Object-Based Computing is published monthly by
Information Technology Solutions, Inc.
400 West Erie, Suite 301
Chicago, IL 60610

FIRST CLASS

ISSUE #3

Object-Based Computing

A resource for the
NeXT Programmer

WorldClock will be reviewed in the upcoming issue of NeXTWORLD

SEapplication
SEDirectory
SEFile
SEFileSystem
SENode
SEOS

Threaded Technologies
339 Wiltsee Avenue
Loveland OH 45140
(513) 677-2106

Trillium Sound Research, Inc.
(403) 284 - 9278
manzara@cpsc.ucalgary.ca
products:
Text-to-Speech Kit

T.Y.C. Software, Inc.
(708) 515 - 0668
coco@ihcoco.att.com
products:
SCSI

Versant Object Technology
(415) 329 - 7549
geoff@osc.com
products:
Versant ODBMS

VNPSoftware
(617) 661 - 4292
info@vnp.com
products:
AccessKit

Workstation AG
+41 91 505094
rgi@wag.chation
products:
DBDragger
FuncEdit
Knob

ZGDV Darmstadt
+49 06151 293863
essmann@igd.fhg.de
products:
Realtime Voice and
Video Communication

ZippyTech
(412) 421 - 9588
ztech@well.sf.ca.us
products:
InetObjects

REVIEWS:

TOOLS AND OBJECTS

Future issues will include a descriptive list of tools and objects available to NeXT programmers. Reviews of two or three objects or programming tools will be featured each month. SEND A COPY of your object or tool with associated literature and documentation for review to:

Edward Shelton, Editor
Object - Based Computing
c/o ITSolutions
400 West Erie, Suite 301
Chicago, IL 60610

We apologize in advance that submissions for review cannot be returned.

We would be happy to list your company—
please let us know how you would like your company listed and what objects are available.

products:
ImageScrollView / MiniView

Digital Composition Systems, Inc.
(415) 673 - 5322
gary@dreyfuss.portal.com
products:
SpreadSheetVue

Digital Tool Works
(617) 742 - 4057
lexcube!equation@bu.edu
products:
Equation

Doberman Systems
(801) 944 - 4329
doberman!mike@esunix.sim.es.com
products:
Simulation Kit

Frontier, Inc.
(800) 448 - 6398
frontinc!john@uunet.uu.net
products:
Clock / Alarm Object
C-Tree Adaptor
DateField
Dials & Gauges
DBKit Companion Objects
Money & Number Field
System Event Timing Object
Time Bomb Object
Time&Billing Timer Object
Validate Text Field

Hot Software
(617) 252 - 0088
info@hot.com
products:
BarCodeKit
SerialPortKit

Impact Software
210 Lake Street
Ithaca, NY 14850
(607) 277-8623
email: impact@impact.shamen.com

Insight Software
(503) 222-2425
info@insight.com
products:
ImageView
ScannerKit

Instituto Balseiro
Universidad Nacional de Cuyo
+54 (944) 61013
roverel@cab.edu.ar
products:
SimulationKit

Itasca Systems, Inc.
(612) 851 - 3155
terry@itasca.com
products:
ITASCA NeXTSTEP Client
ITASCA ODBMS

KCW Consulting
(703) 938 - 4152
curt@kcwc.com
products:
PhoneTones

Lamb Software Design
41-22 735.96.03
lamb@lsd.ch
products:
LSDDistMatrix

Liveware Corporation
(303) 484 - 7607
info@liveware.com
products:
LockOut Object Set

Metaresearch, Inc
(503) 238 - 5728
75270.1262@compuserve.com
products:
Color Digital Eye Objects
SoundWorks Objects

Mouthing Flowers.
(206) 325 - 7870
products:
slugg@mouthers.wa.com

Nightshade Software
(403) 492 - 9343
nightshade@niagara.ucs.ualberta.ca
products:
FilteredFields
GraphMe
NiftyButton

Objective Technologies, Inc.
Suite 1502
7 Dey Street
New York, NY10007
(212) 227-6767
info@object.com
ChooserPalette
GraphPalette
MathPalette
OTDBKit
OTI Extended Text Object
OTI Tabular Text Example Objects
OTString Kit
SmartFieldPalette

RDR, Inc.
Suite 350
10600 Arrowhead Dr.
Fairfax VA 22030
(703) 591-8713
email: info@rdr.com
RDRGadgets
RDRImageView
RDRSelector
RDRSound
RDRSwitchView

Solutions Unlimited
(514) 849 - 3631
darcy@solutions.ca
products:
FloatingImage
SUMatrix

The Stepstone Corporation
(203) 426 - 1875
hotline@stepstone.com
products:
ICpak 101

Stream Technologies, Inc.
+358 0 4357 7340
info@sti.fi
products:
Object Store

Suite Software
(619) 698 - 7550
Customer_Support@suite.com
products:
EnvVar

Object Catalogue

a monthly update
Objects, palettes, and other tools for
NeXT Developers:

ABComputers
(401) 521 - 2829
products:
ProFuse Rule

Anderson Financial Systems, Inc.
(215) 653 - 0911
kit@afs.com
products:
AFSApplication
AFSButton
AFSEventManager
AFSFindPanel
AFSForm
AFSHelpPanel
AFSLookupsPanel
AFSMatrix
AFSMouseCalcPanel
AFSReportPanel
AFSScanPanel
AFSTextField
AFS3DButton/Graph
AFSWindow,AFSPanel
TradeKit

Archetype, Inc.
(617) 890 - 7544
products:
Document Engine

BenaTong
(614) 276 - 7859
benatong@count0.uucp or
chuck@kiwi.swhs.ohio-state.edu
products:
Serial Solutions

Black Diamond Tech.
(312)554 - 5000
products:
Lotus Realtime Object Kit

Black Market Technologies, Inc.
(718) 522 - 5090
info@bmt.gun.com
products:
GridPalette
Multicell

Conextions, Inc.
(508) 689 - 3570
edk@conextions.com
products:
Network Palette

Dept. of Radiology
Ohio State University Hospital
(614) 447 - 9194

This application to application communication channel is provided by the Mesa Object Library Interface (MOLI), and it consists of three classes:

- MesaListen,
- MesaObject
- MesaView

The MesaListen class is much like the AppKit's Speaker/Listener class, it can subscribe to ("Listen" for) specific conditions in the spreadsheet, and when they occur a message is sent to the MesaListen.

One example would be if the prices of gold were updated in the spreadsheet from one external application, a second application running on the gold traders machine could inform Mesa that it would like to listen in and be told when the price of gold in London was 1% higher than in New York. This is quite efficient since the custom application will remain dormant until it is informed of the change.

These actions are all facilitated through the built in Mesa function signal().
One would enter into a cell:=signal(zurich_price / 101% > london_price, "BuyZurich", zurich_price)
Mesa would then send the price of gold in Zurich when it is 1% greater than the price in London to the "port" named BuyZurich.

In our custom application we would subscribe to the BuyZurich "port" when we initialized a subclass of MesaListen. The call would look like

```
goldListener = [[GoldListen alloc]
initToPort:"BuyZurich"],
```

with GoldListen a subclass of MesaListen. Our GoldListen object would have to implement one method -gotMessage:num:forUpperRow:upperCol: lowerRow:lowerCol: that would be called when the signal was triggered.

We see that it is easy enough to get a message from Mesa, but how do we manipulate information inside a worksheet?

Well the MesaObject takes care of that, it is the class that allows connections into worksheets. Once one connects to a worksheet most any action that can be done by a user can be carried out from within the MesaObject.

There are method calls for doing most everything from creating new worksheets to getting values from a range of cells to saving a worksheet under a new name. When a MesaObject is created it must be linked to a worksheet, thus to create a new MesaObject and open a worksheet we write:

```
sheet = [[MesaObject alloc] initToWorksheet:"/files/foo.Mesa"]
```

Now we can send messages to the worksheet through the sheet id. If we wished to force a re-calculation we would send [sheet recalc];, if we would like to make the worksheet the current front window we would send [sheet makeKeyAndOrderFront];. Likewise we can place data into and pull data out of any worksheet, this creates a simple and extensible way to create real time data feeds into MESA worksheets.

The last class that MOLI provides is MesaView, it creates "views" into worksheets. These views can consist of both ranges of cells, or of created graphs. Because these views are actually part of the worksheet they can be edited just as though the user was editing them in Mesa itself. They can accept drag and drop color and fonts just like Mesa itself. The custom programmer does not have to worry about where a color tile has been dropped, MesaView takes care of all the details. The only thing that the custom programmer has to do is create a view into the worksheet with the:

```
GraphView = [[MesaView alloc] initFrame:&re toSheet:tmp toGraph:"Graph Name"]
```

Now when the data or graph is changed in the worksheet, the corresponding

MesaView will also be updated and the user will see all changes automatically. All the work has been done in the MesaView object, imagine having to write one line of code to create a graph from some data. MesaView provides this and more, just like the AppKit all of the work is done for you, all you have to do is instantiate the various objects and have them do all of your work.

Through the use of object oriented programming with Objective-C we can create extensible applications that allow other programs to communicate in a seamless and simple manner to each other. Mesa goes even one step further allowing additions that appear inside the program itself, but that is beyond the scope of this article.

I hope that this has given some insight into the beauty of object oriented programming on the NeXT. With NeXTSTEP 3.0, NeXT has given us even more tools to further inter-application communications. I believe that MOLI is a large step forward and away from static programs toward extensible programs that allow a user to customize the way in which a commercial application works to make them more productive.

If you have any questions about Mesa and MOLI please contact me at:

jaeger@athena.com
or give us a call at
Athena Design, Inc.
(617)-734-6372

encourages re-use rather than re-invention.

Exercises

1.) Bring up Interface Builder and double click the Class Editor window by double clicking on the "Classes" tool-kit icon in the lower left window.

Compare this with the Inheritance structure described above. What happens when you try to use the "Instantiate" selection under the pop-up list labeled "Operations". Try instantiating an Object. Why can't you instantiate a View? Try sub-classing a View and call it, "MyView". If you drag a custom View from the palette and then inspect its attributes what do you see? Can you make the custom view an instance of the MyView class?

2.) What are some sample inheritance structures you might create? What would an inheritance structure for banking objects such as saving accounts, checking accounts, and banking transactions look like? What fields would they have in common? What fields would be unique? What would an inheritance structure for electrical components like wires, resistors, capacitors and transistors look like?

Editor's note --

We would be very interested in your positive and/or negative comments regarding our series on OBC -- please let us know what you think and what you would like to see in the future. We stand ready to deliver the information and suggestions that the programming community is waiting for... please let us know what would help you.

Thank you,

Ted Shelton,
Editor Object-Based Computing
ems@its.com
800 / 394 -4487

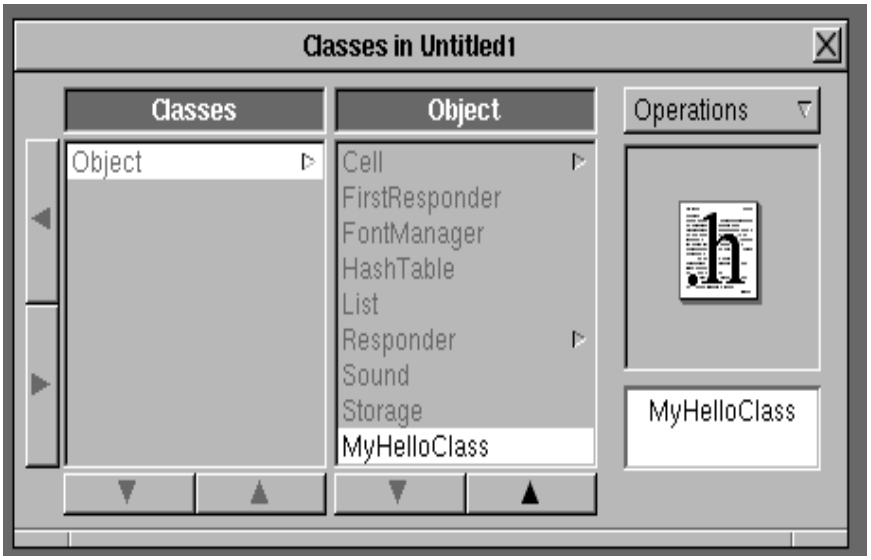


Figure 6.

Object Oriented Applications:

What does that buy you?

Dirk Fromhein, Athena Design, Inc.

We've been hearing about object oriented programming for some time now, but what does it really mean? And most importantly what does it give you?

Because NeXT uses Objective-C many of the things that make a program actually run are done when you start the program, not when you compile the program. This buys a great deal of flexibility that is just not possible with other languages and other systems. When an applications developer uses a more traditional language such as C or C++ to create an application the program is essentially done when the final compile and link is run and the product is shipped to a customer. While there are ways to extend existing programs, by the nature of these languages it will never be very clean.

Here Objective-C shines—one can load additions to the program at run time and never know whether the feature was part of the original program or added later. Because of this the various objects that make up the program can have multiple connections to other objects. The added modules act and are called with the same mechanism that the functions in the already existing program are. This behavior is known as ***run time binding***, one of the most powerful but under used features of Objective-C.

One application that makes extensive and correct use of run time binding is Mesa, a traditional spreadsheet that makes full use of all of the exclusive features that makes a true NeXTSTEP program. Every copy of Mesa comes with a run-time library that allows a custom program to seamlessly communicate in an object oriented manner with the internals of Mesa.

This allows custom programmers to fully utilize the calculation and graphing prowess of this spreadsheet from within a custom application. Custom programmers must merely subclass the given objects to manipulate Mesa and any open worksheets in much the same manner as a user actually performing the actions would be able to.

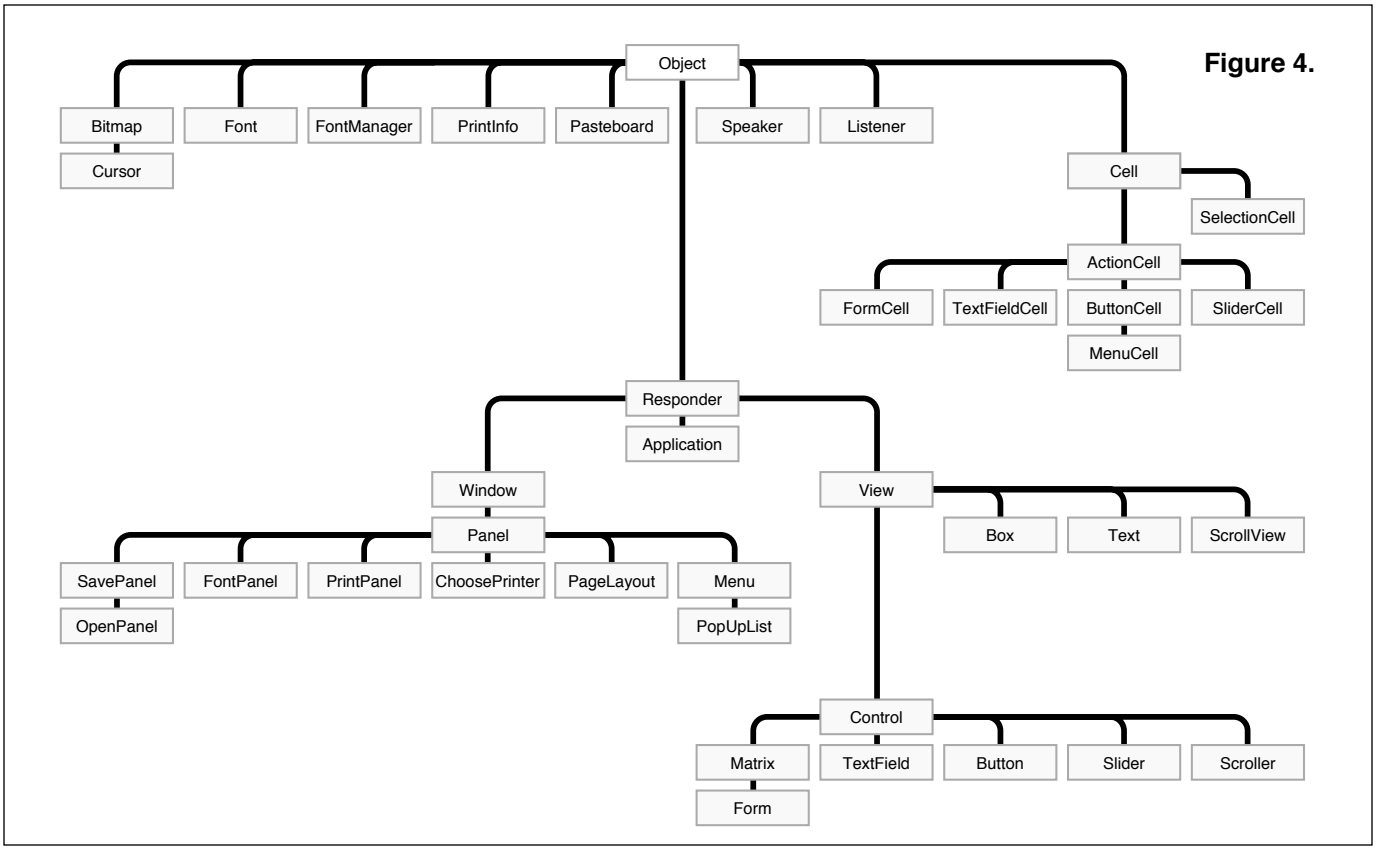


Figure 4.

*Inherited from View*NXRectframe;
NXRect bounds;
id superview;
id subviews;
id window;
struct __vFlags vFlags;
*Inherited from Control*int tag;
id cell;
struct _conFlags conFlags;
Declared in Button(none)

This shows the direct ancestors of the Button class. Below that are the instance variables for each of these levels. The Button class hierarchy might look something like the following figure.

If you create a button and need to know the bounds of it, you could access the bounds of your own button as the variable "bounds". The bounds would actually come from the state variable of the View class. Information, such as bounds, is used over and over again every time you manipulate any on-screen objects. Re-using this code increases productivity and

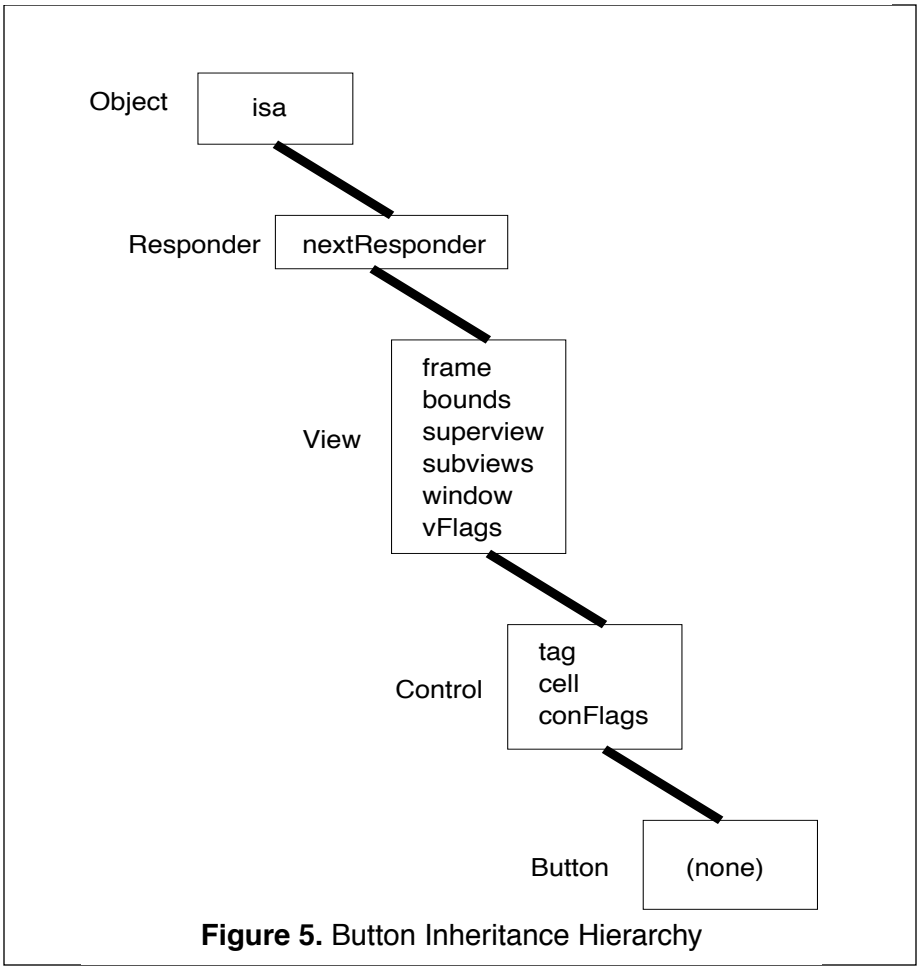
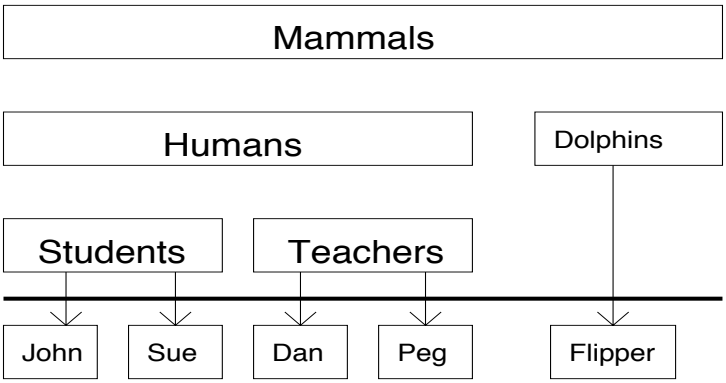


Figure 5. Button Inheritance Hierarchy

400 WEST ERIE, SUITE 301
CHICAGO, IL 60610

Figure 3.



ber they are as different as a car (instance) and a car producing factory (class). This can be difficult for people unfamiliar with Interface Builder because both classes and instances are represented by small windows that are very close together and look similar. Both classes and instances of objects can be changed. Changing the factory that produces cars could change some aspect of every car produced by that factory. But changing one single car would not have a direct effect on all other cars of its class.

We will use some new terminology to describe the relationship between classes. In Figure 3, "Humans" are considered a super-class of "Students" and a sub-class of "Mammals."

When a new class is created, all the instance variables as well as methods defined in its super-class are inherited. It in turn inherits all the instances and methods of all the super-classes above it. This is one of the principal ways that programs are reused in object oriented programming. You find an object in the inheritance tree that most closely matches your needs and then make extensions to it. There are ways to create new structures as well as change existing structures.

Now, take a close look at the inheritance tree for the NextStep Application Kit. This is a tool-kit of objects that can be used to build applications. The structure for this tree is given in the Figure 4 on the next page.

Understanding the structure of the application kit is necessary to be able to use the application kit and extend the user interface objects to meet your needs. At the top of the structure you see a box titled "Object". This is the most generic object in the tree. It has the fewest specialized characteristics of any of the Application Kit objects. Any characteristic of the object class will be shared with all other appkit objects. Directly below the object class is the Responder class. This consists of all objects that can respond to user-generated events such as pressing the mouse and typing on the keyboard. To the lower right of the Responder is the View class. All objects on the screen are a subclass of the View class. Below the View class is the Control class. All classes which are subclasses of the Control will respond to events by sending messages directly to other objects. They can serve as controller inputs to our custom objects.

One example of a control is the Button class, Figure 5. If you take a closer look at the Button class you will see that most of the characteristics of the Button class are not created in the object itself but arise from its location in the inheritance tree. Let's take a closer look at the documentation NeXT provides about the Button class. It can be found in the following path of the NeXT on-line documentation.

INHERITS FROMControl:

View: Responder: Object
INSTANCE VARIABLES

Inherited from Object Class isa;
Inherited from Responder
id nextResponder;

(or occurrence) of the object. The only way to read or write the values of these variables is to use one of the access methods provided with each object. This is often abbreviated to just "method" which refers to the way a program accesses the structures inside an object.

Benefits of Encapsulation

Encapsulating data gives the programmer control of the data types that are passed to the objects. Since the NextStep Objective-C compiler has type-checking built into its messaging, it is easy to catch data type mismatches early in the design process when they are much easier to isolate. This dramatically cuts down time spent with the debugger and greatly enhances the reliability of the final programs.

Once the set of messages that an object can receive is defined, you can then design the object's interface. If, at a later time, you find another more efficient data structure you would like to use inside the object, it can be changed internally and not effect the interface. This means that you can make updates without affecting the other components of your system.
After you have an object that performs some specific function, you can then create a symbolic abstraction of that object using a "view" of it on the screen. The connections to the object can then be done with NextStep's connection based programming tools.

Views of Subroutine Libraries

By creating these views, a user can now graphically manipulate the object and integrate it with other objects. When users make a connection to an object, NextStep asks them which of the access methods they would like to use. This means that non-programmers can start using tools that were previously only accessible to a very small group of experienced programmers. And since it is up to the creators of the objects to validate the correctness of the access methods, a much larger group of people will be able to use the objects without having to debug them.

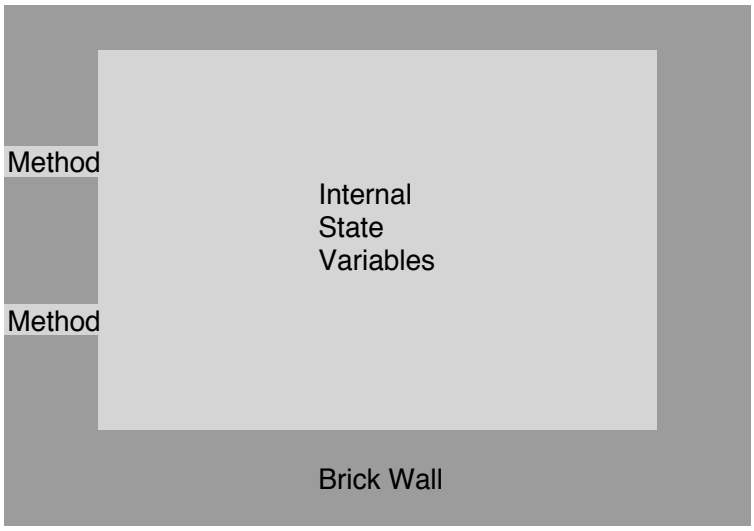


Figure 2. A suggested mental image of an object.

Imagine what the world would be like if the only people who could drive a car were the people who could assemble an internal combustion engine. There certainly wouldn't be the traffic problems we have today. Cars have a simplified user interface: a steering wheel, a brake and a gas peddle. Object-based computing platforms give us these same advantages: easy to use interfaces to traditional subroutine libraries. This helps both the creators and users of a subroutine library system. It helps the creators because the size of their potential user base increases dramatically; it helps users because the amount of training they have to go through decreases dramatically. The number of people who are creating applications with these graphic subroutine libraries will grow exponentially for the next several years.

Before object-based computing people had to use a manual to find out all the arguments to a subroutine, declare all the arguments with the correct data types, pass these in the correct order to subroutines and then, if any of them were wrong, start learning how to use the debugger. Now you can just point to a source object, drag a line to a destination object and click on the message to be sent. Correctness is enforced by the user interface.

INHERITANCE

Before discussing inheritance, it is prudent to clarify the distinction between an instance of an object and a class of objects. The characteristics of a class of objects, like a Ford Truck, is determined by the factory which creates the trucks. If one had a Ford Truck, one would have an instance of the truck. Similarly, there are classes of objects which create new instances of objects. And these are naturally called factory objects.

The second technique used in object oriented programming is Inheritance. Whenever a new class of objects is created, it is always created relative to other classes of objects. These classes fit together into a "tree" of object classes.

Sample Inheritance Tree

The structure of an inheritance tree is very similar to an evolutionary tree (see Figure 3). The most general class is at the top, and each class that has a group of common characteristics would be a lower class. An important point to note is that you can create an instance from any level in the tree. When you think of the difference between an instance of an object and the class of an object, remem-

Object-Based Computing: Part 3

Dan McCreary, Integrity Solutions

This is the third of six articles on Object-Oriented programming. If you have missed the first two we would be happy to send you the earlier articles. For subscribers this service is available at no cost. All others must send $2.50 per issue for shipping and handling.

ENCAPSULATION AND INHERITANCE

You have just been on a whirl-wind tour of the steps required to create your first object. Now, take a step back and look at the details of object creation more closely.

The late 1980's saw the migration of programs traditionally found on large mainframe systems to personal computer platforms - word processors, data bases and spread sheets being the most popular. As these programs became more commonplace, new software companies began entering the personal computer software market with less expensive and more powerful versions of the standard programs. To be competitive, software had to be feature-rich and still run on systems with severe memory limitations. As a result, the marketing divisions of software firms started to promise new versions of products with a large number of new features but which would still run on computer systems with very little memory. Software developers found they couldn't deliver the programs on time. The software had so many features it was called "bigware" and it took so long to develop that it became known as "lateware"[1].

What software developers were finding was that as they tried to add new features they would introduce new bugs. And trying to fix those bugs introduced additional bugs. The result was a drastic decline in software productivity proportional to the size of the project (see Figure 1).

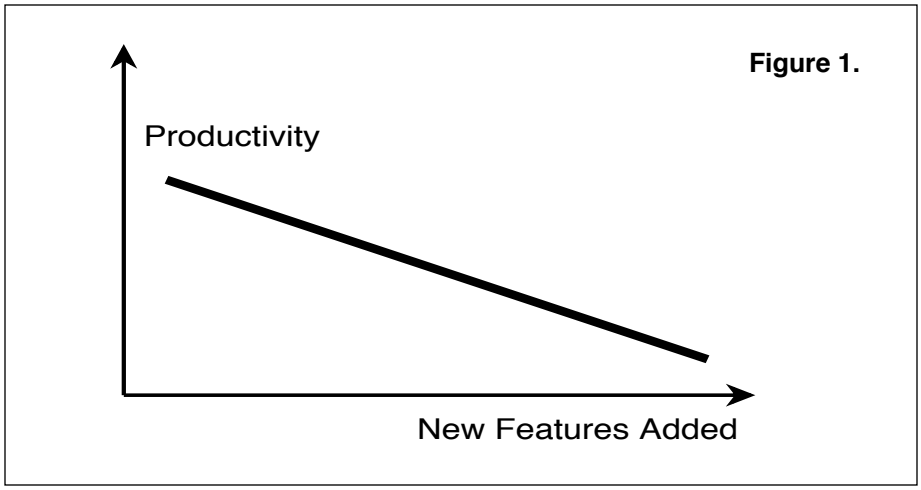


Figure 1.

To solve this problem, the ways in which programs are created and integrated together to create a software system must be examined—and not just at the superficial level of software productivity tools, but rather at the very foundation of software creation.

Researchers have been studying software design techniques for many years. Much of the pioneering research in new programming environments at the Xerox Palo Alto Research Center has shown that there are radically different approaches to creating software that challenge traditional programming methods. One family of techniques that gained a great deal of popularity in the mid-eighties is currently referred to as, "the principals of object oriented programming." These techniques will be introduced to you in this and later chapters. The first technique, known as, "encapsulation" covers the creation of new objects.

ENCAPSULATION

Encapsulation goes by various names. Computer scientists often refer to it as "information hiding" or "data abstraction." In general, these terms imply grouping data and the procedures to access that data together in the same unit. This unit is called an "object." Here are some rules about how you can access the data in an object. One rule is that if you develop an object, you only let people see or change your internal data using the procedures provided with that object. This means the creator of the object has ultimate control over how users access this object and change its internal states. The creator of an object is responsible for creating and testing methods a user will need to read the state of an object as well as checking the validity of inputs used to change the state of an object. The creator of an object (rather than the user of an object) is responsible for the completeness and correctness of all access methods.

To create a mental image of the objects, it might be helpful to imagine a box with a thick brick wall around it, as in Figure 2 on the following page.

Inside the box are the object's data structures: integers, floating point numbers, strings, and other more complicated structures such as linked lists or directed graphs. These are the "instance variables". They hold the state of the object. The word, "instance" is used because there is a different group of these variables associated with each instantiation

1. Newsweek, April, 1989.

Object-Based COMPUTING

Published monthly by
INFORMATION TECHNOLOGY SOLUTIONS INC.

A forum for developments in obejct-based computing in the NeXT community.

NOVEMBER, 1992
ISSUE #3

Volume 1, Number 3, November 1992

editorsDesk

Our Third Issue!

Despite all efforts, we have not yet printed an issue on schedule. We will continue to iterate toward one issue per month. Please bear with us as we learn to produce a regular newsletter, while also providing clients with NeXT systems administration, training and programming services.

Rest assured, your "annual" subscription is being treated as a "12 issue" subscription (not one calendar year).

Since a goal of this newsletter is to give you a picture of what's developing in the object-based community, the onus is on us to stay up-to-date and print information you aren't getting any place else. The newsletter focus is technical, for object oriented computing specialists. Please let us know what information will be of the most value to you.

We feel articles that present or explain a product are valuable. In this issue, Dirk Fromhein of Athena Design has done a wonderful job of opening the door onto their product, Mesa. Although Mesa looks on the outside like a standard spreadsheet application, under the hood it is a powerful object-oriented program which can be integrated into custom applications. Dirk explains some of the ways that a programmer can gain access to Mesa's features through the Mesa API called MOLI.

With this kind of program, object oriented design wins... 500% over all other programming paradigms currently available to computer buyers. The ability to tightly link programs together—both commercial applications from multiple vendors; and commercial and custom applications—is the feature of object-oriented design and programming that has the strongest impact on how we work. I hope all commercial software developers, both NeXT specific and otherwise, take note of Athena's product design.

Someday even the advanced features in the NeXT interface such as "Services" and object-linking will seem trivial next to the multi-vendor integration that our software packages will naturally be capable of.

Until then I will continue cutting and pasting between applications and trying to point the way toward the future utopic computing environment by trying to get all of the power possible out of my NeXT (in order to get this newsletter out on time...).

Yours,

Ted Shelton, Editor
Object-Based Computing
ems@its.com

Submissions Encouraged!

Thank you for letting us know what kind of articles you'd like to read. We are also looking for articles to print... so please consider sharing some of your experiences with the NeXT community.

thisIssue

A little about the articles in this issue.

The third in a series of six articles on object-oriented computing appears in this issue. Written by Dan McCreary of Integrity Solutions, Inc., this is a glimpse at his new book on Object-Based Computing with a focus on the NeXT.

From Dirk Fromhein, Athena Design, Inc., we have an article on object-oriented programming with a specific discussion of Athena's Mesa spreadsheet and how it can be used by programmers.

contents

400 WEST ERIE, SUITE 301
CHICAGO, IL 60610