProc(HWND hwnd, ULONG msg,
                             MPARAM mp1, MPARAM mp2)
{
    Environment *ev = somGetGlobalEnvironment();
    SOMObject somSelf;
    Window sourceWindow;
    HPS hps;
    RECTL rcl;
    char key;
    short vkey, sFlag, kbmState;
    short x, y, w, h;
    SHORT src;
    SHORT winID;
    HWND winHwnd;
}

```

```

if(hwnd)
    somSelf = WinQueryWindowPtr(hwnd, 0);

switch(msg) {
    case WM_CHAR:
        somSelf = WinQueryWindowPtr(hwnd, 0);
        if(somIsObj(somSelf)) {
            sFlag = CHARMSG(&msg)->fs;
            if(sFlag & KC_KEYUP)
                return FALSE;
            if(sFlag & KC_CHAR) {
                key = CHARMSG(&msg)->chr;
                _CharPress(somSelf, ev, key, 0);
            }
        }
        break;
    case WM_SIZE:
        if(somSelf) {
            w = SHORT1FROMMP(mp2);
            h = SHORT2FROMMP(mp2);
            _ReSize(somSelf, ev, w, h);
        }
        break;
#endif NOGRAPHICS
/* we are not going to use a graphics object in our
 * wrapped controls examples, so by processing the
 * WM_ERASEBACKGROUND
 * message in this way, we can get our client area painted
 * for us, free of charge, by the Frame itself
 */
    case WM_ERASEBACKGROUND:
        return (MRESULT)TRUE;
#endif
    case WM_PAINT:
        if (somSelf) {
            hps = WinBeginPaint(hwnd, NULLHANDLE, &rcl);
            _PaintWindow(somSelf, ev, &rcl);
            WinEndPaint(hps);
            return FALSE;
        }
        break;
    case WM_COMMAND:
        /* source of WM_COMMAND is stored in src, should be
         *_PUSHBUTTON */
        src = SHORT1FROMMP(mp2);
        winID = SHORT1FROMMP(mp1);
        winHwnd = WinWindowFromID(hwnd, winID);
        sourceWindow = WinQueryWindowPtr(winHwnd, 0L);
        fprintf(stderr,
                "WM_COMMAND received sourceWindow=%p
                winID=%d %lx %lx\n",

```

```

                sourceWindow, winID, mp1, mp2);
        switch(src) {
            case CMDSRC_PUSHBUTTON:
                _PushedButton(somSelf, ev,
                              sourceWindow);
                break;
            default:
                fprintf(stderr,
                        " pushb.c ERROR: shouldn't have
executed pbSubProc\n");
                break;
        }
        return 0;
    }
    return WinDefWindowProc(hwnd, msg, mp1, mp2);
}

/* The following are stub methods designed to be overridden */

SOM_Scope void  SOMLINK CharPress(Client somSelf,
                                    Environment *ev,
                                    char key,
                                    short kbmState)
{ }

SOM_Scope void  SOMLINK PaintWindow(Client somSelf,
                                      Environment *ev,
                                      RECTL* rcl)
{ }

SOM_Scope void  SOMLINK ReSize(Client somSelf,
                               Environment *ev,
                               short w, short h)
{ }

/*
this method gets called when a user presses down and up on a
pushbutton. The sourceWindow parameter indicates the
particular SOMObject that was pressed.
*/
SOM_Scope void  SOMLINK PushedButton(Client somSelf,
                                      Environment *ev,
                                      Window id)
{ }

void SOMLINK clientInit(SOMClass *cls)
{
    Environment *ev = somGetGlobalEnvironment();
    HAB hab = __get_hab(_Window, ev);

    WinRegisterClass(

```

```
    hab,      /* anchor block handle */
    className, /* Window Client Class name */
    vwClientProc, /* EMClientProc Client procedure */
    CS_SIZEREDRAW, /* Class style */
    4L);          /* Extra bytes */
}
```

PushButton

We have added a method, `DoPress`, that allows us to programmatically press the PushButton. This will result in the visual depression of the button as well as the `WM_COMMAND` message sent to our `Client` resulting in the call to our `PushedButton` abstract method. Other than that, it is the same as the last example.

pushb.idl

```
#include <window.idl> // get definition of your parent

interface PushButton;

// Metaclass definition

interface M_PushButton: M_Window
{

#ifdef __SOMIDL__
implementation {
    //# Class Modifiers
    filestem = pushb;
    functionprefix = m_;

    //# Method Modifiers
    override: CreateWindow;
};

#endif
};

// Class definition

interface PushButton: Window {
    void DoPress();    // programmatic button press
    void DoRelease(); // programmatic button release

#ifdef __SOMIDL__
implementation
{
```

```

//# Class Modifiers
filestem = pushb;
metaclass = M_PushButton;
releaseorder:
    DoPress, DoRelease;

};

#endif

};

```

pushb.c

```

#define PushButton_Class_Source
#include <pushb.ih>

#include <stdio.h>

/*
 * allow for programmatic press of push button
 */
SOM_Scope void SOMLINK DoPress(PushButton somSelf,
                                Environment *ev)
{
    /* PushButtonData *somThis = PushButtonGetData(somSelf);
 */
    HWND hwnd = __get_hwndWindow(somSelf, ev);
    PushButtonMethodDebug("PushButton", "DoPress");
    if(hwnd)
        WinSendMsg(hwnd, BM_CLICK,
                   MPFROMSHORT(TRUE), NULL);
    return;
}

/*
 * allow for programmatic release of push button
 */
SOM_Scope void SOMLINK DoRelease(PushButton somSelf,
                                 Environment *ev)
{
    /* PushButtonData *somThis = PushButtonGetData(somSelf);
 */
    HWND hwnd = __get_hwndWindow(somSelf, ev);
    PushButtonMethodDebug("PushButton", "DoRelease");
    if(hwnd)
        WinSendMsg(hwnd, BM_CLICK, MPFROMSHORT(FALSE), NULL);
    return;
}

```

```

SOM_Scope Window SOMLINK m_CreateWindow(M_PushButton somSelf,
                                         Environment *ev, Window parent, Window owner,
                                         long style,
                                         string initText, short x, short y, short w, short
                                         h)
{
    /* M_PushButtonData *somThis =
    M_PushButtonGetData(somSelf); */
    PushButton pbNew = PushButtonNew();

    HWND hwndParent, hwndOwner, hwnd;
    USHORT idPushButton = _UniqueID(pbNew, ev);
    ERRORID errorid;
    USHORT i;

    M_PushButtonMethodDebug( "M_PushButton", "m_CreateWindow" );

    hwndParent = __get_hwndWindow(parent, ev);
    hwndOwner = __get_hwndWindow(owner, ev);

    __set(hwndParent, pbNew, ev, hwndParent);
    __set(hwndOwner, pbNew, ev, hwndOwner);

    hwnd = WinCreateWindow(
        hwndParent,
        WC_BUTTON,                      /* window class */
        initText,                        /* window text */
        BS_PUSHBUTTON | style,          /* window style */
        (long)x, (long)y,                /* position */
        (long)w, (long)h,                /* size */
        hwndOwner,                       /* owner window */
        HWND_BOTTOM,                    /* placement */
        idPushButton,                   /* child window */
        NULL,                           /* Control data */
        NULL);                          /* pres. params */

    /* store the object pointer, pbNew */
    /* in the PM window's word indexed by QWL_USER */
    WinSetWindowPtr(hwnd, QWL_USER, pbNew);
    __set(hwndWindow(pbNew, ev, hwnd));
    __set(hwndFrame(pbNew, ev, NULLHANDLE));

    return pbNew;
}

```

PushButtons and piano keys

What can we do with just a bunch of push-buttons? In this application we use them as the keys of a small piano—just one octave. We also allow for the pressing of PushButton piano keys by touching keys on your keyboard.

The userData attribute

What is the best way to store SOM object information that is related to that object but not necessarily a part of its? As I mentioned earlier, PM windows give us 32 bits to store user information. We get to that information using one of the PM APIs that allow us to store and query information from that location:

```
WinSetWindowPtr/WinQueryWindowPtr
WinSetWindowULong/WinQueryWindowULong
WinSetWindowUShort/WinQueryWindowUShort
```

We use QWL_USER as the index to find the information from the window words. In our set of wrapped controls, we use that word to store our SOM object reference. Our wrapped controls will provide a similar place that a programmer can use to associate data with his wrapped control. We introduce the userData attribute to the Window class so that all wrapped controls can use this word to store its own information. In this example, the buttons store an index into a frequency table, so that when they are pressed they can "beep" at the correct frequency.

app1.idl

```
#include <client.idl> // get definition of your parent

interface App1: Client // App1 is a subclass of Client
{
    attribute short timesPressed;

#ifdef __SOMIDL__
implementation
{
    /* Class Modifiers
    classinit: app1Init;

    releaseorder:
        _get_timesPressed, _set_timesPressed;

    /* Method Modifiers

    CharPress: override;
    PushedButton: override;

    somInit: override;
};
```

```
#endif  
};
```

app1.c

```
#define App1_Class_Source  
  
#define INCL_DOSPROCESS /* for the DosBeep call used below */  
/*  
#include <app1.ih>  
#include <pushb.h>  
  
#include <stdio.h>  
#include <string.h>  
#include <math.h>  
  
#define TOTALKEYS 13  
static USHORT frequency_table[TOTALKEYS];  
static PushButton PBTTable[TOTALKEYS];  
  
SOM_Scope void SOMLINK somInit(App1 somSelf)  
{  
    App1Data *somThis = App1GetData(somSelf);  
    double freq;  
    short i;  
    char keyText[2];  
    /* using keySave to traverse a C scale so initialize to one  
    less than 'C' */  
    char keySave='B';  
    char thisKey;  
    USHORT whiteKey = 0;  
    static short x=0,y=10,w=30,h=300;  
    short xB=-20, xW=10;  
    Environment *ev = somGetGlobalEnvironment();  
    BYTE fgRGB, bgRGB;  
  
    fprintf(stderr,"somInit of app1.c: begin\n");  
    App1_parent_Client_somInit(somSelf);  
  
    PushButtonNewClass(0,0);  
    freq=220; /* initial frequency (400 Hz or middle A */  
    keyText[1] = '\0';  
    for(i=0;i<TOTALKEYS;i++) {  
        /* initialize the frequency table. Each note on a scale has a  
        * frequency which can be calculated using a formula derived  
        * by knowing that every note is  
        * twice the frequency of the same note an octave below.  
        */  
        frequency_table[i] =
```

```

        (short)(freq * pow((double)2,
(double)i/(double)12.0));
        fprintf(stderr,"-%d freq=%d\n", i+1,
frequency_table[i]);
        switch(i) {
/* these are the 'white keys'. The colors we choose will be
 * a black foreground and white background
*/
        case 0: case 2: case 4:
        case 5: case 7: case 9:
        case 11: case 12:
            fgRGB = 0;
            bgRGB = 255;
            if(++keySave>'G')
                keySave='A';
            keyText[0]=keySave;
            x = xW;
            y = 10;
            w = 70;
            h = 100;
            xW+=75;
/* black keys always follow white keys. reverse not always
true */
            xB+=75;
            break;
        case 1: case 3: case 6: case 8: case 10:
/* No text needed on the black keys. Our code's not
sophisticated */
/* enough to know whether we are dealing with sharps or flats
*/
/* The user will know whether the key between C and D is C# or
*/
/* D-flat */
            keyText[0]='\0';
/* white foreground on black background */
            fgRGB = 255;
            bgRGB = 0;
            x = xB;
            y = 110;
            w = 55;
            h = 200;
            break;
}
PBTTable[i] = _CreateWindow(_PushButton, ev,
    somSelf,      /* parent */
    somSelf,      /* owner */
    WS_VISIBLE,   keyText,
    x, y, w, h);
SetColorFG(PBTTable[i], ev, fgRGB, fgRGB,
    fgRGB);
SetColorBG(PBTTable[i], ev, bgRGB, bgRGB,
    bgRGB);

```

```

        bgRGB);
    __set(userData(PBTable[i], ev, i);
}
}

SOM_Scope void SOMLINK PushedButton(App1 somSelf,
                                    Environment *ev,
                                    Window sourceWindow)
{
    App1Data *somThis = App1GetData(somSelf);
    char strBuf[20];

    _timesPressed++;
    sprintf(strBuf, "Pressed %d times", _timesPressed);
/* set the text on the title bar
 */
    __set_text(somSelf, ev, strBuf);
    DosBeep(frequency_table[__get(userData(sourceWindow, ev)],
                           250);
}

/* CharPress handles keyboard entry and allows a user to sound
notes by pressing the corresponding key on the keyboard. For
example, to hear the note 'F' type on the keyboard either an
'f' or an * 'F'. Use capital 'C' for the top of the octave and
small 'c' for * the bottom.
*/
SOM_Scope void SOMLINK CharPress(App1 somSelf,
                                 Environment *ev,
                                 char key,
                                 short kbState)
{
    App1Data *somThis = App1GetData(somSelf);
    unsigned short i;
    static unsigned short keynum=0;

    switch(key) {
        case 'c':
            keynum = 0;
            break;
        case 'C': /* use capital C for the top of our octave
 */
            keynum = 12;
            break;
        case 'D': case 'd':
            keynum = 2;
            break;
        case 'E': case 'e':
            keynum = 4;
            break;
        case 'F': case 'f':

```

```

        keynum = 5;
        break;
    case 'G': case 'g':
        keynum = 7;
        break;
    case 'A': case 'a':
        keynum = 9;
        break;
    case 'B': case 'b':
        keynum = 11;
        break;
    default:
        if(keynum<(TOTALKEYS-1))
            keynum++;
        else
            keynum=0;
        break;
    }
    _DoPress(PBTable[keynum], ev);
}

void SOMLINK applInit(SOMClass *cls)
{
    /* you could initialize your frequency table here */
}

```

wrap2.c

```

#include <appl.h>
void main(int argc, char *argv[])
{
    App1 obj = App1New();
    _ProcessEvents(_Window, somGetGlobalEnvironment());
    _somFree(obj);
}

```

makefile — for wrapped controls example two

```

LIB = $(LIB)
INCLUDE = .;$(INCLUDE)
SMEMIT = ih;h;c;
SRCS = window. client. pushb. appl.
HDRS = $(SRCS:.ih)
OBJS = $(SRCS:.c=.obj) wrap2.obj

install: wrap2.exe

.SUFFIXES : .ih .sc .lib .def .dll .idl

```

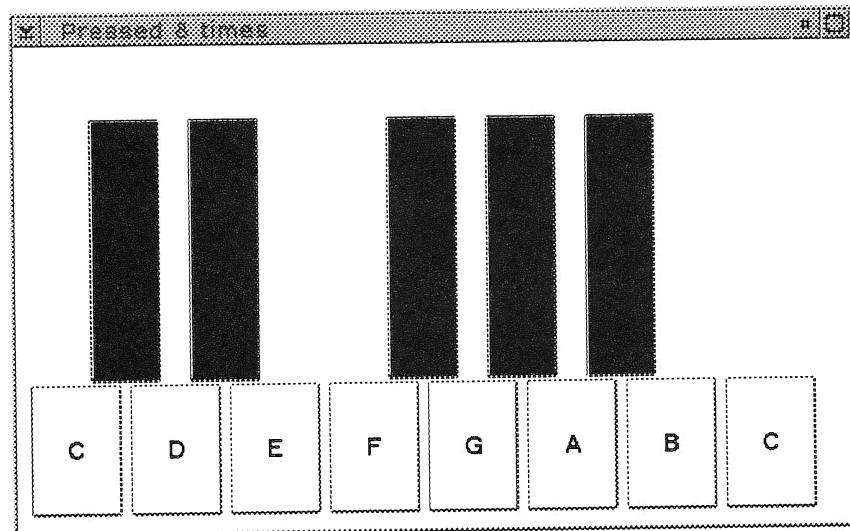


Figure 17.2. Output for WRAP2.EXE

```
run:  
      wrap2  
.c.obj:  
      icc -Q -Ti+ -c $<  
  
.idl.ih:  
      sc $*  
# note the change in the next line:  
# I now make all headers before I make any of the objects.  
# this is a good practice.  
wrap2.exe: $(HDRS) $(OBJS)  
      link386 /NOL /NOI /CO /PM:PM $(OBJS), wrap2.exe,  
nul,somtk;  
window.ih: window.idl  
window.obj: window.ih window.c  
  
client.ih: client.idl  
client.obj: client.ih client.c  
  
pushb.ih: pushb.idl  
pushb.obj: pushb.ih pushb.c  
  
app1.ih: app1.idl  
app1.obj: app1.ih app1.c  
  
wrap2.obj: wrap2.c
```

Run the PushButton Piano

Now, we are ready to run the PushButton piano. At your OS/2 command line enter:

D:\example> wrap2

The application should look like Figure 17.2.

Press your mouse button number one on a key to hear the corresponding note. You can also press a key on the keyboard corresponding to the note you would like to hear.

Exercises

1. Why might it be important to set up a makefile so that all headers are created before any implementation 'C' code is compiled?

18

Let's Wrap!

INTRODUCTION

We have spent a lot of time wrapping a push-button. Now that we know the basics of how to wrap a PM Control, let's wrap a few more. You will soon find that not all PM controls can be wrapped in quite the same way as a push-button. In the next example we will wrap some more controls, and we will note how they differ from the push-button. We'll wrap the following new controls:

- Static Text
- ListBox
- EntryField
- ComboBox

and we will once again use our wrapped `PushButton`. The class hierarchy is shown in Figure 18.1.

We are going to do a lot of things with this final PM example. Among them we will:

- Wrap multiple PM controls.
- Implement a constructor method for our application class, thus making all of our `Window` class objects *instantiable* with a common constructor method. This will enhance our extensibility.

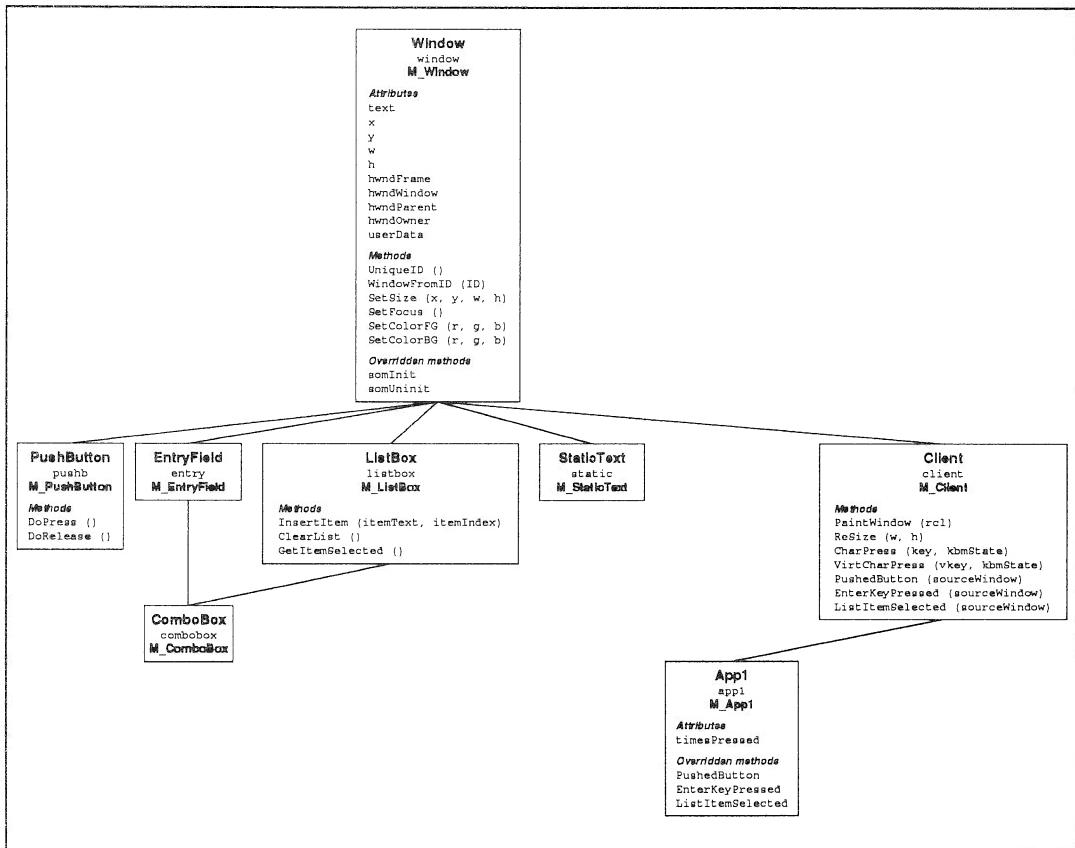


Figure 18.1. WRAP3 Hierarchy

- Build our frameworks into a dynamically linkable library to enhance reusability.
- Use multiple inheritance when we wrap the PM combo box.

Wrapping more PM Controls

As in the case of the push-button, our PM Controls send messages to their owners. In our examples the owner will usually be a Window object subclassed from the **Client** class. In the PushButton, the only message we had to handle in the Client window procedure was the WM_COMMAND message. In most controls, however, the PM message WM_CONTROL is sent by the controls to the owner to inform it of activity taking place within the controls that it owns. See the first section on wrapped controls for an introduction to how these high-level messages are used.

The WM_CONTROL message contains the ID of the control that sent the message. We can use this ID to get the PM window handle using the PM API `WinWindowFromID()`. From the PM window handle we can get the SOMObject

pointer using the familiar `WinQueryWindowPtr(...)` call that we used with the `PushButton` class. Along with the ID, the `WM_CONTROL` message contains a notify code.

```

Window sourceWindow; /* a SOMobject */
SHORT winID, notifyCode;
HWND winHwnd;
.....
case WM_CONTROL:
    winID = SHORT1FROMMP(mp1);
    winHwnd = WinWindowFromID(hwnd, winID);
    sourceWindow = WinQueryWindowPtr(winHwnd, 0L);
    notifyCode = SHORT2FROMMP(mp1);
    if(somIsObj(sourceWindow)) { /* be sure it's a
                                   SOMobject */
        switch(notifyCode) {
        .....

```

Each control has a set of unique notify codes; for example, the listbox notify codes are all #defined using the identifier `LN_` to be integer values 1 through 5, and the combobox notify codes all use prefix `CBN_`. However, the codes of one PM control are not different from the codes of another. For example, the following three notification messages for three different controls are all #defined to equal the integer value of 4:

`EN_CHANGE`, `CBN_LBSELECT`, and `LN_SCROLL`.

The combination of `WM_CONTROL` message, Control ID, and notify code are still not enough to tell us what class of PM control we are dealing with. Therefore, we need another call to help us determine from what class our source control is derived. This is important because we will be calling different methods based on the type of class that is being acted upon. This new call is a SOM method, `_so-mIsA`, to help us ensure that the source control is of a class that can be used for a particular method.

In this example we also use multiple inheritance to help us wrap the PM combo-box. Why does using multiple inheritance here make sense? By definition, the PM combo-box uses all the same methods as the listbox and the entry field. Multiple inheritance allows `ComboBox` objects to inherit the behavior of both of its parents, so it is not necessary to rewrite several of the methods.

WINDOW.IDL and WINDOW.C are the same as in last example

See a previous example for a description and source listing.

Using a metaclass with Client

The `Client` class interface definition has now defined an explicit metaclass, `M_Client`. We did this so that we could also inherit the `Window` class's con-

structor method `CreateWindow`. This modification is a big improvement in our design, because now we are consistent in the way that we create `Window` objects. Up until now, our `Client` and `App` classes were not treated like any other window, but they really should be. Our application objects should be able to be positioned, sized, Z-ordered, etc, just like any other object. Once we create an application, it becomes a flexible, reusable, and interchangeable `Window` object. We would like to either run it alone on the PM desktop or imbed it in another application.

The Client Class

The `Client` class has a few new abstract methods that the client will call when various high level control messages are received. The `Client` class implementation has become much more complicated. We use the control ID, the notify code, and the `_somIsA` method to help us determine the source of the message being sent to our client window.

client.idl

```
#ifndef client_idl
#define client_idl

#include <window.idl> // get parent def

// Metaclass definition

interface M_Client: M_Window
{
    #ifdef __SOMIDL__
    implementation {
        ## Class Modifiers
        filestem = client;
        functionprefix = m_;

        ## Method Modifiers
        override: CreateWindow;
    };
    #endif
};

interface Client: Window // your classname and your parent(s)
{
    void PaintWindow(in RECTL *rcl);
    void ReSize(in short w, in short h);
    void CharPress(in char key, in short kbmState);
    void VirtCharPress(in short vkey, in short kbmState);

    ## Control Events. Command (WM_COMMAND) and Control
}
```

```

//# Notifications (WM_CONTROL)
void PushedButton(in Window sourceWindow);
// this method gets called when a user presses down and
// up on a pushbutton. The
// sourceWindow parameter indicates the particular
// SOMObject that was pressed.

void EnterKeyPressed(in Window sourceWindow);
// this is called when a user hits Enter or NewLine
// in an Entry Field or ComboBox.

void ListItemSelected(in Window sourceWindow);
// this is called when a user makes a selection in a
// ListBox or ComboBox

#ifndef __SOMIDL__
implementation
{
    // Class Modifiers
    releaseorder:
        PaintWindow, ReSize, CharPress, VirtCharPress,
        PushedButton, EnterKeyPressed, ListItemSelected;
    classinit: clientInit;
    metaclass = M_Client;

};

#endif

};

#endif /* #ifndef client_idl */

```

client.c

```

#define Client_Class_Source
#include <client.ih>

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <listbox.h>
#include <combobox.h>

#define NOGRAPHICS      /* */

FNWP vwClientProc;

/* strBuf is used by many of the routines to print messages*/
static char strBuf[100];

```


/* it is sometimes necessary to check whether the **sourceWindow** is actually a SOM Object. This may seem redundant since we have wrapped all of our objects in our application. We know all of our windows have the SOM Object pointer stored in the WindowPtr word. This is not quite true. Some of the controls which we wrap are actually composite windows. A composite window is made up of multiple windows. For example, the comboBox is made up of an entry field, a listbox, and a PushButton. Any of those three windows, or the combobox window itself, are capable of generating PM messages on their own. */

```

        if(somIsObj(sourceWindow) &&
!_somIsA(sourceWindow, ComboBoxNewClass(0,0))
)
{
    _EnterKeyPressed(somSelf, ev,
                     sourceWindow);
    break;
default:
    break;
};

}
break;
case WM_SIZE:
    if(somIsObj(somSelf)) {
        w = SHORT1FROMMP(mp2);
        h = SHORT2FROMMP(mp2);
        _ReSize(somSelf, ev, w, h);
    }
    break;
#endif NOGRAPHICS
case WM_ERASEBACKGROUND:
    return (MRESULT)TRUE;
#endif
case WM_PAINT:
    if (somIsObj(somSelf)) {
        hps = WinBeginPaint(hwnd, NULLHANDLE, &rcl);
        _PaintWindow(somSelf, ev, &rcl);
        WinEndPaint(hps);
        return FALSE;
    }
    break;
case WM_COMMAND:
/* source of WM_COMMAND. */
src = SHORT1FROMMP(mp2);
winID = SHORT1FROMMP(mp1);
winHwnd = WinWindowFromID(hwnd, winID);
sourceWindow = WinQueryWindowPtr(winHwnd, 0L);
switch(src) {
    case CMDSRC_PUSHSBUTTON:
        _PushedButton(somSelf, ev,

```

```

                                sourceWindow) ;
                                break;
default:
    fprintf(stderr, " pushb.c ERROR:
                      shouldn't have executed
                      pbSubProc\n");
    break;
}
return 0;
case WM_CONTROL:
    winID = SHORT1FROMMP(mp1);
    winHwnd = WinWindowFromID(hwnd, winID);
    sourceWindow = WinQueryWindowPtr(winHwnd, 0L);
    if(_somIsObj(sourceWindow)) {
        switch(SHORT2FROMMP(mp1)) {
/* the set of control notification messages,
 * for example LN_... for listbox and CBN_ ... for
 * combobox, are not unique. The following three
 * notification messages for three controls
 * are all #defined as 4:
 *      EN_CHANGE, CBN_LBSELECT, and LN_SCROLL.
 * Therefore we must use the som utility, _somIsA,
 * to help ensure that the control causing the event * is of
 * a class that can be used for a particular
 * method.
 */
        case LN_SELECT:
/* be careful: */
/* could also have been CBN_EFCHANGE,
 * or EN_SETFOCUS */
/* (all #defined as 1) */
        if(_somIsA(sourceWindow,
ListBoxNewClass(0,0))
            && !_somIsA(sourceWindow,
ComboBoxNewClass(0,0)) ) {

_sListItemSelected(somSelf, ev,
                                sourceWindow);
    }
    return 0;
case CBN_ENTER:
    if( _somIsA(sourceWindow,
                  ComboBoxNewClass(0,0)) ) {
        _EnterKeyPressed(somSelf, ev,
                                sourceWindow);
    }
    return 0;
case CBN_LBSELECT:
/* be careful: could also have been
 * EN_CHANGE, or LN_SCROLL

```

```
* (all #defined as 4) */
    if(_somIsA(sourceWindow,
                ComboBoxNewClass(0,0))) {
        _ListItemSelected(somSelf, ev,
                           sourceWindow);
    }
    return 0;
} /* end of switch(sourceWindow) */
} /* end of if(somIsObj(sourceWindow)) */
default:
    break;
}
return WinDefWindowProc(hwnd, msg, mp1, mp2);
}

/*
 * The following are stub methods designed to be overridden
 */

SOM_Scope void  SOMLINK PaintWindow(Client somSelf,
                                      Environment *ev,
                                      RECTL* rcl)
{ }

SOM_Scope void  SOMLINK ReSize(Client somSelf,
                               Environment *ev,
                               short w, short h)
{ }

/*
 * PushedButton is called when a user presses down and up
 * on a PushButton. The sourceWindow parameter indicates
 * the particular SOMObject that was pressed.
 */
SOM_Scope void  SOMLINK PushedButton(Client somSelf,
                                      Environment *ev,
                                      Window id)
{ }

/*
 * EnterKeyPressed is called when a user hits Enter
 * in a client or any of its controls, such as the entry
 * field. The sourceWindow parameter indicates the window in *
 * focus at the time of the Enter Press.
 */
SOM_Scope void  SOMLINK EnterKeyPressed(Client somSelf,
                                         Environment *ev, Window sourceWindow)
{ }

SOM_Scope void  SOMLINK CharPress(Client somSelf,
```


/* In most of the CreateWindow implementations for the other wrapped PM controls, we created our new Window object by saying <className>New(); but in this example ClientNew would not work, so we use the SOM runtime library method _somNew. The reason we do this is because the client class has been designed to be overridden, and the Client implementation here does not know what the name of the class should be. As the writer of this example we happen to know that in this example the class being created is an App1 object, but our Client class will be reused by many application objects, especially now that we have our Controls in a nice DLL so we can't use App1New(). ClientNew() is also wrong because our App1 will not be initialized properly. Using _somNew on the class object somSelf will create an instance of the proper class. */

```

/* M_ClientData *somThis = M_ClientGetData(somSelf); */
Client newClient = _somNew(somSelf);
static ULONG flFrameFlags = FCF_TITLEBAR      |
                           FCF_SYSMENU     |
                           FCF_SIZEBORDER  |
                           FCF_MINMAX     |
                           FCF_SHELLPOSITION |
                           FCF_TASKLIST;

HWND hwndTitleBar, hwndFrame,
hwndClient, hwndParent, hwndOwner;
string className = "Wrapped Controls";
M_ClientMethodDebug("M_Client", "m_CreateWindow");
if(parent) {
    hwndParent = __get_hwndWindow(parent, ev);
    hwndOwner = __get_hwndWindow(owner, ev);
} else {
    hwndParent = hwndOwner = HWND_DESKTOP;
}
/* To make this Client base class smarter, we should give the
option of sometimes creating a standard PM Client here, rather
than creating a Framed standard composite window (as we are
doing on the next line with the PM API WinCreateStdWindow). As
long as our client is a top-level window, or as long as we
want all our clients to come with this canned set of Frame
Creation Flags, this design is fine. It is left as an exercise
for the student to make the Client's implementation of the
CreateWindow method smarter.*/
hwndFrame = WinCreateStdWindow(
    hwndParent,
    0,      /* I turned off the WS_VISIBLE flag here */
    &flFrameFlags,
    className,
    NULL,
    0L,
    (HMODULE) 0,

```

```

        0L,      /* resource ID */
        &hwndClient);
    if((hwndTitleBar = WinWindowFromID (hwndFrame,
                                         FID_TITLEBAR)) != 0L
        WinSetWindowText (hwndTitleBar, className);
        WinSetWindowPos(
            hwndFrame,
            HWND_TOP,
            x, y, w, h,
            SWP_ACTIVATE | SWP_SHOW |
                SWP_MOVE | SWP_SIZE | SWP_ZORDER);
/* store newClient Window pointer in window's user *space */
/* for both the client and frame */
    WinSetWindowPtr(hwndClient, 0L, newClient);
    WinSetWindowPtr(hwndFrame, 0L, newClient);

/* further initialize the new Client object */
    __set_hwndWindow(newClient, ev, hwndClient);
    __set_hwndFrame (newClient, ev, hwndFrame);

    __set_hwndParent (newClient, ev, hwndFrame);
    __set_hwndOwner (newClient, ev, hwndFrame);

    return newClient;
}

```

EntryField

The entry field is a control that allows a user to enter a single line of text. The entry field supports the standard OS/2 marking, cutting, and pasting model. If the parent for the entry field is a Client object, the Client object will call a EnterKeyPressed when Enter is pressed in the entry field.

/* ENTRY.IDL */

```

#ifndef entry_idl
#define entry_idl

#include <window.idl> // get definition of your parent

interface EntryField;

// Metaclass definition

interface M_EntryField: M_Window
{

#ifndef __SOMIDL__
implementation {
    // Class Modifiers

```

```

filestem = entry;
functionprefix = m_;

//# Method Modifiers
override: CreateWindow;
};

#endif
};

// Class definition

interface EntryField: Window // class name and parents
{

#ifndef __SOMIDL__
implementation
{
    //# Class Modifiers
    filestem = entry;
    metaclass = M_EntryField;

};
#endif

};

#endif /* #ifndef entry_idl */

```

ENTRY.C

```

#define EntryField_Class_Source
#define M_EntryField_Class_Source
#include <entry.ih>

SOM_Scope Window SOMLINK m_CreateWindow(M_EntryField somSelf,
                                         Environment *ev, Window parent,
                                         Window owner,
                                         long style, string initText,
                                         short x, short y, short w, short h)
{
    /* M_EntryFieldData *somThis =
M_EntryFieldGetData(somSelf); */
    HWND hwndParent, hwndOwner, hwnd;
    EntryField efNew = EntryFieldNew();
    USHORT idEntryField = _UniqueID(efNew, ev);
    ERRORID errorid;
    USHORT i;

    M_EntryFieldMethodDebug("M_EntryField", "m_CreateWindow");
    /* if style is 0 give the user a default style */
    if(!style)

```

```

        style = WS_VISIBLE | ES_AUTOSCROLL | ES_MARGIN;

        hwndParent = __get_hwndWindow(parent, ev);
        hwndOwner = __get_hwndWindow(owner, ev);

        __set_hwndParent(efNew, ev, hwndParent);
        __set_hwndOwner(efNew, ev, hwndOwner);

        hwnd = WinCreateWindow(
            hwndParent,
            WC_ENTRYFIELD, /* window class */
            initText,      /* window text */
            style,         /* window style */
            (long)x, (long)y,      /* position */
            (long)w, (long)h,      /* size */
            hwndOwner,      /* owner window */
            HWND_BOTTOM,    /* placement */
            idEntryField,   /* child window id */
            NULL,           /* Control data */
            NULL);          /* pres. params */

/* store the object pointer */
/* in the PM window's word indexed by QWL_USER */
        WinSetWindowPtr(hwnd, QWL_USER, efNew);
        __set_hwndWindow(efNew, ev, hwnd);
        __set_hwndFrame(efNew, ev, NULLHANDLE);
        return efNew;
    }
}

```

ListBox

The `ListBox` class corresponds directly to the PM listbox control. The list box contains an array of strings. A single string (shown as a row in the list box) can be selected, or multiple strings can be selected, depending on the style chosen when creating the listbox.

LISTBOX.IDL

```

#ifndef listbox_idl
#define listbox_idl

#include <window.idl> // get definition of your parent

interface ListBox;

// Metaclass definition

interface M_ListBox: M_Window
{

```

```

#ifndef __SOMIDL__
implementation {
    //# Class Modifiers
    filestem = listbox;
    functionprefix = m_;

    //# Method Modifiers
    override: CreateWindow;
};

#endif
};

// Class definition

interface ListBox: Window      // your classname and your
parent(s)
{
    short InsertItem(in string itemText, in short
itemIndex);
    // itemIndex can be the zero-based index indicating
    // the desired placement of the itemText, or it can
    // be one of the listbox constants:
    // LIT_SORTASCENDING, LIT_SORTDESCENDING, LIT_END
    void ClearList();
    string GetItemSelected();
}

#ifndef __SOMIDL__
implementation
{
    //# Class Modifiers
    filestem = listbox;
    metaclass = M_ListBox;
    releaseorder:
        InsertItem, ClearList, GetItemSelected;

};
#endif

};

#endif /* listbox_idl */

```

LISTBOX.C

```

#define ListBox_Class_Source
#define M_ListBox_Class_Source
#include <listbox.ih>

```

```

/*
 * itemIndex can be the zero-based index indicating the
 * desired placement of the itemText, or it can be one
 * of the listbox constants:
 * LIT_SORTASCENDING, LIT_SORTDESCENDING,
 * LIT_END.
 */

SOM_Scope short SOMLINK InsertItem(ListBox somSelf,
Environment *ev, string itemText, short
itemIndex)
{
    /* ListBoxData *somThis = ListBoxGetData(somSelf); */
    HWND hwnd = __get_hwndWindow(somSelf, ev);
    ListBoxMethodDebug("ListBox", "InsertItem");

    if(hwnd)
        return (short)WinSendMsg(hwnd, LM_INSERTITEM,
                               MPFROMSHORT(itemIndex),
                               MPFROMP(itemText));
    else
        return -1;
}

SOM_Scope void SOMLINK ClearList(ListBox somSelf,
Environment *ev)
{
    /* ListBoxData *somThis = ListBoxGetData(somSelf); */
    HWND hwnd = __get_hwndWindow(somSelf, ev);

    ListBoxMethodDebug("ListBox", "ClearList");
    if(hwnd)
        WinSendMsg(hwnd, LM_DELETEALL, 0, 0);
    return;
}

SOM_Scope string SOMLINK GetItemSelected(ListBox somSelf,
                                         Environment *ev)
{
    /* ListBoxData *somThis = ListBoxGetData(somSelf); */
    short itemIndex, itemLength;
    static string itemText=NULL;
    HWND hwnd = __get_hwndWindow(somSelf, ev);
    ListBoxMethodDebug("ListBox", "GetItemSelected");

    itemIndex = (SHORT)WinSendMsg(hwnd, LM_QUERYSELECTION,
                                 NULL, NULL);
    /* single selection style makes mp1 and mp2 ignored */
    if(itemIndex == LIT_NONE)
        return NULL;
}

```

```
    itemLength = (SHORT)WinSendMsg(hwnd,
LM_QUERYITEMTEXTLENGTH,
                                MPFROMSHORT(itemIndex),NULL);
    itemText = (string)malloc(itemLength+1);
    WinSendMsg(hwnd, LM_QUERYITEMTEXT,
                MPFROM2SHORT(itemIndex, itemLength+1),
itemText);
    return itemText;
}

SOM_Scope Window SOMLINK m_CreateWindow(M_ListBox somSelf,
                                         Environment *ev, Window parent,
                                         Window owner,
                                         long style, string initText,
                                         short x, short y, short w, short h)
{
    /* M_ListBoxData *somThis = M_ListBoxGetData(somSelf); */
    ListBox lbNew = ListBoxNew();
    HWND hwndParent, hwndOwner, hwnd;
    USHORT idListBox = _UniqueID(lbNew, ev);
    ERRORID errorid;
    USHORT i;

    M.ListBoxMethodDebug ("M.ListBox", "m_CreateWindow");

    if(!style)
        style = WS_VISIBLE | LS_NOADJUSTPOS;
    hwndParent = __get_hwndWindow(parent, ev);
    hwndOwner = __get_hwndWindow(owner, ev);

    __set_hwndParent(lbNew, ev, hwndParent);
    __set_hwndOwner(lbNew, ev, hwndOwner);

    hwnd = WinCreateWindow(
        hwndParent,
        WC_LISTBOX, /* window class*/
        initText, /* window text */
        style, /* window style */
        (long)x, (long)y, /* position */
        (long>w, (long)h, /* size */
        hwndOwner, /* owner */
        HWND_BOTTOM, /* placement */
        idListBox, /* child window id */
        NULL, /* Control data */
        NULL); /* pres. params */

    /* store object pointer */
    /* in the PM window's word indexed by QWL_USER */
    WinSetWindowPtr(hwnd, QWL_USER, lbNew);
    __set_hwndWindow(lbNew, ev, hwnd);
```

```
    __set_hwndFrame(lbNew, ev, NULLHANDLE);  
  
    return lbNew;  
  
}
```

Static Text

This class corresponds directly to the PM static text control and is similarly used for read-only labels. `StaticText` objects will never take the focus; rather attempts to set focus on the `StaticText` object will result in focus being set on the next sibling in the view tree.

STATIC.IDL

```
#ifndef static_idl  
#define static_idl  
  
#include <window.idl> // get definition of your parent  
  
interface StaticText;  
  
// Metaclass definition  
  
interface M_StaticText: M_Window  
{  
  
#ifdef __SOMIDL__  
implementation {  
    //## Class Modifiers  
    filestem = static;  
    functionprefix = m_;  
  
    //## Method Modifiers  
    override: CreateWindow;  
};  
#endif  
};  
  
// Class definition  
  
interface StaticText: Window // classname and parent(s)  
{  
  
#ifdef __SOMIDL__  
implementation  
{  
    //## Class Modifiers  
    filestem = static;  
    metaclass = M_StaticText;
```

```
};

#endif

};

#endif /* #ifndef static_idl */
```

STATIC.C

```
#define StaticText_Class_Source
#define M_StaticText_Class_Source
#include <static.ih>

SOM_Scope Window SOMLINK m_CreateWindow(M_StaticText
                                         somSelf,
                                         Environment *ev,
                                         Window parent,
                                         Window owner,
                                         long style,
                                         string initText,
                                         short x, short y,
                                         short w, short h)
{
    /* M_StaticTextData *somThis =
       M_StaticTextGetData(somSelf); */
    StaticText stNew = StaticTextNew();

    HWND hwndParent, hwndOwner, hwnd;
    USHORT idStaticText = _UniqueID(stNew, ev);
    ERRORID errorid;
    USHORT i;

    M_StaticTextMethodDebug("M_StaticText",
                           "m_CreateWindow");

    M_StaticTextMethodDebug("M_StaticText",
                           "m_CreateWindow");

    if(!style)
        style = WS_VISIBLE | SS_TEXT | DT_CENTER
               | DT_VCENTER;

    hwndParent = __get_hwndWindow(parent, ev);
    hwndOwner = __get_hwndWindow(owner, ev);

    __set(hwndParent, ev, hwndParent);
    __set(hwndOwner, ev, hwndOwner);

    hwnd = WinCreateWindow(
        hwndParent,
```

```

        WC_STATIC,           /* window class */
        initText,            /* window text */
        style,               /* window style */
        (long)x, (long)y,    /* position */
        (long)w, (long)h,    /* size */
        hwndOwner,           /* owner window */
        HWND_BOTTOM,         /* placement */
        idStaticText,        /* child window id */
        NULL,                /* Control data */
        NULL);               /* pres. params */

/* store the object pointer */
/* in the PM window's word indexed by QWL_USER */
WinSetWindowPtr(hwnd, QWL_USER, stNew);
__set_hwndWindow(stNew, ev, hwnd);
__set_hwndFrame(stNew, ev, NULLHANDLE);

return stNew;
}

```

PUSHB.IDL and PUSHB.C are the same as the previous example

/* See Previous example for source code */

ComboBox

This corresponds to the PM ComboBox control, which has features of both the list box and the entry field. We use multiple inheritance to capture this behavior. By definition, all the methods that are supported by the ListBox and EntryField are also supported by the ComboBox.

COMBOBOX.IDL

```

#ifndef combobox_idl
#define combobox_idl

#include <entry.idl>      // get definition of your parents
#include <listbox.idl>

interface ComboBox;

// Metaclass definition

interface M_ComboBox: M_Window
{

#ifdef __SOMIDL__
implementation {
    /* Class Modifiers
    filestem = combobox;

```

```

        functionprefix = m_;

        //## Method Modifiers
        override: CreateWindow;
    };
#endif
};

// Class definition
// identify the name of this class and your parent(s)
interface ComboBox: EntryField, ListBox
{
}

#ifndef __SOMIDL__
implementation
{
    //## Class Modifiers
    filestem = combobox;
    metaclass = M_ComboBox;

};
#endif

};

#endif /* #ifndef combobox_idl */

```

COMBOBOX.C

```

#define ComboBox_Class_Source
#define M_ComboBox_Class_Source
#include <combobox.ih>

SOM_Scope Window SOMLINK m_CreateWindow(M_ComboBox somSelf,
                                         Environment *ev, Window parent, Window owner,
                                         long style, string initText,
                                         short x, short y, short w, short h)
{
    /* M_ComboBoxData *somThis = M_ComboBoxGetData(somSelf); */
    ComboBox comboNew = ComboBoxNew();
    HWND hwndParent, hwndOwner, hwnd;
    USHORT idComboBox = _UniqueID(comboNew, ev);
    ERRORID errorid;
    USHORT i;
    M_ComboBoxMethodDebug ("M_ComboBox", "m_CreateWindow");

    if(!style)      /* give a default */
        style = WS_VISIBLE | CBS_DROPDOWN;
    hwndParent = __get_hwndWindow(parent, ev);
    hwndOwner = __get_hwndWindow(owner, ev);

```

```

    __set_hwndParent(comboNew, ev, hwndParent);
    __set_hwndOwner(comboNew, ev, hwndOwner);

    hwnd = WinCreateWindow(
        hwndParent,
        WC_COMBOBOX,           /* window class */
        initText,               /* window text */
        style, /* window style */
        (long)x, (long)y,       /* position */
        (long)w, (long)h,       /* size */
        hwndOwner,              /* owner window */
        HWND_BOTTOM,            /* placement */
        idComboBox,             /* child window id */
        NULL,                  /* Control data */
        NULL);                 /* pres. params */

/* store the object pointer */
/* in the PM window's word indexed by QWL_USER */
WinSetWindowPtr(hwnd, QWL_USER, comboNew);

__set_hwndWindow(comboNew, ev, hwnd);
__set_hwndFrame(comboNew, ev, NULLHANDLE);

return comboNew;

}

```

App1. An application object

All of our wrapped control objects are created using a constructor method, CreateWindow, but our application class is not. Why not? In this example, we are going to use our application objects in a very similar fashion to how we use our controls.

app1.idl

```

#ifndef app1_idl
#define app1_idl
#include <client.idl> // get def of your parent

interface StaticText;
interface EntryField;
interface ListBox;
interface ComboBox;

interface App1;

// Metaclass definition

interface M_App1: M_Client

```

```
{  
  
#ifdef __SOMIDL__  
implementation {  
    //## Class Modifiers  
    filestem = app1;  
    functionprefix = m_;  
  
    //## Method Modifiers  
    override: CreateWindow;  
};  
#endif  
};  
  
interface App1: Client // App1 is subclass of Client  
{  
    attribute short timesPressed;  
  
#ifdef __SOMIDL__  
implementation  
{  
    //## Class Modifiers  
    metaclass = M_App1;  
    releaseorder:  
        _get_timesPressed, _set_timesPressed;  
  
    //## Method Modifiers  
  
    PushedButton:     override;  
    EnterKeyPressed: override;  
    ListItemSelected: override;  
  
    //## Instance variables  
  
    StaticText msgField;  
    EntryField ef;  
    ListBox lb;  
    ComboBox cb;  
  
};  
#endif  
};  
  
#endif /* #ifndef app1_idl */
```

app1.c

```
#define App1_Class_Source
```

```

#include <app1.ih>
#include <static.h>
#include <entry.h>
#include <listbox.h>
#include <combobox.h>
#include <pushb.h>

#include <stdio.h>
#include <string.h>

SOM_Scope void SOMLINK PushedButton(App1 somSelf,
                                      Environment *ev,
                                      Window sourceWindow)
{
    App1Data *somThis = App1GetData(somSelf);
    App1MethodDebug("App1", "PushedButton");
    _ClearList(_lb, ev);
}

SOM_Scope void SOMLINK EnterKeyPressed(App1 somSelf,
                                       Environment *ev,
                                       Window sourceWindow)
{
    App1Data *somThis = App1GetData(somSelf);
    char strBuf[256];
    App1MethodDebug("App1", "EnterKeyPressed");

    if(sourceWindow==_ef || sourceWindow==_cb) {
        /* only interested in Entry Field and Combo Box */
        EntryField efTmp = sourceWindow;
        char *efText = __get_text(sourceWindow, ev);

        sprintf(strBuf, "EntryField Text: %s",
                /* set the text on the title bar */
                __set_text(_msgField, ev, strBuf));
        __InsertItem(_lb, ev, efText, LIT_SORTASCENDING);
        __InsertItem(_cb, ev, efText, LIT_SORTASCENDING);

        __set_text(_ef, ev, "");
        free(efText);
    }
    return;
}

SOM_Scope void SOMLINK ListItemSelected(App1 somSelf,
                                         Environment *ev,
                                         Window sourceWindow)
{
    App1Data *somThis = App1GetData(somSelf);
    char strBuf[256];
}

```

```
    string itemText;

    App1MethodDebug ("App1", "ListItemSelected");

    itemText = _GetItemSelected(sourceWindow, ev);
    if (!itemText)
        return;
    sprintf(strBuf, "Selection from List or Combo: %s",
itemText);
        /* set the text on the title bar */
    __set_text (_msgField, ev, strBuf);
    free(itemText);
        /* Method _GetSelectedItem allocates
         * memory for my string that I must
         * free. */

}

SOM_Scope Window  SOMLINK m_CreateWindow(M_App1 somSelf,
                                         Environment *ev,
                                         Window parent,
                                         Window owner,
                                         long style,
                                         string initText,
                                         short x, short y,
                                         short w, short h,
                                         somToken userStruct)
{
    /* M_App1Data *somThis = M_App1GetData(somSelf); */
    App1Data *somThat;
    ListBox lb;
    StaticText st;
    PushButton pb;
    App1 appOne;
    short i;
    BYTE fgRGB=255, bgRGB=0;

    M_App1MethodDebug ("M_App1", "m_CreateWindow");

    appOne = M_App1_parent_M_Client_CreateWindow(somSelf, ev,
                                                parent, owner, style, initText, x, y, w, h, userStruct);

    somThat = App1GetData(appOne);

    fprintf(stderr, "M_CreateWindow of app1.c: begin\n");

    EntryFieldNewClass(0,0);
    StaticTextNewClass(0,0);
    ListBoxNewClass(0,0);
```

```

ComboBoxNewClass(0,0);

PushButtonNewClass(0,0);
somThat->ef = _CreateWindow(_EntryField, ev,
                           appOne,      /* parent */
                           appOne,      /* owner */
                           0L, "",      10, 60, 300, 40, (somToken)0);
st = _CreateWindow(_StaticText, ev,
                   appOne,      /* parent */
                   appOne,      /* owner */
                   WS_VISIBLE | SS_TEXT | DT_LEFT | DT_BOTTOM,
                   "Enter text for next ListBox item:",
                   10, 110, 300, 40, (somToken)0);
pb = _CreateWindow(_PushButton, ev,
                   appOne,      /* parent */
                   appOne,      /* owner */
                   WS_VISIBLE | BS_NOPOINTERFOCUS, /* a PM style
for buttons who won't take focus */
                   "Clear List Box",
                   10, 170, 300, 60, (somToken)0);
somThat->msgField = _CreateWindow(_StaticText, ev,
                                   appOne,      /* parent */
                                   appOne,      /* owner */
                                   WS_VISIBLE | SS_TEXT | DT_LEFT | DT_VCENTER,
                                   "Messages:",
                                   10, 10, 560, 40, (somToken)0);
somThat->lb = _CreateWindow(_ListBox, ev,
                            appOne,      /* parent */
                            appOne,      /* owner */
                            0L,NULL, /* no style and no initial text */
                            320, 60, 250, 230, (somToken)0);
somThat->cb = _CreateWindow(_ComboBox, ev,
                            appOne,      /* parent */
                            appOne,      /* owner */
                            0L, NULL, /* no style indicated (I will
get default) and no initial text */
                            10, 60, 300, 200, (somToken)0);

_SetColorFG(somThat->ef, ev, fgRGB, fgRGB, fgRGB);
/* black foreground */
_SetColorBG(somThat->ef, ev, bgRGB, bgRGB, bgRGB); /* on
white background */
_SetColorFG(somThat->lb, ev, 255, 255, 255);
_SetColorBG(somThat->lb, ev, 255, 0, 0);
_SetColorFG(somThat->msgField, ev, 255, 255, 255);
_SetColorBG(somThat->msgField, ev, 0, 0, 255);
_SetFocus(somThat->ef, ev);

}

```

A Constructor Example

Soon, SOM will add new constructor support. Chapter 24 covers this new area. You may wonder how constructor support is handled without these functions. Until this example, the test program has done very little other than create an instance of our application object and then start the `ProcessEvents` method. Now the `wrap3` program calls an `App1` metaclass constructor instead. In this way, we are not simply blindly accepting the default look and feel of Application window; rather, we are exercising control over the size, position, and other attributes when we create an instance of this class.

wrap3.c

```
#include <app1.h>
void main(int argc, char *argv[])
{
    Environment *ev = somGetGlobalEnvironment();
    App1 obj = App1New();
    obj = _CreateWindow(_App1, ev,
                        (Window)NULL, /* parent will be
                                         HWND_DESKTOP */
                        (Window)NULL, /* owner as well */
                        0L, "Wrapped Controls Three!",
                        20, 20, 580, 330, (somToken)0);
    _ProcessEvents(_Window, somGetGlobalEnvironment());
    _somFree(obj);
}
```

Build our wrapped controls into a DLL

In this example, we are going to package our wrapped controls into a dynamically linkable library (DLL). We do this to facilitate the efficient reuse of useful objects without having multiple copies of the binaries filling up our hard drive with duplicated code. For most of the examples in this book, the number of compiled classes (in *.obj files) is quite small, so we simply linked them with our application code to create a single stand-alone executable. However, this meant that if three applications each wanted to use our framework, they would each link to our framework, resulting in three large executable programs with each containing the wrapped control code.

With a DLL, we can store the framework code in a single central location which can be accessed by all three executables. The executables, instead of linking to the object files, now link to an import library which contains references to the public entry points of our dynamically linkable library. The import library is created using the tool `IMPLIB` with our definition file. The import library is then placed in a directory identified by the environment variable `LIB`. The DLL

is created like our executables using LINK386. The DLL must then be placed somewhere in the search path which is identified in your CONFIG.SYS file by LIBPATH.

The key differences between this makefile and our last, is that when compiling the objects that will be linked for our DLL we use the compiler options -Gs- and -Ge-. The -Ge- option tells us to use the version of the runtime library that assumes a DLL is being built. The -Gs- option is for removing stack probes. Once again, for a listing of what the compiler options mean just type:

```
icc ?
```

makefile—for wrapped controls example three

```
LIB = $(LIB)
INCLUDE = .;$(INCLUDE)
SMEMIT = ih;h;c;
SRCS = window. entry. static. listbox. pushb. combobox.
client.
HDRS = $(SRCS:.ih) app1.ih
OBJS = $(SRCS:.obj)

install: wrap3.exe

.SUFFIXES : .ih .sc .lib .def .dll .idl

all: ctrl.dll wrap3.exe

.c.obj:
    icc -Gs- -Ge- -Q -Ti+ -c $<

.idl.ih:
    sc $*

wrap3.exe: $(HDRS) ctrl.dll ctrl.lib app1.obj wrap3.obj
    link386 /NOL /NOI /CO /PM:PM app1.obj wrap3.obj,
wrap3.exe, nul,somtk ctrl;

ctrl.dll: $(SRCS:.obj)
    link386 /NOL /NOI /CO /PM:PM $(OBJS), ctrl.dll, nul,somtk,
ctrl.def
    implib /nologo ctrl.lib ctrl.def

ctrl.lib: ctrl.def
    implib /nologo ctrl.lib ctrl.def

window.ih: window.idl
window.obj: window.ih window.c

client.ih: client.idl
```

```

client.obj: client.ih client.c

entry.ih: entry.idl
entry.obj: entry.ih entry.c

static.ih: static.idl
static.obj: static.ih static.c

listbox.ih: listbox.idl
listbox.obj: listbox.ih listbox.c

combobox.ih: combobox.idl
combobox.obj: combobox.ih combobox.c

pushb.ih: pushb.idl
pushb.obj: pushb.ih pushb.c

# the appl and wrap3 object code are compiled without the
# -Gs- and -Ge- flags.
appl.ih: appl.idl
appl.obj: appl.ih appl.c
    icc -Q -Ti+ -c $*.c

wrap3.obj: wrap3.c
    icc -Q -Ti+ -c $*.c

```

The definition file

The SOM compiler has an emitter option that can help us create the definition file.

```
sc -sdef <filestem>
```

This will emit a .def file for a single class. To combine several classes into a single DLL, you can merge those files. The LIBRARY line must identify the name of the DLL that combines all your classes. Then, in the EXPORTS section, you must include three entries for each class that is packaged in your DLL. Through these three entry points all of your class methods and data can then be accessed. The .def file is used to create both your import library file and your DLL.

ctrl.def

```

LIBRARY CTRL INITINSTANCE
DESCRIPTION 'Wrapped Controls Class Library'
PROTMODE
DATA MULTIPLE NONSHARED LOADONCALL
EXPORTS
    ClientCClassData
    ClientClassData
    ClientNewClass
    ComboBoxCClassData

```

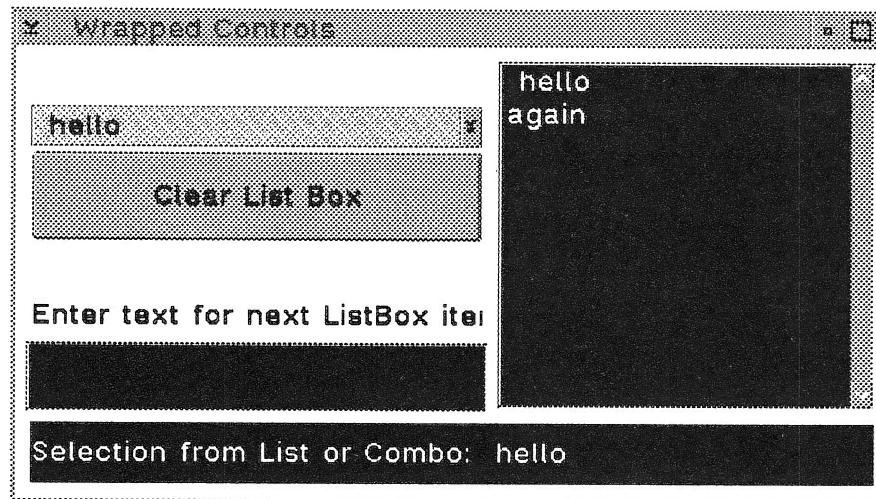


Figure 18.2. Output for WRAP3.EXE

```
ComboBoxClassData
ComboBoxNewClass
M_ComboBoxCClassData
M_ComboBoxClassData
M_ComboBoxNewClass
EntryFieldCClassData
EntryFieldClassData
EntryFieldNewClass
M_EntryFieldCClassData
M_EntryFieldClassData
M_EntryFieldNewClass
ListBoxCClassData
ListBoxClassData
ListBoxNewClass
M_ListBoxCClassData
M_ListBoxClassData
M_ListBoxNewClass
PushButtonCClassData
PushButtonClassData
PushButtonNewClass
M_PushButtonCClassData
M_PushButtonClassData
M_PushButtonNewClass
StaticTextCClassData
StaticTextClassData
StaticTextNewClass
M_StaticTextCClassData
M_StaticTextClassData
M_StaticTextNewClass
```

```
WindowCClassData  
WindowClassData  
WindowNewClass  
M_WindowCClassData  
M_WindowClassData  
M_WindowNewClass
```

Running Wrap3

To run wrap3.exe, just type **wrap3** at your command line:

```
[d:\example\wrap3] wrap3
```

and you will see a set of PM controls arranged nicely on your screen as you can see in Figure 18.2.

By typing into your entry field or combo box you can add items to your list box. (The items will be sorted for you.) The push button is set up to clear the list box. The message box at the bottom of the window informs you of various activity occurring in your application.

POSSIBLE ENHANCEMENT TOWARD BETTER EXTENSIBILITY

We have come a long way with our study of wrapped controls, but we could have gone even further. We are not quite as extensible as we could be using SOM. To add an object to our client area, we have prerequired our Client objects to know about the objects that would be contained in the Client. There should not be such a prerequisite. SOM provides a method for dynamically loading libraries: SOMInitClassModule. To make this package of classes more flexible SOMInitModule would have to be added to the list of exports in the definition file. SOMInitModule could then be used to instantiate each new class in the library.

We have seen a few examples of using PM. Now, let's take a look at some common OS/2 constructs.

19

A File System Example

OS/2 OPERATING SYSTEM

This chapter is the first of two chapters that briefly examine a couple of the features of the OS/2 Operating System. We'll look at the file system to see how we can use object-oriented design to encapsulate a basic file and some of its behavior including the more sophisticated extended attributes. In the next chapter, we'll examine how objects can be used to encapsulate an OS/2 event semaphore. We'll see how semaphores are used on OS/2 to help manage its powerful multi-tasking environment.

THE OS/2 FILE SYSTEM

One of the most commonly used OS/2 features is its file system. There is nothing unique about the basic file system. Its hierarchical arrangement of directories and files is very similar to the file systems you may have seen on DOS and UNIX. We will create a basic file/directory class that wraps this basic and often-used I/O functionality, and also encapsulate some of the OS/2-unique extended attributes. As we create the class, we will keep in mind that there are some useful similarities between file objects and other non-file system objects such as semaphores, shared memory, and named pipes.

File System Name Space

In UNIX, you are taught that everything is a file. This is because files, directories, printers, queues, and devices are all treated as files. All of these files have a place in the overall hierarchical file system. They are opened and closed, and they are read from and written to. OS/2 does not attempt to treat everything in a similar fashion, but as we work on our class library, we can wrap many of our OS/2 operating system objects in such a way that many of our OS/2 objects are files. You will see that we create a class called `GenericFile` from which classes of a variety of types can be subclassed. A unifying feature of OS/2 that is common to the file objects and non-file systems objects is the way they make use of the *file system name space*. This is where named objects are managed and where the names of these objects are guaranteed to be unique. Before we look at the file system, we will see the place on which your file system is stored. When you see a file named `D:\example\koenen\test.c` the `D:` tells you which logical drive stores your file. The logical drive is a type of device.

Devices

Devices come in two different types: *block* and *character*. Devices that can be accessed randomly, such as your hard drive, or diskette drive, are block devices. Devices that handle stream I/O like your keyboard, and serial and parallel ports are character devices. Files are stored on block devices which can be partitioned into one or more logical block units or logical drives. These logical drives are identified using letters from A through Z.

Installable File System

OS/2 has an *installable file system* architecture that allows for the presence of multiple file systems in the same environment. The file systems are made up of *file system objects*, which are organized in a hierarchy of objects. The system of files and directories with which most of you are familiar is one example of file system objects. The file system on your OS/2 machine, whether it uses the FAT (File Allocation Table) file system or the HPFS (High Performance File System), is a collection of directories and files arranged in a tree. The top of the tree is a directory called the *root* and is designated by a lone back-slash. Each logical drive has a root directory. The root directory can contain files and other directories, which in turn can also contain files and more subdirectories.

Each file has an *absolute pathname* which uniquely identifies a particular file. A pathname looks like:

`D:\KOENEN\SRC\HELLO.C`

This pathname tells us that a file named `HELLO.C` is located on logical drive `D:` and is found in the `\KOENEN\SRC` subdirectory.

Attributes and Extended Attributes

A standard set of information called *attributes* is maintained for each file object. As you have probably guessed these will form the basis for our objects' attributes as we define them in our IDL. This set of attributes includes information like the name of the file, its size, and the last time it was changed. Additional information called *extended attributes*—like ICON information or simple strings of characters to catalog or describe the file—can be stored by applications with each file.

EAVIEW Example

In this example, we create a class called `GenericFile`. This class will be used in a later example as the base class for our event semaphore class which, as mentioned earlier, is a non-file system object. The `File` class is then subclassed from `GenericFile`. We prefix the name of this basic `File` class with an EA because we want to encapsulate this feature that OS/2 files can have. Working with extended attributes can get rather ugly sometimes, so a nice class that encapsulates all the ugliness and hides it from the user constitutes a rather nice act toward humanity. This is what object-oriented programming is all about!

This example is going to be slightly different from past examples. It will sketch out a possible interface for an Extended Attribute File class but will implement only enough of the methods needed to create an EAViewer application. This example is based on an example you can find in the *OS/2 Application Programmer's Guide*. This book is recommended to fill in some of the empty methods and to include the functionality that adds or changes extended attributes.

GenericFile. Everything on OS/2 is a GenericFile.

The first OS/2 class that we introduce is the `GenericFile` class. Okay, so the name is not too sexy, but you can name it what you want. We only have two attributes that we introduce here. These attributes, `path` and `name`, can be used to identify files, semaphores, named pipes, and the like. There is also some behavior that you may think is common among these various OS/2 files. Why not have methods that resemble functions like `open` and `close`? That would not be a bad idea, but we will hide, or encapsulate, that functionality within our `somInit/somUninit` or constructor methods. This is assuming that someone who instantiates a `GenericFile` object will want it opened and initialized at creation time. On the other hand, some programmers may prefer to expose these methods to the user. A more sophisticated user might not mind having multiple steps of initialization so that he or she can gain flexibility.

gfile.idl

```
#include <somobj.idl>

interface GenericFile: SOMObject
{
```

```

        attribute string path;
        attribute string name;
#ifndef __SOMIDL__
implementation
{
    releaseorder:
        _get_path, _set_path,
        _get_name, _set_name;

// by 'passing through' this #define and include,
// allsubclasses will get these for free
// Remember to put any additional #defines above
// subclass's include <file>.ih
    passthru C_h_before =
        "#define INCL_DOS"
        "#include <os2.h>"
;

override: somInit;
override: somUninit;

//# Data modifiers
// path and name are strings, so we will handle the
// memory
// storage when a 'set' is attempted.
    path: noset;
    name: noset;
};

#endif

};

```

gfile.c

```

#define GenericFile_Class_Source
#include <gfile.ih>

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/*
 *Method from the IDL attribute statement:
 *"attribute string path"
 */

SOM_Scope void  SOMLINK _set_path(GenericFile somSelf,
                                  Environment *ev,
                                  string path)
{
    GenericFileData *somThis = GenericFileGetData(somSelf);

```

```

GenericFileMethodDebug ("GenericFile", "_set_path");
if (_path)
    free (_path);
_path = (string)malloc(strlen(path)+1);
strcpy (_path, path);
}

/*
*Method from the IDL attribute statement:
* "attribute string name"
*/
SOM_Scope void  SOMLINK _set_name(GenericFile somSelf,
                                    Environment *ev,
                                    string name)
{
    GenericFileData *somThis = GenericFileGetData(somSelf);
    GenericFileMethodDebug ("GenericFile", "_set_name");
    if (_name)
        free (_name);
    _name = (string)malloc(strlen(name)+1);
    strcpy (_name, name);
}

SOM_Scope void  SOMLINK somInit(GenericFile somSelf)
{
    GenericFileData *somThis = GenericFileGetData(somSelf);
    GenericFileMethodDebug ("GenericFile", "somInit");
    _name = NULL;
    _path = NULL;
    GenericFile_parent_SOMObject_somInit(somSelf);
}

SOM_Scope void  SOMLINK somUninit(GenericFile somSelf)
{
    GenericFileData *somThis = GenericFileGetData(somSelf);
    GenericFileMethodDebug ("GenericFile", "somUninit");
    if (_name)
        free (_name);
    if (_path)
        free (_path);

    GenericFile_parent_SOMObject_somUninit(somSelf);
}

```

EAFile, A File Class with Extended Attributes

The EAFile class defines what you normally think of when you think of a file on the OS/2 operating system. These files can be viewed with DIR, their contents

can be examined with TYPE <filename>, and they can be COPY'ed, MOVE'ed, or RENAME'ed.

eafile.idl

```
#include <somcls.idl>
#include <gfile.idl>

interface EAFile;

typedef somToken HDIR;

interface M_EAFile: SOMClass
{

    EAFile getFirstFile(in string lookfor);
    // retrieve the first file that matches a pattern
    // indicated in the string 'lookfor.' wildcards
    // supported. When using wildcards, you can cycle
    // through all the files that match the pattern, by
    // following up with repeated calls to getNextFile.
    // Also used for a single file.

    EAFile getNextFile();
    // must be used after getFirstFile

#ifndef __SOMIDL__
implementation

{
    releaseorder:
        getFirstFile, getNextFile;

    /* define an instance variable */
    HDIR hdir;

};

#endif
};

interface EAFile: GenericFile
{
    readonly attribute long fileLength;
    // can be queried but not gotten.

    attribute long currentPosInFile;
    // reading and writing to the file will be in reference
    // to this current point.
```

```

    readonly attribute long permissions;
    // use flags for r/w/x permissions (that's read/write/execute)

    readonly attribute long date;
    // date last touched

    string readLine(in long lineLength);
    // retrieves a line of text from the file.
    // if lineLength set to -1 will read to the next newline

    void writeLine(in somToken writeBuffer, in long
bufferLength);
    // write out the contents of a buffer to a file for simple
    // string buffers that are null terminated you
    // could allow for a -1 bufferLength

    void listEAs();
    // make a listing of all of the extended attributes
    // also return the listing in the form of an array
    // of strings, terminated by a NULL

    somToken getFirstEA(in string EAName);
    somToken getNextEA();
    somToken getEAValue(in somToken EAttr);
    void addExAttr(in somToken EAttr, in somToken EAttrVal);

#ifndef __SOMIDL__
implementation
{
    metaclass = M_EAFile;  //# identify EAFile's metaclass
releaseorder:
    _get_fileLength, _get_currentPosInFile,
    _set_currentPosInFile,
    _get_permissions, _set_permissions,
    _get_date, _set_date,
    readLine, writeLine, listEAs,
    getFirstEA, getNextEA, getEAValue,
    addExAttr;

    override: somInit;
};

#endif
};

```

eafile.c

```

#define EAFILE_Class_Source
#define M_EAFile_Class_Source

#define INCL_DOS

```

```
#include <os2.h>

#include <eafile.ih>

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/*****************/
/* Function prototypes. */
/*****************/
VOID allocMem(PVOID *pvMessage,
              ULONG ulSize);
BOOL readEA(char *path);
VOID GetEAsFromFile(CHAR *szFilename,
                     DENA2 *eaList,
                     ULONG ulEnumCount);
VOID PrintData(USHORT *pEAData);

/*****************/
/* Defines. */
/*****************/
#define MAX_GEA 500L
/*
 * retrieves a line of text from the file.
 * if lineLength set to -1 will read to the next newline
 */
SOM_Scope string SOMLINK readLine(EAFile somSelf,
                                   Environment *ev,
                                   long lineLength)
{
    EAFileData *somThis = EAFileGetData(somSelf);
    EAFileMethodDebug("EAFile", "readLine");

    return;
}

SOM_Scope void SOMLINK writeLine(EAFile somSelf,
                                 Environment *ev,
                                 somToken writeBuffer,
                                 long bufferLength)
{
    EAFileData *somThis = EAFileGetData(somSelf);
    EAFileMethodDebug("EAFile", "writeLine");

}

/*
 * make a listing of all of the extended attributes
 * also return the listing in the form of an array
```

```
* of strings, terminated by a NULL
*/
SOM_Scope void SOMLINK listEAs(EAFile somSelf, Environment
*ev)
{
    EAFileData *somThis = EAFileGetData(somSelf);
    EAFileMethodDebug("EAFile","listEAs");
    readEA(__get_name(somSelf, ev));

    return;
}

SOM_Scope somToken SOMLINK getFirstEA(EAFile somSelf,
Environment *ev,
string EAName)
{
    EAFileData *somThis = EAFileGetData(somSelf);
    EAFileMethodDebug("EAFile","getFirstEA");

    return;
}

SOM_Scope somToken SOMLINK getNextEA(EAFile somSelf,
Environment *ev)
{
    EAFileData *somThis = EAFileGetData(somSelf);
    EAFileMethodDebug("EAFile","getNextEA");

    return;
}

SOM_Scope somToken SOMLINK getEAValue(
    EAFile somSelf,
    Environment *ev,
    somToken EAttr)
{
    EAFileData *somThis = EAFileGetData(somSelf);
    EAFileMethodDebug("EAFile","getEAValue");

    return;
}

SOM_Scope void SOMLINK addExAttr(EAFile somSelf,
Environment *ev,
somToken EAttr,
somToken EAttrVal)
{
    EAFileData *somThis = EAFileGetData(somSelf);
    EAFileMethodDebug("EAFile","addExAttr");
```

```
}

SOM_Scope void  SOMLINK somInit(EAFile somSelf)
{
    EAFileData *somThis = EAFileGetData(somSelf);
    EAFileMethodDebug("EAFile", "somInit");

    EAFile_parent_GenericFile_somInit(somSelf);
}

/*
 * retrieve the first file that matches a pattern in
 * the string 'lookfor'. Standard wildcard support.
 * When using wildcards, you cycle through all that
 * match the pattern, by following up with calls to
 * getNextFile.
 * Also used for a single file.
 */

SOM_Scope EAFile  SOMLINK getFirstFile(M_EAFile somSelf,
                                         Environment *ev,
                                         string lookfor)
{
    M_EAFileData *somThis = M_EAFileGetData(somSelf);
    EAFile returnObj;
    APIRET retVal;
    FILEFINDBUF3 buf;
    ULONG srchcnt = 1;
    HDIR hdir=HDIR_SYSTEM;

    /* the next vars are for the EA info management */
    CHAR *pAlloc=NULL;           /* Buffer for returned EA data.*/
    ULONG ulEntryNum = 1;        /* Current EA counter.          */
    ULONG ulEnumCnt;            /* Number of EAs to return.    */
    FEA2 *pFEA;                 /* Pointer to returned values. */
    M_EAFileMethodDebug("M_EAFile", "getFirstFile");

    if (!(retVal = DosFindFirst(lookfor,
                                &hdir,
                                FILE_NORMAL + FILE_HIDDEN + FILE_SYSTEM +
                                FILE_READONLY,
                                &buf,
                                sizeof(buf),
                                &srchcnt, 1L))) {
        returnObj = _somNew(somSelf);
        __set_name(returnObj, ev, buf.achName);
/*      _listEAs(returnObj, ev); */
/*      _hdir = hdir; */
/*      readEA(buf.achName); */
    }
}
```

```

        } else {
            return NULL;
        }

/* **** */
/* Allocate enough room for any attribute list for      */
/* the returned EA buffer and initialize the pFEA to   */
/* the start of this buffer. The pFEA buffer is         */
/* used to traverse the returned buffer.                 */
/* **** */
allocMem((PPVOID)&pAlloc, MAX_GEA);
pFEA = (FEA2 *) pAlloc;
ulEnumCnt = 1;
if(DosEnumAttribute(1,
                     buf.achName,
                     ulEntryNum,
                     pAlloc,
                     MAX_GEA,
                     &ulEnumCnt,
                     1L) ) {
    printf ("\nERROR\n");
}

return returnObj;
}

/*
 * must be used after getFileFirst
 */
SOM_Scope EAFile  SOMLINK getNextFile(M_EAfile somSelf,
Environment *ev)
{
    M_EAfileData *somThis = M_EAfileGetData(somSelf);
    EAfile returnObj;
    APIRET retVal;
    FILEFINDBUF3 buf;
    HDIR hdir = _hdir;
    ULONG srchcnt = 1;
    M_EAfileMethodDebug("M_EAfile", "getNextFile");
    if(! (retVal = DosFindNext(hdir,
                               &buf,
                               sizeof(buf),
                               &srchcnt))) {
        returnObj = _somNew(somSelf);
        __set_name(returnObj, ev, buf.achName);
        return returnObj;
    } else
        return NULL;
}

VOID allocMem (PVOID *ppv, ULONG cb)

```

```
{  
    BOOL failed;  
  
    failed = (BOOL) DosAllocMem(ppv, cb, fPERM|PAG_COMMIT);  
    if (failed) {  
        fprintf(stderr, "ERROR: Memory is full\n");  
        *ppv = NULL;  
        exit(1);  
    }  
    return;  
}  
  
BOOL readEA(CHAR *path)  
{  
    CHAR *pAlloc=NULL;           /* Buffer for returned EA data. */  
    ULONG ulEntryNum = 1;         /* Current EA counter. */  
    ULONG ulEnumCnt;             /* Number of EAs to return. */  
    FEA2 *pFEA;                 /* Pointer to returned values. */  
  
    /******  
     * Allocate enough room for any attribute list for the  
     * returned EA buffer and initialize the pFEA to the  
     * start of this buffer. The pFEA buffer is used to  
     * traverse the returned buffer.  
     */*****  
    allocMem((PPVOID)&pAlloc, MAX_GEA);  
    pFEA = (FEA2 *) pAlloc;  
  
    /******  
     * Loop through all the EAs in the file. A break is */  
    /* issued when no EAs are left to be processed. */  
    /******  
    for(;;) {  
        /******  
         * Use DosEnumAttribute to get a buffer (pAlloc) containing  
         * basic EA information about the EA. The actual data for the  
         * EA is not returned by this function. The data is acquired  
         * by using DosQueryFileInfo.  
        */*****  
        ulEnumCnt = 1;  
        if(DosEnumAttribute(1,  
                           path,  
                           ulEntryNum,  
                           pAlloc,  
                           MAX_GEA,  
                           &ulEnumCnt,  
                           1L) ) {  
            fprintf(stderr, "ERROR -- path=%s enumCnt=%ld\n",  
                    path, ulEnumCnt);  
            break;          /* An error occurred. */  
        }  
    }
```

```

***** */
/* If the first EA for the file is not available there */
/* are no EAs associated with this file. */
***** */

    if (ulEnumCnt != 1) {
        if(ulEntryNum==1) {
            printf("                ::> none\n");
        } else {
            printf("\n");
        }
        break;
    }

    ulEntryNum++;
***** */
/* Print the attribute name. The first attribute offset on */
/* the line differently from the rest to account for the */
/* filename which also appears on the same output line. */
***** */

    if(pFEA->szName)
        printf("          %10s\n",
               pFEA->szName);

***** */
/* Open the file, get the attribute value, print */
/* EA values information. */
***** */

    GetEAsFromFile(path, (DENA2 *)pAlloc,
                    ulEnumCnt);
}

DosFreeMem(pAlloc); /* Free the buffer. */
return (TRUE);
}

VOID GetEAsFromFile(CHAR *szFileName,
                    DENA2 *eaList,
                    ULONG ulEnumCount)

{
    HFILE      FileHandle;
    ULONG      ulActionTaken;
    ULONG      ulFileAttributes = FILE_READONLY;
    ULONG      ulOpenFlag = OPEN_ACTION_FAIL_IF_NEW |
                           OPEN_ACTION_OPEN_IF_EXISTS;
    ULONG      ulOpenMode = OPEN_SHARE_DENYREADWRITE |
                           OPEN_ACCESS_READONLY;
    ULONG      ulEACount;
    ULONG      ulEASize;

```

```

EAOP2          eaop2;
USHORT        *pEAData;
APIRET        rc = 0;

/***** */
/* Open the file for Extended Attribute access. It must      */
/* be opened for read access with denyWrite sharing mode. */
/***** */

rc = DosOpen(szFileName,
              &FileHandle,
              &ulActionTaken,
              100L,
              ulFileAttributes,
              ulOpenFlag,
              ulOpenMode,
              0L);

if (rc == 0) {
    for (ulEACount=0;ulEACount<ulEnumCount;ulEACount++) {

/***** */
/* Allocate storage for the EAs and setup the GEA2LIST      */
/* based on the list retrieved from DosEnumAttribute. The   */
/* storage size is based on the following formula :          */
/* */
/*      4 Bytes - for oNextEntryOffset                      */
/*      1 Bytes - for usFlags                                */
/*      1 Bytes - for cbName                                */
/*      2 Bytes - for cbValue                               */
/* _____ */                                              */
/*      8 Bytes - Total                                     */
/* */
/* This value (8) is added to the length of the             */
/* attribute name (cbName) plus 1 byte for the NULL       */
/* terminator plus the length of the actual attribute    */
/* value information (cbValue).                           */
/* */
/* _____ */                                              */
/*      8 + cbName + 1 + cbValue                         */
/* */
/***** */

    if (rc == 0) {
        ulEASize = 8 +
                    (eaList[0].cbName + 1) +
                    eaList[0].cbValue;
        eaop2.fpGEA2List = malloc(ulEASize);
        eaop2.fpFEA2List = malloc(ulEASize);

        if ((eaop2.fpGEA2List != NULL) &&

```

```

        (eaop2.fpFEA2List != NULL)) {
        eaop2.fpGEA2List->list->oNextEntryOffset =
0;
        eaList[0].cbName;
        strcpy(eaop2.fpGEA2List->list->szName,
               eaList[0].szName);
        eaop2.fpGEA2List->cbList =
        uLEASize;
    } else {
        rc = -1;
    }
}

if (rc == 0) {
/* **** */
/* Use DosQueryFileInfo to get the Full EA data */
/* for the EA specified in fpGEA2List. */
/* **** */
    eaop2.fpFEA2List->cbList = uLEASize;
    rc = DosQueryFileInfo(FileHandle,
                           FIL_QUERYEASFROMLIST,
                           &eaop2,
                           uLEASize * 2);

/* **** */
/* If attribute information was found, allocate */
/* enough storage to hold the data and copy data */
/* into the new storage. Pass the EA data to the */
/* printing routine. */
/* **** */
    if (rc == 0) {
        pEAData =
malloc(eaop2.fpFEA2List->list->cbValue);

        if (pEAData != NULL) {
            memcpy((UCHAR *)pEAData,
                   (PBYTE)eaop2.fpFEA2List +
                   sizeof(FEA2LIST) +
                   eaop2.fpFEA2List->list->cbName,
                   eaop2.fpFEA2List->list->cbValue);
            PrintData(pEAData);
            free(pEAData);
        } else {
            fprintf(stderr, "- Memory allocation
error\n");
        }
    } else {
        fprintf(stderr, "- DosQueryFileInfo RC =
%d\n", rc);
    }
}

```

```

        free(eaop2.fpGEA2List);
        free(eaop2.fpFEA2List);
        eaList = (DENA2 *)((PBYTE)eaList +
eaList[0].oNextEntryOffset);
    } else {
        fprintf(stderr, "- Memory allocation error.\n");
    }
}
DosClose(FileHandle);
} else {
    fprintf(stderr,
    "- Unable to open file - DosOpen RC = %d.\n",rc);
}
return;
}

VOID PrintData(USHORT *pEAData)
{
    CHAR      szPrintBuffer[1024];
    USHORT    usEAType;
    USHORT    usEALen;
    BYTE     bByteValue;
    ULONG    i;

/******
/* pEAData points to memory that contains three */
/* pieces of information for the EA: EA Type, EA length,*/
/* and EA data. For multiple value EAs, this info */
/* is slightly different. */
/*
/* usEAType is stored at the start and has a length 2. */
/* usEALen is follows usEAType and also has 2 bytes */
/* The EA data follows the usEALen and a length of */
/* usEALen bytes. */
/*
/* After storing type and length we move pEAData pointer */
/* to point to the value itself. In order to move pEAData */
/* a byte at a time we use the (PBYTE) cast. */
/***** */

    usEAType = (USHORT)*pEAData;
    pEAData = (USHORT *)((PBYTE)pEAData + 2);
    usEALen = (USHORT)*pEAData;
    pEAData = (USHORT *)((PBYTE)pEAData + 2);

    printf("%4d - ", usEALen);

/******
/* Print EA information based on the EA type. */
/***** */

    switch(usEAType) {

```

```

        case EAT_EA      :
        case EAT_ASCII   :
            strncpy(szPrintBuffer,
                    UCHAR *)pEAData,usEALen);
            szPrintBuffer[usEALen] = '\0';
            printf("\\"%s\"\n", szPrintBuffer);
            break;

        case EAT_BINARY   :
            for (i=0;i<usEALen;i++) {
                bByteValue = (BYTE)*pEAData;
                if (bByteValue<=0x0F) {
                    printf("0");
                }
                printf("%x",bByteValue);
                pEAData = (USHORT *)(
                    (PBYTE)pEAData + 1);
            }
            printf("\\"\\n");
            break;

        case EAT_METAFILE :
            printf("METAFILE Type\\n");
            break;

        case EAT_BITMAP    :
        case EAT_ICON      :
            printf("ICON or BITMAP Type\\n");
            break;

        case EAT_MVMT      :
        case EAT_MVST      :
            printf("MVMT or MVST multivalue Type\\n");
            break;

        default :
            printf("Unknown listType EA Type: %d\\n",
usEAType);
        }
    }
    return;
}

```

Eaview. An EA viewer

The eaview.c source shows how easy it is to use our file framework to view the extended attributes of files.

eaview.c

```
#include <eafile.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    EAFile a;
    Environment *ev = somGetGlobalEnvironment();

    M_EAFile clsObj = EAFileNewClass(0,0);
    if(argc>1)
        a = _getFirstFile(clsObj, ev, argv[1]);
    else
        a = _getFirstFile(clsObj, ev, "*.*");
    while(a) {
        printf("%18s\n", __get_name(a, ev));
        _listEAs(a, ev);
        _somFree(a);
        a = _getNextFile(clsObj, ev);
    }
    return(0);
}
```

makefile **** for multiple inheritance example

```
CC = icc
HDRS = gfile.ih eafile.ih
OBJS = eaview.obj gfile.obj eafile.obj
TARGET = eaview.exe

.c.obj:
    icc -Q -Ti+ -c $<

eaview.exe: gfile.obj eafile.obj eaview.obj
    link386 /CO /PM:VIO /NOL $(OBJS),eaview.exe,nul,somtk;

hdrs: $(HDRS)

gfile.obj: gfile.c gfile.ih

gfile.ih: gfile.idl
    sc -u gfile

eafile.obj: eafile.c eafile.ih

eafile.ih: eafile.idl
    sc -u eafile

eaview.obj: eaview.c eafile.ih
```

Now run EAView

To run eaview, go to an interesting directory like c:\os2 and type:

```
C:\OS2> eaview
```

You may get some error messages if you try to open files that are opened and being used by a current process.

As you can see, we've hidden from our innocent class user some of the ugly underside of OS/2 and given him or her the ability to use Extended Attributes. We could easily continue to encapsulate the functionality for adding EAs. The number of ugly sides to OS/2 is quite big, so the potential good that we can bring the world as object-oriented programmers on OS/2 is immense! Isn't this exciting?! Let look at another side of the OS/2 operating system, the semaphore. We'll look at how we can use an Event Semaphore to help us manage multi-threaded and multi-process applications.

20

IPC with Semaphores

A SEMAPHORE EXAMPLE

This next example is going to use semaphores to create a multi-threaded application and a multi-process application. The processes will interact with one another. This interprocess interaction is commonly known as *interprocess communication* or IPC. The semaphore acts like a traffic light, and to the various processes and threads decide whether they will pay attention to these traffic lights as the semaphores are posted and cleared. The OS/2 operating system keeps processes from colliding with each other, so your process normally does not need to even think about other processes. But if you want to synchronize your activity with the activities of another thread or process, you can do so by using the semaphore.

Multi-Tasking with OS/2

What is multi-tasking? A *task* is either a *process* or a *thread*. A process is an executable program. When you start a program at your OS/2 command line, you are beginning a *process*. You also begin a new process when you use an OS/2 API, DosExecPgm, or DosStartSession. A *process* is a relatively heavyweight and self-contained form of a task that requires a lot of overhead to be loaded into memory from disk. A *thread* is a piece of code that can be dispatched by the OS/2 scheduler. The OS/2 scheduler allocates CPU time on a thread basis, rather than to entire processes. A thread can be as big as an entire process in the case of

single-threaded applications, or it can be one of many threads in a single process, in what are called multi-threaded applications. Thread Number One is launched when you begin a process, the threads beyond Number One are created from within the application using the API, *DosCreateThread*, or the C library call, *_beginthread*.

Each task is meant to be run concurrently in a multitasking environment, but occasionally you will want to coordinate or synchronize two or more of the concurrently running tasks. To do this, some kind of IPC mechanism is necessary that allows one thread to notify another thread of an action or event. OS/2 provides a number of IPC mechanisms to aid in this communication and coordination such as shared memory, named pipes, queues, and semaphores. This next example will examine a specific type of semaphore called an *event semaphore*. To be more specific, this example will use a *shared* event semaphore.

SEMAPHORES

OS/2 has a rich set of semaphore functions. There are three types of semaphores: *mutual exclusion* (mutex), *event*, and *multiple wait* (muxwait). Each of these three types of semaphores can be either *shared* or *private*. For a complete description of the variations of semaphores and the general topic of multitasking, I recommend *Client/Server Programming with OS/2 2.1* by Orfali and Harkey. In this chapter we will only give a quick overview to help us set the stage for our example and to plan ways to encapsulate a Semaphore.

Mutex Semaphores

Mutex semaphores are used to protect critical regions. What is a critical region? When programming in a multi-threaded environment, you will often hear questions like: "Is your code reentrant?" or "Is your code multi-thread safe?" This is important because more than one thread can be marching through the same code section at about the same time. If you aren't careful, Thread Two can start resetting variables that Thread One had set a few lines ago in Thread One's execution. To keep other threads from interfering with variables critical to the execution, you can say to the OS/2 world "Hey, keep out of this section of code until I'm done!" A mutex semaphore can be used in this case. A mutex semaphore is either *unowned*, or it can be *owned* by a single thread. At the beginning of your *critical* section you request a mutex semaphore by using the OS/2 API *DosRequestMutexSem*. This call will block the current thread until ownership can be obtained. Once you are given *ownership*, you can then proceed through your critical section of code. When your critical section is finished, you relinquish ownership with *DosReleaseMutexSem*, so that the other threads can proceed through the critical section. It is like passing over a one-lane bridge. The car on the bridge at the moment had better *own* that bridge.

Event Semaphores

An event semaphore is used to signal the occurrence of an event. The event can be of interest to multiple processes which wait for the event to occur. In our example we will put a timer on a second thread, and the main thread will wait until the simple timer expires, or times out. We will have a short second program that shows how a second process can be used to prematurely shut down our "waiting" process. This will allow us to kill the application before it times out.

In OS/2 terminology, when the event occurs, the semaphore will be *posted*, and while waiting on an event, the semaphore is *reset*. These are the only two states. Don't let the terminology confuse you. "Posted" is simply "on," and "reset" is simply "off." This is slightly confusing to those who have used "set" and "clear" terminology to describe the same thing. It is a simple binary phenomenon with a new name to keep this world of ones and zeroes interesting.

Muxwait Semaphores

If you want to wait on multiple event semaphores or multiple mutex semaphores (NOT both), you can do this with a single wait through the use of *muxwait* semaphores. You can wait for any or all of up to 64 semaphores. For example, think of yourself as a complicated multithreaded process; think of your sense of sight as a single thread. Sometimes you sit and wait watching for a single visual cue. When your cue becomes visible, you act. That action is the event semaphore. Now, suppose you are in the bell tower watching for one of two different visual cues giving you a different message to act on, for example, the famous "one if by land and two if by sea." If your sense of vision is equipped with an OS/2 muxwait semaphore, you can respond to either the single light or to the two lights just like Paul Revere.

Timeout and Kill the Timeout examples

The following is a listing of the code for two simple IPC examples:

gfile.idl and gfile.c are the same as in the first OS/2 example

See previous example for description and source listing of the GenericFile class.

Semaphore—A Class to Help with Interprocess Communication

We have chosen to create a single Semaphore class in this example. The methods and attributes introduced here are tailored to our need for a shared event semaphore. It would be easy to expand on this class to support other semaphore types. Semaphore is a subclass of GenericFile. Semaphore always has a path name of \SEM32\.

semaphor.idl

```
#include <somcls.idl>
#include <gfile.idl>

interface Semaphore;

typedef somToken HEV;

interface M_Semaphore: SOMClass
// use the M_Semaphore class as a constructor
{

    Semaphore CreateEventSem(in string semName);
    // returns a named Event Semaphore object

#ifndef __SOMIDL__
implementation
{
    releaseorder:
        CreateEventSem;

};

#endif
};

interface Semaphore: GenericFile
{
    readonly attribute HEV hEventSem;

    long WaitForEvent(in unsigned long waitType);
// WaitForEvent blocks current thread for a
// time controlled by the waitType parameter.
// Blocking will stop when an event (associated with
// somSelf) occurs.
// waitType's include:
// -1, or SEM_INDEFINITE_WAIT
// Wait indefinitely, blocking thread until event occurs or
// is posted. 0, or SEM_IMMEDIATE_RETURN
// No wait, quick check to see if event has been posted.
// Any positive value. A timeout in milliseconds.
// return values include:
// 0. when event indicated in somSelf is posted.
// ERROR_TIMEOUT. timeout occurs with no event posted.
// ERROR_INTERRUPT.

    void PostSemaphore();
    // for posting a semaphore. for informing other threads
    // or processing
    // that an event has occurred.
```

```

unsigned long ResetSemaphore();
// to reset a semaphore.
// returns the count of posts made before this current
// reset.

#ifndef __SOMIDL__
implementation
{
    metaclass = M_Semaphore; //# identify Semaphore's
                           // metaclass
    releaseorder:
        _get_hEventSem, WaitForEvent, PostSemaphore,
ResetSemaphore;

    override: somInit;
    override: somUninit;
};

#endif
};

```

[semaphor.c](#)

```

#define Semaphore_Class_Source
#define M_Semaphore_Class_Source

#define INCL_DOSERRORS
#include <semaphor.ih>

#include <stdio.h>

/*
 *
 * WaitForEvent blocks current thread for a period of time
 * controlled by the waitType param. Blocking will stop when
 * an event (associated with somSelf) occurs.
 *
 * waitType's include:
 * -1, or SEM_INDEFINITE_WAIT
 * Wait until event occurs or is posted.
 * 0, or SEM_IMMEDIATE_RETURN
 *   No wait, check to see if event has been posted.
 * Any positive value. A timeout in milliseconds.
 *
 * return values include:
 * 0. when event indicated in somSelf is posted.
 * ERROR_TIMEOUT. timeout with no event posted.
 * ERROR_INTERRUPT.
 */

```

```

    SOM_Scope long  SOMLINK WaitForEvent(Semaphore
                                         somSelf, Environment *ev,
                                         unsigned long waitType)
    {
        SemaphoreData *somThis = SemaphoreGetData(somSelf);
        APIRET rc;
        SemaphoreMethodDebug("Semaphore", "WaitForEvent");

        return (long)DosWaitEventSem(_hEventSem,
                                     waitType);
    }

/*
 * for posting a sem. for informing others or
 * processing that an event has occurred. */
SOM_Scope void  SOMLINK PostSemaphore(Semaphore
                                      somSelf, Environment *ev)
{
    SemaphoreData *somThis =
        SemaphoreGetData(somSelf);
    SemaphoreMethodDebug("Semaphore", "PostSemaphore");

    DosPostEventSem(_hEventSem);
}

/*
 * to reset a semaphore.
 * returns the count of posts made before this
 * current reset.
 */
SOM_Scope unsigned long  SOMLINK ResetSemaphore(Semaphore
                                                 somSelf, Environment *ev)
{
    SemaphoreData *somThis = SemaphoreGetData(somSelf);
    APIRET rc;
    unsigned long postCount;

    SemaphoreMethodDebug("Semaphore", "ResetSemaphore");
    rc = DosResetEventSem(_hEventSem, &postCount);
    return postCount;
}

SOM_Scope void  SOMLINK somInit(Semaphore somSelf)
{
    SemaphoreData *somThis = SemaphoreGetData(somSelf);
    Environment *ev = somGetGlobalEnvironment();
    SemaphoreMethodDebug("Semaphore", "somInit");

    Semaphore_parent_GenericFile_somInit(somSelf);
}

```

```

/* make call to parent first because the GenericFile class */
/* will initialize the path attribute to NULL */
/* After that we can set path to our SEM32 value */

    __set_path(somSelf, ev, "\\SEM32\\");
}

SOM_Scope void  SOMLINK somUninit(Semaphore somSelf)
{
    SemaphoreData *somThis = SemaphoreGetData(somSelf);
    SemaphoreMethodDebug("Semaphore", "somUninit");

    DosCloseEventSem(_hEventSem);

    Semaphore_parent_GenericFile_somUninit(somSelf);
}
/*
 * returns a named Event Semaphore object
 */
SOM_Scope Semaphore SOMLINK CreateEventSem(M_Semaphore
                                             somSelf, Environment *ev,
                                             string semName)
{
    /* M_SemaphoreData *somThis = M_SemaphoreGetData(somSelf); */
    SemaphoreData *somThat;
    string semPath;
    char semFullName[256];
    HEV hEvSem;
    APIRET rc;
    BOOL32 fState=0;          /* initialize to "set" */
    ULONG flAttr=0;           /* flAttr is ignored with named
semaphores */
    Semaphore semNew;

    M_SemaphoreMethodDebug("M_Semaphore",
                           "CreateEventSem");

    if(!semName) {
        return (Semaphore)NULL;
    }
    semNew = SemaphoreNew();
    somThat = SemaphoreGetData(semNew);
    semPath = __get_path(semNew, ev);
    strcpy(semFullName, semPath);
    __set_name(semNew, ev, semName);

    /* not supporting unnamed semaphores in this
     * example so just return a NULL if
     * NULL for a semaphore name */
}

```

```

/* set up the semaphore name using the standard path/name
   combo which is also used for files in the file system. */
strcpy(semFullName, semPath);
strcat(semFullName, semName);

***** initialize the event semaphore ****/
flAttr = 0;
if(rc = DosCreateEventSem(semFullName, &hEvSem,
                           flAttr, fState)) {
    if(rc==ERROR_DUPLICATE_NAME) {
        hEvSem = 0; /* set to 0 for named
                      DosOpenEventSem */
        rc = DosOpenEventSem(semFullName,
                             &hEvSem);
    } else {
        fprintf(stderr, "DosCreateEventSem error! semName=%s
rc=%ld 1 hEvSem=%lx\n",
                semFullName, rc,hEvSem);
        _somFree(semNew);
        return (Semaphore)NULL;
    }
}
somThat->hEventSem = hEvSem;

return semNew;
}

```

Timeout.c. A Simple Multithread Example

In timeout.c, we have a simple multithread example. Thread One launches Thread Two and then waits for an event to be posted on Thread Two. Thread One could just wait indefinitely until the event semaphore is posted, but our Thread One waits for about a second at a time and then prints out a little message each second before continuing to wait for the Semaphore for another second. Thread Two sleeps for a TIMEOUT interval (which is #defined in our code) and then posts the event semaphore.

timeout.c

```

#include <stdio.h>

#include <semaphor.h>

#define STACKSIZE 4096
#define TIMEOUT_VALUE 20000

VOID timeoutThread (Semaphore semEvent);
string Name1="KillTO"; /* kill the TimeOut application */

```

```

int main(int argc, char *argv[])
{
    Semaphore a;
    short cnt=0;
    Environment *ev = somGetGlobalEnvironment();
    M_Semaphore clsObj = SemaphoreNewClass(0,0);

    a = _CreateEventSem(clsObj, ev, Name1);

    _beginthread ((void (*)(void *))timeoutThread, NULL,
STACKSIZE, a);

    printf("Hi.\n");
    DosSleep(1000); /* pause a second */
    printf("I'm waiting to be killed.\n");
    while(_WaitForEvent(a, ev, 1000)) { /* wait 1000
                                         milliseconds */
        printf("%d\n", ++cnt); /* count while waiting to
                               time out */
    }
    printf(" Aaaaaaa! I'm dead\n");

    _somFree(a);

    return(0);
}

VOID timeoutThread (Semaphore semEvent)
{
    Environment *ev = somGetGlobalEnvironment();
    short i;

    DosSleep(TIMEOUT_VALUE);
    _PostSemaphore(semEvent, ev);
    _endthread();
}

```

killto.c

Let's kill the timeout example from another process. This is a simple follow-up example to our `timeout.c` example. We named our shared event semaphore `KillTO`. The `killto.c` source can also create a `Semaphore` object with the same name. The `killto.exe` program can kill the `timeout.exe` program before it is timed out by the sleeping Thread Two.

killto.c ** a program designed to kill the timeout program***

```

#include <stdio.h>

#include <semaphor.h>

```

```

string Name1="KillTO"; /* kill the timeout application */

int main(int argc, char *argv[])
{
    Semaphore a;
    Environment *ev = somGetGlobalEnvironment();
    M_Semaphore clsObj = SemaphoreNewClass(0,0);

    a = _CreateEventSem(clsObj, ev, Name1);

    fprintf(stderr,"I am the great timeout killer!!\n", a);
    _PostSemaphore(a, ev);

    _somFree(a);

    return(0);
}

```

makefile ** for semaphore example***

```

CC = icc
HDRS = gfile.ih semaphor.ih
OBJS = gfile.obj semaphor.obj timeout.obj killto.obj
TOBJS = timeout.obj gfile.obj semaphor.obj
KOBJS = killto.obj gfile.obj semaphor.obj
TARGET = timeout.exe

.c.obj:
    icc -Q -Ti+ -c $<

all: $(HDRS) $(OBJS) timeout.exe killto.exe

timeout.exe: $(TOBJS)
    link386 /CO /PM:VIO /NOL /NOI
    $(TOBJS),timeout.exe,nul,somtk;

killto.exe: $(KOBJS)
    link386 /CO /PM:VIO /NOL /NOI
    $(KOBJS),killto.exe,nul,somtk;

hdrs: $(HDRS)

gfile.obj: gfile.c gfile.ih

gfile.ih: gfile.idl
        sc -u gfile

semaphor.obj: semaphor.c semaphor.ih

```

```
semaphor.h: semaphor.idl
    sc -u semaphor

# use the -Gm+ option for this multi-threaded example.
timeout.obj: timeout.c semaphor.h
    icc -Q -Ti+ -Gm+ -c timeout.c

killto.obj: killto.c semaphor.h
```

Now run **timeout.exe** and **killto.exe**

`timeout` can be run by itself:

```
D:\example\sem> timeout
```

Timeout will simply die after Thread Two finishes sleeping.

Running the `killto.exe` program is easiest if you have two windows up and running next to each other. In one window, start the `timeout` program as before. In the second window, in a similar fashion, start `killto` after `timeout` is begun but before `timeout` has timed out.

CONCLUSION

We have looked at SOM, PM, and the OS/2 API. We have seen some good practical examples that show you how to develop OO frameworks. These frameworks should be reusable and general. In the next chapters, we will go into more detail. We will look at a C++ framework that is in use today. We will also see how to develop friendly user interfaces. For your programming pleasure, we have included these samples on the diskette that comes with this book. Enjoy!

Section III: Experts

21

ObjectPM: A Consultant's View

INTRODUCTION

This chapter describes the ObjectPM class library and illustrates how it is used to write object oriented OS/2 applications in C++. Since its introduction in 1990 by Raleigh Systems Incorporated (RSI), the ObjectPM class library has evolved through use to the point where it now consists of over 200 classes. Reviewing a full-fledged commercial class library such as ObjectPM provides a good opportunity for understanding the use of OOP on OS/2. We discuss concepts of code reuse and class frameworks in order to understand how object technology is applied to build and use such a tool.

Reusing Code

As the most advanced PC operating system available, OS/2 is famous for the power, sophistication, and flexibility of its interfaces. It is also difficult to program. Over a thousand API calls provide programming interfaces to OS/2's subsystems, such as the kernel, Presentation Manager (PM), and the graphics engine (GPI). Add to this extensive list of calls the 400 or so PM messages, 475 API data structures, CUA-91 features, and the SOM Workplace Shell classes, and you have a system whose magnitude and complexity is truly a force to be reckoned with.

The ability to reuse previously-developed code on OS/2 is therefore an extremely important consideration, and this explains the importance of using OOP

on OS/2. OOP enables programmers to take software written for a particular application, package it, and use it in other applications. This concept is the basis for traditional subroutine libraries. But subroutines cannot be customized by their users. With object-oriented technology, class libraries of reusable software components can be extended and customized according to individual application demands. This gives you the ability to create new components by customizing existing components, or by assembling two or more simpler components into more complex objects. With well-designed OOP class frameworks, you can:

- Code applications faster by re-using existing software instead of re-inventing it.
- Increase the reliability of software by using components that have been proven to operate correctly.
- Allow software to be maintained easier because of a modularized approach.
- Allow application developers to deal with the complexity of modern systems.

Because application developers don't have to know the implementation details of reusable components, they can concentrate on the specifics of their application and leave the details of the middleware and foundation libraries to engineers and products that specialize in those areas.

As suggested above, a significant reason for the importance of reusable code on OS/2 is the amount of effort it takes to implement even the simplest of applications using primitive interfaces. The Presentation Manager's APIs, though extremely powerful and flexible, are also very low-level. Many of these APIs require data structures that must be laboriously filled in element-by-element. Even the famous "Hello World" application, which is about three lines of code when text-based, requires nearly a 100 lines of code as a minimal PM application when expressed in C:

```
/* hello.c
 *
 * Hello World. This program creates a standard PM frame
 * window and displays the text "Hello World" centered in the
 * client window.
 */
#define INCL_WIN
#define INCL_GPI

#include <os2.h>      /* include the PM header files */

MRESULT EXPENTRY MyWindowProc(HWND, ULONG, MPARAM, MPARAM);

HAB hab;                  /* anchor block handle */
HMQ hmq;                  /* message queue handle */
```

```

int main(void)
{
    HWND hwndFrame, hwndClient;
    LONG flCreate;
    QMSG qmsg;

    hab = WinInitialize(0);           /* initialize this
                                      * thread for PM */
    hmq = WinCreateMsgQueue(hab, 0); /* create the message
                                      * queue      */

    WinRegisterClass(               /* register client
                                      * window class */
        hab,                      /* anchor block
                                      * handle      */
        (PSZ) "MyWindow",          /* window class name */
        (PFNWP) MyWindowProc,       /* address of window
                                      * procedure   */
        CS_SIZEREDRAW,             /* class style */
        0 );                      /* no extra window
                                      * words      */

    flCreate = FCF_STANDARD;         /* frame creation
                                      * flags */

    hwndFrame = WinCreateStdWindow(
        HWND_DESKTOP,              /* desktop window is
                                      * parent      */
        0,                         /* std window styles */
        &flCreate,                  /* frame control flag */
        "MyWindow",                /* client window
                                      * class name */
        "",                         /* no window text */
        0,                          /* no special class
                                      * style      */
        (HMODULE) 0L,               /* resource is in
                                      * .EXE file   */
        ID_WINDOW,                 /* frame window
                                      * identifier   */
        &hwndClient);              /* client window
                                      * handle */

    WinSetWindowText(hwndFrame, "Hello World Demo");

    /* enter message loop and stay here until a WM_QUIT msg
     * is posted */
    while( WinGetMsg( hab, &qmsg, 0L, 0, 0 ) )
        WinDispatchMsg( hab, &qmsg );

    WinDestroyWindow(hwndFrame);      /* destroy all
                                      */

```

```
        * resources now      */

    WinDestroyMsgQueue(hmq);
    WinTerminate(hab);
    return 0;
}

/*
 * client window procedure
*/
MRESULT EXPENTRY MyWindowProc(HWND hwnd, ULONG msg, MPARAM mp1, MPARAM mp2)
{
    switch(msg)
    {
        case WM_PAINT:
        {
            HPS     hps;
            RECTL   rc, windowRect;
            SWP     swp;
            PSZ pText = "Hello World";

            /* allocate a pres. space for drawing
             * (rc = invalid rect) */
            hps = WinBeginPaint( hwnd, 0L, &rc );
            GpiErase(hps);

            /* determine the window rectangle for
             * centering the text */
            WinQueryWindowPos(hwnd, &swp);
            windowRect.x = 0;
            windowRect.y = 0;
            windowRect.cx = swp.cx;
            windowRect.cy = swp.cy;

            /* draw the text centered in the client window */
            GpiSetBackMix(hps, BM_LEAVEALONE);
            WinDrawText(hps, strlen(pText), pText, &rc,
                        CLR_BLACK, 0,
                        DT_CENTER | DT_VCENTER);

            WinEndPaint(hps);      /* drawing is complete */
            break;
        }

        default:
            return WinDefaultWindowProc(hwnd, msg, mp1, mp2);
    }
    return (MRESULT) 0;
}
```

Of course, this program does much more than the `stdio` equivalent. The text displayed to a user is drawn in a window that can be sized, moved, and closed. And the text will always be centered properly when it is drawn. Now, this is still a very simple application. Imagine a much more sophisticated application that contains several window and dialog procedures. It is not uncommon for one of these procedures to process dozens of messages and span over twenty pages of code.

The main reason that PM applications require so much effort is that the PM API calls are usually very low level. One solution to this problem would be to write an additional set of functions that provide high-level functionality. Although this is done to some small degree, most PM programmers take the "editor inheritance" approach to reuse. This technique involves simply finding a generic or "close" application, copying it into a new set of files, and then customizing it to what the application is suppose to do.

The results are programs that typically have many sequences of program steps that duplicated over and over. The sad part is that most of these sequences have only minor differences between them. Since most editors have better cut-and-paste features that the C language can provide for reuse, the editor approach is usually taken.

The problems with programs of this nature are well known. They tend to be large, difficult to understand, difficult to maintain, and provide little in the way of reusable software components that can be used in new software. The solution to these problems is to use object technology to build on a set of highly reusable classes that hide the low level details and supply a collection of methods that are high level, and logically organized. Application specific functionality is then added with new classes that are inherited from the existing classes, in which the new classes modify existing methods and add new ones. This way, applications need only assemble existing objects and add new ones, without having to reinvent the wheel every time.

Layered Software Development

With object technology, you can build class library packages in layers, using standard components wherever possible in each layer. In the approach recommended by RSI for ObjectPM there are three layers. These are: foundation, model, and application. The foundation layer provides the basic services that are common to applications in general. These services range from providing simple building block classes, such as those for files and text strings, to sophisticated application frameworks, communications frameworks, and persistence managers. In essence, the foundation layer provides the middleware, which includes all services that lie inbetween the application models and the services provided by the operating system and other servers. The ObjectPM class library is an example of a foundation class library.

The model layer, which uses the services of the foundation layer, can be used to provide reusable models of business operations or other real-world entities. For example, model libraries can be built for accounting and inventory, or for systems such as weather systems and flight simulation. A model can provide a

rich description of components and processes which together specify both the structure and behavior of the entity being modeled. Object technology lends itself very well to modeling.

This model-building strategy has some important advantages. All the higher-level objects, as well as the model itself, are general purpose in nature and can be reused in future products. In addition, solutions built on the model are more flexible than solutions based directly on low-level objects. This makes software more maintainable because changes can usually be applied to the model layer without making changes whatsoever to the classes or code in the application layer.

The top layer consists of the actual applications that fulfill specific needs. Instead of the traditional top-down structure that is common in information technology today, these applications are a collection of assembled components; where high-level objects, models, and foundation objects are "glued" together by connecting objects via methods and messages.

The idea behind layered software is to contain the bulk of an application's code in reusable class libraries. As the designers of class libraries such as ObjectPM looked to generalize and identify reusable classes, you should try to do the same within your applications.

OBJECTPM

ObjectPM is a class library intended to provide C++ applications with a true object-oriented interface to OS/2. The main purpose of ObjectPM is to relieve the application programmer of much of the harsh details of coding directly to OS/2's APIs, and to provide a framework for building reusable GUI components. As discussed earlier, the OS/2 APIs are numerous, and the PM interface exists as both API calls and a large collection of messages. ObjectPM reduces the complexity of this system by breaking down the major API categories (Dos, Win, and Gpi, etc), into over 200 classes or "software packages" where each class provides a clean and consistent interface to a single logical component. Instead of working with API's, ObjectPM programmers assemble new, existing, and customized objects into finished applications. Instead of using API data structures, a smaller number of data classes are used, and these are easier to create, initialize, and destroy.

To illustrate these points, here is another version of "Hello World" which has been written in C++ by utilizing the ObjectPM class library:

```
#define InclGraphics
#include <ObjectPM.hpp>

class wAppWindow : public wFrameWindow
{
public:
    wAppWindow();      // wAppWindow constructor
    void Paint();      // paint routine
};
```

```

// bMainThread::Start. Program execution starts here.
// All ObjectPM apps must supply this method.

short bMainThread :: Start()
{
    wAppWindow app;          // Construct top level frame window
    Exec();                  // Go to the message loop.

    return 0;
}

wAppWindow :: wAppWindow()
{
    CreateWindow(OpmStdFrame); // Create the frame
                               // window
    SetCaption("Hello World Demo"); // Set the frame window
                                     // caption
    Show();                     // Make the frame visible
}

void wAppWindow :: Paint()
{
    WindowPS()->Erase();      // Erase the client window
    wTextPen t(WindowPS());    // Create new wTextPen object
    t.SetColor(ClrBlack);      // Set the pen color to black
    t.Display(WindowRectangle(),
              DfCenter | DfVCenter,
              "Hello World");
}

```

The first thing that you may notice about this application is that it is much smaller than the C language version. There are a number of reasons for this. One is that the PM API calls such as WinRegisterClass, WinCreateStdWindow, and WinGetMsg, etc., have been encapsulated by methods that provide a higher level functionality (such as the CreateWindow method). Another important reason is in the way ObjectPM handles events. Note that this application does not declare, register, or implement a window procedure. Instead, it overrides an “event method” called Paint which the ObjectPM kernel calls when it receives a WM_PAINT message for a window managed by a wAppWindow object. Allowing methods to handle the PM messages is a very powerful feature. It allows the code to be more modular (as opposed to the classic “case” statement used in the first program), and allows these methods to be reused and customized by subclasses.

This is a good example of a powerful OOP idiom. It is characterized by publication of a method that the application developer is not intended to ever invoke. Instead, application code is intended to override this method, and code within the class library will invoke it when appropriate. Sometimes the phrase “class framework” is used to describe this approach, and applications are viewed as

components that "plug" various special behaviors into the framework by subclassing and overriding methods. Using this technique, class frameworks such as ObjectPM can provide and organize all the necessary building blocks of a complete application.

The ObjectPM class library is actually two class libraries. The first one, called the ObjectPM BaseSet library is a portable foundation class library that provides:

- Application, task, and thread management
- Inter-thread communications
- File, directory, and extended attribute classes
- National language support
- Date and time management
- String and universal character management
- INI File profile classes
- Condition recording and reporting facilities for errors
- Thread safe collection classes
- Abstract data type framework

The second class library, the ObjectPM WindowManager, provides a large scale object-oriented interface to the OS/2 Presentation Manager. This includes:

- event management
- support for windows, controls, and custom controls
- graphics tools and pictures
- high level data forms management.

The WindowManager represents a philosophy that is different from other commercial class libraries in that portability is not the main concern. Today, it would be very easy to find a host of GUI application frameworks that boast source code portability to other operating platforms. Although in theory this sounds good (and is proper for many applications), a library of this type cannot support features that are unique to one particular environment. For Microsoft Windows and DOS platforms, this is not much of a problem since these platforms are the most limited in functionality. OS/2 is a completely different story. Developers who code to OS/2 want to make use of multi-threading, interprocess communications, CUA-91 controls, Workplace Shell integration, and PM graphics. The ObjectPM WindowManager provides these things.

Thus, the ObjectPM framework provides a large amount of generic functionality to an application or model, but in addition, establishes OS/2-specific policies for such things as event management, the use of graphical objects, thread creation and management, and a common interface of high level data types. On a larger scale, the ObjectPM framework was designed to integrate with other foundation class libraries such as those that provide communications and persistence classes. Together, these frameworks can provide a significant and powerful set of components for building full blown distributed information systems and model class libraries.

Hello: A Sample Program

This section reworks the C++ “hello world” application illustrated earlier, to provide a more detailed introduction to many of the classes and objects used to create an application in ObjectPM. This elaborated version is one of the sample programs shipped with ObjectPM.

The program illustrated here creates a standard frame window and displays one of five messages in the client window. The message can be changed by choosing the “Next” or “Prev” menu options, or by pressing the left or right arrow keys on the keyboard. The application also contains an “About box” which provides some basic information about the program in a dialog window, and a “confirm exit” dialog.

When an ObjectPM application is started, control is passed to a function called `main` contained within the library. The `main` function initializes the application by creating an instance of a class called `bMainThread`. Once the object is created, `main` calls the virtual method `Start` of the `bMainThread` class. All ObjectPM applications therefore override the `bMainThread::Start` method because this is where execution of the application starts. Let’s look at the `bMainThread::Start` method for Hello.

```
void bMainThread :: Start()
{
    HelloWindow w;           // Construct the top level frame window
    Exec();                  // Go to the message loop.
}
```

The first thing this method does is create an instance of the `HelloWindow` object. As we will see shortly, this creates the frame window for the application as well as other objects that are associated with the frame window.

Once these objects are created, there is nothing left to do except paint the “client” window and wait for commands from the user. All Presentation Manager programs are event-driven. For example, it is common for text-based program to poll the keyboard to see if a key has been pressed. In Presentation Manager programs, the system tells the application that a key is ready by sending it a message. In other words, a “keystroke” event occurred.

To get messages and process them, Presentation Manager programs must implement a routine that continually gets and dispatches messages. This is called the message loop. In ObjectPM, the message loop is contained in the `bThread::Exec` method. This method is usually called from `bMainThread::Start` and does not return until the application is given the command to close.

When `Exec` returns, it is time to close down, and in the Hello application this is done in elegant way supported by C++. The instance of the `HelloWindow` object was created in the form of an automatic variable. In other words, it was created on the stack. Upon exit from `Start`, the object goes out of scope and so its C++ destructor is executed. Thus the object, its associated window, and the objects it created are all destroyed via the `HelloWindow` destructor method and

the other destructors in the `wWindow` class hierarchy, completely cleaning up and closing down the application.

Let us now focus on the `HelloWindow` object. It is defined as follows:

```
class HelloWindow : public wFrameWindow
{
    private:
        wMenu *menubar;
        wIcon *icon;
        wAccelTable *acceltab;
        short msg;

        long MenuCommand(wCommandMsg);
        long Close(wMessage);
        void Paint();

    public:
        HelloWindow(); ~HelloWindow();
};
```

The `HelloWindow` class is derived from the `wFrameWindow` class contained in the ObjectPM library. `wFrameWindow` in turn is derived from the `wWindow` class. These classes, along with the other classes derived from `wWindow`, form the window class hierarchy (described in more detail later). All ObjectPM applications that create window objects must do so by deriving a class from one of the ObjectPM window classes, such as `wFrameWindow`, and then creating an object that is an instance of that class.

Inside the `HelloWindow` class definition, four data members are declared. The `menubar`, `icon`, and `acceltab` members will hold pointers to dynamically created `wMenu`, `wIcon`, and `wAccelTable` objects. These objects are created in the constructor of `HelloWindow` and are created dynamically because they must exist for the duration of the application. The `msg` member represents which message will be displayed by the `Paint` method.

The three methods: `MenuCommand`, `Close`, and `Paint` are special methods called “message handlers.” These methods are called from the ObjectPM library when the appropriate event occurs (such as a selecting an item from the menu). They are virtual, overloaded functions that are initially defined in `wWindow` class which is the base class of the window class hierarchy. There are many of these special methods including:

- Create
- TimeOut
- Destroy
- ButtonClick
- InitDlg
- Activate
- Paint

- Close
- MenuCommand
- MenuAction
- PushButtonCommand
- MouseMove
- Control
- ScrollBarChange
- Size
- SaveApplication
- Move
- KeyStroke
- FocusChange
- Semaphore
- FormAction

Each of these methods represents an event or message generated by the Presentation Manager. If you need to process a message that is not included in one of the standard ObjectPM event methods, you can install a message handler method, using the AddMessageHandler method of wWindow, that would be called any time the desired message was sent or posted to the window.

Defining objects derived from a class in the wWindow hierarchy such as wFrameWindow, involves three things:

- 1 . Define the data that is private to the new class.
- 2 . Choose which message handler methods your window will process. This done by including the desired methods in the class definition. ObjectPM will take the default action for all methods that are not included. For example, if the window has a menu bar, it will be necessary to override the MenuCommand method.
- 3 . Define public methods that can be called from elsewhere in the application. This is done to allow manipulation of the window in some way that is specific to the new class. In many cases, this step is not necessary.

The remainder of the class definition specifies a public constructor and destructor for creating and destroying HelloWindow objects. The following code is for the constructor:

```
HelloWindow :: HelloWindow()
{
    CreateWindow(OpmStdFrame);
    SetCaption("Hello World");
    SetSwitchTitle("Hello World Demo Program");
    SetIcon(icon = new wIcon(ResIcon, I_OBJPM));
    msg = 0;

    // create and setup the menu bar
    menubar = new wMenu(this, MB_MESSAGE,
```

```

        "Message\");

menubar->SetSubMenuItemItems(MB_MESSAGE, MI_NEXT,
                             "Next\t\x1a;Previous\t\x1b");

// add an "About..." item to the system menu
wSysMenu sm(this);
sm.AppendSeparator();
sm.AppendItem(SC_ABOUT, "~About...");

// create accelerator for the left and right arrow keys
acceltab = new wAccelTable(this);
acceltab->Set(VkRight, MI_NEXT, AsVirtualKey);
acceltab->Set(VkLeft, MI_PREV, AsVirtualKey);

Show();
}

```

In the above code, the statement:

```
CreateWindow(OpnStdFrame);
```

creates a window. For a number of reasons, windows are not displayed visually when they are first created. The `OpnStdFrame` identifier is a `#define` that sets a number of frame creation flags that specify the frame controls and options:

Option	Meaning
FaShellPosition	Allows the PM to choose the size and position
FaTaskList	Creates a Task Manager entry
FaSysMenu	Creates a system menu
FaSizeBorder	Creates a sizing border
FaTitleBar	Creates a titlebar
FaMinMax	Creates minimize and maximize buttons

The next two lines:

```
SetCaption("Hello World");
SetSwitchTitle("Hello World Demo Program");
```

are used to set the text that appears in the titlebar of the frame window and in the Task Manager's switch list. The following statement dynamically creates an `wIcon` object from an icon defined in the resource file and sets that icon to be shown when the frame window is minimized.

```
SetIcon(icon = new wIcon(ResIcon, I_OBJPM));
```

Note that the methods CreateWindow, SetCaption, SetSwitchTitle, and SetIcon are members of the wFrameWindow class and thus are inherited by the HelloWindow class.

The code:

```
menubar = new wMenu(this, MB_MESSAGE, "Message\\\" );
menubar->SetSubMenuItems(MB_MESSAGE, MI_NEXT,
"Next\t\x1a;Previous\t\x1b");
```

creates the menu bar for the frame window. It defines one submenu with the name wMessage. The next line defines the actual menu items in the wMessage submenu.

Menus in ObjectPM may be defined such as this or constructed from a menu definition in a resource file. In the notation above, submenus end with a backslash (\\"), menu items end with a semicolon (;), separators are indicated with a minus sign(-), and the menu items are separated from the accelerator key text with a tab character (\t). This scheme allows menus to be easily created and modified.

The "MB_MESSAGE" and "MI_NEXT" identifiers are defines that set the id's of the submenu and menu items. When a menu item is selected, the ID of the menu item is used to determine which item was selected.

As illustrated above, the Hello application contains a system menu. Most frame windows do. Often times it is desired to modify the contents of this menu by adding new items or deleting items. The code fragment:

```
wSysMenu sm(this);
sm.AppendSeparator();
sm.AppendItem(SC_ABOUT, "~About...");
```

creates a temporary wSysMenu object that allows the application to work with the system menu. Note that creating a wSysMenu object does not actually create the menu. It simply registers the object with the system menu of "this" frame window, and allows the application to use the methods in the wMenu class to work with it. Once the needed items are appended to the menu, the application does not need this object anymore.

The last object to be created is the wAccelTable object. Creating an wAccelTable object in turn causes an accelerator table to be created for the frame window. Accelerators are a neat little trick implemented by the Presentation Manager. These allow key strokes to be linked to a menu item. In the Hello application, pressing the right arrow key causes the "Next" menu item to be selected, and pressing the left arrow key causes the "Previous" menu item to be selected. The code to create our accelerator table is:

```
acceltab = new wAccelTable(this);
acceltab->Set(VkRight, MI_NEXT, AsVirtualKey);
acceltab->Set(VkLeft, MI_PREV, AsVirtualKey);
```

Finally, Show(); causes the frame window to become visible.

When a frame window is created, the frame window will in turn create a number of "child" windows such as the titlebar, the min/max buttons, etc. The frame window manages these windows automatically and takes care of drawing them. The frame window also creates a window called the "client" window. Your application will appear in this window, and thus the application is responsible for painting it. The Show method shown above will cause the frame to draw itself, and then post a Paint message to the client window. When the paint message arrives, the Paint method of the HelloWindow object will be called. For this sample application, this code appears as follows:

```
void HelloWindow :: Paint()
{
    static char *msgtext [LAST_MSG+1] = {
        "Hello World",
        "Welcome to OS/2 Object Oriented Programming",
        "Introducing ObjectPM",
        "A C++ Class Library and Application Framework",
        "Have a Good Day!!!"
    };

    WindowPS() ->Erase();
    wTextPen t (WindowPS());
    t.SetColor(ClrBlue);
    wFont f (TmsRmn14, UnderscoreFont, WindowPS());
    t.Display(WindowRectangle(), DfCenter+DfVCenter,
    msgtext [msg]);
}
```

Before we get into this code, a few concepts need explaining. Within the Presentation Manager and ObjectPM is an object called a presentation space. This object is a virtual graphics medium that is drawn on, and provides independence from the output device connected to it. Applications do not draw on windows. They draw on presentation spaces. A window is an output device. This mechanism allows a picture drawn in a window to be drawn on a laser printer by connecting a different output device to the presentation space.

Presentation spaces are created in one of two ways. First, the application can create one by creating an instance of a wPressSpace object. Applications that need to draw in units other than pels, or that need advanced graphics capabilities should do this.

The second way of creating a presentation space is to let ObjectPM do it. If the Paint method is to be called, and no presentation space has been created by the application for the window, the ObjectPM library will create one just prior to calling the Paint method. A pointer to the wPressSpace object is returned from the wWindow::WindowPS method. Thus the statement:

```
WindowPS() ->Erase();
```

erases the visible portion of the presentation space and thus erases the client window. Understanding this concept is key to doing any drawing in the Presentation Manager, no matter what platform you choose.

The next fragment of code:

```
wTextPen t(WindowPS());
```

creates a `wTextPen` object used for drawing text, and connects the pen to the `wPresSpace` object. Five primary objects within `ObjectPM` are used for drawing. These are:

<code>wPen</code>	Draws lines, arcs, polygons, paths, and areas.
<code>wTextPen</code>	Draws and formats text.
<code>wFillPattern</code>	Defines patterns and colors used to fill graphical objects
<code>wBitmap</code>	Used to draw, size, and move bit images.
<code>wFont</code>	Defines a font to be used with a <code>wTextPen</code> or <code>wFillPattern</code> .

The code:

```
wFont f(TmsRmn14, UnderscoreFont, WindowPS());
```

creates a `wFont` object representing a 14pt-underlined Times Roman font, and sets it as the current font. The last line of code in the `Paint` method finally draws the text using the blue `wTextPen` and the Times-Roman font:

The statement:

```
t.Display(WindowRectangle(),
DfCenter+DfVCenter,
msgtext[msg]);
```

first calls the `wWindow::WindowRectange` method which returns a `wRect1` object representing the position and size of the client window. The `wTextPen::Display` method is then called, which displays the appropriate text horizontally and vertically centered in the rectangle.

Upon exit from the `Paint` method, both the `wTextPen` and the `wFont` will be destroyed.

At this point, both the frame and client window's are drawn. The application has nothing left to do except wait for a menu command. Because of the accelerator keys, a menu command can be generated by choosing a menu item, or pressing the left or right arrow keys. Let's say for example the "Next" menu item was chosen. When this occurs, the `ObjectPM` library will call the `MenuCommand` method and pass it a `wCommandMsg` object. Here is the code for the `MenuCommand` method:

```

long HelloWindow :: MenuCommand(wCommandMsg m)
{
    switch(m.usCmd())
    {
        case MI_NEXT:
            msg = msg == LAST_MSG? 0: msg + 1;
            break;

        case MI_PREV:
            msg = msg == 0? LAST_MSG: msg - 1;
            break;

        case SC_ABOUT:
        {
            AboutDialog aboutWin(this);
            return FALSE;
        }
    }
    RePaint();
    return FALSE;
}

```

The operation of this method is very simple. The usCmd method of the wCommandMsg object returns the id of the selected menu item. If the menu item was MI_NEXT or MI_PREV, then the msg member will be adjusted accordingly and the RePaint method of the wWindow class will be called to invalidate the window. This will cause the ObjectPM library to call the Paint method at a later time.

If the "About..." menu item is selected from the system menu, the MenuCommand method will be called with the id set to SC_ABOUT. In this case, an instance of the AboutDialog class is created. This class implements a simple dialog window that displays some basic information about the Hello application. To create a dialog window, you must first draw it using the dialog box editor. The definition of the dialog window must then be compiled into the program as a resource using the resource compiler. Within the code, a class must be defined which is derived from the wDialogWindow class. The definition of the AboutDialog class looks like this:

```

class AboutDialog : public wDialogWindow
{
    private:
        long InitDlg(wInitDlgMsg)
        {
            ChangePosition(PosCenter, OwnerWindow());
            return FALSE;
        }

        long PushButtonCommand(wCommandMsg)
        {

```

```

        Dismiss();
        return FALSE;
    }

public:
    AboutDialog(wWindow *owner) : wDialogWindow (D_ABOUT)
        { CreateWindow(owner); }
};
```

Notice that the code is in-lined along with the definition of the class methods. In fact, all the processing for the dialog window is contained within this class definition. The `AboutDialog` method is the constructor for the class. It passes the resource ID of the dialog window to the `wDialogWindow` constructor, and then it calls the `wDialogWindow::CreateWindow` method to create the actual window.

The `InitDlg` and `PushButtonCommand` methods are both message handlers. The ObjectPM library calls the `InitDlg` method after all the dialog controls are created and before the window is actually shown. The `AboutDialog` class uses the `InitDlg` method to implement the code needed to center the dialog window inside the main window.

Whenever a push-button is clicked in a dialog window, the `PushButtonCommand` method is called. Like the `MenuCommand` method, a `wCommandMsg` object is passed that indicates which button was clicked. Since the About dialog window contains only one push-button, it is not hard to figure it out. The code for this method simply dismisses (destroys) the dialog window.

OBJECTPM ARCHITECTURE

For the remainder of this chapter, we cover a number of detailed aspects of the ObjectPM class library in order to further demonstrate application of the object-oriented paradigm to OS/2 sub-systems. These topics include multi-threading, condition handling, windows, event handling, control windows, graphics, and data forms.

Applications and Threads

The top level class of the BaseSet library is the `bApp` class, which represents the application as a whole. This class contains methods for querying country and code page information, setting and querying of current modules and message files, and an interface to an application-global dictionary that applications can use to store objects by symbolic names.

To support threading, ObjectPM supplies the classes `bThread`, `bMainThread`, and a collection of semaphore and guardian classes that provide inter-thread synchronization and event signaling. The class heirarchy for these classes is as shown in Figure 21.1.

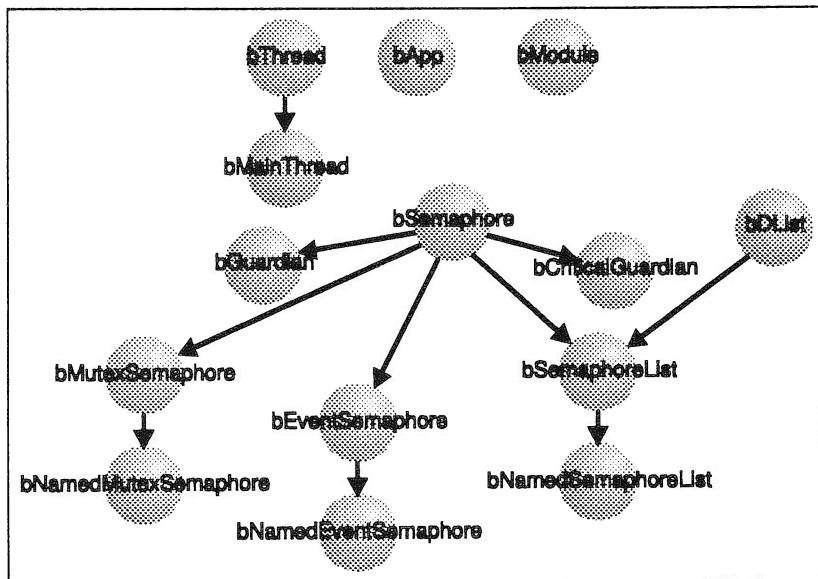


Figure 21.1. Thread Framework

As mentioned above, although a program starts at a function called `main`, ObjectPM programs do not need to supply this function. Instead, the `main` function in ObjectPM creates an instance of the `bMainThread` class, which initializes the class library, registers the primary thread, and then calls the user-supplied `bMainThread::Start` method. This method is where execution begins for ObjectPM programs.

Starting applications in this way allows the ObjectPM class library to do the work of initialization that all Presentation Manager programs must do themselves. However, this mechanism is not cast in granite. Applications can supply their own `main` function and create the instance of `bMainThread` themselves. This approach is in fact required for non-PM applications that want to use the ObjectPM class library.

Multi-threading is achieved in ObjectPM applications by creating instances of classes that are derived from the `bThread` class. Each class derived from `bThread` must override the `Start` method. When an application creates a derived `bThread` object, the execution point for that thread will begin at the `Start` method. The `bThread` class provides all the methods needed to control threads including setting priorities, suspending, resuming, and sleeping.

To synchronize access to shared objects between threads requires semaphores. In ObjectPM, OS/2 semaphores are implemented in the `bMutexSemaphore`, `bNamedMutexSemaphore`, `bEventSemaphore`, `bNamedEventSemaphore`, and `bSemaphoreList` classes. These semaphores can be used for both mutual exclusion and event signaling purposes. The `bSemaphoreList` classes provide a thread with the ability to wait on multiple semaphores at one time.

To demonstrate these classes, we'll create a PM program that displays one of three messages in a second thread depending on a menu selection. The main

thread of this application will make use of a bSemaphoreList object and a collection of bEventSemaphore's in order to send command events to the drawing thread.

The source code for this application follows. Note how the constructor for DrawThread chains construction upwards to its parent class, bThread:

```

class DrawThread : public bThread
{
public:
    DrawThread(bThread *parent, const char *name)
        : bThread(parent, name, 0, 4096,
                   ThreadAncBlock) { }
    short Start(); // entry point for new thread
};

class AppWindow : public wFrameWindow
{
private:
    wMenu *menubar;
    wIcon *icon;
public:
    pbSemaphoreList semList;
    AppWindow();
    ~AppWindow();
    long MenuCommand(wCommandMsg);
    void Paint() { WindowPS()->Erase(); }
};

short MainThread :: Start() // Program execution starts here
{
    AppWindow w;           // Construct top level frame
                           // window

    // add the address of the AppWindow object to the
    // application's dictionary and start the second thread
    ThisApp->PutObject("APPWINDOW", &w);
    DrawThread t(ThisThread, "DrawThread");

    Exec();                // Go to the message loop.

    // Send 'Self Destruct' cmd to second thread
    ((bEventSemaphore*)((*w.semList)[CMD_QUIT]))->Post();

    // Elevate the priority of the second thread and wait up
    // to 2 seconds for it to kill itself.

    t.SetPriority(PrtdMaximum);
    for (int i = 0; i < 20; i++)
    {
        if (!t.IsAlive())

```

```
        break;

    Sleep(100);      // Give up the CPU for 100ms
}
return 0;          // Exit the process
}

AppWindow :: AppWindow()
{
    CreateWindow(OpnStdFrame);           // Create the frame
    SetCaption("Semaphore Demo");       // Set captions
    SetSwitchTitle("Semaphore Demo");
    SetIcon(icon = new wIcon(ResIcon, I_OBJPM)); // Set the
                                                // app's icon

    menubar = new wMenu(this, MB_MESSAGE, "~Message\\\");
    menubar->SetSubMenuItems(MB_MESSAGE, MI_FIRST,
    "First;Second;Third");

    // Create the semaphore list (mux semaphore)
    semList = new bSemaphoreList(FALSE);

    // ...and the individual semaphores
    semList->Add(new bEventSemaphore(MI_FIRST));
    semList->Add(new bEventSemaphore(MI_SECOND));
    semList->Add(new bEventSemaphore(MI_THIRD));
    semList->Add(new bEventSemaphore(CMD_QUIT));
    semList->ResetAll(); // Reset all the semaphores
    Show();              // Make the frame visible
}
AppWindow :: ~AppWindow()
{
    // Destroy the semaphores and the list
    semList->RemoveAll();

    delete semList;
    delete menubar;
    delete icon;
}

long AppWindow :: MenuCommand(wCommandMsg m)
{
    // select the proper semaphore by index and post
    // the event.
    ((bEventSemaphore*)
        ((*semList)[m.usCmd() - MI_FIRST]))->Post();
    return FALSE;
}

***** Class members running in the second thread *****
```

```

short DrawThread :: Start()
{
    AppWindow *appWinp = (AppWindow
*)ThisApp->GetObject("APPWINDOW");
    bEventSemaphore *semCmd;

    for (;;)
    {
// Wait for one of the semaphores to clear. When
// one does,
// the Wait method will return a pointer to the Semaphore
// object that actually cleared.

        semCmd = (bEventSemaphore*) (appWinp->semList->Wait());

        if ( semCmd )
        {
            if (semCmd->SemaphoreId() == CMD_QUIT)
            {
                Quit();
                return 0;
            }

            wPressSpace ps(appWinp);
            wTextPen t(&ps);

            t.SetBackMix(BmOverPaint);
            t.Printf(20, 20, "%d menu item chosen!      ",
                    semCmd->SemaphoreId());
            semCmd->Reset();           // "re-arm" the semaphore
        }
    }
}

```

The Initialization code of this application creates a semaphore list object (which wraps the OS/2 mux-wait semaphores) and then creates and adds four event semaphores to the list. Once this is done, a DrawThread object is created which starts the second thread. Remember that the Start method will be called as the entry point for the new thread, which is implemented in the last method in the code listing. In this method, the second thread blocks on the semaphore list by invoking the Wait method on the bSemaphoreList object, and waits for work to do.

When the user selects a menu item, the MenuCommand method is called with an argument that contains the ID of the selected item. That ID is used to select the associated bEventSemaphore object from the semaphore list, and the Post method is invoked on it. This causes the second thread to wake up and the value returned from the Wait method will be the pointer to the bEventSemaphore object that was posted. The remaining code of the Start method gets the ID of

the semaphore, uses it to display a message in the client window, rearms the semaphore, and then waits again on the semaphore list for a new command.

One other noteworthy item about this program is the way it terminates. To clean up properly, the application must destroy the objects that it creates. But OS/2 provides no way for one thread to destroy another. To get around this, the application uses one of the event semaphores to tell the draw thread to quit, which it uses in the last code section of the `bMainThread::Start` method. Once the `quit` command has been posted, the main thread goes into a loop to wait for the second thread to die. When the `IsAlive` method returns false, the actual thread managed by the `DrawThread` object has now stopped and it is now safe to destroy the object.

In addition to the semaphore classes, ObjectPM provides a set of guardian classes that applications can use to serialize method calls on objects. Guardians are objects that contain a pointer to a mutex semaphore contained within an object and are created as automatic variables upon entry of a scope or method. Upon construction, the guardian requests the semaphore. If the requesting thread is blocked, that means that another thread is currently executing a method on the object. When OS/2 grants the request, the calling thread now "owns" the object until the guardian object goes out of scope. When this happens the guardians destructor is called which releases the semaphore.

Conditions and Facilities

All systems and application software systems need to have ways of handling and reporting error conditions. Most programs will implement one or more functions that are called by other parts of the program to display or record an error condition and take some action (usually based on a response from the user). With layered software systems, such as object-oriented programs that are based on a stack of class libraries, the task of identifying errors and reporting them to the user can be very difficult. To help with this, ObjectPM supplies a number of classes and methods that implement an error management system.

The ObjectPM `bThread` and `bApp` classes work together to form a system for recording conditions and dispatching the appropriate handlers. When the `bApp` class is instantiated, it allocates a pool of `bCondition` objects. Each `bCondition` object is a data object that describes a condition type, facility, condition code, location, and condition specific arguments. A condition can be any of the following:

- An error
- A fatal error
- A warning
- An OS/2 exception
- Application defined

The interface to this system is through methods in the `bThread` class. To record a condition, the application or class library method must call the `bThread::NewCondition` method (or one of the ObjectPM helper macros) which causes

the bThread class to allocate a bCondition object and add it to its list of conditions. Using an I/O stream-like interface, an application can add information and arguments to a bCondition object.

For example, the following code fragment records a “File not found” error condition for the current thread, records the location of the error, an argument (the filename), and asserts the condition by terminating the statement with the constant, “Tell”:

```
CONDITION(FAC_APP,CtError,E_FILENOTFOUND) ATLOC(A_FILEOPEN, 0)
    << "MEMO.DOC" << Tell;
```

Conditions can be stacked. This allows applications and systems that are built on layers of class libraries (or facilities) to record a “trail” of errors, originating from a call in a low-level subsystem all the way up to the application layer. To do this, a condition statement would be used like the one above, but a constant “Remember” would be used instead of “Tell.”

Each layer or subsystem is called a facility. A class called bFacility allows the name, ID, message file, and help file names to be declared for each facility so that ObjectPM can do most (or all) of the work of formatting error strings to be displayed or logged. By default, error messages are extracted from standard OS/2 message files. However, developers can override a method of the bFacility class to build error strings any way they want.

In order to report or handle conditions, applications must create one or more objects derived from the bConditionHandler class and install them using the bThread::AddEventHandler method. Objects of this type indicate the type(s) of conditions they can handle and the address of a method to be invoked to handle the condition. For example, the following sample code declares and installs a condition handler for errors and an application facility:

```
#define FAC_HELLOWORLD      20          // facility ID
#define FAC_APPMSGFILE      "hello.msg" // message file name
#define FAC_APPHELPFILE     "hello.hlp"  // help file library
#define FAC_APPHELPOFFSET   2000        //
offset=help_panel_id-error_code

class AppErrorHandler : public bConditionHandler
{
    private:
        bString appName;

    public:
        AppErrorHandler(char *appName, long condType);
        ~AppErrorHandler();

        short ProcessErrors(bCondition *c); // error handler
};

short bMainThread :: Start()           // Program execution
// starts here
```

```

{
    // attach the error handler
    AppErrorHandler errorHandler("hello", CtError);
    AddEventHandler(&errorHandler);

    // Create and register the application facility
    bFacility *fac = new bFacility(FAC_HELLOWORLD, "App",
FAC_APPMSGFILE);
    fac->SetHelpFilename(FAC_APPHELPFILE, FAC_APPHELPOFFSET);
    ThisApp->AddFacility(fac);

    HelloWindow w;      // Construct the top level frame window
    Exec();             // Go to the message loop.
}

// AppErrorHandler constructor. This method primarily sets
// the type of condition(s) that can be handled and the
// handler methods address

AppErrorHandler :: AppErrorHandler(char *_appName, long
condType)
    : bConditionHandler(condType, this,
EVENTIMETHOD(AppErrorHandler,
ProcessErrors)),
        appName(_appName)
{
}

```

With the above code, this application is now ready to process error conditions.

As mentioned before, a list of bConditionHandler objects is maintained for each bThread object. When conditions are asserted, ObjectPM searches the current thread for a handler that matches the condition type (e.g. error, warning, etc.). If one is not found, the parent thread is searched, and so on, until a handler is found. Once a handler is identified, the appropriate method is called with a list of one or more bCondition objects that describe the error.

For example, if the CONDITION form above were executed in our sample program, the error handler would be found and the ProcessErrors method would be invoked with the condition object as an argument. The next code fragment illustrates the implementation of the ProcessErrors method:

```

short AppErrorHandler :: ProcessErrors(bCondition *c)
{
    // get the pointer to the application frame window from the
    // application's object dictionary
    wWindow* dialogParent = (wWindow
*) (ThisApp->GetObject("AppFrame"));
    if (!dialogParent)
        dialogParent = DesktopWindow;

    // show the error in the ObjectPM standard error dialog
}

```

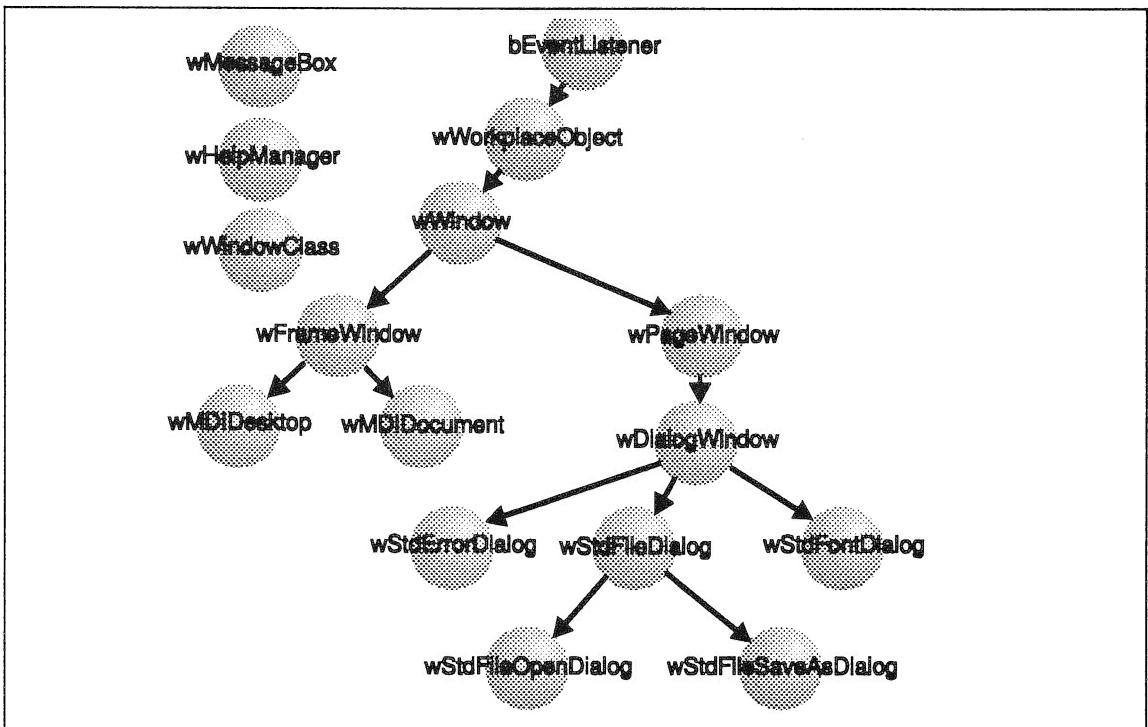


Figure 21.2. Window Class Hierarchy

```

wStdErrorDialog errDlg(dialogParent, c);
if (!errDlg.GetResult())
{
    // exit the process if the user chose "Abort"
    exit(0);
}
return TRUE;
}
  
```

Note that this method makes use of the ObjectPM **wStdErrorDialog** class for displaying the error to the user. Like all of the standard dialogs, this class provides a number of customization features so that applications can customize its appearance and operation.

Windows

The heart of a GUI class library are the primary window classes. These classes provide the functionality for application windows, dialogs, controls, as well as implementing a system for routing and presenting window messages. The primary window classes are illustrated in the class hierarchy in Figure 21.2. The **bEventLister**, **wWorkplaceObject**, and **wWindow** classes work together to form the foundation for ObjectPM window objects and provide methods

that control the behavior of all windows. The bEventListener class has no methods or data. It serves as a base class for taking the address of event handling methods. As a result, any class derived from bEventListener can contain methods for processing window messages. The wWorkplaceObject class provides the base class for all window and non-window objects that support the drag and drop protocol. This class provides methods and data members for setting an objects type, name, supported rendering mechanisms/formats, and drag image. This framework greatly simplifies the code applications need to provide in order to support drag and drop.

The wWindow class defines the interface common to all ObjectPM window objects and implements the event manager framework. An instance of this class represents a Presentation Manager window.

A window is a rectangular area of the screen where an application displays output and receives input from the user. All windows share a common set of properties. These properties include:

- A Size
- A Position
- Style and state settings
- A parent
- An owner
- A window handle
- A window ID

Many of the methods of the wWindow class are dedicated to setting and querying the value of these properties.

In many instances, a window is referred to in the sense of an “application window.” An application window usually refers to a top-level frame window of an application, including the frame, titlebar, min-max buttons, system menu, and a “client” window. The client window is the large area of the window used by the applications themselves. The ObjectPM class for creating and manipulating top-level frame windows is wFrameWindow. The methods of this class deal strictly with the operations of the frame, whereas the inherited methods from the wWindow class are used to work with the client window. For example, invoking the GetFrameSize method on a wFrameWindow object, which is defined in the wFrameWindow class, will return the size of the entire frame window. Invoking the GetSize method on the object, which is inherited from wWindow, will return the size of the client window.

The wFrameWindow class is used by applications to build the top-level frame window. It is also possible to create wFrameWindow objects that will appear within other wFrameWindows. Some applications use frame windows inside a frame window quite extensively. An example of this is the OS/2 Help Facility. In this subsystem, each frame window contains a different help panel. Applications of this type use a framework known as the Multiple-Document-Interface or MDI. To build an application with this type of interface, applications need to first create an instance of a class derived from the wMdiDesktop class. Each

frame window within the outer desktop frame must be an instance of a class derived from wMdiDocument.

Together, the wMdiDesktop and wMdiDocument classes implement all MDI features. These features include the proper menu handling between the desktop and documents, tiling and cascading of document windows, hiding and showing of document frame controls, and a popup dialog for selecting a document to activate. Since MDI behavior is not implemented in PM (as it is in Windows), these classes represent a significant amount of work and are key features of the ObjectPM framework.

Another category of primary window classes are "page" and "dialog" windows. Page windows, implemented by the wPageWindow class, are used for changing client windows. In many PM applications (including the Workplace Shell), opening an item or selecting a menu item results in the creation of a new frame window which is popped up on top of the frame with which the user was working. If the application has a highly nested nature, then the number of frame windows can be excessive, causing the application to be overly complex and a cluttered desktop. Instead, the application could use an approach which allows the client window to change to new views or "pages."

An example of an application of this style is the Query Manager that comes with IBM's DB2/2 and Database Manager. This application uses buttons on one client window page to form a "main menu," and then it changes the client window and menu bar depending on the actions selected by the user. The wPageWindow class of ObjectPM defines a client window, a menu bar, and, optionally, a list of child pages. When an application needs to change client windows, all it has to do is call the ChangePage method of the wFrameWindow class and pass the appropriate wPageWindow object as an argument.

Since wDialogWindow is derived from wPageWindow, a modeless dialog window can also be used as a client window. This technique can eliminate the need to code a keyboard interface.

Dialogs are an important component GUI applications because they are generally used to obtain input from the user and inform the user of important events. ObjectPM supplies the wDialogWindow class for building application-defined dialogs as well as a collection of commonly used standard dialogs for opening and saving files, choosing fonts, displaying errors, and presenting messages.

Using standard PM coding techniques, it is necessary to code a separate window procedure for each dialog that the application uses. These window procedures follow the same model as the window procedure in the "Hello World" program listed at the beginning of this chapter. As PM programmers know, these dialog window procedures can be very complex as well as difficult to set up. ObjectPM dialog objects follow the same event management scheme as standard windows, where the events (window messages) normally processed by a window procedure are routed to the methods of the wDialogWindow class and its subclasses. This makes it simple to code dialog boxes, allows the code to be more modular, and, most importantly, allows the processing methods to be re-used by subclasses.

To demonstrate how a dialog would be coded using ObjectPM, the following section of code from a simple phone book program pops up a dialog for entering a new name and phone number.

This first section of code illustrates the definition and implementation of the dialog:

```
#define NAME_LEN 25
#define PHONE_LEN 14

class NewEntryDialog : public wDialogWindow
{
    private:
        wEntry *nameEdit, *phoneEdit;

    public:
        NewEntryDialog(wWindow *owner);
        ~NewEntryDialog();

        // event methods
        long InitDlg(wInitDlgMsg);
        long PushButtonCommand(wCommandMsg);

        // public data members
        bString name, phone;
};

// This method is the constructor. It passes the resource id
// of the dialog template to the base class constructor,
// initializes its data members and then creates the dialog //
itself.

NewEntryDialog :: NewEntryDialog(wWindow *owner)
                : wDialogWindow(D_NEWENTRY)
{
    nameEdit = NULL;
    phoneEdit = NULL;

    // create the physical dialog window. Since this is a
    // modal dialog,
    // this call will not return until the dialog is dismissed.
    CreateWindow(owner);
}

NewEntryDialog :: ~NewEntryDialog()
{
    delete nameEdit;
    delete phoneEdit;
}
```

```

// This method is called after dialog window and all child
// windows
// have been created but has not been made visible to the
// user yet.

long NewEntryDialog :: InitDlg(wInitDlgMsg)
{
// first create control window objects for the entry fields
    nameEdit = new wEntry(this, E_NAME);
    phoneEdit = new wEntry(this, E_PHONE);

    // set the edit field lengths
    nameEdit->SetMaxLength(NAME_LEN);
    phoneEdit->SetMaxLength(PHONE_LEN);

    // center the dialog
    ChangePosition(PosCenter, OwnerWindow());

// return FALSE meaning that the initial focus window has not
// been changed. once this method returns, the dialog will
// be made visible to the user.
    return FALSE;
}
// This method is called when the OK or Cancel
// buttons are selected

long NewEntryDialog :: PushButtonCommand(wCommandMsg m)
{
    if (m.usCmd() == CMD_OK)
    {
// save the text in the entry fields and make sure
// they were filled in
        name = nameEdit->GetText();
        phone = phoneEdit->GetText();

        if (name.IsBlank() || phone.IsBlank())
        {
            wMessageBox(this,
                        "Edit Error",
                        "Please fill in all fields!",
                        MbOk | MbIconExclamation)

            return 0;    // do not dismiss dialog
        }
        Dismiss(TRUE); // dismiss with TRUE result
    }
    else Dismiss(FALSE) // user canceled dialog

    return 0;
}

```

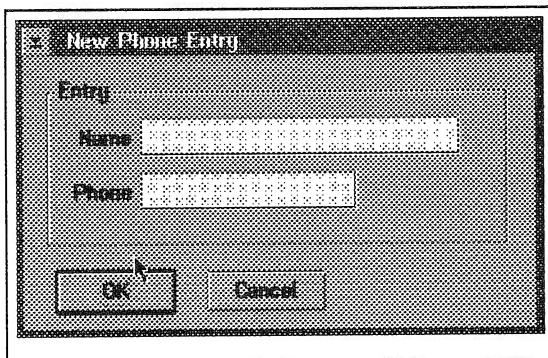


Figure 21.3. Phone Example Dialog

Note that this code makes use of the `wMessageBox` class to display an error message dialog when the user does not fill in one or more of the edit fields. The `wMessageBox` class is one of the standard dialog classes that are used a great deal by ObjectPM applications.

The actual dialog window used in this implementation was defined using a dialog box editor. This tool also allows the window and resource identifiers to be defined, and saves the dialog box definition as a dialog template which is compiled into the application's resources. Figure 21.3 illustrates the dialog window.

The next code fragment demonstrates the use of our dialog class:

```
// This method is called when the user selects "New..." from
// the "Item" menu

void PhoneBookApp :: NewItem(wMessage)
{
    // create the dialog window by creating an instance of
    // NewEntryDialog.
    // The constructor will not return until the user
    // dismisses the dialog.
    NewEntryDialog dlg(this);

    if (dlg.GetResult())
    {
        // if the user selected "Ok", then create a new phone
        // book entry and add it to the list.
        AddEntry( new PhoneBookEntry(dlg.name, dlg.phone) );
    }
}
```

As you can see, the dialog class is simple and easy to use, illustrating one of the strongest features of object oriented programming — the ability to encapsulate and hide complex software with an easy to use interface.

Event Management

One of the most important aspects of the wWindow class is that of handing messages. The Presentation Manager is an event driven system. When an event happens, (such as a button click, mouse move, or keystroke), the Presentation Manager builds a message that identifies the event including relevant parameters, and dispatches it to the proper window. All Presentation Manager C programs must supply a "window procedure" for processing these messages.

In ObjectPM, most events are handled by a collection of virtual methods defined in the wWindow class along with a collection of message classes. If an application needs to process an event, such as "Left Mouse Button Down," it simply overrides the Window class's ButtonClick method, providing the appropriate code to process the button event in this method. Many of these "event" methods are passed message objects when they are called. For example, the ButtonClick method is passed a wButtonClickMsg object when it is called. The **wButtonClickMsg** object has methods that are used to determine the button number, and whether the button was clicked, released, or double-clicked.

The class declaration below illustrates the wButtonClickMsg definition.

```
class wButtonClickMsg : public wMessage
{
    private:
        ushort bnum;
        ushort activity;

    public:
        wButtonClickMsg(ulong msg, MPARAM mp1, MPARAM mp2,
wWindow *pw=0);
        ushort& usButtonNum() { return bnum; }
        ushort& usActivity() { return activity; }
        ushort& x() { return SHORT1FROMMP(mp1); }
        ushort& y() { return SHORT2FROMMP(mp1); }
        wPointl Position() { return wPointl((long)x(),
(long)y()); }
};
```

In addition to the virtual methods, the event mechanism used to handle conditions can also be used for window messages. Using this scheme, a method of any class can be called when a specific message is delivered to a particular window object. This mechanism can be used to implement owner-draw controls and menus, and to redirect command events from menus and push-buttons. For example, the next code fragment is from our phone book program which illustrates the technique for binding a message to a particular method:

```
class PhoneBookApp : public wFrameWindow
{
    private:
        PhoneEntryList entries;
        wMenu *menubar;
```

```
wIcon *icon;

public:
    PhoneBookApp()
    ~PhoneBookApp()

    void Paint();
    long NewEntry(wMessage);
    ...
};

// This method is the constructor for the top level frame

PhoneBookApp :: PhoneBookApp()
{
    CreateWindow(OpnStdFrame);
    SetCaption("Phone Book");

    // construct the frame's menu and icon from the app's
    // resources
    menubar = new wMenu(this, MPhoneBookId);
    SetIcon(icon = new wIcon(ResIcon, IPhoneBookId));
    // install handler for the "new entry" menu item
    AddMessageHandler(this, EVENTMETHOD(PhoneBookApp,
NewEntry),
                           WM_COMMAND, MiAddEntry);
    Show();
}

```

In the PhoneBookApp constructor, the AddMessageHandler method is used to install a handler that will process all WM_COMMAND messages for this window that have a command ID of MiAddEntry. The EVENTMETHOD macro is used to simplify the syntax of taking the address of the NewEntry method. This technique completely eliminates the need for the dreaded switch statements found in typical PM programs. It allows these methods to be reused by subclasses and makes structure of the application more modular.

The Forms Manager

In the example program above, we demonstrated a much simpler way of coding a dialog window than the traditional method used by PM programmers. Although this method allows us to create dialogs easier and manage events better, it still leaves the programmer with the burden of dealing with raw control windows.

Consider a dialog that has 20 or more edit fields along with radio buttons and check boxes. The amount of code necessary to populate, validate, and transfer data from these controls would be quite extensive. Add to this the requirement to deal with various data types such as dates, money, and numbers, and now this code must also parse, validate, and convert these data fields appropriately.

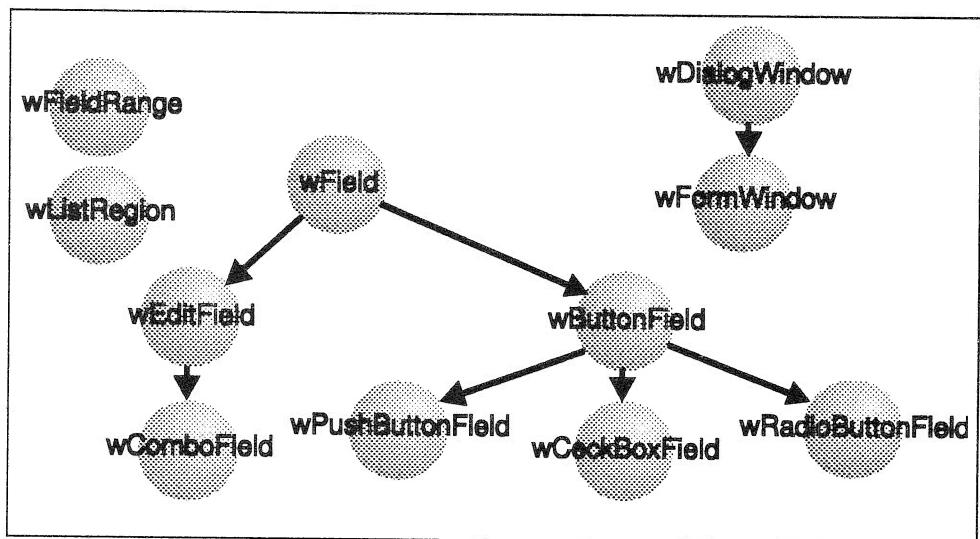


Figure 21.4 Forms Manager Hierarchy

This is why complex dialogs can get so out of hand. It is an unfortunate reality that PM and Windows programmers have had to deal with these limitations for so long. Even features such as edit masks, which have been available in UI libraries for mainframes and DOS for years, are still not supported in the graphical environments.

The ObjectPM Forms Manager is a collection of classes that are aimed at solving these problems. The Forms Manager provides applications with the ability to process dialog windows as data forms. It changes a standard dialog window into a "form," and changes edit controls and buttons into "fields." The class hierarchy of the FormsManager and it's datatype framework is shown in Figure 21.4.

The wFormWindow class represents the dialog window itself. A wFormWindow contains a collection of one or more wField objects and a collection of zero or more wListRegions. When a form is created, a virtual event method of the wFormWindow class called Init is invoked. This gives the application the opportunity to load the data into the appropriate variables before the form is shown. Likewise, there are two other event methods: Validate, and Save which are called by the Forms Manager when it is time to validate and save the data on the form.

The wField class is the base class for all fields. Like the wFormWindow class, it handles events, but at a field level. Applications can supply methods that will be called by the FormsManager whenever the user enters or exits the field, or when the field requires user-level validation. The wEditField class provides data entry fields. This class, along with the FormsManager itself, uses a collection of "field" data-type objects for converting field data-types to and from their string representations. The base class for the data conversion classes is fDataType which is defined as follows:

```

class fDataType
{
protected:
    ushort id;

public:
    fDataType();
    virtual ~fDataType();

    ushort GetID()           { return id; }
    void SetID(ushort _id)   { id = _id; }

    virtual
        short ValidateChar(bUChar c, const char *fmt)
        = 0;
    virtual
        short Validate(const char *s, const char *fmt)
        = 0;
    virtual
        short SetFromString(const char *s,const char *fmt)
        = 0;
    virtual
        short SetFromNativeType(const void*v,bool fInitBlank)
        = 0;
    virtual
        short SetFromObject(fDataType *obj)
        = 0;
    virtual
        short CopyToString(char*dest,short len,const char*fmt)
        =0;
    virtual short CopyToNativeType(void *dest) = 0;
    virtual short CopyToObject(fDataType *obj) = 0;
    virtual bool IsLessThan(fDataType *obj) = 0;
    virtual bool IsGreaterThan(fDataType *obj) = 0;
    virtual bool IsEqual(fDataType *obj) = 0;
    virtual fDataType *Copy() = 0;
    virtual
        bString DefaultMask(const char *fmt, ushort fldlen);
    virtual bool IsNil() = 0;
    void SetToNil() { SetFromString("", ""); }
};


```

The `fDataType` class defines a framework for string/character validation, string parsing and formatting, and the setting and getting of native data types (such as short, long, float, etc). ObjectPM supplies more than a dozen data type classes derived from this class which provides the Forms Manager with a healthy set of edit field data types. Application developers who need to add new data types derive their data type classes from `fDataType` and implement the above methods. Making the data type known to the Forms Manager is done by cre-

ating an instance of the class and adding it to the repository of data type objects through methods of the bApp class.

To show you an example of how this system works, let us replace the dialog in the phone book application with a form. The code listing below illustrates the definition and implementation of the form:

```
#define NAME_LEN 25
#define PHONE_LEN 14

class NewEntryDialog : public wFormWindow
{
public:
    NewEntryDialog(wWindow *parent);

    // public data members
    bString name, phone;
};

NewEntryDialog :: NewEntryDialog(wWindow *parent)
    : wFormWindow(D_NEWENTRY, 2, 0, DlgModal)
{
    wEditField *ef;

    // register the 'Name' entry field as a required string field
    ef = new wEditField(FtString,
                        NAME_LEN, E_NAME, &name, NULL,
                        FsRequired);
    AddField(ef);

    // now add the 'Phone' field along with an edit mask
    ef = new wEditField(FtString,
                        PHONE_LEN, E_PHONE, &phone, NULL,
                        FsRequired | FsMustFill);
    ef->SetEditMask("(999) 999-9999");
    AddField(ef);

    // register the Ok and Cancel buttons
    SetActionButtons(CMD_CANCEL, CMD_OK, 0);

    // put up the dialog
    FormUp(parent);
}

// This method is called when the user selects "New..." from
// the "Item" menu

void PhoneBookApp :: NewItem(wMessage)
{
    // create the form by creating an instance of
    // NewEntryDialog.
```

```
// The constructor will not return until the user
// dismisses the dialog.
NewEntryDialog dlg(this);

if (dlg.GetResult() == AcFormExit)
{
    // if the user selected "Ok", then create a new phone
    // book entry and add it to the list.
    AddEntry( new PhoneBookEntry(dlg.name, dlg.phone) );

}
}
```

Notice that the application does not have to worry about setting or getting the contents of the edit field. The forms system automatically converts the data objects to and from a string form and validates the format. Although there is less code in this version of the dialog, it has more features. There is an edit mask set into the Phone field and it is filtered properly so that the user can only enter numbers. In addition, both fields are verified that they have been entered, and the Phone field is checked to make sure that the phone number is complete.

The last form class we will cover is the wListRegion class. Objects of this class are used to manipulate row-oriented data. The central control of a **wListRegion** is a listbox for displaying the list of data rows. Unlike text in a normal listbox, the text in a **wListRegion**'s listbox can be formatted into sub-columns. Options are provided to set column justification, hidden columns, and static text. In addition, the application can associate a **wEditField** object with a column. Thus when the user selects an item in the listbox, the **wListRegion** automatically copies the text from the column to the appropriate **wEditField** so that the user changes the data. If the user clicks the right mouse button inside the listbox, all edit fields will be cleared and the **wListRegion** will be set-up so the user can add a new row of data to the list.

The ObjectPM Forms Manager is a powerful framework that is constantly being improved. In internal projects and in many of RSI's customers' applications, its features and time-saving abilities have been much appreciated.

PM Graphics

As we have seen, C++ is well-suited for programming Presentation Manager applications. With features such as encapsulation, inheritance, and virtual methods, you can build clean and elegant models of PM objects and a powerful event handling system. Other C++ features such as operator overloading and cast operators, while strictly unrelated to inheritance and subclassing, are also quite valuable for graphics programming.

Consider the class definitions below that show the definitions of **wPoint1**, **wDimension**, and **wRect1** which are fundamental classes for representing the position and size of graphical objects.

```

class wCoord
{
protected:
    long a;
    long b;
public:
    wCoord () { a = 0; b = 0; }
    wCoord (long A, long B) { a = A; b = B; }
};

class wPointl : public wCoord
{
public:
    wPointl () { }
    wPointl (long X, long Y,
              wWinCoordUnit type = DevUnits);

    long xSet(long X, wWinCoordUnit type = DevUnits);
    long ySet(long Y, wWinCoordUnit type = DevUnits);
    long xSet(long X, wPressSpace *ps);
    long ySet(long Y, wPressSpace *ps);

    long& x() { return a; }
    long& y() { return b; }

    wPointl operator+(wPointl p2);
    wPointl& operator+=(wPointl p2);
    wPointl operator-(wPointl p2);
    wPointl& operator-=(wPointl p2);
    int operator==(wPointl p2);
    int operator!=(wPointl p2);
};

class wDimension : public wCoord
{
public:
    wDimension () { }
    wDimension (long X, long Y, wWinCoordUnit type =
DevUnits);

    long SetWidth(long X, wWinCoordUnit type = DevUnits);
    long SetHeight(long Y, wWinCoordUnit type = DevUnits);
    long& Width() { return a; }
    long& Height() { return b; }

    wDimension operator+(wDimension d2);
    wDimension& operator+=(wDimension d2);
    wDimension operator-(wDimension d2);
    wDimension& operator-=(wDimension d2);
    int operator==(wDimension d2);
    int operator!=(wDimension d2);
};

```

```

};

class wRectl
{
    private:
        long xL, yB, xR, yT;

    public:
        wRectl() { xL = 0L; yB = 0L; xR = 0L; yT = 0L; }
        wRectl(long XL, long YB, long XR, long YT,
wWinCoordUnit type = DevUnits
);
        wRectl(wPointl o, wDimension s);

        long SetLeft(long XL, wWinCoordUnit type = DevUnits);
        long SetBottom(long YB,
                      wWinCoordUnit type = DevUnits);
        long SetRight(long XR,
                      wWinCoordUnit type = DevUnits);
        long SetTop(long YT, wWinCoordUnit type = DevUnits);

        long& Left()           { return xL; }
        long& Bottom()         { return yB; }
        long& Right()          { return xR; }
        long& Top()             { return yT; }

        wPointl Position()     { return wPointl(xL, yB); }
        operator wPointl()      { return Position(); }
        wDimension Size()
        { return wDimension(xR - xL, yT - yB); }
        operator wDimension()  { return Size(); }

        bool PtInRectangle(wPointl p);
        wRectl Stretch(long amt);
        bool IsEmpty();
        void Normalize();

        int operator==(wRectl &r2);
        int operator!=(wRectl &r2);
        wRectl& operator+=(wPointl &p);
        wRectl& operator-=(wPointl &p);
        wRectl& operator+=(wDimension &d);
        wRectl& operator-=(wDimension &d);
        wRectl& operator|(wRectl& r2);
};

}

```

The `wCoord` class forms the base class of the coordinate classes. Each of these classes is based on a pair of two long integer values. These data types and layout are important for obvious performance reasons. The binary representations of these objects must look like the equivalent API data structures if, for example,

we are to pass the address of a wPointl object as an argument in an API call that takes a POINTL structure.

Although they hold the same type of data, the wPointl and wDimension classes differ in their semantic meaning. The wPointl class is used to define points in a coordinate space and is generally used to indicate the position of windows or drawing tools. The wDimension class is used to define size of windows or graphics objects. In this case, the "x" component defines the width, and the "y" component defines the height.

wRectl objects consist of a pair of points to define a rectangle. One point specifies the lower-left corner, and one point specifies the upper-right corner. Thus, all rectangles have a position and size. The type of rectangles used by ObjectPM (and most of PM) are called inclusive-exclusive, meaning that the top-most row and right-most column of pels are not included in the rectangle.

The classes just described are used for specifying the position and size of graphics objects in general. The types of coordinates used by these objects are up to the application. For example, if these objects are being used along with some drawing tools to render a drawing, then the coordinates used will be world coordinates (such as twips or millimeters). If you were using them to position and size a window, then you would use window coordinates for the position, and device coordinates for the size. So, these classes provide a number of overloaded operator and cast methods whose purpose is support for translation and scaling operations. For example:

A point can be translated (moved) to another position by adding a another point to it:

```
wPointl margin(20, 15);
wPointl pos = win->GetPosition();
pos += margin;
```

Likewise, a rectangle can also be translated by adding a wPoint:

```
wRectl r(0, 0, 40, 20);
r += wPointl(50, 10);
```

Or scaled by adding or subtracting a wDimension:

```
wRectl r(0, 0, 40, 20);
r += wDimension(10, 10);
```

The size and position of a wRectl can be determined using cast operators:

```
wRectl r = win->WindowRectangle();
wDimension d = (wDimension)r; /* window size */
wPointl p = (wPointl)r; /* window position */
```

The intersection and union of two rectangles can be taken using the overloaded &= and |= operators:

```
wRectl a(0, 0, 40, 20);
wRectl b(20, 10, 45, 15)
wRectl i = a;
```

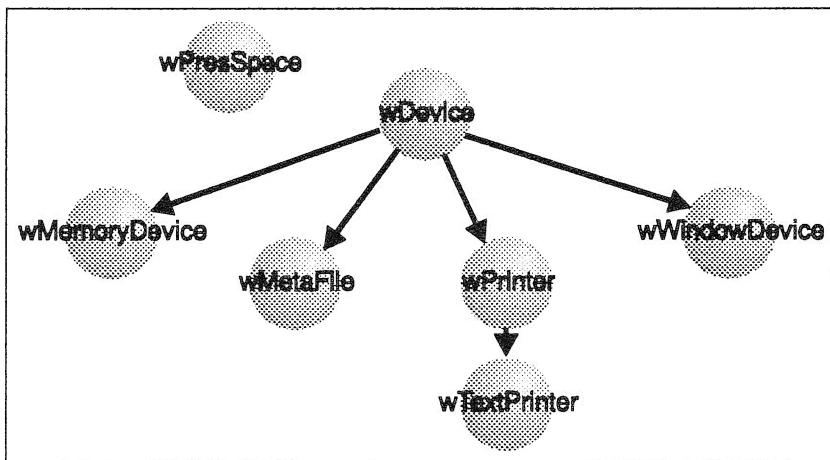


Figure 21.5 Presentation Spaces and Devices

```

a &= b;
wRectl u = a;
a |= b;
  
```

And objects can be compared:

```

wRectl r1 = win->WindowRectangle();
wRectl r2(10, 10, 35, 20);
if (r1 == r2)
{
    ....
}
  
```

Techniques using the overloaded and cast operators of C++ can go along way in eliminating clutter from code.

Presentation Spaces and Devices

As mentioned earlier, in the OS/2 Presentation Manager (as well as ObjectPM) there is a core object type called a presentation space. A presentation space object defines a virtual drawing surface, or medium, that is device independent. An output device, such as a window or printer can be attached to the presentation space in order to present graphics output to the user. When an application is drawing on a presentation space, it needs to care very little, or at all, about the actual output device used. This is how the Presentation Manager achieves its device independence.

In ObjectPM, the central object used for all graphics drawing is defined by the `wPresSpace` class. `wPresSpace` objects can be created for a particular window, or for any of the `wDevice` objects. Figure 21.5 illustrates the presentation space and device class hierarchy.

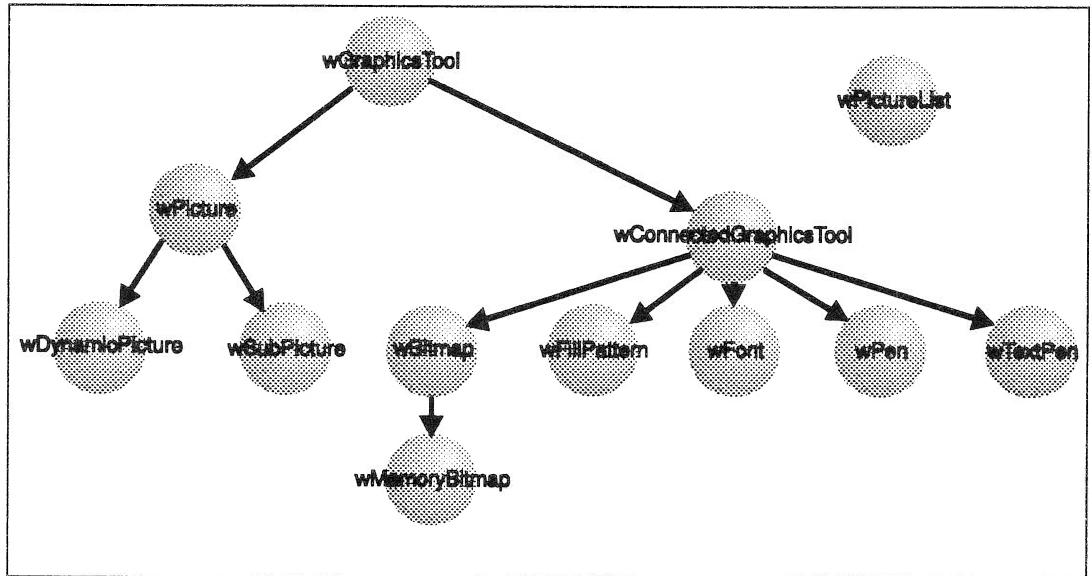


Figure 21.6. Graphics Tools Hierarchy

Any one of the above `wDevice` objects can be created and attached to a `wPresSpace` object. In general, you create the `wDevice` object first and then use the `wDevice` object to create the `wPresSpace` object. For windows, this is not necessary unless the application needs to change output devices for a particular `wPresSpace`. There is a constructor provided to create `wPresSpace` objects that are associated with `wWindow` objects.

`wMemoryDevice` objects allow the output to be written to memory. This `wDevice` type is used primarily by the `wMemoryBitmap` class. The `wMetaFile` device allows the output to be written to a standard OS/2 metafile. The `wPrinter` classes are used to define hard-copy output devices such as printers and plotters. And the `wWindowDevice` allows a window to be defined as an output device.

The `wPresSpace` objects provide applications with a virtual drawing surface. In addition, these objects also maintain attribute and resource information. At any given point in time, a `wPresSpace` object can have:

- A current `wPen`
- A current `wTextPen`
- A current `wFillPattern`
- A current `wFont`
- A current `wBitmap` (Image)
- A current output `wDevice`
- A current path or area
- A current clip path and clip region
- A current open `wPicture`
- A current color table

- A current graphics-cursor position
- A collection of transformations
- Current page units and page size

Note that for each type of object itemized in this list, there can only be one that is current. The `wPressSpace` class maintains a list called the `wGraphicsToolList`. Each time a drawing tool, such as a `wPen` or `wFillPattern` is created or connected, it is added to the `wGraphicsToolList`s of the appropriate `wPressSpace` object. Although a `wPressSpace` can have many `wPen` objects in its tool list, only one can be current. Setting tools as current is accomplished with the `PenDown` and `SetAsCurrent` methods.

DRAWING TOOLS

The drawing tools are objects that either produce graphics output (such a `wPen` and `wTextPen`), or modify the characteristics of a tool that produces output (such as `wFonts` and `wFillPatterns`). The drawing tool hierarchy is shown in figure 21.6.

The `wConnectedGraphicsTool` class forms the base class and provides a very important characteristic. These tools are “connectable”, which means one tool can be used on one or many `wPressSpaces`. Some objects, such as `wRegions`, must be created for, and remain associated with, one individual `wPressSpace`. The tools itemized above can be disconnected from one `wPressSpace` and then connected to another. It also means that a `wPressSpace` can be destroyed, and its tools will continue to exist. This is important because in many instances `wPressSpace` objects are temporary. Unless specifically overridden, the `ObjectPM` library creates a temporary `wPressSpace` object each time the `wWindow::Paint` method is called. This is done because the PM's presentation spaces are not a limitless resource. In their normal form, presentation spaces can consume a considerable amount of memory. Using this connected tool mechanism, though, it will not be necessary to reconstruct the tools each time you have to paint the window.

The `wPen` class is used to create objects that render line-oriented output. `wPens` can draw lines, arcs, circles, boxes, splines, and fillets. `wPens` can also sketch and fill paths and areas. A path is the only mechanism available in PM to draw lines that are wider than one pel. An area allows one or more polygons to be sketched by a `wPen` and then filled.

When a `wPen` object must fill a drawn object, such as a box or area, it uses the `wPressSpace`'s current `wFillPattern` object. The `wFillPattern` class is provided to define pattern bitmaps and colors used for filling.

`wTextPens` are pen objects used to draw text. The `wTextPen` class also has methods to change attributes such as mix modes, colors, baseline angle, and character direction. A method also exists to measure the width and height of any character string. `wTextPens` render their output using the `wPressSpace`'s current font. In `ObjectPM`, the `wFont` class is provided for creating

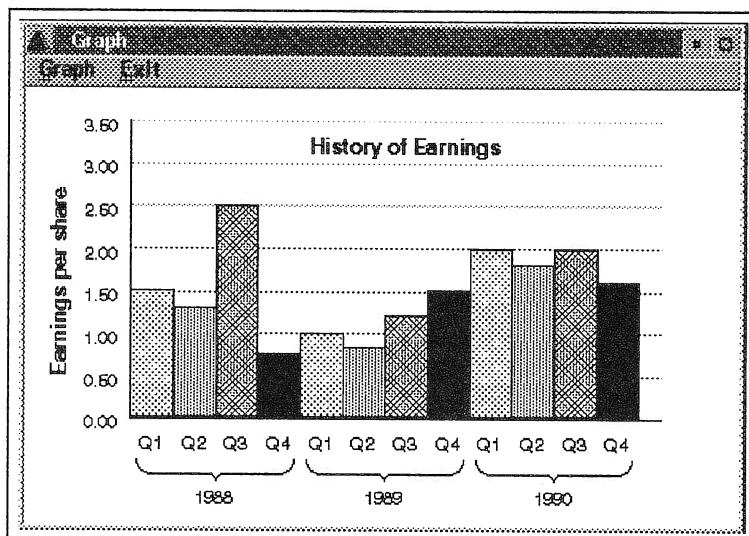


Figure 21.7 Graphics Example.

and manipulating PM fonts. This class allows both bitmap (image) and scalable outline (postscript) fonts to be created.

The wBitmap and wMemoryBitmap classes are provided to support the construction and painting of bitmapped images. The wBitmap class allows a bitmap to be constructed from one of the system's bitmaps, a bitmap from the application's resources, or from an initialized array in memory. The wMemoryBitmap class allows bitmaps to be created that are not predetermined. wMemoryBitmap objects contain their own wPresSpace objects, which allow tools such as wPens and wTextPens to draw on them. wMemoryBitmaps are also useful for capturing the contents of the screen or a particular window.

To summarize, RSI has provided ObjectPM with a collection of classes that encapsulate use of the PM graphics interface. These classes form a presentation space, a set of output devices, and a set of rendering tools. This architecture varies little from the underlying architecture of PM's graphics engine. Although this architecture is not clearly evident from its C language API calls, using OOP the important ideas are exposed and unimportant details are hidden.

To show these classes in action, the following example illustrates the code for the graphics example program shown in Figure 21.7.

```
// Application class
class GraphWindow : public wFrameWindow
{
    private:
        wIcon *icon;
        wMenu *menubar;
        wFont *scFonts[2];
        wFont **fonts;
        wWindowDevice *winDC;
```

```
short aRotation;
long xGraphPos, yGraphPos;
double scale;
wSubPicture *bracket;

void Paint();
void CreateFonts(wFont **);
void CreateBracket(wDimension siz);
void PrintGraph();
long MenuCommand(wCommandMsg);

public:
    GraphWindow();
    ~GraphWindow();

};

short bMainThread :: Start() // Program execution starts here
{
    GraphWindow w;           // Construct the top level frame window
    Exec();                  // Go to the message loop.

    return 0;
}

GraphWindow :: GraphWindow()
{
    CreateWindow(OpnStdFrame);           // Create the frame
    SetCaption("Graph");               // Set captions
    SetSwitchTitle("Graph Demo Program");
    SetIcon(icon = new wIcon(ResIcon, I_OBJPM)); // Set the
                                                // app's icon
    menubar = new wMenu(this, M_GRAPH);

    // Setup the initial position, scaling, and rotation...
    xGraphPos = 25;
    yGraphPos = 25;
    scale = 1.0;
    aRotation = 0;

    // Create a device representing the client window
    winDC = new wWindowDevice(this);

    // Create a 'normal' presentation space that supports
    // attaching different output devices. This PS
    // will use Low Metric units.
    WindowPS() =
        new wPresSpace(winDC,
                      PuLoMetric,
                      PcReAssociate + PcRetain);

    CreateFonts(scFonts); // build font array
```

```

        fonts = scFonts;

        CreateBracket(wDimension(370, 50)); // create the bracket
                                         // 'subpicture'
        Show();
    }

// This method creates the two fonts necessary for the
// drawing. Since font searching takes a while, we will
// create the fonts and keep them around instead of creating
// them each time we need to paint.

void GraphWindow :: CreateFonts(wFont **f)
{
    wFontDefList *fonts = 0;

    // find the 'Helvetica' PostScript font
    fonts = WindowPS()->EnumFonts("Helvetica");
    if (fonts && fonts->Entries() > 0)
    {
        // create the font from it's font definition
        f[0] = new wFont((*fonts)[0], WindowPS());
        f[0]->ChangePointSize(8);
        delete fonts;

        // likewise create a 'Helvetica Bold' PostScript font
        fonts = WindowPS()->EnumFonts("Helvetica Bold");
        f[1] = new wFont((*fonts)[0], WindowPS());
        f[1]->ChangePointSize(10);
        delete fonts;
    }
    else
    {
        // Helvetica PostScript font has not been installed
        f[0] = new wFont(SysMono10, 0, WindowPS());
        f[1] = new wFont(SysMono12, 0, WindowPS());
    }
}

// This method is called to paint the graph onto the
// presentation space

void GraphWindow :: Paint()
{
    static ushort vals[] = { 150, 130, 250, 75, 100, 83, 120,
150,
                           200, 180, 200, 160 };
    char wbuf[40];
    int i;
}

```

```
// Create drawing tools to work with. Unlike fonts,
// pens and textpens are fast to create.
wPen pen(WindowPS());
wFillPattern fillPat(WindowPS());
fillPat.SetBackMix(BmOverPaint);
wTextPen textPen(WindowPS());

// Adjust the page orientation

WindowPS()->Erase();
WindowPS()->SetPageRotation(aRotation);
WindowPS()->SetPageScale(scale, scale, TransformAdd);
WindowPS()->SetPageOrigin(wPoint1(xGraphPos * 10,
yGraphPos * 10),
TransformAdd);

// Draw the axis of the graph in wide line (at page
// origin) We must use a path to do this since it
// is the only mechanism available to draw lines
// wider than 1 pel.

pen.MoveTo(0, 700);
pen.SetGeomLineWidth(3);
pen.BeginPath();
pen.LineTo(0, 0);
pen.LineTo(1250, 0);
pen.PaintPath();

// Draw the horizontal grid lines in pale grey
penSetColor(ClrPaleGray);
for (i = 1; i < 8; i++)
{
    pen.MoveTo(2, i * 100);
    pen.LineTo(1250, i * 100);
}

// Draw the colored bars using a wFillPattern
// object to
// to fill the interior of the boxes drawn with wPen
penSetColor(ClrBlack);
for (i = 0; i < 12; i++)
{
    switch (i % 4)
    {
        case 0:
            fillPat.SetColor(ClrDarkRed);
            fillPat.SetPattern(PatDense8);
            break;
        case 1:
            fillPat.SetColor(ClrDarkBlue);
```

```

        fillPat.SetPattern(PatDense7);
        break;
    case 2:
        fillPatSetColor(ClrDarkGreen);
        fillPat.SetPattern(PatDense6);
        break;
    case 3:
        fillPatSetColor(ClrRed);
        fillPat.SetPattern(PatSolid);
        break;
    }
    pen.Box(
        wRect1(wPoint1(100 * i, 2),
               wDimension(101, vals[i] * 2)),
        DoOutlineFill);
}

// Now using an 8 point outline font,
// draw the horizontal units
textPen.PenDown(fonts[0]); // PenDown causes the wTextPen
                           // and the wFont
                           // to be the current text pen
                           // and font

long yBaseline =
    -((long)((float)textPen.MaxCharHeight() * 1.5));
for (i = 0; i < 12; i++)
{
    textPen.Printf(
        (100 * i) + 23,
        yBaseline,
        "Q%d",
        (i % 4) + 1);

// draw the brackets and years
for (i = 0; i < 3; i++)
{
    .
    char year[5];
    wPoint1 p;

    p.x() = (i * 400) + 15;
    p.y() = -((textPen.MaxCharHeight() * 2) + 50);
    bracket->Draw(p, TransformAdd);

    sprintf(year, "%d", 1988 + i);
    textPen.Display(
        p.x() +
        (185 - (textPen.TextSize(year).xWidth())) /
    2),
    p.y() - 50, year);
}

```

```

// Draw the y-axis units
float o = 0.0;
for (i = 0; i < 8; i++, o += .5)
{
    char nBuf[8];

    sprintf(nBuf, "%.2f", o);
    textPen.Display(
        -(30 + textPen.TextSize(nBuf).xWidth()), // x-pos
        (i*100)-(textPen.MaxCharHeight()/2), // y-pos
        nBuf); // text
}

// draw the y-axis title
textPen.PenDown(fonts[1]);
textPen.SetBaselineAngle(wPoint1(0, 10));
textPen.Display(wPoint1(-150, 100), "Earnings per share");
textPen.SetBaselineAngle(wPoint1(1, 0));

// draw the graph's title
textPen.Display(425, 625, "History of Earnings");
WindowPS() -> SetPageOrigin(wPoint1(0, 0));
}

```

Although this program uses a rather brute force method for drawing a bar graph, it demonstrates how the drawing tools and presentation spaces are used. Notice that in the constructor for wGraphWindow a presentation space is created for this application and associated with a wWindowDevice object. In the constructor call of the wPresSpace, the "PcReAssociate" flag is used to build a PS that can connect to different output devices. Initially the PS will use a window as its output device, but if we want to print the graph example, we could connect the PS to a printer device. The code for the PrintGraph method below illustrates how this might be done:

```

void GraphWindow :: PrintGraph()
{
    wFont *prFonts[2];

    HourGlass();

    // using a wPrinterInfo object, get the setup information
    // for the default printer object on the WPS desktop
    wPrinterInfo pi;
    wPrinterSetup *prSetup = pi.GetDefaultPrinter();

    // create the printer device object using the setup and
    // connect it to the presentation space.
    wPrinter printer(prSetup);
    WindowPS() -> AssociateDevice(&printer);
}

```

```

    printer.StartPrintJob("Hardcopy Graph");

    // re-create the fonts for the printer
    CreateFonts(prFonts);
    fonts = prFonts;

    // invoke paint to draw the graph on the PS
    Paint();

    // end the print job which causes the drawing to be spooled
    // to the printer.
    printer.EndPrintJob();

    // clean up
    delete prFonts[0];
    delete prFonts[1];
    fonts = scFonts;
    delete prSetup;

    // re-connect the window to the PS
    WindowPS()->AssociateDevice(winDC);
    HourGlass();
}

```

The last thing to point out about this program is the technique used to draw the horizontal brackets (braces). This is done with a feature of PM known as "Retained Graphics," which allows the "graphics orders" or operations of a drawing to be stored in block memory as the drawing is being constructed and then "played back" to render the drawing so that we can see it. Retained graphics provides advanced features such a hit testing and bounds testing of graphic objects as well as the ability to scale, position, rotate, and edit these objects.

The retained graphics features are supported by ObjectPM in the wPicture, wDynamicPicture, wSubPicture, and wPictureList classes. The sample program makes use of the wSubPicture class to "record" the shape of a curly brace so that it can be used repeatedly by the paint routine. The code in the CreateBracket method below shows how a subpicture is created:

```

void GraphWindow :: CreateBracket(wDimension siz)
{
    long basHeight, tailHeight, basArc, tailArc;

    // create the subpicture which opens a PM "segment
    // bracket". In this mode, we will record all graphics
    // operations instead of drawing them
    bracket = new wSubPicture(WindowPS());
    wPen p(WindowPS());

    basHeight = siz.yHeight() / 3;
    tailHeight = siz.yHeight() - basHeight;

```

```

siz.xWidth() /= 2;
basArc = (long)((float)basHeight * 0.7) + 0.5;
tailArc = (long)((float)tailHeight * 0.7) + 0.5;

// set the coordinate space origin to the "point" of the
// curly brace using a "translate" model transformation
WindowPS()->SetModelTransl(wPointl(siz.xWidth(), 0),
TransformAdd);
// run through a loop twice. First to draw the right
// side of the brace, and then secondly, draw left
// side using a reflection transform

for (short i = 0; i < 2; i++)
{
    p.MoveTo(0,0);
    p.Arc(wPointl(basHeight - basArc, basArc),
           wPointl(basHeight, basHeight));
    p.LineTo(siz.xWidth() - tailHeight, basHeight);
    p.Arc(wPointl((siz.xWidth()-tailHeight)+tailArc,
                  basHeight + (tailHeight - tailArc)),
           (wPointl&)siz);

    if (!i) // set reflection transformation on the
           // x-axis
        WindowPS()->SetModelScale(-1, 1, wPointl(0, 0),
                                     TransformPreempt);
}
// we're done, stop recording graphics orders
bracket->Close();
}

```

Once the subpicture is created, it can be rendered onto a presentation space at any time. The next code fragment, which is taken from the GraphWindow::Paint method above, shows how the bracket subpicture is used:

```

// draw the brackets and years
for (i = 0; i < 3; i++)
{
    char year[5];
    wPointl p;

    p.x() = (i * 400) + 15;
    p.y() = -(textPen.MaxCharHeight() * 2) + 50;

    // draw the bracket at the specified location
    bracket->Draw(p, TransformAdd);

    ...
}

```

There are two basic types of wPicture objects: "chained," and "called." The wPicture and wDynamicPicture classes implement the first type. These wPicture objects are collected in a list, which allows an entire drawing to be drawn by simply drawing the list. The wPictureList class is a derived linked-list class that manages the list of wPicture objects. wPictures are kept in a list for the purpose of setting the drawing order. The first wPicture in the list is drawn first, followed by the next, until the last wPicture object in the list is drawn. Thus arranging the list, changes the 'Z' order of the pictures, allowing you to control which wPictures are drawn on top other wPictures.

wDynamicPictures are a special kind of picture. When they are drawn, the Presentation Manager uses the XOR raster mode. This allows the picture to be drawn and then removed very quickly. Thus, wDynamicPictures are used for animation and mouse dragging.

The second type of wPicture object is implemented in the wSubPicture class. This is the class we used in our example program. These type of pictures are used to draw generic pictures that are intended to be reused. For example, if you were going to draw a house, you could use a wSubPicture to draw a window. Then, each time you need to draw another house, just draw the wSubPicture giving it the position and size you want. These types of pictures have no ordering on their own, and they can be called when drawing wPictures and wDynamicPictures as many times as needed.

This completes our case study of RSI ObjectPM. To summarize this chapter, ObjectPM is an extensive and useful C++ class library, providing a comprehensive encapsulation of OS/2 and Presentation Manager. It was created in one step, but was developed through an iterative cycle of implementation and learning. This is true of most evolving object-oriented designs, where classes are created, combined, and split as new useful relationships and commonalities are discovered. Through the object-oriented strengths of code reuse and inheritance, Raleigh Systems has crafted a powerful and extensible library of classes to aid C++ programmers in developing applications for OS/2.

THE AUTHOR

John Pompei is Vice President of Research and Development and co-founder of Raleigh Systems. He is a self taught programmer who has been actively programming for 15 years. He has five years of experience in OOP, primarily with C++ but also with SOM. Recently, he has begun to work with distributed object technology. John has graciously provided this chapter to illustrate how C++ and OOP can be used to develop powerful class libraries.

22

A Practical Methodology for OOUI Design

INTRODUCTION

Today, more than ever, the user interface is being recognized as crucial to the success of interactive software. Concurrent with the increasing interest in object-oriented programming (OOP) techniques has been a rise in interest in object-oriented user interfaces (OOUIs) and design methodologies. Many aspects of OOP design are equally applicable in the design of an OOUI. Activities such as identification of object classes, states, responsibilities, and relationships might be done using any of the popular methodologies for Object-Oriented Analysis and Design (OOA/OOD), such as Shlaer-Mellor, Booch, Rumbaugh, or others. The major difference is that in OOUI design the focus is on objects *perceived by users*. That is, OOUI design is concerned with objects that represent the user's information, and relationships that support the user's tasks. Contrary to popular belief, this does not include UI toolkit objects, controls, and widgets, such as entry fields, menus, and buttons. These are elements of the programmer's environment. While they are used in *presentations* of the user's information, they are not the objects with which typical users are concerned. User objects include memos, reports, annotations, calendars, diaries, movies, project plans, dates, salary amounts, and the like.

The primary role of the UI designer is to define the user object model and to provide a mapping into appropriate implementations using facilities of targeted systems. Definition of the UI object model is crucial to the creation of an intuitive, easy to use, and productive interface. This model establishes the objects,

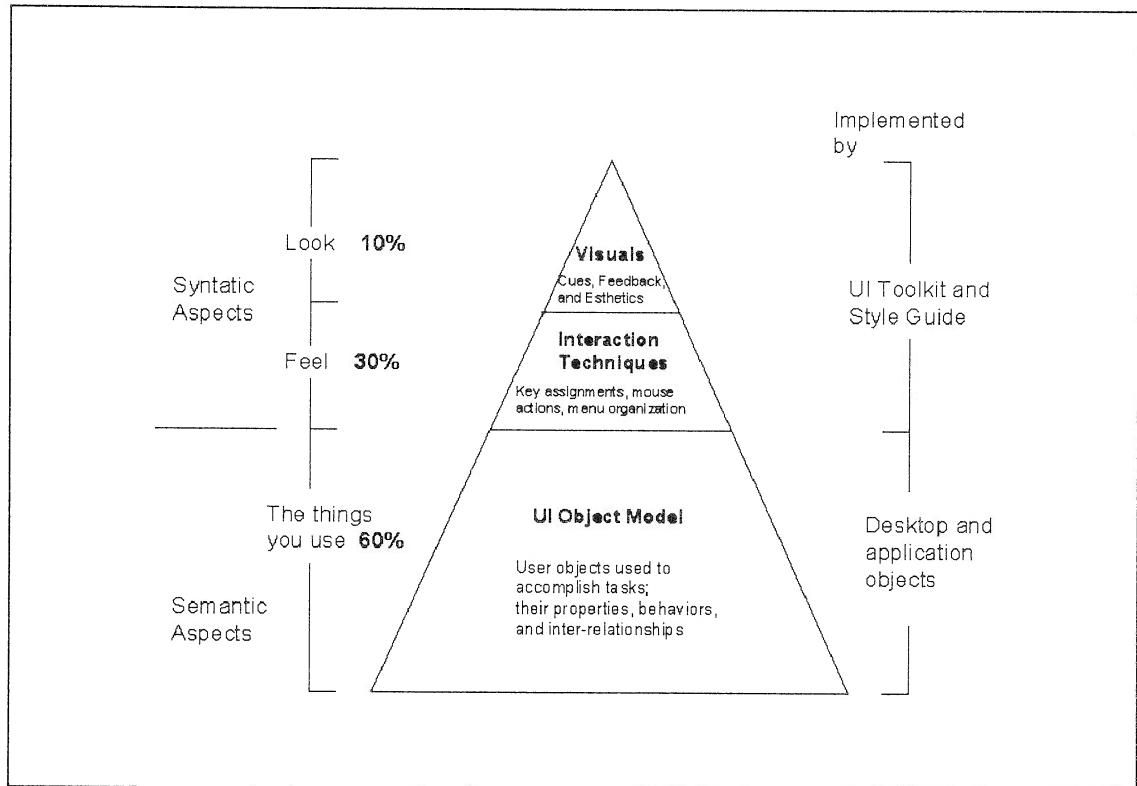


Figure 22.1. Look and Feel of User Interfaces

behaviors, and relationships users must know in order to accomplish their tasks. Given a particular task to accomplish, a user must figure out how to use the objects, behaviors, and relationships provided through the UI to accomplish the task. Because the UI object model is so important to the achievement of users' goals it is usually credited with being the most important single aspect contributing to the overall usability of a product. The relative importance of the UI object model as compared with the "look and feel" aspects typically thought of when the term *user interface* is mentioned, is shown in Figure 22.1.

The "look and feel" aspects of a user interface include key assignments and sequences, mouse techniques, visual cues and feedback indications, and visual esthetics. In a linguistic sense, these aspects can be considered to be the *lexical* and *syntactic* elements of the interface. The object model constitutes the *semantic* aspects. The "look and feel" aspects can contribute significantly to interaction efficiency and error rates. Consistent use of good techniques throughout an interface will tend to maximize efficiency and minimize user errors. The semantic aspects reflected in the user object model contribute to learning time and task mastery. An interface with simple, easy to remember "look and feel" techniques

can still be a nightmare to learn if the user object model is not well-defined, intuitive or obvious, and suited to the tasks the user expects to perform.

While the popular OOA/OOD methodologies guide a designer in identifying object classes, behaviors, and relationships, difficulties often arise in actual implementations. These difficulties are sometimes encountered in the mapping from abstractions to specific methods of presentation and interaction provided by an implementation environment, such as the Presentation Manager and Workplace Shell in IBM's OS/2. This mapping is depicted in Figure 22.1.

The intent of this chapter is to introduce a practical methodology that designers should find helpful in mapping a user interface object model into views supporting the user's tasks, using menu bars, pop-up menus, and drag/drop techniques.

OBJECT MODELS AND VIEWS

Most of the current popular user interfaces implement some degree of data-view separation in which the user's information is presented in one or more views of user objects. Each view presents a particular subset of information, or information attributes, and provides mechanisms allowing the user to interact with information to accomplish specific tasks. The relationship between user objects and views is shown in Figure 22.2.

In order to support the tasks required in the use of a typical object such as a calendar, the designer must often provide several different types of views. For a calendar, views might be provided for a year, a month, and a day at a time, as well as views to support scheduling of appointments and changing calendar attributes. The objects, object properties, behaviors, and inter-relationships defined in the object model support all of the desired user-tasks. Likewise, the collective set of views provide representations and interaction techniques for the objects. Each view typically supports some subset of the overall user-task requirement, although some tasks might be supported in more than view in order to provide a different perspective, or address different needs of specific types of users.

In order to accommodate a variety of situations and users having various degrees of computer experience, the designer typically must provide multiple interaction techniques for certain tasks within each view. It is crucial to the usefulness of the interface that this combination of views and interaction techniques support all of the user's task requirements, enabling the novice user to learn how to use the object while simultaneously providing shortcuts and high efficiency for more experienced users. In addition, the techniques should be complimentary. Techniques used by the novice should lead naturally to a mastery of an object's functionality as well as to a higher level of interaction efficiency. Choices found in menu bars, and pull-down menus should have useful and intuitive relationships with those found in pop-up menus. Likewise, actions performed as the result of a drag/drop should compliment those in menus. In general, interaction mechanisms should be synergistic, providing the user with

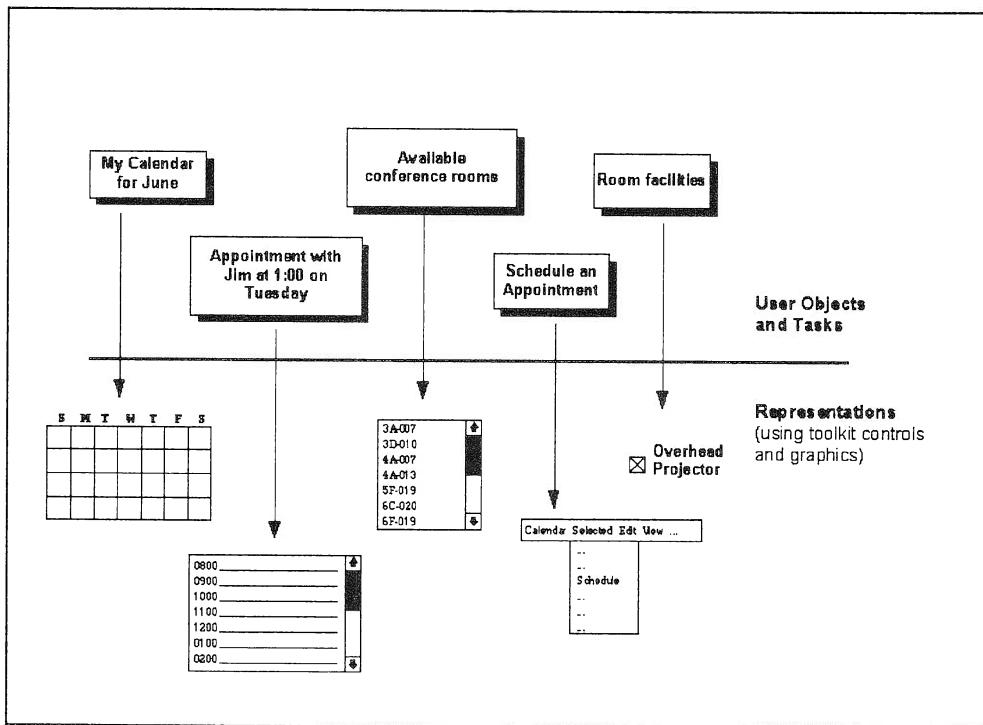


Figure 22.2. Relationships Between Tasks and Views

alternatives to meet the needs of varying situations, and reinforcing the underlying object relationships and roles.

IBM's Common User Access (CUA) defines an object-oriented user interface in which the primary interaction techniques are based on menu bars, pop-up menus, and drag/drop. This interface is exemplified by the Workplace Shell in OS/2.

The CUA guidelines are intended for use by application developers. They address a broad range of user interface design considerations including window and dialog, layout, guidelines for use of various UI controls, keyboard mappings, mouse button usage, and terminology. The guidelines are available in *Object-Oriented Interface Design, IBM Common User Access Guidelines*.

THE OVTT METHODOLOGY

The methodology introduced in this chapter is intended to help designers define relationships between user *objects*, the *views* in which objects are presented, the set of user *tasks* to be supported in each view, and a standard set of interaction *techniques* for performing the tasks as in Figure 22.3. To help maintain a focus on all these four key elements of the methodology, objects, views, tasks and tech-

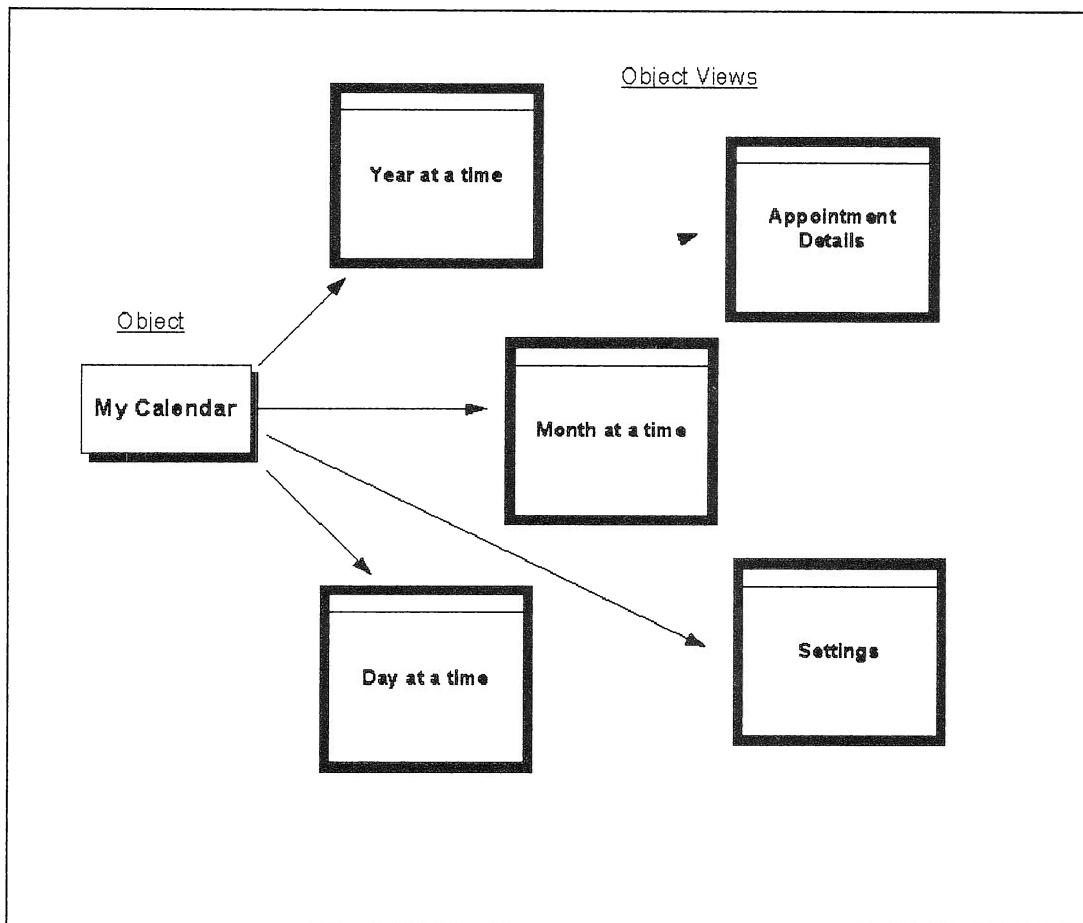


Figure 22.3. Relationships Between Objects and Views

niques, is referred to as OVTT. While this title may imply a specific sequence or ordering of steps it should be understood that the methodology is actually iterative. Obviously, a designer cannot begin without some notion of what user tasks must be supported. This much is necessary to begin identification of the abstract object classes. However, as mentioned previously these steps are addressed by a thorough object analysis phase. Therefore, it might be viewed that the OVTT methodology picks up in the middle where object analysis leaves off and before coding begins, as in Figure 22.4.

The user tasks to be supported must be considered in identification of views, and the form of presentation in a view must be considered to enable specific interaction techniques. However, it is often desirable to allow users to modify properties of views. That is, the definition of views often leads to identification of new tasks. Likewise, the mapping of interaction techniques provided in a view to tasks supported by the view can lead to changes in presentation form for the view. Therefore, a precise ordering of steps in this methodology is not practical,

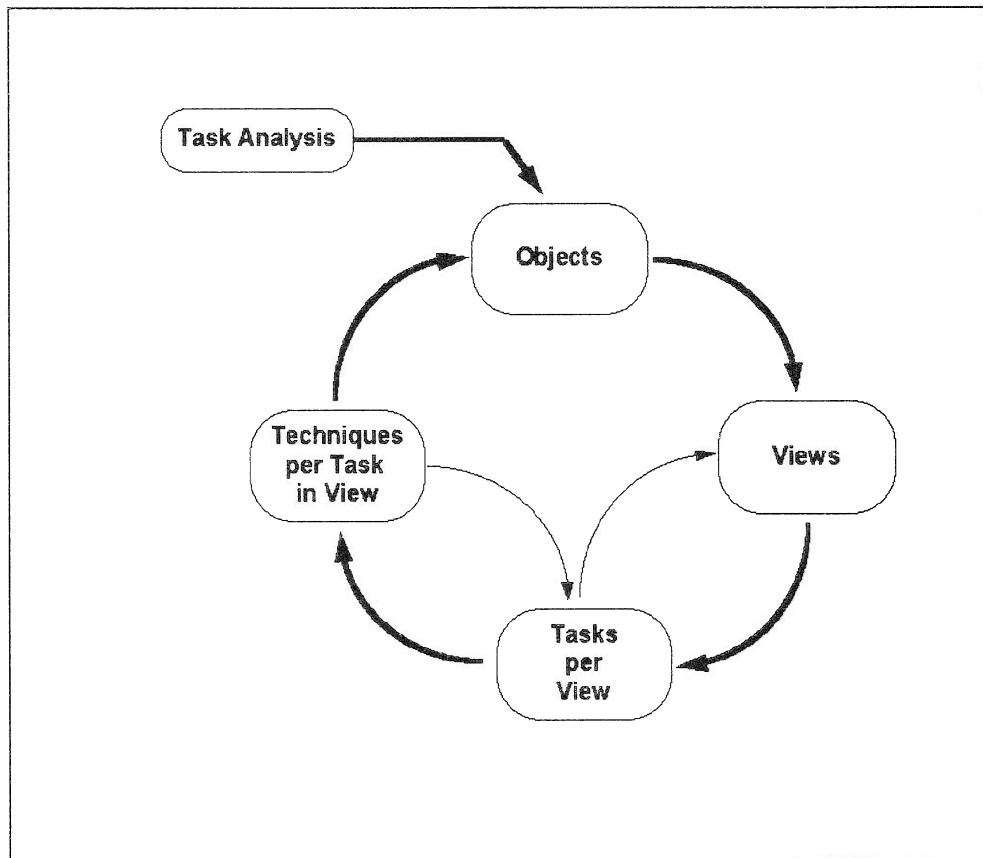


Figure 22.4. The OOVVT Methodology

or even desirable. In general, the process begins with identification of user objects, based on at least a rough idea of the tasks to be performed, with subsequent partitioning into views, each of which support a subset of tasks. Tasks are mapped to the standard interaction techniques within each view. Iteration occurs whenever new tasks are identified or new techniques become available. The data-view separation is constantly assessed to insure that all required tasks are supported in at least one view, and that multiple technique alternatives are provided to accommodate various task situations and levels of user experience.

The remainder of this chapter is devoted to an example using the OVTT methodology to begin the design of a simple Calendar object. A simple object model is assumed and tasks are identified based on typical real-world calendar-related tasks. Multiple views of the calendar are identified along with the tasks supported in each view. Finally, within each view the tasks are mapped to interaction techniques using menu bars, pop-up menus, and drag/drop. The complete design process resulting in a full-function calendar requires multiple iterations and is beyond the scope of this chapter. Points of omission, such as lack of definitions for Help and Settings views, should be recognized. However, the example

should prove an adequate starting point enabling you to appreciate the intent of the OVTT approach and be able to apply it in your design activities.

A SIMPLE CALENDAR EXAMPLE

The OVTT methodology is an approach for mapping the results of a previously completed object analysis into implementation. Therefore, the example will first identify the results of some analysis, presumed to have been done previously.

The design of any product intended to be used by people should begin with an analysis of users' needs and skills. This phase is often called Task Analysis. There are many approaches to gathering user task requirements. For purposes of this example we will assume a simple task analysis has been done. It will identify the key tasks that users must be able to accomplish using the Calendar, and note user skill levels or special needs.

We also presume a simple level of object analysis. The results of this analysis are documented in an object model for the calendar. An object model identifies object classes, properties, behaviors, and inter-relationships. Because we are concerned with design of a user interface, the object model is concerned only with objects that represent the user's information and tasks. Objects from the computer domain, such as buttons and menus, are not considered at this level of analysis.

Following the descriptions of the presumed task requirements and object model, the OVTT methodology is applied to generate the definitions of views, menus, and drag/drop techniques that accomplish the user's task requirements.

User Tasks and Skill Requirements

For purposes of this simple example a pragmatic approach provides an adequate starting point. We can simply list from experience some of the activities for which calendars are known to be used.

- Determining what day of the week a particular date falls on. "What day is my birthday next year?"
- Determining when recurring events will occur. "What day is Christmas next year."
- Noting events that will occur on a particular date. "The concert will be Sunday at 2:00 PM."
- Calculating time intervals. "The video taped lecture will start at 4:15 and is 1:47 long. What time will it be over?"
- Noting characteristics of an event. "The subject of the meeting is Sales Trends in 1994. It will be in Conference Room 121. Dave will present. Jill and Steve will also attend. Remind me 5 minutes in advance."

At this point, it is probably best to err on the side of too little definition rather than too much. Since the entire process is iterative, it is always possible, even

desirable, to revise prior results as new discoveries are made in subsequent phases.

In terms of user skills, we will assume that this calendar must be usable by computer novices with no prior tutorials or training on the calendar itself. We will assume the users understand how to use a mouse, windows, and menus. While the calendar must be usable by novices, it must also be efficient and productive for users as they become more skilled in the interface. In other words, it must provide shortcuts and personalization capabilities to fit individual user's needs and preferences.

The Calendar Object Model

We begin with a very abstract definition of a calendar. By being very general, abstract definitions often prod us to recognize potential organizations and relationships that otherwise might not be obvious. If we were to restrict our notion of a calendar to a table of days and dates hanging on a wall or sitting on a desk we might miss an entire domain of possible uses and relationships to other tasks. Such an object model can best be generated quickly through brainstorming by a small group of individuals who are skilled in developing abstractions and who understand the subject domain.

Objects

The following definitions have been generated from such an object analysis. They provide an abstract notion of a calendar and related aspects:

A **calendar** is an ordered collection of points in time. At this degree of abstraction we don't have to be concerned with precisely which units of time the calendar uses. It suffices to note that points in time might be identified using any of a variety of units, such as days or hours, and that the units are typically relative to terrestrial, lunar, or some other basis. The Gregorian calendar and several standards based on Chinese dynasties are examples. What is important to note is the potential to vary these aspects of the calendar, in other words, to consider making them user-settable properties of the calendar object.

An **event** is some activity associated with a point in time. The activity could be anything that a user might find useful, such as a reminder for a meeting, a due-date for completion of a task, or the starting time for some automated procedure. By recognizing this generality we can make explicit support decisions for a variety of calendar triggered mechanisms, such as audible alarms, pop-up windows, task scheduling, and so forth. Also, the association between an event and its point in time might be relative. In other words, the user might be allowed to assign some events to particular points in time, and to move them later. Other types of events might be inextricably bound to points in time. Historical events are examples. Again,

we have identified a potential property that users might be allowed to manipulate, depending on our task needs.

An **agenda** is a collection of ordered events. It has at least a beginning event and an ending event. It is important to note that an agenda has a duration. Because the events are ordered they are assigned points in time that are typically ascending, but through this abstract definition process we are prompted to think about the usefulness of an agenda that is organized in time-descending order, an agenda that runs “backward”. Such a property of an agenda could be useful in animations and slide shows. Remember, we are using the term “agenda” in a very general sense to represent a sequence of events, not just in the traditional sense of a meeting agenda.

A **plan** is a collection of inter-related agendas. Traditional project plans are typically composed of multiple time-lines. Multiple agendas might be used to support project planning tasks. Also, an agenda might be used to control the animation of a single “actor” in an elaborate animated presentation. The collection of “actor scripts” (a plan) would constitute a “show”.

An **interval** is the duration between one event and another. An interval is therefore relative to the units in which time is measured. Imagine how useful it might be to have a Calendar object that calculated intervals between two points in time. The requestor might not even have to know what units the calendar or events were recorded in. For example, it might be possible to ask a Calendar “how many years after Columbus’ first voyage did Sir Walter Raleigh make his first landing in the New World?” If such events were recorded as objects, in a multimedia encyclopedia for instance, a user interface for calculating intervals using the Calendar could provide very fast and convenient date/time calculation capabilities.

By now it should be obvious that the domains in which such an abstract calendar object might be used are numerous. Although this is a very simple and incomplete object analysis, it should be evident that a level of object abstraction can lead to recognition of potential capabilities and relationships that otherwise might be overlooked. Capabilities and relationships that are not of immediate use can be filtered out during the iterative process when current user task needs are considered. However, this abstract definition step itself should prompt additional thinking about the adequacy of the user-task definitions as they were initially identified. The power of the iterative process is realized through continuing refinement and filtering at each phase of design and development.

Tasks

The task definitions are also abstract in the beginning. The names associated with tasks at this level of design are a short-hand for convenience. They may or

may not actually be used in the resulting user interface. Actions can be accomplished using a variety of techniques, such as drag/drop, that do not require terminology in the user interface.

A **find** action determines what day of the week a particular date falls on. It also provides a list of standard events, such as holidays, that is tailorable by the user.

A **create** action allows users to create new instances of standard calendar types, and they can create events, agendas, and plans. Users might also be allowed to create new types of calendars, but that is beyond the scope of this example.

A **change** action allows users to change the information associated with calendars, events, agendas, and plans.

A **delete** action allows users to discard a calendar, event, agenda, or plan.

An **open** action allows users to view and access the information in a particular calendar, event, agenda, or plan.

A **schedule** action associates an event with a point in time on a calendar. Agendas and plans are collections of events so they can be scheduled as well.

An **unschedule** action disassociates an event with a point in time but preserves the event and its information. The event might then be scheduled at a different point in time or on a different calendar later. Agendas and plans can also be unscheduled.

A **now** action returns to the current point in time, within some resolution that could be specified as a user preference.

Potential for compound actions often becomes evident during this phase of design. For example, a **move** action can be implemented using **unschedule** and **schedule**. Since convenience and efficiency for the user are goals, the interface should provide such a move capability for scheduled events, agendas, and plans.

At this point we have enough information to begin using the OVTT methodology to map the object model and task requirements into views, task groups, and techniques. Other sources of information will continue to cause change and iteration. For example, during menu design we will consult a user interface style guide for the platform on which we intend to deliver the calendar. For this example we will reference IBM's Common User Access (CUA) Guidelines, and refer to the Workplace Shell as an exemplary implementation.

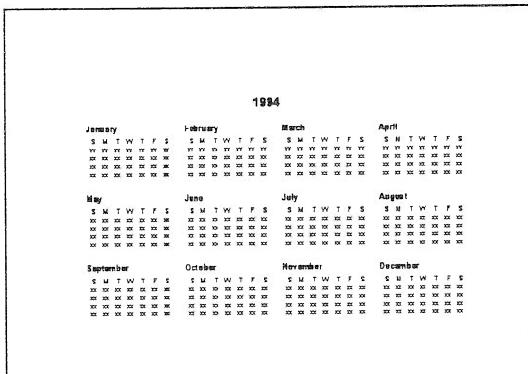


Figure 22.5. A Sample Calendar

Calendar Views

The role of a view is to present information and enable interactions that allow users to accomplish a set of tasks. For anything other than trivial objects it is typically not possible to accomplish this for all desired tasks using a single view. Thus, one of the designer's jobs is to identify a set of views that collectively support all of the user's task needs, while structuring the subsets of information and capabilities presented (the views) in a logical, intuitive, and efficient manner.

Referring back to our user description, we begin with a pragmatic starting point by determining what types of calendar "views" users are accustomed to seeing. For the business and home domains an initial list of views can be generated by visiting local business supply and greeting card stores. Typical calendar views include:

Yearly: a year-at-a-time, displaying twelve months, where each month is a table of days and dates organized by week.

Monthly: a month-at-a-time, displaying a single table of days and dates organized by week. Each day presents an overview of events associated with it.

Daily: a day-at-a-time, displaying a days-worth of points in time in which events can be scheduled. Includes an overview of events associated with each point. Figure 22.5 shows an example of a typical calendar style found in popular use.

Frequently, a view of one time period, such as a month, will display preceding and following periods of time. For example, the monthly view for June typically includes scaled-down monthly presentations of May and July. The scaled-down presentation of these adjacent periods provides context and continuity on both sides of the current point of focus, which is June, with-

out unduly distracting our attention. They often resemble the presentation of that particular period of time as it appears in the view of the *next larger* period of time. For example, the scaled-down presentation of May and July used in the monthly view of June are the same presentations as used for *all* the months in the Yearly view. Given careful attention to such details during object design and implementation, these views could provide very consistent appearance and behavior throughout the calendar. For example, a monthly-view *programmer's object* (not a user object) might be implemented. It could provide various size views of itself, or even be smoothly scalable, to support scaled-down presentations. It could also allow users to open a *daily* view for a particular date by double-clicking on the date. This montly-view object could then be used throughout the calendar implementation, where ever a month-at-a-time was to be displayed. This would prove very useful, for example, by allowing users to double-click on a date to open a daily view, *everywhere* the monthly presentation appeared. In other words, the dates would be "live" everywhere, even in the scaled-down presentations of the month.

Each view must enable user interactions to accomplish the user's tasks, such as finding out what day Christmas falls on in the year 2015. But, there are two additional sets of actions that must be addressed because the calendar is being implemented for use on a computer. First, the views themselves have properties, such as font and color, which should be modifiable by users. Second, there must be some way for users to navigate between sections of the calendar to "turn the pages", and to select different views.

The set of actions that allow users to personalize the properties of an object, such as the calendar, are presented in another type of view, called a Settings view in CUA. The Settings views provide a standardized way for users to access and change properties for all types of objects. CUA prescribes a particular type of view that uses a tabbed notebook. The notebook is a programming component that is provided in the Presentation Manger's Toolkit with IBM OS/2, and with systems using OSF's Motif toolkit. The notebook provides not only a familar appearance but standard interaction techniques for "turning the pages" as well.

Users must also be able to "turn the pages" of the calendar itself. That means each view of the calendar must provide mechanisms that allow users to advance to the next period of time, go backwards to the prior period, or to specify some specific period, such as "November 1945".

View Descriptions: Contents, Tasks, and Techniques

For this simple example we will define three views of the calendar, a year view, a month view, and a day view. Views for describing events, such as appointments, for changing properties, and to provide Help will also be necessary but aren't shown here. Each view will support a specific set of user tasks. To support the tasks each view must present task-related information and enable interaction techniques. First, we concentrate on the presentation of information to support the tasks. Note that some user tasks might be accomplished by *view-*

ing only, without provision for any other user interaction. For example, the task of determining what day a particular date falls on can be accomplished simply by viewing the calendar, once the desired page of the calendar is found.

While the additional views are necessary for a complete design, these three views of the calendar are sufficient for this introductory example.

For convenience we can use tables, such as those enabled in most word processors, to summarize the capabilities of each view. Each row of a table represents a specific user-task supported in that view. The columns of the table then represent the interaction techniques to be supported. The information content of each view can be described in prose or a simple list. It is often useful to do an initial graphic design at the same time. This allows factors such as information overload, complexity, and visual clutter to be managed in the iterative process. The general form of the description for each view is as follows:

Object Name—View Name

A paragraph of prose or a list that identifies the information content of the view; identifies what is selectable and modifiable by the user; also includes an overview of the purpose of the view in terms of the types of tasks that are supported.

User Task	Technique 1	Technique 2	Technique 3
Task 1	How the user performs the task using this technique	How the user performs the task using this technique	How the user performs the task using this technique
Task 2
Task n

Notes and additional clarifying information. Shortcut techniques for specific tasks can be included here.

Support for additional techniques can be accommodated in several ways. If a technique is to be used pervasively, it should be added as a column in the table. For example, if we decided to add a tool palette capability it would be represented in an additional column. Techniques that are related to something already in the table can be noted within an existing column. For example, accelerator keys for menu items can be noted along with the menu choices that support them. Shortcuts and other special techniques that are not pervasively available across many tasks can be described in the notes section following the table. For example, the double-click technique performs a default task that can be identified in the notes.

Decisions for support of particular interaction techniques should be based on an understanding of who the users will be, their skills and training, as well as user interface guidelines and standards for the targeted environments. As a

rule-of-thumb, all tasks should be supported by using the menu bar. It is visible and can be used by both mouse and keyboard oriented users. Visibility of the controls is a popular user interface design principle for which the menu bar provides a useful implementation.

Pop-up menus and drag/drop are hidden mechanisms. While they are more convenient, users must learn where pop-up menus are available and which actions occur as the result of drag/drop. Both of these techniques can make the interface more efficient. Drag/drop is also limited in terms of the actions that are possible, even when keyboard augmentation is used as a modifier. It is best to assign the most common and intuitive actions to drag/drop techniques.

Regardless of which interaction techniques are ultimately supported, you should try to develop an explicit criteria for the use of each technique and be consistent in your mapping of tasks to techniques. One of the benefits of the OVTT methodology is that it makes these decisions explicit and visible, in a tabular form that can be quickly scanned and analyzed.

The interaction techniques to be supported in this example are a menu bar, pop-up menus, and drag/drop. Several other interaction techniques, such as accelerator keys and tool palettes, could also be supported. Each additional interaction technique can be accommodated in the methodology by adding a column in the tables.

Calendar Views, Tasks, and Techniques

For completeness it is convenient to treat an object's icon as a view. Over time, as the graphics capabilities of systems improve, the icons of objects are likely to become more informative and interactive, thereby becoming views in a first-class sense. Currently, icons provide a thumb-nail sketch for an object, and they enable actions on the object in support of user-tasks. Therefore, an object's icon support should be designed concurrently with the rest of the object's views and interaction techniques.

Calendar—Icon

A simple image of a generally recognizable calendar. Several different icons may be provided to accommodate user preferences, or cultural and national differences. The Workplace Shell allows users to change the icon of an object by simply dragging a new icon to the current icon in the Settings view of the object.

The calendar's icon is intended to allow the user to manipulate the calendar as a whole. In general, users can use the icon to put the calendar where they want it, to copy it, delete it, print it, and to open various views of its contents (yearly, monthly, daily, settings, etc.).

User Task	Pop Up Menu	Drag/Drop
Open the yearly view	<i>Open > Yearly (1)</i>	n/a

Open monthly view	<i>Open > Monthly</i>	n/a
Open daily view	<i>Open > Daily</i>	n/a
Open settings view	<i>Open > Settings</i>	n/a
Open help view	<i>Help</i>	n/a
Create another calendar	<i>Create Another...</i>	Drag or n/a (2)
Copy Calendar	<i>Copy...</i>	Ctrl-Drag
Move Calendar	<i>Move...</i>	Drag or Shift-Drag
Create shadow of the calendar	<i>Create Shadow...</i>	Ctrl-Shft-Drag (4)
Delete Calendar	<i>Delete...</i>	Drag the icon to shredder
Print default calendar view	<i>Print...</i>	Drag icon to printer

Notes:

1. The appearance of a “>” indicates use of a cascade menu for the choices that follow.
2. Only if the Template setting has been set in the Settings view. Otherwise use the menu.
3. Only if the Template setting has *not* been set. Otherwise, use Shift+Drag to Move.
4. This is a standard CUA specified technique for creating a shadow.

As a short-cut, a double-click of the mouse Select button will be defined to open the yearly view. This technique might be defined to open the last used view instead, depending on user scenarios, usage patterns, and user preferences. Given an equally divided user preference, a user setting could be provided.

Calendar—Year View

The yearly view displays a monthly calendar for each of the twelve months. Each monthly calendar is arranged as dates in a table of days-of-the-week (Sunday through Saturday). December of the previous year and January of the following year are displayed to provide surrounding context. The dates of national holidays are highlighted. Today's date is uniquely highlighted in the monthly calendar and is spelled out at the top of the calendar. The

current year is displayed in a large push button surrounded by two additional push buttons that are "page turners" to the prior and next years, as in figure 22.6.

The user can select any date or the name of any month from January through December of the current year as well as the December calendar for the prior year and the January calendar for the next year.

The menu bar choices are as prescribed by the cua guidelines. The *Calendar* menu provides actions on the calendar as a whole. These are, in fact, the same actions as provided by the calendar's icon, but the icon is likely covered up by windows on the desktop so this menu provides convenient access to these common actions. The *Selected* menu provides actions on the selected item within the view, such as the currently selected date. The *Edit* menu provides standard actions such as Undo/Redo, Cut, Copy, and Paste. The *View* menu provides actions that change the view in the current window. The *Window* menu lists the various calendar windows that may be open simultaneously, and the *Help* menu provides standard CUA prescribed Help choices.

This view is intended primarily to support identifying days of the week associated with arbitrary dates and specific holidays. It provides techniques for accessing different years. It also provides access to views containing more detailed information, such as monthly and daily views.

User Task	Menu Bar	Pop Up Menu	Drag/Drop
All tasks supported for the icon	<i>Calendar</i> (choices are same as icon's pop-up menu)	From the background of the view: same as icon's pop-up menu)	(1)
Determine day of the week for scheduled event or holiday	<i>Edit...</i> <i>Find...</i> (2)	<i>Find...</i> from the year, name of month, date:	n/a
Display next (prior) year	<i>View</i> <i>next year</i> <i>prior year</i>	<i>next year/prior year</i> from the background of the view	n/a
Open Details for a month	<i>Selected</i> <i>Open</i> With the name of a month selected	<i>Open</i> from the month's name or anywhere in the month	n/a
Open details for a date	<i>Selected</i> <i>Open</i>	<i>Open</i> from the date	n/a

Print the current view of the calendar	<i>Calendar Print</i>	<i>Print from the background of the view</i>	Drag the year to the printer (1)
Print the view of a month	<i>Selected Print</i> With the name of a month selected	<i>Print from the month's name or anywhere in the month</i>	Drag the name of a month to a printer
Print the day view of a date	<i>Selected Print</i>	<i>Print from the date</i>	Drag a date to the printer

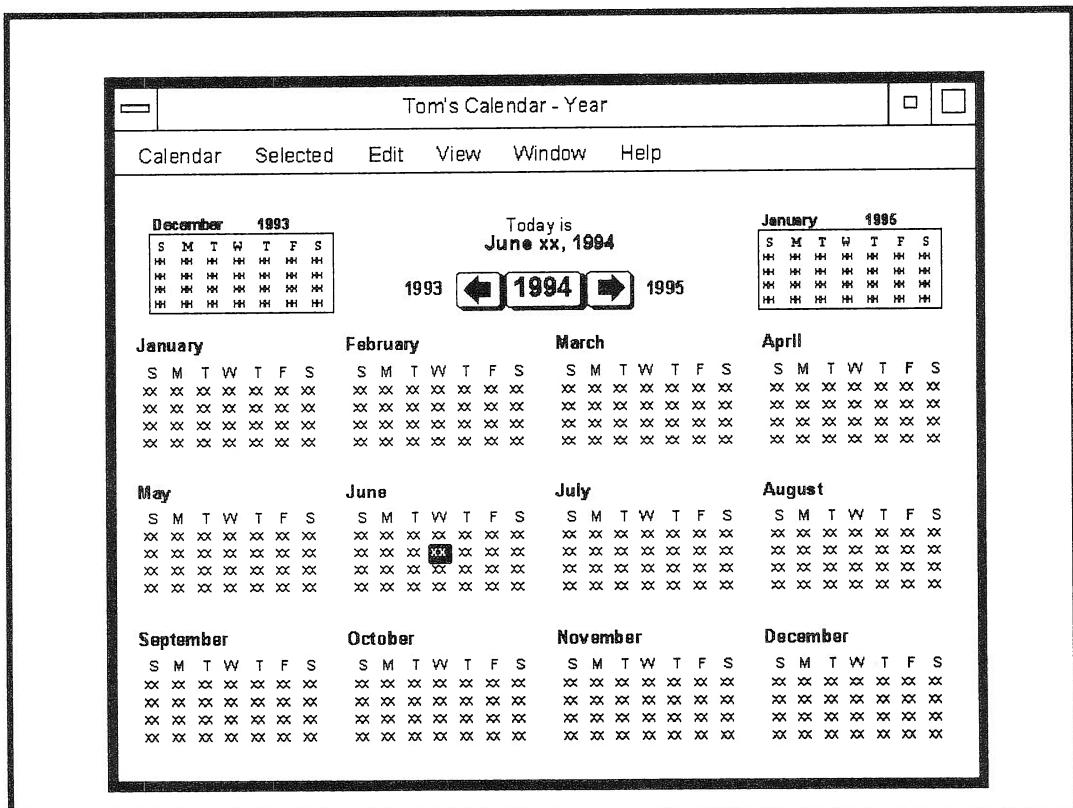


Figure 22.6. Calendar Year View

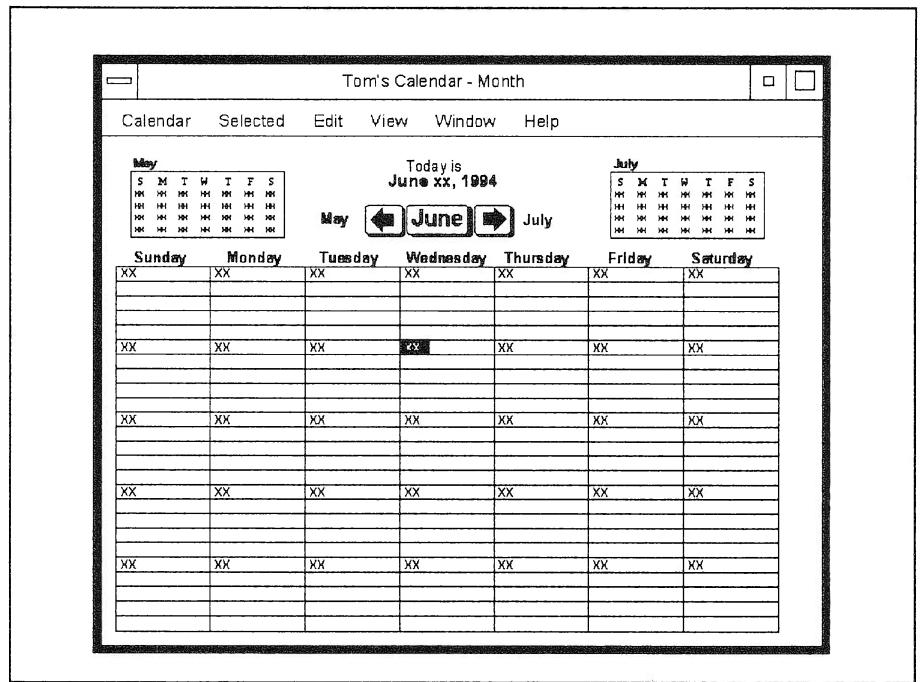


Figure 22.7. Calendar Month View

Notes:

1. Drag/drop actions for the calendar as a whole are best provided by including in the view an icon representing the calendar, for example a small icon in the window's title bar (not shown here).
2. The Find choice displays a window containing lists of scheduled events recorded in the calendar. Lists of national holidays, religious holidays, and personal holidays (birthdays and anniversaries) will be provided. These lists are editable by the user. The lists pertain to the year from which the window was opened but the year is also editable and selectable by the user. The result of doing a Find is changing the view in the current window to display the same level of view (year, month, or day) that includes the found event.

The user can also display the prior (next) year by clicking on the appropriate push buttons surrounding the year at the top of the calendar. Clicking on the year push button causes a window to appear in which the user can enter a specific year or pick one from a scrollable list.

Double-clicking on the name of a month opens a window containing the monthly view of that month. Double-clicking on a date causes a window to open containing the daily view for that date. Similarly, double-clicking on today's date at the top of the view causes the daily view for today to open in another window.

Calendar—Month View

The month view in Figure 22.7 displays a weekly calendar for each of the weeks in the month. Each week is arranged as days-of-the-week (Sunday through Saturday). Each day of the week provides an overview of events scheduled on that day. The prior month and next month are displayed to provide surrounding context. The dates of national holidays are highlighted and the name of the holiday is displayed in the overview for the day. Today's date is uniquely highlighted in the week and it is spelled out at the top of the calendar. The current month is displayed in a large push button surrounded by two additional push buttons that are "page turners" to the prior and next months.

The user can select any date from the current month as well as from the prior month or the following month. They can also select an event in the overview for the day. The menu bar choices are similar to those provided for the year view.

This view is intended primarily to provide an overview of scheduled events for the month. It provides techniques for accessing different months and it provides access to views containing more detailed information about the events, such as day views and event views (not shown in this example).

User Task	Menu Bar	Pop-up Menu	Drag/Drop
All of the tasks supported for the icon	<i>Calendar</i> (choices are the same as the icon's pop-up menu)	On the background of the view: (same as the icon's pop-up menu)	(1)
Display next (prior) month	<i>View</i> <i>Next month</i> <i>Prior Month</i>	<i>Next year / Prior month</i> from the background of the view	n/a
Open the details for a day	<i>Selected</i> <i>Open</i>	<i>Open</i> from the date	n/a
Open the details for an event	<i>Selected</i> <i>Open</i> with the event selected	<i>Open</i> from the event in the day overview	
Print the current view of the calendar	<i>Calendar</i> <i>Print</i>	<i>Print</i> from the background of the view	Drag the month to the printer (1)

Print a day	Select <i>Print</i>	<i>Print</i> from the date	Drag a date to the printer
Print an event	Select <i>Print</i> with an event selected	<i>Print</i> from the event	Drag an event to the printer
Move a scheduled event from one day to another	<i>Move</i> with an event selected	<i>Move</i> from the event	Drag the event to another day (2)
Copy an event to one or more other days	<i>Copy</i> with an event selected	<i>Copy</i> from the event	Ctrl-drag the event to another day (2)
Other tasks from the year view	(3)		

Notes:

- (1) Drag/drop actions for the calendar as a whole are best provided by including in the view an icon representing the calendar, for example a small icon in the window's title bar (not shown here).
- (2) The properties of the event, including scheduled time, remain the same when it is dropped on a different date. If this causes a time conflict, a window containing both events might be displayed to help the user resolve the conflict. If the scheduled events involve other people who's phone numbers are recorded as well, the calendar might support auto-dialing using those numbers to further assist the user.
- (3) Many of the other tasks from the year view are applicable here as well. For example, the Find choice should still be provided under *Edit* in the menu bar.

The user can also display the prior (next) month by clicking on the appropriate push buttons surrounding the month at the top of the view. Clicking on the month push button causes a window to appear in which the user can enter a specific month or pick one from a scrollable list.

Double-clicking on a date causes a window to open containing the day view for that date. Similarly, double-clicking on today's date at the top of the view causes the day view for today to open in another window. Double-clicking on an event causes a window to open containing a description of the event.

Calendar—Day View

The day view in Figure 22.8 displays schedulable time intervals for a day. The view is arranged as a list of times. The interval between the times is specified by the user in the properties of the calendar. Each time slot provides an overview of

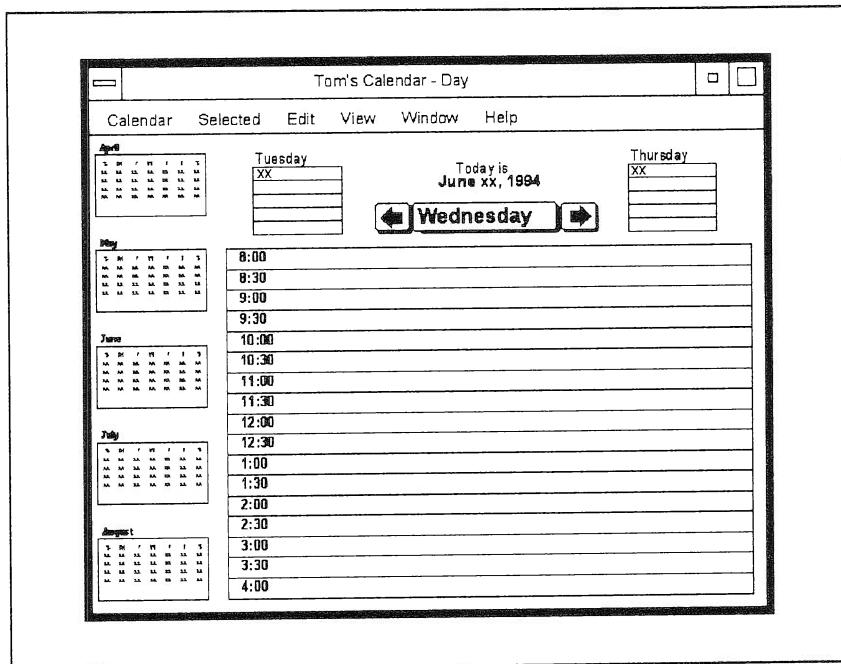


Figure 22.8. Calendar Day View.

events scheduled in that interval. This overview is more detail than provided for events in the month view, but is typically not the entire set of information, which can be displayed in the event view. The prior day and next day, as well as the current month and surrounding months are displayed to provide context and access to closely related events. Today's date is spelled out at the top of the calendar. The current day is displayed in a large push button surrounded by two additional push buttons that are "page turners" to the prior and next days.

The user can select an event in a time slot, select a time slot, or select a range of time slots from the list. The user can also select an event from the day overviews, or any date from the current or surrounding months. The menu bar choices are similar to those provided for the month view.

This view is intended primarily to allow scheduling of events for the day. It provides techniques for accessing different days and it provides access to the event views (not shown in this example).

User Task	Menu Bar	Pop-Up Menu	Drag/Drop
All of the tasks supported for the icon	<i>Calendar</i> (choices are the same as the icon's pop-up menu)	On the background of the view: (same as the icon's pop-up menu)	(1)

Display next (prior) day	<i>View</i> <i>Next day</i> <i>Prior day</i>	<i>Next day / Prior day</i> from the background of the view	n/a
Open the details for a day	<i>Selected</i> <i>Open</i> With a date selected	<i>Open</i> from the date	n/a
Open the details for an event	<i>Selected</i> <i>Open</i> With the event selected	<i>Open</i> from the event in the day overview	n/a
Schedule an event	<i>Selected</i> <i>Open</i> with the time slot selected	<i>Open</i> from a time slot in the list or from the day overview	n/a
Print the current view of the calendar	<i>Calendar</i> <i>Print</i>	<i>Print</i> from the background of the view	Drag the day to the printer (1)
Print a day	With a date selected: <i>Selected</i> <i>Print</i>	<i>Print</i> from the date	Drag a date to the printer
Print an event	With an event selected: <i>Selected</i> <i>Print</i>	<i>Print</i> from the event	Drag an event to the printer
Move a scheduled event from one time slot or day to another	With an event selected: <i>Move</i>	<i>Move</i> from the event	Drag the event to another day, date, or time slot (2)
Copy an event to one or more other time slots or days	With an event selected: <i>Copy</i>	<i>Copy</i> from the event	Ctrl+drag the event to another day, date, or time slot (2)
Other tasks from the year view	(3)		

Notes:

- (1) Drag/drop actions for the calendar as a whole are best provided by including in the view an icon representing the calendar, for example a small icon in the window's title bar (not shown here).
- (2) The properties of the event, including scheduled time, remain the same when it is dropped on a different date. If this causes a time conflict, a window containing both events might be displayed to help the user resolve the conflict. If the scheduled events involve other people whose phone numbers are recorded as well, the calendar might support auto-dialing using those numbers to further assist the user.
- (3) Many of the other tasks from the month view are applicable here as well. For example, the Find choice should still be provided under *Edit* in the menu bar.

The user can also display the prior (next) day by clicking on the appropriate push buttons surrounding the day at the top of the view. Clicking on the day push button causes a window to appear in which the user can enter a specific day or pick one from a scrollable list.

Double-clicking on a date causes a window to open containing the day view for that date. Double-clicking on an event or a time slot causes a window to open containing a description of the event, or allowing an event to be scheduled in that time slot.

CONCLUSION

While this example has been brief and is incomplete, it should provide an adequate basis for an understanding of the OVTT approach. As you pursue design activities keep in mind the goals of: understanding the user's task needs, identifying objects that enable users to accomplish their tasks, defining views that present information associated with specific tasks, and support of interaction techniques that enable those tasks in a consistent, easy to learn, and productive manner.

The OVTT methodology addresses several key aspects of object-oriented interface design, but there are many others. Definition of object properties and sequence of flow between windows are but two that are alluded to in this chapter. As with the adoption of any new methodology you will likely find it easier to weave aspects of this one into approaches that work for you, that you're already comfortable with. This is not a methodology that must be rigorously followed as shown in the example. What is important is that you appreciate the concepts and intent involved. If the table approach works for you that's fine. If it doesn't and instead you find a way to address the goals using your own approach, that's fine too.

Object-oriented user interfaces have introduced a new paradigm for users and for developers. For users, these interfaces tend to be more simple and productive, allowing them to focus on their information as opposed to computer-based artifacts such as programs. As a result the feedback from users continues to be

positive. On the other hand, developers typically invest a significant amount of effort learning and building experience-based methodologies over long periods of time. Methodologies for *design* of object-oriented user interfaces are still evolving. The OVTT methodology introduced in this chapter should prove useful in those stages of design involving identification of the *objects* to be presented, the *views* of those objects, the *tasks* supported in each view, and interaction *techniques* provided to accomplish the tasks.

THE AUTHOR

Dick Berry has been with IBM for over 25 years. He is one of the most respected user interface designers there. He has worked with the team that developed Common User Access (CUA), a scheme for common user interfaces across all applications. He is now working with a team to define the next generation of computer desk tops.

Section IV: SOM 2.1

23

Constructors and Destructors in SOM

INTRODUCTION

The next two chapters provide a preview of coming attractions for SOM. These chapters are based on preliminary expectations for the SOMobjects 2.1 Toolkit, which is currently expected to be released some time in 1994, and on work being done by IBM and others to make it easier to program with SOM. Readers should not assume that SOMobjects 2.1 or any other future products will actually include any of the capabilities described here. But we feel that readers may be interested in the general directions that these chapters describe.

For example, an exciting recent development for SOM is use of the SOM API by C++ compiler vendors to implement C++. At the time of this writing, both IBM and MetaWare are developing DirectToSOM C++ compilers. The advantage of the DirectToSOM approach is that classes programmed in C++ become SOM classes, useful to other languages, and useful within backwards-compatable binary class libraries.

To support DirectToSOM work, a number of extensions have been added to the SOM API. While these extensions are not yet officially supported by the SOMobjects Toolkit, we expect that they may be provided in the near future.

The extended SOM API includes new methods in SOMObject and SOMClass whose purpose is support for C++-style constructors and destructors. Furthermore, SOMobjects Toolkit language bindings can provide special support for these, with the result that native C++ users (i.e., users of non-DirectToSOM

compilers) may use C++ constructors provided by SOM language bindings to initialize SOM objects.

Early versions of SOM kernels and emitters supporting the extended API have been made available to SOMobjects Toolkit users via various SOM user support forums for evaluation and testing. This is independant of the DirectTo-SOM C++ compilers that are being developed. The purpose of this chapter is to introduce new ideas in the SOM API that are related to initializing and uninitalizing SOM objects, and to help you use these ideas if you have access to a SOM system that supports these ideas. In this chapter, we will call such a system ESOM (for Experimental SOM). As explained above, you should not expect that ESOM will be exactly reflected in any future SOMobjects product, but by understanding ESOM we believe that you will be better equipped for using some of the enhanced capabilities that appear in future products.

QUICK REVIEW OF INITIALIZERS IN SOM

Before discussing new facilities, we begin with a quick review of how objects are initialized in the general availabilty release of SOM2.0. The SOMobjects Toolkit Users Guide provides complete details on these matters.

Object initialization is a separate activity from object creation in SOM. Initialization is a capability supported by an object's methods. Methods, of course, are always invoked on an object that has been *created* as an instance of some specific class. The class's instance method table determines the behavior of the object, and thus determines the object's initialization behavior.

Object creation is the act that enables the execution of methods on an object. In SOM, this basically involves storing a pointer to a method table into a word of memory. This single act converts raw memory into a (uninitialized) SOM object that starts at the location of the method table pointer.

The instance variables encapsulated by an object must be brought into some consistent state before general use of a newly-created object makes sense. This is the purpose of initialization methods. Because, in general, every ancestor of an object's class contributes different instance data to an object, it is generally appropriate that each of these ancestors contribute to the initialization of the object. The way this is done in SOM 2.0 is for initializer methods to chain "parent-initializer" calls upwards, thereby allowing initializer method procedures contributed by all ancestors of an object's class to execute. This chaining is not supported in any special way by the SOM 2.0 API. It is simply one of the possible idioms available to users of OOP in SOM, easily available to a SOM class designer as a result of the support provided by SOMobjects Toolkit emitters for parent-method calls.

The SOM API, therefore, does not constrain initialization to be done in any particular way, or require that any particular ordering of ancestor class's method procedures be used, or even that all ancestors be involved.

However, SOM does provide an overall framework that SOM class designers can easily tie into in order to implement default initialization of SOM objects. This framework is provided by the object initialization method, `somInit`, intro-

duced by `SOMObject` and supported by the SOM Toolkit template emitters, whose implementation templates for overridden methods automatically chain parent-method calls upwards through parent classes. Also, some of the class methods that perform object creation call `somInit` automatically.

In SOM 2.0, `SOMClass` introduces six methods for creating new objects: `somNew`, and `somNewNoInit`, which dynamically allocate new memory to hold the created object, and the methods `somRenew`, `somRenewNoInit`, `somRenewNoZero`, and `somRenewNoInitNoZero`, which operate on previously allocated memory. `somNew`, `somRenew`, and `somRenewNoZero` call `somInit`, thereby combining object creation with object initialization. In SOM, such methods are called *object constructors*.

`somInit` best serves the purpose of a default initializer because it takes no arguments, but SOM programmers have the option of introducing other initialization methods that take arguments, and introducing new class methods as object constructors that create an object (often using `somNewNoInit`), and then a non-default initializer.

INITIALIZERS IN ESOM

ESOM makes no major changes in the above framework. All code written using the above approach (and, of course all existing class binaries) continue to be fully supported and useful within ESOM. It simply adds a few new capabilities to SOM and to SOM IDL. In the context of initializers (we'll discuss destructors later), there are essentially two important enhancements:

- ESOM recognizes initializers as a special kind of method, and supports a specific mechanism for ordering the execution of ancestor initializer method procedures. This is based on information provided in the SOM IDL implementation section for classes.
- ESOM introduces a new default initializer method that uses this execution mechanism, and allows SOM class designers to declare additional initializers. The template emitters provide special support for methods declared as initializers.

Two new SOM IDL modifiers are supported by ESOM: `directinitclasses`, and `init`. The first of these is what controls the order of execution of initializer method procedures provided by the different ancestors of the class of an object. The second is the modifier used to indicate that a method is an initializer method — i.e., that it both uses and supports the new ESOM object initialization protocol being discussed.

In ESOM, every class has a `somDirectInitClasses` attribute (introduced by `SOMClass`). For any given class, this is simply the sequence of ancestor classes whose initializer method procedures the given class wants to directly invoke. The default (when no `directinitclasses` modifier is given in the class's IDL declaration) is simply the given class's parents — in left-to-right order. Using this information and the actual runtime class hierarchy above them,

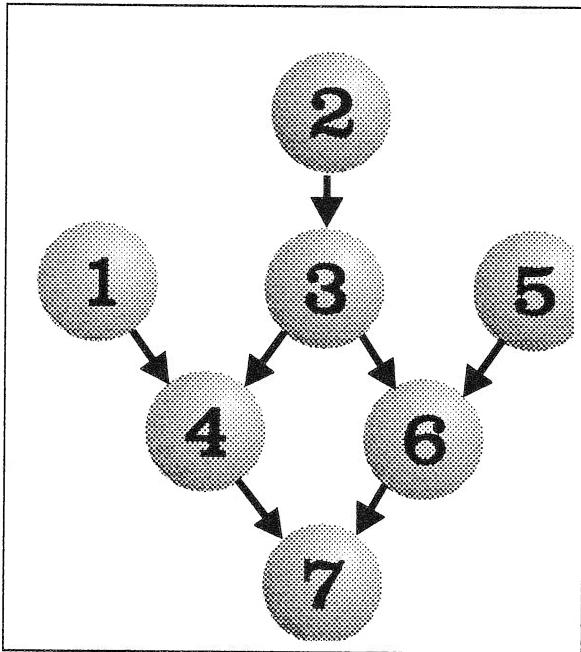


Figure 23.1 Example Initialization Ordering

all classes in ESOM inherit from `SOMClass` the ability to create a data structure called a `somInitCtrl`. This structure represents a particular visit ordering that reaches each class in the transitive closure of `somDirectClasses` exactly once. The ordering is basically the same that C++ uses: when initializing a given object of some specific class, recursively, for each class whose initializer method procedure should be run, first run the initializer method procedures of all of the class's `directinitclasses` (when they have not already been run by other class's initializers), in left-to-right order.

Figure 23.1 illustrates this by showing an inheritance hierarchy along with the initialization ordering produced when an instance of the class labelled 7 is created, and each class simply uses its parents as `directinitclasses`.

The `somInitCtrl` data structure for any given class is based on a global view of the class hierarchy, and is used by SOM initializers to guide their actions so the required initialization ordering is used. For example, this structure is what tells node 6 in Figure 23.1 not to try to run node 3's initializer code (because it has already been executed). Further details associated use of the `somInitCtrl` structure are not important to a general SOM user. For example, the code that deals with this structure is generated automatically, either by a DirectToSOM compiler (in order to use the SOM API to implement C++ classes), or by ESOM Toolkit implementation binding emitters. All initializers take this data structure as an initial inout parameter, and its type is defined in `soma-pi.h`. Also, all initializers return void.

Thus, the new ESOM default object initializer introduced by `SOMObject` takes a `somInitCtrl` as its (only) argument, and returns void. Here is the IDL

for this method.

```
interface SOMObject {
  ...
  void somDefaultInit(inout somInitCtrl ctrl);
  //
  // Default initializer for any SOM object. SOMObject's
  // procedure is appropriate for any class – it calls
  // the directinitclasses initializers, and then calls
  // the somInit method procedure defined by the class
  // (if somInit was overridden by the class).
  ...
  implementation {
    ...
    somDefaultInit: init;
    ...
  };
}
```

IMPLEMENTING NEW CLASSES WITH INITIALIZERS

As suggested above, ESOM programmers can easily declare and implement new classes with initializers. Classes can have as many initializers as desired, and subclasses can invoke whichever of these that they want (just as in C++). The initializers available for a given class include `somDefaultInit` plus any new initializer methods that the class declares. The implementation templates provided by the template emitter for a class's initializers are specially-tailored for that specific class. This is done both for reasons of efficiency, and to make it easy to modify automatically-generated method procedure templates for initializer methods.

Initializers are declared as in the following IDL example:

```
interface NewClass : SOMObject {

  void withName(inout somInitCtrl ctrl,
                in string name);
  void withXY(inout somInitCtrl ctrl,
              in long x,in long y);

  implementation {
    callstyle=oidl;
    releaseorder: withName, withXY;
    withName: init;
    withXY: init;
  };
}
```

When initializers are introduced by a class, as illustrated by NewClass in the above example, the ESOM C and C++ implementation template emitters automatically generate an appropriate “default” implementation for the corresponding initializer method procedures. A default initializer method procedure differs from those provided for other methods. In particular, the body of an initializer method procedure consists of two main sections: the first section calls to ancestors of the class to invoke their initializers; the second section is used by the programmer to perform any “local” initializations appropriate to the instance data of the class being defined. The ancestor class initializer calls and the somInitCtrl argument that is passed together arrange for the second sections to execute exactly once during the initialization of an object and produce exactly the desired ordering of “local” initializations (for example, as illustrated in Figure 23.1).

In the first section, the ancestor classes whose initializers are called are the directinitclasses for the class whose initializer is being defined. As mentioned earlier, the default for this is simply the parents of the new class. *Under no circumstances should the number or ordering of parent-initializer calls made in the first section be changed by a programmer.* These aspects of the generated template, and also the somInitCtrl argument passed to an initializer, are computed based on the directinitclasses of the new class. If you want to change the number or ordering of the parent-initializer calls, then you can use the directinitclasses modifier for this purpose. However, the parent-initializer calls are designed to be modified (i.e., replaced) by the programmer, as now described.

Each call to an ancestor initializer is made by using a special parent-call-like macro defined for this purpose within the implementation bindings. Macros are defined for all possible ancestor initialization calls. Initially, a template calls the default ancestor initializers provided by somDefaultInit, but the programmer may replace any of these calls with a different initializer call (as long as it is to the same ancestor). Non-default initializer calls will generally take additional arguments other than the control argument.

In the second section, the programmer provides any code that may be needed for initialization. For example, as a result of the above IDL declaration for NewClass, the C++ template emitter, xc, will produce the following implementation template for NewClass.

```
/*
 * This file was generated by the SOM Compiler and Emitter
 * Framework.
 */
SOM_Scope void SOMLINK
    withName(NewClass *somSelf,
             somInitCtrl* ctrl,
             string name)
```

```

{
    NewClassData *somThis; /* set by BeginConstructor */
    somInitCtrl globalCtrl;
    somBooleanVector myMask;
    NewClassMethodDebug("NewClass", "withName");

    NewClassBeginConstructor_withName;
    NewClass_Init_SOMObject_somDefaultInit(somSelf, ctrl);

/*
 * local NewClass initialization code
 * added here by programmer
 */
}

SOM_Scope void SOMLINK
    withXY(NewClass *somSelf,
           somInitCtrl* ctrl,
           long x,
           long y)
{
    NewClassData *somThis; /* set by BeginConstructor */
    somInitCtrl globalCtrl;
    somBooleanVector myMask;
    NewClassMethodDebug("NewClass", "withXY");

    NewClassBeginConstructor_withXY;
    NewClass_Init_SOMObject_somDefaultInit(somSelf, ctrl);

/*
 * local NewClass initialization code
 * added here by programmer
 */
}

```

A more complete example that loads instance variables and illustrates ancestor initializer calls to non-default initializers will be provided later. For now, it is useful to note the behavior of an initializer that is passed a NULL control argument. When this happens, the initializer assumes that the object being initialized is an instance of the same class as that providing the initializer, and therefore invokes the class method that computes the somInitCtrl for the class to obtain the correct control structure.

USING INITIALIZERS

If you want to construct an instance of a class supported by a particular initializer, it is simple to use the basic SOM API directly for this purpose. To do this, you invoke `somNewNoInit` on the class object to create an instance, and then invoke the desired initializer on the new object, passing a NULL control argu-

ment in addition to whatever other arguments may be required by the initializer. For example, to construct an instance of NewClass, declared above using the withName initializer, a SOM programmer could write:

```
NewClass *nc = (NewClass*) (_NewClass->somNewNoInit());
nc->withName(0, "a string");
```

Also, the general object constructor somNew, can be invoked on a class to create and initialize objects. In ESOM, this creates an object and then invokes somDefaultInit on it (this will end up invoking somInit code if this is appropriate).

Usage bindings can hide details associated with initializer use in various ways. For example, the SOMObjects Toolkit C usage bindings for some class X already provide a convenience macro XNew() that first assures existence of the class object, and then calls somNew on it to create and initialize a new object. C macros that invoke non-default initializers can also be easily provided.

In the C++ bindings, initializers are represented as overloaded constructors. Because overloading in C++ is done with respect to signature, this is straightforward as long different initializer methods for a class do not take the same argument types. When a collision between the signatures of different initializers prevents supporting them all by C++ constructors, the initializer methods are still available explicitly by name, as illustrated in the above example of direct use of the SOM API. In the case of NewClass, the result is that C++ users would be able to construct a NewClass object using (for example) any of the following expressions:

```
NewClass *nc1 = new NewClass;
NewClass *nc2 = new NewClass("a string");
NewClass *nc3 = new NewClass(1, 2);
```

This is everything necessary for declaration, implementation, and use of ESOM initializers. Before presenting a complete example, we discuss ESOM destructors.

DESTRUCTORS IN ESOM

The purpose of a destructor is to “uninitialize” an object before its storage is freed. This is an important consideration because it allows freeing storage not contained within the body of the object. Destructors in ESOM operate in much the same way as initializers. They take a control argument (in this case, of type somDestructCtrl) and are supported by ESOM template emitters. The somDestructCtrl structure computed by an ESOM class exactly reverses the order in which initializers are executed.

Only a single destructor is needed for any class of objects, so SOMObject introduces the method that provides this function, somDestruct. As with the default initializer method, somDefaultInit, a class designer that with nothing

special to do in the way of uninitialization does not need to be concerned about somDestruct. On the other hand, if you do need to perform uninitialization of instance data introduced by your class, then you should override somDestruct and place the necessary uninitialization code in the generated template. As an alternative, you could also just override somUninit (as you always did, before ESOM). This works because ESOM will arrange for somUninit to be called in this case.

A COMPLETE EXAMPLE

In this example, classes ctor::A through ctor::C provide progressively more initializers, which provides good examples of chaining initialization upwards through non-default initializers. Code relies on the guarantee that each class's initialization code, or destruction code, runs exactly once when an object is initialized or destroyed. Here is the IDL:

```
#include <somobj.idl>
module ctor {
    interface A : SOMObject {
        readonly attribute long a;
        implementation {
            callstyle = oidl;
            releaseorder: _get_a;
            somPrintSelf: override;
            somDefaultInit: override, init;
            somDestruct: override;
        };
    };

    interface B : SOMObject {
        readonly attribute long b;
        void BwithInitialValue(
            inout somInitCtrl ctrl,
            in long initialValue);
        implementation {
            callstyle = oidl;
            releaseorder: _get_b, BwithInitialValue;
            somPrintSelf: override;
            somDefaultInit: override, init;
            BwithInitialValue: init;
            somDestruct: override;
        };
    };
}
```

```

interface C : A, B {
    readonly attribute string c;
    void CwithInitialValue(
        inout somInitCtrl ctrl,
        in long initialValue);
    void CwithInitialString(
        inout somInitCtrl ctrl,
        in string initialString);
    implementation {
        callstyle = oidl;
        releaseorder: _get_c, withInitialString;
        somPrintSelf: override;
        somDefaultInit: override;
        CwithInitialValue: init;
        CwithInitialString: init;
        somDestruct: override;
    };
};

```

From the above, we see that the class `ctor::A` has one initializer method, `somDefaultInit`. `ctor::B` has two: `somDefaultInit` and `BwithInitialValue`. Finally, `ctor::C` has three initializer methods: `somDefaultInit`, `CwithInitialValue`, and `CwithInitialString`. Below is a completed C++ implementation template, followed by a test program. Code written by a programmer (as opposed to being automatically generated) is highlighted in bold.

```

/*
 * This file was generated by the ESOM Compiler.
 * Generated using:
 *     ESOM incremental update: 2.15
 */
#define SOM_Module_ctorexample_Source
#define VARIABLE_MACROS
#define METHOD_MACROS
#include <ctorExample.xih>
#include <stdio.h>

SOM_Scope SOMObject* SOMLINK
ctor_AsomPrintSelf(ctor_A *somSelf)
{
    ctor_AData *somThis = ctor_AGetData(somSelf);
    ctor_AMethodDebug("ctor_A","ctor_AsomPrintSelf");
    printf("{an instance of %s at location %x with
(a=%d)}\n",
           somSelf->somGetClassName(), somSelf, __get_a());
    return (SOMObject*)((void*)somSelf);
}

```

```
SOM_Scope void SOMLINK
    ctor_AsomDefaultInit(ctor_A *somSelf, somInitCtrl* ctrl)
{
    ctor_AData *somThis; /* set by BeginConstructor */
    somInitCtrl globalCtrl;
    somBooleanVector myMask;
    ctor_AMethodDebug("ctor_A", "somDefaultInit");

    ctor_ABeginConstructor_somDefaultInit;
    ctor_A_Init_SOMObject_somDefaultInit(somSelf, ctrl);

/*
 * local A initialization code added here by programmer
 */
    _a = 1;
}

SOM_Scope void SOMLINK
    ctor_AsomDestruct(ctor_A *somSelf,
                      octet doFree,
                      somDestructCtrl* ctrl)
{
    ctor_AData *somThis; /* set by BeginDestructor */
    somDestructCtrl globalCtrl;
    somBooleanVector myMask;
    ctor_AMethodDebug("ctor_A", "ctor_AsomDestruct");
    ctor_ABeginDestructor;

/*
 * local A deinitialization code added by programmer
 */
    _a = 0;

    ctor_AEndDestructor;
}

SOM_Scope SOMObject* SOMLINK
    ctor_BsomPrintSelf(ctor_B *somSelf)
{
    ctor_BData *somThis = ctor_BGetData(somSelf);
    ctor_BMethodDebug("ctor_B", "ctor_BsomPrintSelf");
    printf("{an instance of %s at location %X with (b=%d)}\n",
           somSelf->somGetClassName(), somSelf, __get_b());
    return (SOMObject*)((void*)somSelf);
}

SOM_Scope void SOMLINK
    ctor_BsomDefaultInit(ctor_B *somSelf, somInitCtrl* ctrl)
{
    ctor_BData *somThis; /* set by BeginConstructor */
```

```

= ctor_BGetData(somSelf);
    somInitCtrl globalCtrl;
    somBooleanVector myMask;
    ctor_BMethodDebug("ctor_B", "somDefaultInit");

    ctor_BBBeginConstructor_somDefaultInit;
    ctor_B_Init_SOMObject_somDefaultInit(somSelf, ctrl);

/*
 *local B initialization code added by programmer
 */
    _b = 2;
}

SOM_Scope void SOMLINK
ctor_BBwithInitialValue(ctor_B *somSelf
                        somInitCtrl* ctrl,
                        long initialValue)
{
    ctor_BData *somThis; /* set by BeginConstructor */
    somInitCtrl globalCtrl;
    somBooleanVector myMask;
    ctor_BMethodDebug("ctor_B", "withInitialValue");

    ctor_BBBeginConstructor_withInitialValue;
    ctor_B_Init_SOMObject_somDefaultInit(somSelf, ctrl);
/*
 *local B initialization code added here programmer
 */
    _b = initialValue; // note use of initializer argument
}

SOM_Scope void SOMLINK
ctor_BsomDestruct(ctor_B *somSelf
                  octet doFree,
                  somDestructCtrl* ctrl)
{
    ctor_BData *somThis; /* set by BeginDestructor */
    somDestructCtrl globalCtrl;
    somBooleanVector myMask;
    ctor_BMethodDebug("ctor_B", "ctor_BsomDestruct");
    ctor_BBeginDestructor;
/*
 * local B deinitialization code added by programmer
 */
    _b = 0;

    ctor_BEndDestructor;
}

```



```

/*
 * local C initialization code added by programmer
 */
_c = (char*) SOMMalloc(initialValue+1);
sprintf(_c, "%d", initialValue);
}

SOM_Scope void SOMLINK
ctor_CCwithInitialString(ctor_C *somSelf,
                        somInitCtrl* ctrl,
                        string initialString)
{
    ctor_CData *somThis; /* set by BeginConstructor */
    somInitCtrl globalCtrl;
    somBooleanVector myMask;
    ctor_CMethodDebug("ctor_C", "withInitialString");

    ctorCBeginConstructor_withInitialString;
    ctor_C_Init_ctor_A_somDefaultInit(somSelf,ctrl);
    ctor_C_Init_ctor_B_withInitialValue(somSelf
                                ctrl,
atoi(initialString)-11);

/*
 * local C initialization code added by programmer
 */
_c = (char*) SOMMalloc(sizeof(initialString)+1);
strcpy(_c, initialString);
}

SOM_Scope void SOMLINK
ctor_CsomDestruct(ctor_C *somSelf,
                  octet doFree,
                  somDestructCtrl* ctrl)
{
    ctor_CData *somThis; /* set by BeginDestructor */
    somDestructCtrl globalCtrl;
    somBooleanVector myMask;
    ctor_CMethodDebug("ctor_C", "ctor_CsomDestruct");
    ctor_CBeginDestructor;
/* local C deinitialization code added here by programmer */
SOMFree(c);

    ctor_CsomDestructEndDestructor;
};


$$\begin{array}{l}
// \\
// \text{Test Program} \\
//
\end{array}$$


```

```

main()
{
    SOM_TraceLevel = 1;
    int i;
    ctor_A *a = new ctor_A;
    a->somPrintSelf();
    delete a; printf("\n");

    ctor_B *b = new ctor_B();
    b->somPrintSelf();
    delete b; printf("\n");

    b = new ctor_B(22);
    b->somPrintSelf();
    delete b; printf("\n");

    ctor_C *c = new ctor_C;
    c->somPrintSelf();
    delete c; printf("\n");

    c = new ctor_C(44);
    c->somPrintSelf();
    delete c; printf("\n");

    c = new ctor_C("66");
    c->somPrintSelf(); delete c;
}

```

Here's the program output.

```

% a.out
"ctorExample.C": 31:      In ctor_A:somDefaultInit
"ctorExample.C": 18:      In ctor_A:ctor_AsomPrintSelf
"./ctorExample.xih": 285:      In ctor_A:ctor_A_get_a
{an instance of ctor::A at location 20062628 with (a=1)}
"ctorExample.C": 53:      In ctor_A:ctor_AsomDestruct

"ctorExample.C": 89:      In ctor_B:somDefaultInit
"ctorExample.C": 76:      In ctor_B:ctor_BsomPrintSelf
"./ctorExample.xih": 612:      In ctor_B:ctor_B_get_b
{an instance of ctor::B at location 20062E88 with (b=2)}
"ctorExample.C": 134:      In ctor_B:ctor_BsomDestruct

"ctorExample.C": 112:      In ctor_B:BwithInitialValue
"ctorExample.C": 76:      In ctor_B:ctor_BsomPrintSelf
"./ctorExample.xih": 612:      In ctor_B:ctor_B_get_b
{an instance of ctor::B at location 20062E88 with (b=22)}
"ctorExample.C": 134:      In ctor_B:ctor_BsomDestruct

"ctorExample.C": 177:      In ctor_C:somDefaultInit

```

```

"ctorExample.C": 31:      In ctor_A:somDefaultInit
"ctorExample.C": 89:      In ctor_B:somDefaultInit
"ctorExample.C": 164:     In ctor_C:ctor_CsomPrintSelf
"./ctorExample.xih": 285:      In ctor_A:ctor_A_get_a
"./ctorExample.xih": 612:      In ctor_B:ctor_B_get_b
"./ctorExample.xih": 1006:     In ctor_C:ctor_C_get_c
{an instance of ctor::C at location 20063798 with
(a=1, b=2, c=3)}
"ctorExample.C": 253:     In ctor_C:ctor_CsomDestruct
"ctorExample.C": 134:     In ctor_B:ctor_BsomDestruct
"ctorExample.C": 53:      In ctor_A:ctor_AsomDestruct

"ctorExample.C": 202:     In ctor_C:CwithInitialValue
"ctorExample.C": 31:      In ctor_A:somDefaultInit
"ctorExample.C": 112:     In ctor_B:BwithInitialValue
"ctorExample.C": 164:     In ctor_C:ctor_CsomPrintSelf
"./ctorExample.xih": 285:      In ctor_A:ctor_A_get_a
"./ctorExample.xih": 612:      In ctor_B:ctor_B_get_b
"./ctorExample.xih": 1006:     In ctor_C:ctor_C_get_c
{an instance of ctor::C at location 20063798 with
(a=1, b=33, c=44)}
"ctorExample.C": 253:     In ctor_C:ctor_CsomDestruct
"ctorExample.C": 134:     In ctor_B:ctor_BsomDestruct
"ctorExample.C": 53:      In ctor_A:ctor_AsomDestruct

"ctorExample.C": 229:     In ctor_C:CwithInitialString
"ctorExample.C": 31:      In ctor_A:somDefaultInit
"ctorExample.C": 112:     In ctor_B:BwithInitialValue
"ctorExample.C": 164:     In ctor_C:ctor_CsomPrintSelf
"./ctorExample.xih": 285:      In ctor_A:ctor_A_get_a
"./ctorExample.xih": 612:      In ctor_B:ctor_B_get_b
"./ctorExample.xih": 1006:     In ctor_C:ctor_C_get_c
{an instance of ctor::C at location 20063798 with (a=1, b=55,
c=66)}
"ctorExample.C": 253:     In ctor_C:ctor_CsomDestruct
"ctorExample.C": 134:     In ctor_B:ctor_BsomDestruct
"ctorExample.C": 53:      In ctor_A:ctor_AsomDestruct
%

```

In SOM 2.0, metaclasses were used to introduce constructors for objects. As illustrated by the above example, recognizing initializers as a special kind of method in ESOM and reflecting this in IDL allows usage bindings to provide constructors directly instead of requiring class implementors to define constructors by using of metaclasses for this purpose. Metaclass programming is a powerful capability provided by SOM, but requiring the use of metaclasses to implement constructors is an expensive approach to handling a very common and important requirement of OOP. The support provided by ESOM for this is much more efficient in terms of execution speed as well as required memory use.

24

Metaclass Programming in SOM

INTRODUCTION

As in the previous chapter about object constructors, this chapter provides a look at coming SOM attractions. After summarizing concepts important to metaclass programming in general, we will describe a “metaclass cooperation framework.” This is not currently supported by a SOMobjects Toolkit product, but we believe that future SOMobjects Toolkit products may provide facilities similar to those described here.

What can you do with metaclasses in SOM? You can define the implementations of classes — what data they encapsulate and what procedures execute their methods. So, by subclassing from existing metaclasses that provide an initial capability for implementing classes, you can implement new kinds of classes that have new class variables and new methods (in addition to those that are inherited).

When different metaclasses are automatically combined into SOM-derived metaclasses (as described in Chapter 6), these kinds of enhancements never conflict with each other. This is because both the methods and the data of objects (in this case classes) are segregated into groups corresponding to their introducing class when combined using multiple inheritance.

In addition to introducing new class variables and methods, a metaclass programmer can also override inherited methods. For example, `somInitMIClass` is an important class method that is often overridden. This method determines the procedure pointers that are stored in the instance method table during ini-

tialization of class objects. Since metaclass programmers can override `somInitMIClass`, they can store whatever is desired in the instance method table. The method `somOverrideSMethod` can be used for this purpose. But, the power to explicitly load a class's instance method is a double-edged sword. Given automatic use of derived metaclasses in SOM, which combine unrelated metaclasses to support polymorphism when explicit metaclasses are used, how can metaclass programmers know that what they put in a class's instance method table using a `somInitMIClass` method procedure won't be undone by (or undo) what some other metaclass puts there?

To understand why this problem arises, recall that derived metaclasses in SOM chain four special methods upwards allowing initialization and uninitialization of the class variables introduced by ancestor metaclasses. These methods are `somInit`, `somInitMIClass`, `somClassReady` and `somUninit`. So, if different unrelated metaclasses are used as parents of an automatically derived metaclass, different, generally unrelated `somInitMIClass` method procedures will be executed in sequence beginning with the explicit metaclass, if any. These unrelated `somInitMIClass` method procedures may interfere with each other.

To solve this problem and allow effective use of the power offered by `somInitMIClass`, a "metaclass cooperation framework" can be provided to allow metaclasses to register method procedures as wanting to "cooperate" in the execution of inherited methods (instead of simply overriding these methods). This chapter describes one such cooperation framework, again using the name ESOM (Experimental SOM) to refer to some hypothetical system that provides the described capabilities. For simplicity, we assume use of the methods `somInit` and `somUninit`, which are supported by ESOM as described in the previous chapter. Remember, *ESOM is not an IBM product*.

In ESOM, cooperation takes the form of a sequence of calls to registered cooperative method procedures that is terminated by a call to a final method procedure that returns the result of the method call. The cooperation framework consists of two metaclasses: `SOMMCooperative`, and `SOMMCooperativeRedispatched`. Here is a top-level view of the IDL that describes these. There are six important methods.

```
#include <somclss.idl>
//*****
// SOMMCooperative and SOMMCooperativeRedispached provide
// core functionality upon which cooperative metaclasses
// are built. Metaclasses whose instances may be subclassed
// should be cooperative.
//
// A cooperative metaclass follows these guidelines:
//
// 1. Only the four following methods may be overridden
//    in IDL:
//      a. somInit,
//      b. somInitMIClass,
//      c. somClassReady, and
//      d. somUninit
//
// 2. If it is necessary to "override" any methods other
//    than the above four methods, the metaclass should instead
//    inherit (directly or indirectly) from SOMMCooperative
//    or SOMMCooperativeRedispached, and should then use
//    cooperative methods to achieve the desire results.
//    Generally, these methods are invoked by a class on
//    itself during execution of somInitMIClass code defined
//    by its metaclass.
//
//*****
// - Define the Metaclass Cooperation Framework interfaces
//
interface SOMMCooperative : SOMClass {
// method definitions for:
//   sommAddCooperativeInstanceMethod
//   sommAddCooperativeClassMethod
//   sommRequestFirstCooperativeInstanceMethodCall
//   sommRequestFirstCooperativeClassMethodCall
//   sommRequestFinalCooperativeClassMethodCall
//   sommSatisfyRequests
};
//
//*****
// A SOM programmer who wants to place redispach stubs in a
// metaclass's instances' instance method tables (to receive
// control when methods are dispatched) should:
//
// 1. Define the metaclass by inheriting (directly or
//    indirectly) from SOMMCooperativeRedispached. This will
//    automatically install redispach stubs in the instance
//    method tables of all objects whose metaclass is (or is
//    derived from) the defined metaclass, and will
//    automatically handle all details associated with method
//    dispatch (including correct handling of parent method
//    calls).
//
```

```

// 2. Use an appropriate method from SOMMCooperative to
//    register a    method procedure to cooperate on
//    somDispatch.
// //*****=====
// 
interface SOMMCooperativeRedispatched : SOMMCooperative {
//
// SOMMCooperative adds no methods of great importance.
// Its purpose is to automatically provide redispatch
// handling for programmer-defined metaclasses.
};

```

Metaclasses derived from SOMMCooperative inherit not only somInitMIClass, introduced by SOMClass but also the ability to register cooperative method procedures using methods that record registration information in class variables inherited from (and initialized by) SOMMCooperative.

Thus, as different metaclasses' somInitMIClass method procedures register different cooperative methods during the initialization of a class object, class variables incrementally accumulate information (recall that class variables include the instance method table, introduced by SOMClass). Later, once the class has been registered (using somClassReady), whenever a method call is resolved through the method table, the sequence of method procedures that want to cooperate on the method invocation is executed first, followed by a final call to the original method procedure (i.e., what the content of the method table would have been if not for cooperation on the method). Of course, the cooperation sequence for a method may be empty, in which case the method table entry will simply be the final method procedure.

In addition to the capabilities available from SOMMCooperative, metaclasses derived from SOMMCooperativeRedispatched acquire the following trait: they automatically arrange that after the sequence of cooperative method procedures for any given method has executed and before the final call, the redispatch stub for the method is invoked. Then, when somDispatch is called by the redispatch stub, all method procedures cooperating on somDispatch are executed, and then, finally, the "final" method is called. This result is arranged by careful use of the SOMMCooperative methods and the overall SOM API. Classes in SOM are indeed quite useful for packaging reusable functionality, especially at this level. SOMMCooperative, and SOMMCooperativeRedispatch package code whose execution involves subtle interactions within the SOM abstract machine, and, at the same time, give SOM programmers the ability to easily create powerful metaclasses whose execution is cooperative.

SOMMCooperative also provides methods for affecting the ordering of the cooperation sequence for a method. These methods allow somInitMIClass code written by a metaclass programmer to *request* that a given procedure be the first in the cooperation sequence for a given method, or that it be the "final" method procedure, whose result is returned to the caller. These are the sommRequest methods previewed in the above IDL. Clearly, these requests may fail as a result of unrelated metaclasses issuing conflicting requests. Requests are therefore saved in a "request block," and are examined when the sommSatisfyRequests call is invoked. Request blocks for different derived metaclass an-

cestors are granted (or failed) as single atomic actions, incrementally building up a class's behavior, and providing the ability to recognize failure.

The cooperation framework can not completely prevent the possibility of interference between different metaclasses combined into derived metaclasses. This seems impossible in general — it is a question of how much can be done cooperatively, and also of making sure that a metaclass is informed when its requests cannot be granted, due to earlier granted request with which it conflicts. Then, if recovery from a failed request is not possible, a metaclass can invoke an inherited method that will print an informative error message and terminate execution. The alternative would be that metaclass code would simply fail mysteriously as a result of combination with other metaclasses in SOM-derived metaclasses. The metaclass framework prevents this.

When defining a new metaclass, you can always introduce new methods and class variables without special concern for interference between metaclasses. But, when you want to write code that participates in the execution of an inherited method (either an instance method, or a class method), then, for all but the four special methods automatically chained for SOM-derived metaclasses, the cooperation framework should be used instead to achieve the same result. This is always possible.

Often the purpose that would be achieved by overriding an inherited method can be served without requesting any particular ordering of the cooperation sequence for the method. But there are also times when it is important to know that a given method procedure will be the first called (it is then possible to break the cooperation chain and return a method result before any other method procedures are called). Also, it is sometimes necessary for a given method procedure to be the final method called after all other cooperative methods are called. These two cases reproduce the two possibilities provided by conventional overriding, while also preventing interference between metaclasses. In the first case, it is like an override without parent method calls, and in the second case, it is like an override with parent method calls.

The following analogy seems a good way to understand the top-level idea of the cooperation framework. In SOM, as in most OOP models, parent-method calls provide a way of invoking "other" functionality necessary for supporting the overall semantics of a method call. Specifically, parent method calls allow linking separate pieces of code together along lines of class inheritance. The metaclass cooperation framework provides metaclasses with the ability to dynamically construct cooperation chains along other lines — between method procedures defined by separate, unrelated metaclasses that have been combined into SOM-derived metaclasses.

EXAMPLES

To provide a simple usage example for the metaclass framework, we consider the idea of classes that simply count the number of their instances that exist at any time. This example deliberately ignores important aspects of the SOM API

to focus on use of the cooperation framework. We'll tackle a full blown, realistic example later.

To aid understanding, we first show how a metaclass that implements an instance count might be implemented without use of the cooperation framework, and then we show how the same objective can be achieved with the framework. We name the (uncooperative) metaclass SOMMCounted. An object whose metaclass is derived from SOMMCounted will be counted, and will have a class object with an instanceCount attribute.

Without use of the cooperation framework, the IDL for the desired metaclass might appear as follows:

```
interface SOMMCounted : SOMClass {
    readonly attribute long instanceCount;
    implementation {
        somMethodProc* doFree; // explained below
        somInit: override; // to initialize instanceCount
        somNew: override; // to increment instanceCount
        somInitMIClass: override // explained below
    };
};
```

The first thing to note concerning the above IDL is that SOMMCounted cannot be reliably combined into a SOM-derived metaclass with any other metaclasses that overrides somNew. The reason is that somNew is not one of the four methods handled specially by SOM-derived metaclasses. If some other metaclass overrides somNew and this metaclass (call it X) and SOMMCounted are automatically combined into a SOM-derived metaclass during subclassing, then either X's somNew will be installed in the instance method table of an instance of the derived metaclass, or SOMMCounted's somNew will be installed. Because neither of these two metaclasses knows about the other, there is no way that both somNew method procedures will get executed. In spite of this, we continue the example.

An initial C++ implementation template for SOMMCounted could be completed as follows:

```
#define SOMMCounted_Class_Source
#define VARIABLE_MACROS
#include <counted.xih>

static char *somFreeName = "somFree";
static somId somFreeId = &somFreeName;

SOM_Scope void SOMLINK somInit(SOMMCounted *somSelf)
{
    SOMMCountedData *somThis = SOMMCountedGetData(somSelf);
    _instanceCount = 0;
    SOMMCounted_parent_SOMClass_somInit(somSelf);

}
```

```

SOM_Scope SOMObject SOMLINK somNew(SOMMCounted *somSelf)
{
    SOMMCountedData *somThis = SOMMCountedGetData(somSelf)
    _instanceCount++;
    return (SOMMCounted_parent_SOMClass_somNew(somSelf));
}

/*
 * This local procedure is not produced by the template
 * emitter. It was written specially, and will be placed
 * in a counted object's method table by the following
 * somInitMIClass code so that when somFree is executed,
 * the instance count for the object's class can be
 * decremented. The class variable _doFree is used to
 * hold the original method table content.
 */
SOM_Scope void SOMLINK SOMMCounted_somFree(SOMObject obj)
{
    SOMClass objClass = SOM_GetClass(obj);
    SOMMCountedData *somThis = SOMMCountedGetData(objClass);
    _instanceCount--;
    _doFree(obj); /* _doFree is set in somInitMIClass, below */
}

SOM_Scope void SOMLINK
    somInitMIClass(SOMMCounted somSelf,
                  long inherit_vars,
                  string className,
                  SOMClass_SOMClassSequence* parentClasses,
                  long dataSize,
                  long dataAlignment,
                  long maxStaticMethods,
                  long majorVersion,
                  long minorVersion)
{
    SOMMCountedData *somThis = SOMMCountedGetData(somSelf);
    /*
     * Parent method call to chain somInitMIClass upwards.
     */
    parent_somInitMIClass(...);
    /*
     * Record the original content of the instance mtab
     * entry for somFree in a class variable.
     */
    _doFree =
        somClassResolve(somSelf, SOMObjectClassData.somFree);
    /*
     * Replace original somFree entry with a pointer to
     * the local method (which decrements the count and
     * then calls doFree.
     */
    _somSelf->somOverrideSMETHOD(
        somFreeId,
        SOMMCounted_somFree);
}

```

Although the above solution is not cooperative, due to its override of `somNew`, the handling of the instance method table entry for `somFree` is instructive, since this provides a clue to the way that the cooperation framework is designed. In `SOMMCounted::somInitMIClass`, the original content of the `somFree` entry in `somSelf`'s instance method table is remembered, and this is later called by the locally-registered routine for `somFree`, after decrementing the class's instance count. This is very similar to a parent method call, but is not based directly on inheritance. Rather, it is simply based on whatever the content of the instance method table is when `SOMMCounted::somInitMIClass` saves the `somFree` entry. If some other metaclass participating in the initialization of the class object identified here as `somSelf` had already done something similar to what `SOMMCounted` does with `somFree`, then that metaclass would have saved the original content of the `somFree` entry, and `SOMMCounted` would then save (and later invoke) this metaclass's `somFree` method procedure, after which the other metaclass would invoke the original `somFree` entry. This is how a chain of method procedures can be built up dynamically by different metaclass's `somInitMIClass` method procedures, first saving and then replacing the content of an instance method table entry. This is basically how the cooperation framework works.

If the technique illustrated above for `somFree` were all that a metaclass programmer needed to avoid interference, there would be little need for a cooperation framework. Metaclass programmers could simply use this technique to achieve the desired results. But complications arise from providing control over the cooperation chain ordering and also from handling parent method calls when redispach stubs are placed in method tables. As a result, the methods introduced by `SOMMCooperative` are extremely important — they provide a correct solution to a number of difficult problems whose combined solution requires an intimate understanding of the overall SOM API, and they offer a simple-to-use interface for metaclass programmers.

These methods are now described in more detail, completing the earlier top-level IDL view.

```
interface SOMMCooperative : SOMClass {
    somMethodProc** sommAddCooperativeInstanceMethod(
        in somId methodId,
        in somMethodProc* coopProc);
```

This method installs a cooperative override in the receiver's instance method table, and is the cooperation framework analogy to the technique illustrated in the above example for handling `somFree`. The returned result is the location of the method procedure pointer that must be invoked by `coopProc` to continue the cooperation chain. This location is maintained by a class in order to support cooperation chain ordering.

```
somMethodProc** sommAddCooperativeClassMethod(
    in somId methodId,
    in somMethodProc* coopProc);
```

This method installs a cooperative override in the receiver's class's instance method table. In other words, this method allows a class to change its own behavior (as opposed to the behavior of its instances) by modifying the instance method table of the class of which it is an instance. This is how cooperation on class methods (such as somNew) is achieved. The returned result is the location of the method procedure pointer that must be invoked by coopProc to continue the cooperation chain.

The above two methods are especially important because they can be used by metaclasses without any possibility of interference. In contrast, metaclasses using the request methods below may interfere with each other. As mentioned earlier, this possibility is handled by allowing each metaclass to build up a request block by making requests and then ask to have the request block satisfied.

```
boolean sommSatisfyRequests();
```

When this method is invoked, the class's current request block is checked to see if any new requests conflict with previously-granted requests. If so, none of the new requests are granted and FALSE is returned. Otherwise all the new requests are granted. Instead of returning a result, the request methods themselves all accept an extra output argument which is the address of a variable that the caller wants loaded with the location of its cooperation chain method pointer (if the request is satisfied).

```
void sommRequestFirstCooperativeInstanceMethodCall(
    in somId methodId,
    in somMethodProc* coopProc,
    out somMethodProc** chainProcAddrAddr);
```

This method *requests* a cooperative override in the receiver's mtab. It is similar to sommAddCooperativeInstanceMethod, but requires coopProc to be the first cooperation chain method procedure that is called when the indicated method is invoked on an instance of the class being initialized. The coopProc argument is the address of the cooperative method procedure being registered, and the chainProcAddrAddr argument is the address of the caller's variable into which the address of the somMethodProc* that will continue the cooperation chain will be placed if the request is satisfied.

```
void sommRequestFirstCooperativeClassMethodCall(
    in somId methodId,
    in somMethodProc* coopProc,
    out somMethodProc** chainProcAddrAddr);
```

This method *requests* a cooperative override in the receiver's class's mtab. It is similar to sommAddCooperativeClassMethod but requires coopProc to be the first cooperation chain method procedure that is called when the indicated method is invoked on the class being initialized. The coopProc argument is the address of the cooperative method procedure being registered, and the chainProcAddrAddr argument is the address of the caller's variable into which the

address of the address of the somMethodProc* that will continue the cooperation chain will be placed if the sommSatisfyRequests method call is successful.

```
void sommRequestFinalClassMethodCall(
    in somId methodId,
    in somMethodProc* methodProc);
```

This method *requests* that the indicated methodProc be called to provide the final semantics for the indicated class method. Note that no output argument is used to support cooperation—the final method simply returns a result. In ESOM, this method is currently only provided for class methods. This concludes a listing of the SOMMCooperative methods.

Using the Cooperation Framework

Using the cooperation framework methods described above, we can now improve on the previous example to implement a cooperative metaclass SOMMCoopCounted as follows. We will cooperate on the class method somNew, and the instance method somFree. No special ordering of cooperation chain methods is required. Here is the IDL for our new metaclass:

```
interface SOMMCoopCounted : SOMMCooperative {

    readonly attribute long instanceCount;
    implementation {
        somMethodProc** doFree;
        somMethodProc** doNew;
        somInit: override; // to init instanceCount
        somInitMIClass: override // to register cooperation
    };
};
```

A completed implementation template might appear as follows:

```
#define SOMMCoopCounted_Class_Source
#define VARIABLE_MACROS
#include <coopcounted.xih>

static char *somFreeName = "somFree";
static somId somFreeId = &somFreeName;
static char *somNewName = "somNew";
static somId somNewId = &somNewName;

SOM_Scope void SOMLINK somInit(SOMMCoopCounted *somSelf)
{
    SOMMCoopCountedData
        *somThis = SOMMCoopCountedGetData(somSelf);
    SOMMCounted_parent_SOMClass_somInit(somSelf);
    _instanceCount = 0;
}

SOM_Scope void SOMLINK
```

```

somInitMIClass(SOMMCountedCounted somSelf,
               long inherit_vars,
               string className,
               SOMClass_SOMClassSequence* parentClasses,
               long dataSize,
               long dataAlignment,
               long maxStaticMethods,
               long majorVersion,
               long minorVersion)
{
    SOMMCoopCountedData
        *somThis = SOMMCoopCountedGetData(somSelf);
    _parent_somInitMIClass(...);
    _doFree = somSelf->sommAddCooperativeInstanceMethod(
        somFreeId,
        SOMMCoopCounted_somFree);
    _doNew = somSelf->sommAddCooperativeClassMethod(
        somNewId,
        SOMMCoopCounted_somNew);
}

#define DO_Free \
    ((somTD_SOMObject_somFree) (*_doFree))

SOM_Scope void SOMLINK
    SOMMCoopCounted_somFree(SOMObject *obj)
{
    SOMClass objClass = SOM_GetClass(obj);
    SOMMCoopCountedData
        *somThis = SOMMCoopCountedGetData(objClass);
    _instanceCount--;
    DO_Free(obj); /* cooperate on somFree */
}

#define DO_New \
    ((somTD_SOMClass_somNew) (*_doNew))

SOM_Scope void SOMLINK
    SOMMCoopCounted_somNew(SOMMCoopCounted somSelf)
{
    SOMMCoopCountedData
        *somThis = SOMMCoopCountedGetData(somSelf);
    _instanceCount++;
    return DO_New(somSelf); /* cooperate on somNew */
}

```

Other Metaclasses

The above SOMMCoopCounted metaclass was fairly simple, but it illustrates a typical metaclass that requires none of the sommRequest methods. Although a variety of useful metaclasses can be provided by simply using cooperative instance and class methods, it is also instructive to see an example of a metaclass

that must make specific sommRequest calls in order to do its job. For this purpose, we consider one possible implementation for a metaclass already provided by ESOM — the Single Instance Metaclass. Before beginning this example, however, we take the opportunity to mention some other metaclasses provided by ESOM.

In addition to the basic metaclass cooperation framework provided by SOMM-Cooperative and SOMMCooperativeRedispached, ESOM provides a small number of other metaclasses whose purpose is to package additional advanced SOM capabilities. These metaclasses have been programmed using the cooperation framework, and currently include the following:

- SOMMSingleInstance
- SOMMBeforeAfter
- SOMMTraced
- SOMMReplicable

SOMMBeforeAfter provides a special kind of redispach handling that allows execution of specially registered method procedures before and after the execution of a method. This is used by SOMMTraced to display the arguments that are passed in the method call and the returned result. In SOMMReplicable, before/after behavior is used to automate the duties required for use of the SOM Toolkit Replication Framework.

The desired semantics of SOMMSingleInstance is as follows. A class that is an instance of SOMMSingleInstance (or some metaclass derived from SOMM-SingleInstance) should only allow one instance of itself to exist at any time. This kind of a class is useful in a variety of situations when a single object can provide a shared resource to multiple users. The idea is that once an instance of some “single instance” class has been created, subsequent requests for new instances will result in the user getting back a reference to the existing instance, rather than a new instance.

Single Instance Metaclass Example

There are a number of ways of implementing such a metaclass in SOM. The approach used by ESOM is to implement SOMMSingleInstance so that its instances (classes) return a reference-counted “proxy” for the single class instance (instead of the single instance itself) when somNew is invoked on them. The proxy is implemented as an instance of a dynamically-created subclass of the original single instance class (this is done to arrange that the proxy will support the right interface). The proxy class uses redispach stubs via the cooperation framework to redirect method calls made on a proxy object to the “single instance” to which it corresponds, and it uses abstract inheritance so that proxies themselves contain no data. This approach has a number of important benefits, but is a bit too complex to present here. An alternative approach useful for tutorial purposes is illustrated here.

In the approach illustrated here, proxies for the “single instance” are not used — the single instance itself is returned when somNew is invoked. This allows a

simpler solution than used by ESOM. Nevertheless, this example is realistic in terms of the design issues that it addresses. Also, it provides the opportunity of pointing out some important aspects of the SOM API — aspects you should understand if you are going to be programming metaclasses in SOM.

The approach illustrated here is based on the idea of reference-counting the number of pointers to the “single instance” that are given out. Our approach will be somewhat similar to that taken for SOMMCounted in this respect, but there are important differences. These will be explained below, following the IDL for SOMMSingleInstance.

```
SOMMSingleInstance : SOMMCooperative {
    implementation {
        SOMObject *singleInstance;
        long refCount;
        somMethodProc** doNew;
        somMethodProc** doNewNoInit;
        somMethodProc** doRenewNoInitNoZero;
        somMethodProc** doFree;

        somInitMIClass: override;
    };
}
```

In this example, SOMMSingleInstance adds no new class methods to those inherited from SOMMCooperative. It introduces new class variables for holding a pointer to a class’s single instance and a reference count for the number of references to the single instance that have been given out. It also introduces class variables for holding cooperation chain links for the class methods somNew, somNewNoInit, somRenewNoInitNoZero, and the instance method somFree. It overrides somInitMIClass to provide a control point for initialization of all these class variables.

The reason for cooperation on the two other class methods besides somNew is that somNew is not the only way that SOM objects may be created, and we would like SOMMSingleInstance to work over as wide a range of object creation approaches as possible.

Basically, SOM objects can be created three different ways: somNew can be called, somNewNoInit can be called, or any of the four somRenew class methods can be called. Both somNew and somNewNoInit begin by allocating storage for a new object. To turn this raw memory into a SOM object, they call somRenewNoInitNoZero. This is the method that loads the first word of a SOM object with its method table pointer, turning storage into an actual SOM object. Following this, somNewNoInit just returns to the caller, while somNew first invokes the method somInit on the new instance before returning.

Without exception, somRenewNoInitNoZero is always called when a new object is created via any of the somNew or somRenew methods. It may also be called by SOM users directly, after raw memory for a SOM object has been allocated by the user. This raises the following question: what can be done if a single instance currently exists at some given address, and a user calls somRenewNoInitNoZero with a different address? Loading the first word at the

passed address would create a second class instance, which would violate the intended semantics for SOMMSingleInstance. In this illustration we would handle this by documenting single instance classes as requiring the use of somNew (or somNewNoInit) for creating "new" instances. The circumstance described above is then handled as an error. One reason why ESOM uses proxies is that they solve this problem — proxies can be created anywhere, independently of their corresponding single instance.

Another interesting complication concerns the fact that, as with the earlier SOMMCoopCounted example, we use somFree for decrementing the reference count. But somFree should only be applied to objects that have been created using somNew or somNewNoInit. Luckily, we have just required that this be so, therefore this issue will not be considered further here. (This problem is also solved by using proxies.)

Having addressed the above issues, the major remaining design issue concerns the ordering of cooperation sequences. We want the SingleInstance method procedure for somFree to be first in its cooperation chain for this method so that actual object uninitialization and freeing of storage can be prevented when necessary. As mentioned earlier, this is similar to an override of an inherited instance method. In this case, somFree can decrement a reference count, and simply return (without freeing the object) if the count has not reached zero. On the other hand, when the count reaches zero, a "parent call" through the cooperation chain can be made to actually free the single instance. Similar considerations apply to the required cooperation chain positions of the SingleInstance method procedures for somNew, somNewNoInit, and somRenewNoInitNoZero. Actual creation of the single instance is left to the somRenewNoInitNoZero method procedure, since this method is the single common aspect for all object creations.

Here, then, is the implementation code.

```
#define SOMMSingleInstance_Class_Source
#define VARIABLE_MACROS
#include "sngliccls.xih"
/*
 * static definitions for somIds. This is more
 * efficient than using somIdFromString.
 */
static string somNewName = "somNew";
static somId somNewId = &somNewName;
static string somNewNoInitName = "somNewNoInit";
static somId somNewNoInitId = &somNewNoInitName;
static string somRenewNoInitNoZeroName
    = "somRenewNoInitNoZero";
static somId somRenewNoInitNoZeroId
    = &somRenewNoInitNoZeroName; static string
somFreeName = "somFree";
static somId somFreeId = &somFreeName;
```

```

/*
 * Cooperative Procedures
 */
#define DO_New \
    ((somTD_SOMClass_somNew) (*_doNew))

SOM_Scope SOMObject* SOMLINK
    sngl_somNew(SOMMSingleInstance *somSelf)
{
    SOMMSingleInstanceData
        *somThis = SOMMSingleInstanceGetData(somSelf);
    if (!_singleInstance) return DO_New (somSelf);
    else {
        refCount++;
        return _singleInstance;
    }
}

#define DO_NewNoInit \
    ((somTD_SOMClass_somNewNoInit) (*_doNewNoInit))

SOM_Scope SOMObject* SOMLINK
    sngl_somNewNoInit(SOMMSingleInstance *somSelf)
{
    SOMMSingleInstanceData
        *somThis = SOMMSingleInstanceGetData(somSelf);
    if (!_singleInstance) return DO_NewNoInit (somSelf);
    else {
        refCount++;
        return _singleInstance;
    }
}

#define DO_RenewNoInitNoZero \
    ((somTD_SOMClass_somRenewNoInitNoZero) (*_doRenewNoInitNoZero))

SOM_Scope SOMObject* SOMLINK
    sngl_somRenewNoInitNoZero(
        SOMMSingleInstance *somSelf,
        void* obj)
{
    SOMMSingleInstanceData
        *somThis = SOMMSingleInstanceGetData(somSelf);
    if (!_singleInstance) {
        singleInstance = DO_RenewNoInitNoZero (somSelf, obj);
        refCount = 1;
        return _singleInstance;
    }
    else if ( _singleInstance == obj ) return obj;
    else {
        somPrintf( "Error: somRenewNoInitNoZero was invoked "
                    "on an instance of SOMMSingleInstance but "
                    "the argument memPtr did not point to the "
                    "single instance.");
        SOM_Error( SOM_FatalCode(...) );
    }
}

```

```

#define DO_Free \
    ((somTD_SOMObject_somFree) (*_doFree))

SOM_Scope void SOMLINK
    sngl_somFree(SOMObject *obj)
{
    SOMMSingleInstanceData
        *somThis = SOMMSingleInstanceGetData(SOM_GetClass(obj));
    _refCount--;
    if (!_refCount) {
        DO_Free (obj);
        singleInstance = NULL;
    }
}

/*
 * Class Initialization.
 */

SOM_Scope void SOMLINK
    somInitMIClass(SOMMSingleInstance *somSelf,
                    long inherit_vars,
                    string className,
                    SOMClass_SOMClassSequence*
parentClasses,
                    long dataSize,
                    long dataAlignment
                    long maxStaticMethods
                    long majorVersion,
                    long minorVersion)
{
    SOMMSingleInstanceData
        *somThis = SOMMSingleInstanceGetData(somSelf);
    _parent_SOMMCoop_SOMMCooperative_somInitMIClass( ... );

    somSelf->sommRequestFirstCooperativeClassMethodCall(
        somNewId,
        sngl_somNew,
        &(_doNew));
    somSelf->sommRequestFirstCooperativeClassMethodCall(
        somNewNoInitId,
        sngl_somNewNoInit,
        &(_doNewNoInit));
    somSelf->sommRequestFirstCooperativeClassMethodCall(
        somRenewNoInitNoZeroId,
        sngl_somRenewNoInitNoZero,
        &(_doRenewNoInitNoZero));
    somSelf->sommRequestFirstCooperativeInstanceMethodCall(
        somFreeId,
        sngl_somFree,
        &(_doFree));

    if (!somSelf->sommSatisfyRequests()) {
        somPrintf("Error: SingleInstance requests not granted\n");
    }
}

```

```

        SOM_Error( SOM_FatalCode(1) );
    }
    _singleInstance = NULL;
    _refCount = 0;
}

```

One may complain that the above implementation of `SingleInstance` uses the cooperation framework correctly, but it isn't very cooperative — it makes many requests that may conflict with those of other metaclasses. Use of the cooperation framework makes it possible to detect this when it happens and issue an informative error, but it should be clear that metaclasses that use few or no `sommRequest` calls to the cooperation framework have a better chance of being successfully combined with other metaclasses. Luckily, use of proxies in the actual ESOM implementation of `SingleInstance` requires no use of `sommRequest` methods. As a result, the ESOM `SingleInstance` metaclass will not conflict with other metaclasses. While on this subject, we should mention that the cooperation framework is actually implemented using itself, but that no requests are required for this purpose. As a result, cooperative metaclasses can never interfere with operation of the cooperation framework itself.

This concludes a demonstration of using the cooperation framework to implement a simple `SingleInstance` metaclass. This example illustrated the use of `sommRequest` methods in the cooperation framework, and provided an opportunity for discussing the different ways that SOM objects can be created. A final example of using the cooperation framework is now provided to illustrate the use of `SOMCooperativeRedispached`. We'll implement a simple trace facility. This requires no `sommRequest` method calls.

A Trace Metaclass

It is almost too simple to create a rudimentary trace facility using `SOMMCooperativeRedispached`. All that is necessary to produce a tracing metaclass `SOMMTraced` is to subclass from `SOMMCooperativeRedispached`, and cooperate on `somDispatch`. For this example, we just print the name of the method that is being invoked when the `SOMMTraced` method procedure for `somDispatch` is invoked. All classes that have `SOMMTraced` as an explicit metaclass, or are derived from classes for which this is true will then print a message whenever a method is invoked on one of their instances. The actual ESOM trace metaclass uses the SOM Interface Repository to determine the types of the arguments being passed, and uses this information to display the values of the passed arguments and the returned result.

Here is the IDL:

```

SOMMTraced : SOMMCooperativeRedispached {
    implementation {
        somMethodProc** doDispatch;
        somInitMIClass: override;
    };
}

```

A simple implementation for SOMMTraced then appears as follows:

```
#define SOMMTraced_Class_Source
#define VARIABLE_MACROS
#include "traced.xih"
#include <stdio.h>
/*
 * static definitions for somIds
 */
static string somDispatchName = "somDispatch";
static somId somDispatchId = &somDispatchName;

/*
 * Cooperative Procedures
 */

#define DO_Dispatch \
((somID_SOMObject_somDispatch) (*_doDispatch)) \
SOM_Scope boolean SOMLINK \
    traced_somDispatch(SOMObject *obj,
void *RetVal,
somId methodId,
va_list ap)
{
    SOMMTracedData
        *somThis = SOMMTracedGetData(SOM_GetClass(obj));
    printf("%s called on:\n ", somStringFromId(methodId);
    obj->somPrintSelf();
    DO_Dispatch(obj, RetVal, methodId, ap);
}

/*
 * Class Initialization.
 */

SOM_Scope void SOMLINK
    somInitMIClass(SOMMTraced *somSelf,
                    long inherit_vars,
                    string className,
                    SOMClass_SOMClassSequence* parentClasses,
                    long dataSize,
                    long dataAlignment,
                    long maxStaticMethods,
                    long majorVersion,
                    long minorVersion)
{
    SOMMTracedData
        *somThis = SOMMTracedGetData(somSelf);
    _parent_somInitMIClass( ... );
```

```
_do_Dispatch = somSelf->somAddCooperativeInstanceMethod(  
    somDispatchId,  
    traced_somDispatch);  
}
```

This concludes our discussion of the use of the ESOM metaclass cooperation framework. This framework plus the additional utility metaclasses provided by ESOM provide a solid and useful foundation, supporting a large percentage of the uses for which new metaclasses might be developed by SOM programmers.

Bibliography

References and Additional Reading

The OS/2 2.1 Application Programmer's Guide by Jody Kelly, et al, Van Nostrand Reinhold.

The C Programming Language, Second Edition by Brian W. Kernighan and Dennis M. Ritchie, published by Prentice Hall.

Client/Server Programming with OS/2 2.1, Third Edition by Bob Orfali and Dan Harkey, Van Nostrand Reinhold.

Programming the OS/2 Presentation Manager by Charles Petzold, Microsoft Press.

The C++ Programming Language by Bjarne Stroustrup, Addison-Wesley Publishing Company, Reading, Massachusetts.

For Additional Reading

Reflections on Metaclass Programming in SOM by Scott Danforth, OOPS-LA Conference Proceedings, 1994.

The Annotated C++ Reference Manual by Margaret A. Ellis and Bjarne Stroustrup, Addison Wesley.

C++ Primer by Stanley B. Lippman, Addison Wesley.

Encapsulation and Inheritance in OOP Languages by Alan Snyder, OOPS-LA Conference Proceedings, 1986.

The Common Object Request Broker: Architecture and Specification by The Object Management Group Inc. and x/Open.

Index

- AboutDialog class, 339-340
abstract classes, 130
 in C++, 62
accelerator keys, 41
affinity groups, 123
APIs (Application Programming Interfaces), 15
 for execution and control, 16-17
app1.c, 200-201, 231, 255-258, 283-286
App1 class, 282-286
 in PMApp program, 200
 PM controls and, 230-233
app1.idl, 200, 231, 254, 282-283
application objects, 282-286
argument conversion, 54-55
ascout.c, 166-167
ascout.idl, 166
attributes, 294.
 metaclass, accessing, 161-162
 in SOM, 145-151
 C emitter option, 150-151
 identifying the IDL compiler, 147
 instance variables or attributes, 145-146
 modifying attributes, 147
 modifying attributes in implementation section, 146
 releaseorder: statement, 147-148
- bConditionHandler class, 346-347
bEventListener class, 348-349
bitmaps, 42
block devices, 293
bMainThread class, 340-341
boolean somClassDispatch, 114
boolean somDispatch, 114
boxdraw.c, 218-220
boxdraw.idl, 217
box drawing program, 205-221
bThread class, 340-341
Button1Click, 196
- buttons, 39-40
 radio, 40
- C++, 48-71
 generic procedures in, 60-66
 abstract classes, 62
 constructors, 60-62
 templates, 64
 inheritance in, 49-60
 argument conversion, 54-55
 definition of object structure, 50-53
 multiple inheritance, 66-70
 polymorphism, 56
 tree and graph inheritance, 66-68
 uses of C++ objects, 50
 virtual functions, 57-60
 interfacing with other domains, 70-71
 SOM API and, 88-89
- callbacks, 71
- CallStyle, 132
- C compiler, 144
- C emitter, 150-151
- character devices, 293
- check boxes, 40
- child class (subclass), 9
- child processes, 17
- child windows, 34, 180
- ClassData structure, 90
- class declarations, multiple inheritance and, 164
- classes, 7-14.
 abstract, 130
 collection, 11-12
 SOM, 73-87
 definitions of terms, 74-75
 implementing, 81-87
 object interface definitions and, 76-78
 as objects, 75

types and, 74-75
using, 76-80
Workplace Shell, 127-128

classinit, Window metaclass and, 235-236

class libraries.
layered development of, 326-327

class methods
metaclasses and, 102-105
SOM API and, 115-121

client.c, 197-200, 208-212, 226-229, 247-251, 265-272

Client class
in box drawing program, 205, 207
metaclasses and, 263-264
in PMApp program, 196-197
PM controls and, 224
PushButton program and, 246
wrapping PM controls and, 264

client.idl, 197, 207-208, 226, 246-247, 264-265

client procedures, 191

client programs, in SOM, 142-143

clipboard, 40

collection classes, 11-12

colors of PM controls, 236-238

COMBOBOX.C, 281-282

combo boxes, 41, 280-282

COMBOBOX.IDL, 280-281

compile time bindings, 72

constructors, 60-62
example of, 155-158
metaclasses as, 154-155
Window metaclass and, 236

container control, 134
in Workplace Shell, 127

containers, 42

containment relationships, 12-13

controls, 33, 39-42
PM, 222-233
changing colors of, 236-238

CORBA (Common Object Request Broker Architecture) standard, 73.

CreateBracket method, 372

ctrl.def, 289-291

CUA (common user access) convention, 34

definition files, 289

derived metaclasses, 105-108

desktop window (workplace), 180

desktop windows, 34

destructors, in ESOM (Experimental SOM), 408-409

device context, 44-45

device independence, 33

devices
block and character, 293
presentation spaces and, 363-365

dialog boxes, 41

dialog windows, 350

diamond inheritance problem, 13

dispatch resolution, 94-99

DLLs (Dynamic Link Libraries), 29-30
performance considerations for, 30
types of information stored within, 29-30
wrapped controls in, 287-288

DoPress method, 251

DosCreateThread, 17-18

DosEnterCritSec, 18

DosExecProgram, 17

DosExitCritSec, 18

DosExitList, 17

DosGetInfoBlocks, 17

DosKillProcess, 17

DosKillThread, 18

DosResumeThread, 18

DosSelectSession, 19

DosSetExceptionHandler, 27

DosSetPriority, 18

DosSetSignalFocus, 27-28

DosStartSession, 18

DosStopSession, 18

DosSuspendThread, 18

DosWaitChild, 17

drop down menus, 41

dynamic methods, 117

eafile.c, 298-308

EAFFile class, 296-297

eafile.idl, 297-298

eaview.c, 309

encapsulated objects, 5-7

ENTRY.C, 273-274
entry fields, 40, 272-274
ENTRY.IDL, 272-273
enumerations, 165
environment variables, for SOM compiler, 143
error conditions, ObjectPM class library and, 345-348
ESOM (Experimental SOM)
 destructors in, 408-409
 initializers in, 403-405
 metaclass programming in, 418-420
ev, 141
event management, ObjectPM class library and, 354-355
events, 32
event semaphores, 20, 313
exception management, 27-28
explicit metaclasses, 102-105
Extended Attribute File class, 294
extended attributes, 28, 29, 294
 Workplace Shell and, 134

facilities, ObjectPM class library and, 346
FAT (file allocation table), 28
fDataType class, 355-358
file management, 28-29
file system, 292-310
 installable, 293
File System Name Space, 293
file system objects, 293
fills, 45
font management, 33
fonts, 45-46
friends, 162

GenericFile class, 294
GetData macro, 162
gfile.c, 295-296
gfile.idl, 294-295
global variables, 236
GRAPHIC.C, 214-215
Graphic class, 203-205
GRAPHIC.IDL, 213-214
graphics, 43-47, 183, 203-221

ObjectPM class library and, 359-363
graphics programming interface (GPI), 33, 43-44, 203
grClearWindow, 215
grDrawBoxAt, 216
grDrawLineAt, 215-216
grDrawStringAt, 216-217
Grpahic class, 212-213
guardian classes, in ObjectPM class library, 345

handles, 20-21
 window, 34-35, 36
hello.c, 325-328
"Hello World" program, 329-330, 332-340
HPFS (high performance file system), 28

icons, 42
IDL compiler, identifying, 147
IDs, in Workplace Shell, 133
images, 47
implementation file, SOM, 138-142
implementation section, modifying attributes in, 146
include files, non-object, 110-113
inheritance, 9-14
 in C++, 49-60
 argument conversion, 54-55
 definition of object structure, 50-53
 multiple inheritance, 66-70
 polymorphism, 56
 tree and graph inheritance, 66-68
 uses of C++ objects, 50
 virtual functions, 57-60
of implementation, 10-11
of interface, 11-12
multiple, 13-14
SOM API and, 99-102
 window procedures and, 35
initializers, in SOMobjects 2.1 Toolkit, 402-408

ESOM (Experimental SOM),
 403-405
 example of, 409-416
 implementing new classes with,
 405-407
 input focus, 181
 installable file system, 28-29, 293
 installation program, 133-134
 instance method table, 91
 instance variables, 6
 attributes and, 145-146
 interface definition, SOM, 138-139
 Interface Definition Language (IDL),
 74
 OIDL and, 132
 SOM classes and, 76-81
 interprocess communication (IPC), 17,
 19-23
 object strategy for IPC services, 22
 with semaphores, 311-321

 keyboard focus, 43
 keyboard input, 43
 killto.c, 319-321

 labels, 40
 lazy write, 28
 lines, 44-45
 LINK386 command, 144
 ListBox class, 274
 list boxes, 40, 274-278
 LISTBOX.IDL, 274-275
 logical drives, 293

 main.c, 174-175
 main function, 340-341
 main windows, 180
 makefiles
 for multiple inheritance example,
 176, 309
 for PMAppl number one, 189
 for PMAppl program, 201
 for semaphore example, 320-321
 for wrapped controls, 232
 for wrapped controls example
 three, 288-289

 for wrapped controls example two,
 258-259
 markers, 46
 memory, shared, 17, 19, 23-24
 memory management, 23-27
 API for, 26-27
 object considerations and, 24-25
 paging and, 24
 segmentation and, 25
 swapping and, 26
 thunks and, 25-26
 memory objects, 24
 MenuCommand method, 338-339
 menus, 41-42
 message loops, 332-333
 message queues, 32, 36
 message routing, 181
 messages, 36-39
 creation and destruction of, 38
 elements of, 36
 notification, 39
 posting vs. sending, 37
 metaclasses, 153-162
 accessing other data with, 161-162
 automatic derivation of, 107-108
 as constructors, 154-158
 derived, 105-108
 explicit, 102-105, 154
 implicit, 154
 in SOM, 75
 SOM API and, 99-100
 in SOMobjects 2.1 Toolkit, 417-435
 examples, 421-435
 metaclass cooperation frame-
 work, 418-420, 426-427
 single instance metaclass exam-
 ple, 428-433
 trace metaclass, 433-435
 uses of, 154-155
 metaclass incompatibilities, 106
 method descriptor, 102
 method resolution, SOM API and,
 89-99
 dispatch resolution, 94-99
 name-lookup resolution, 92-94
 offset resolution, 90-92
 methods, 6-7

dynamic, 117
overriding, 152
static, 117

mouse input, 43

multi-line-edit (MLE) controls, 40

multiple inheritance, 163-177
in C++, 66-70
tree and graph inheritance, 66-68

class declarations with, 164
enumerations and, 165
makefile for example of, 309
uses for, 163-164

multitasking, 16, 311-312

Mutex semaphores, 18, 20, 312

MuxWait semaphores, 20, 313

M_Window metaclass, 234-235.

name-lookup resolution, 92-94

non-object include files, 110-111

notebook, 42

notebook control, 134
in Workplace Shell, 127-128

notification messages, 39

object constructors, 403

object methods, SOM API and, 113-115

Object-Oriented Programming (OOP), 3-4
benefits of, 4-5
reusing code and, 324-325

object-oriented user interfaces (OOUIs), design of, 375-398
calendar example, 381-397
calendar views, 385-397
object analysis, 381-383
task definitions, 383-384
user tasks and skill requirements, 381-382

"look and feel" aspects and, 376-377

multiple interaction techniques, 377-378

OVTT (objects, views, tasks, and techniques) methodology, 378-381

primary concerns in, 375

UI object model and, 375-376

ObjectPM BaseSet library, 331

ObjectPM class library, 324-374
applications and threads in, 340-345
class libraries included in, 331
conditions and facilities in, 345-348
drawing tools in, 365-374
retained graphics features, 372
event management and, 354-355

Forms Manager and, 355-359
graphics and, 359-363
guardian classes in, 345
"Hello World" program written using, 329-330, 332-340

presentation spaces and, 337-338, 363-365

primary drawing objects in, 338

termination of programs and, 345

windows in, 348-353

ObjectPM Forms Manager, 355-359

ObjectPM WindowManager, 331

objects, composition of, 5-6

offset resolution, 90-92

IDL (Object Interface Definition Language), 132

OO user interfaces, 126

OS/2
characteristics of, 16
organization of, 15-19
units of execution and protection in, 16

overload methods, SOM API and, 101

overriding methods, 152

owner-owned relationships
Mutex semaphores and, 312
of windows, 181

pages, 23

page windows, 350

paging, 24

paint messages, 38-39

Paint method, 337

parent-child relationships of windows, 180

parent class, 9

pathname, 293

patterns, 45

pipes, 21
 PMAPP.C, 187-188
 PMAPP.IDL, 186-187
 PMApp program, 184-202
 classes of, 191-202
 PM controls, 222-233
 changing colors of, 236-238
 PM programs, initialization of, 184
 polymorphism, 12
 pop up menus, 42, 184
 postscr.c, 168-172
 postscr.idl, 167-168
 preemptive multitasking, 16
 Presentation Manager (PM), 7, 32-47,
 178-202
 messages and, 36-39, 181-183
 creation and destruction of, 38
 initiators of, 181
 object model of, 35-36
 window management functions of,
 32
 windows and, 33-35
 presentation space, 44
 presentation spaces, 204-205, 337-338,
 363-365
 PrintGraph method, 371-372
 ProcessErrors method, 347-348
 processes, 16, 311
 child, 17
 termination of, 17
 ProcessEvents method, 236
 protection concept of, 16
 proxy object, 130
 PUSHB.C, 230
 pushb.c, 252-253
 PUSHB.IDL, 229
 pushb.idl, 251-252
 pushbutton, 39-40
 PushButton class, 229-233
 PushButton-piano program, 253-260
 running, 260
 userData attribute and, 254
 PushButtons, 234-260

 Query Manager, 350
 queues, 19, 21-22

 radio buttons, 40
 redispach stubs, 99, 112, 118
 refactoring class hierarchies, 93
 releaseorder: statement, 147-148
 resource compiler, 42
 resources, 42
 reusing code, 324-325
 running a program, 144
 RUNPM.C, 189

 scroll bars, 41
 semaphor.c, 315-318
 Semaphore class, 313
 semaphores, 19-21, 311-321
 event, 20, 313
 Mutex, 18, 20, 312
 MuxWait, 20, 313
 in ObjectPM class library, 341, 344
 posted, 313
 reset, 313
 semaphor.idl, 314-315
 sessions, 18-19
 SetFocus method, 238
 setup string, in Workplace Shell, 133
 shared memory, 19, 23-24
 siblings, 180
 signal focus, 27-28
 signals, exception management, 27-28
 sliders, 41
 SMEMIT environment variable, 140,
 150
 soFindClass, 123
 somAddDynamicMethod, 116, 117
 somAddStaticMethod, 99, 112, 116,
 117
 somAllocate, 115
 SOM API, 73, 88-124
 class manager methods and,
 121-124
 class methods and, 115-121
 metaclasses and
 derived metaclasses, 105-108
 explicit metaclasses, 102-105
 method resolution and, 89-99
 dispatch resolution, 94-99
 name-lookup resolution, 92-94
 offset resolution, 90-92

non-object include files and, 110-113
object methods and, 113-115
subclassing and inheritance with, 99-102
`somapi.h`, 110
`somApply`, 112
`SOMobject`, 185
`somBuildClass`, 112
`somCheckVersion`, 121
`SOMClass`, 403
`SOMClass` class, 75
`somClassDispatch`, 114
SOM classes, 73-87
 definitions of terms, 74-75
 implementing, 81-87
 object interface definitions and, 76-78
 as objects, 75
 types and, 74-75
 using, 76-80
 Workplace Shell, 127-128
`SOMClassMgr`, 110
`SOMClassMgrObject`, 110, 121
`somClassReady`, 117-118
`somClassResolve`, 91-92, 111
SOM compiler, 88-89, 143-144
`somDataResolve`, 111
`somDeAllocate`, 115
`somDescendedFrom`, 121
`somDispatch`, 94-99, 114, 118
`somDumpSelf`, 115
`somDumpSelfInt`, 115
`somEnvironmentNew`, 110, 121
`somFindClass`, 122
`somFindClsInFile`, 122
`somFindMethod`, 120
`somFindMethodOk`, 120
`somFindSMethod`, 120
`somFindSMethodOk`, 120
`somFree`, 113
`SomGetClass`, 92
`somGetClass`, 114
`somGetInitFunction`, 123
`somGetInstancePartSize`, 118-119
`somGetInstanceSize`, 118-119
`somGetInstanceToken`, 119
`somGetMemberToken`, 119
`somGetMethodData`, 119
`somGetMethodDescriptor`, 120
`somGetMethodIndex`, 120
`somGetMethodToken`, 119
`somGetName`, 121
`somGetNthMethodData`, 120
`somGetNthMethodInfo`, 120
`somGetNumMethods`, 120
`somGetNumStaticMethods`, 120
`somGetParents`, 121
`somGetPClsMtabs`, 121
`somGetRelatedClasses`, 123
`somGetVersionNumbers`, 121
`som.h`, 110
`somIds`, 110-111
`somInit`, 113, 115, 152, 185, 186
`somInitCtrl` data structure, 404
`somInit` method, 402-403
`somInitMIClass` method, 99-101, 116, 117
 in SOMobjects 2.1 Toolkit, 417-420
`somInstanceDataOffsets`, 121
`somInterfaceRepository`, 123
`somIsObject`, 112
`SOMLINK`, 141
`somLocateClassFile`, 122
`somLookupMethod`, 92, 120
`SOMMCooperative` metaclass, 418-420
`SOMMCooperativeRedispatched` metaclass, 118, 418-420
`somMergeInto`, 122
`somMethodData`, 119
`somMethodDataStruct`, 119
`somMethodPtr`, 120
`somNew`, 115-116
`somNewNoInit`, 115-116
`SOMObject` class, 75
SOM objects
 creating and using, 79-80
 implementation of, 74
 interface definitions for, 76-78
SOMobjects 2.1 Toolkit, 401-435
 initializers in, 402-408
 ESOM (Experimental SOM), 403-405
 example of, 409-416

implementing new classes with,
 405-407
 metaclass programming in, 417-435
 examples, 421-435
 metaclass cooperation frame-
 work, 418-420, 426-427
 single instance metaclass exam-
 ple, 428-433
 trace metaclass, 433-435
 SOMobjects Toolkit, 73
 somOverrideMTab, 118
 somOverrideSMethod, 117
 somParentNumResolve, 111
 somPrintSelf, 114-115, 115
 somRegisterClass, 123
 somRegisteredClasses, 123
 somRenew, 115-116
 somRenewNoInit, 115-116
 somRenewNoZero, 115-116
 somResolve, 90-92, 111
 somResolveByName, 111
 SOM_Scope, 141
 somSelf, 141
 somSetClassData, 118
 somSetMethodDescriptor, 120
 somSubstituteClass, 117, 122-123
 somSupportsMethod, 121
 somtypes.h, 110
 somUninit, 115, 152, 185
 somUnloadClassFile, 123
 somUnregisterClass, 123
 som.xh, 110
 speaker.c, 173-174
 Speaker class (example), 166-177
 speaker.idl, 172-173
 STATIC.IDL, 278-279
 static member functions, 71
 static methods, 117
 StaticTest objects, 278-280
 subclassing, 9, 74
 SOM API and, 99-102
 superclass, 9
 System Object Model (SOM), 4, 7,
 72-124.
 attributes in, 145-151
 C emitter option, 150-151
 identifying the IDL compiler,
 147
 instance variables or attributes,
 145-146
 modifying attributes, 147
 modifying attributes in imple-
 mentation section, 146
 releaseorder: statement, 147-148
 introduction to, 72-75
 terminology, 74-75
 metaclasses in, 417-435
 examples, 421-435
 metaclass cooperation frame-
 work, 418-420, 426-427
 single instance metaclass exam-
 ple, 428-433
 trace metaclass, 433-435
 overriding methods in, 152
 simple example of, 138-144
 client program, 142-143
 emitter framework, 140-141
 environment variables, 143
 implementation file, 138-142
 interface definition, 138-139
 invoking SOM methods, 142-143
 macros, 141-142
 SOM compiler, 143-144
 targetObj, 122
 templates, C++, 64
 test3.c, 201, 220-221
 text, 40
 display of, 45
 threads, 16, 17-18, 311-312
 in ObjectPM class library, 341-345
 priority of, 18
 thunks, 25-26
 timeout.c, 318-319, 321
 timers, 22
 value sets, 41
 virtual functions, in C++ objects, 57-60
 void somFree, 113
 void somInit, 113
 void somUninit, 113
 wBitmap class, 366

wButtonClickMsg class, 354
wConnectedGraphicsTool class, 365
wCoord class, 359-362
wDevice class, 363-364
wDialogWindow class, 389, 350
wDimension class, 362
wDynamicPicture class, 372
wDynamicPictures class, 374
wField class, 355
wFont class, 338, 365-366
wFormWindow class, 355
wFrameWindow class, 349
WinCreateStandardWindow, 34
WinCreateWindow, 34
WinDispatchMsg, 185
window.c, 193-196, 240-246
Window class, 32
 in PMApp program, 192
 PM controls and, 224
 for PushButtons, 234-238
WindowFromID, 241-242
window handles, 34-35, 36
WINDOW.IDL, 192-193
window.idl, 238-240
Window metaclass, 234-236
window procedures, 33-35, 182-183
 inheritance and, 35
Windows, 16
windows
 child, 34
 creating, 34-35
 desktop, 34
 desktop (window), 180
 dimensions of, 34
 main, 180
 in ObjectPM class library, 348-353
 owner-owned relationships of, 181
 painting, 38-39
 parent-child realtionships, 180
 standard, 34
wListRegion class, 359
WM_BUTTON1CLICKED, 196
WM_CCONTROL, 224
WM_CLOSE, 38
WM_COMMAND, 224, 262
WM_CONTROL, 262-263
WM_DESTROY, 38
wMdiDesktop class, 349-350
wMdiDocument class, 350
wMemoryBitmap class, 366
wMemoryDevice class, 364
wMessageBox class, 353
WM_PAINT, 38-39
WM_QUIT, 38
Workplace Shell (WPS), 125-134
 as an OO user interface, 126
 extended attributes and, 134
 IDL and, 132-133
 IDs in, 133
 levels of compliance with, 133-134
 OIDL and, 132
 setup string in, 133
 SOM and, 73
writing programs for, 127-132
 classes, 127-132
 WPAbstract, 130-131
 WPFileSystem, 128-130
 WPObject, 128
 WPTransient, 131-132
WPAbstract, 128, 130-131
wpAddObjectGeneralPage, 128
wpAddObjectWindowPage, 128
wpAddSettingPage, 128
wPageWindow class, 350
WPClassManager, 128
wpClose, 128
wpCnrInsertObject, 128
wpCnrRemoveObject, 128
wpCnrSetEmphasis, 128
WPCountry, 130-131
WPDataFile, 129
WPDisk, 131
wpDraggedOverObject, 128
wpDragOver, 128
wpDrop, 128
wpDroppedOnObject, 128
wPen class, 365
WPFileSystem, 128-130
WPFolder, 128
wpFolder, 128
wPicture class, 372, 374
wPictureList class, 372
WPKeyboard, 130-131
WPMouse, 131

WPObject, 128
wPointl class, 362
WPPrinter, 131
wpPrintFiffFile, 130
wpPrintMetaFile, 130
wpPrintPrinterSpecificFile, 130
wpPrintUnknownFile, 130
WPProgram, 130
wpQueryAttr, 129
wpQueryEASize, 129
wpQueryLastAccess, 129
wpQueryLastWrite, 129
wpQuerySize, 129
wPresSpace class, 338, 363-364
wpSetAttr, 129
wpSetRealName, 129
wpSetType, 129
WPTransient, 131-132
wrap1.c, 232
wrap2.c, 258
wrap3.c, 287
wrapping PM controls, 222-224, 261-291
application objects, 282-286
Client class and, 264
combo boxes, 280-282
entry fields, 272-274
list boxes, 274-278
putting wrapped controls into a dynamically linkable library (DLL), 287-288
StaticTest objects, 278-280
wRectl class, 362-363
wStdErrorDialog class, 348
wSubPicture class, 372, 374
wTextPen class, 338, 365
wWindow class, 348-349
wWorkplaceObject class, 348-349

Z order, 34, 180

OBJECTS for OS/2®

Scott H. Danforth, Paul Koenen, and Bruce Tate

Straight from IBM's OS/2 software development labs comes this timely reference: A complete book that addresses Object-Oriented Programming (OOP) in the framework of OS/2. This unique reference teaches OOP not strictly as a tool, but as a way of thinking. The authors have synthesized years of experience in software development under OS/2 into a practical collection of philosophy, tips, and analysis for software developers in search of advanced techniques.

The book begins with a concise guide to basic OOP principles of object definition and function, using simple examples like a data stack called StackofPizzas. The authors follow an incremental progression of more sophisticated OOP concepts, through class, inheritance, and initializers and destructors, all in the context of familiar OS/2 features like Presentation Manager (PM), Workplace Shell (WPS), and the SOM API.

The text is packed with insight, humor and most of all, usable advice on OOP:

- Using PM as a traffic cop
- Controlling and manipulating graphics
- Implementing new classes with initializers
- Working within ObjectPM architecture and OVTT Methodology

Most of the examples are written in C with SOM. The book comes with a disk containing pertinent examples in reusable C++ code. This advanced reference enables programmers to better visualize and organize their work within the object-oriented environment, resulting in tighter, easier-to-maintain code produced at considerable savings in time and money.

Scott H. Danforth is a Development Staff Engineer at IBM, where he works with a group responsible for the kernel of SOM, the IBM System Object Model. He has implemented and applied for numerous patents related to object-oriented programming.

Paul W. Koenen has nine years of experience as a systems evaluator and integrator in the telecommunications and computer industries. He works with object technology at IBM, where he is currently designing development aids such as portable class libraries and frameworks to aid developers in building sophisticated portable applications.

Bruce Tate has held many positions with IBM, including Database Manager development. He has received several patents related to visual query systems. He is currently developing visual programming tools for the SOM environment.

*OS/2 is a registered trademark of IBM Corporation.
OS/2 accredited logo is a trademark of IBM Corp. and
is used by Van Nostrand Reinhold under license.*

Cover photo by Michel Tcherevkoff, The IMAGE Bank.

ISBN 0-442-01738-3

90000



9 780442 017385

Van Nostrand Reinhold
115 Fifth Avenue, New York, NY 10003