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

FIRST CLASS

PagerKit and PagerNX

Chicago based Information Technology Solutions has announced the availability of PagerKit and PagerNX for NEXTSTEP.

Providing a complete solution to alphanumeric and numeric paging, the PagerKit gives the NEXTSTEP programmer both a command line and object kit access to a fully featured pager engine, pager definition methods, and person record.

In addition, a full NEXTSTEP application for sending pager messages is included.

The total price for the combined PagerKit and PagerNX.app is $249. The PagerNX application is available separately for $149.

Michael Manning, ITS programmer responsible for the PagerKit says, "I wanted to have a robust set of tools for connecting to an on-line service and exchanging simple types of messages.".

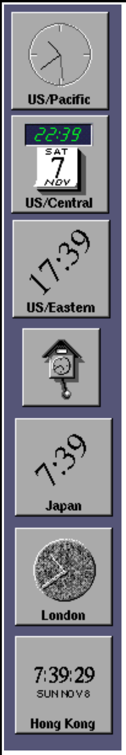
Michael notes that the "chatman" module used to set up conversations with Pager companies' on-line systems can be adapted to many other purposes.

"For one project", added Michael, "we used 'chatman' to connect to a GEISCO mailbox and retrieve simple email messages into an automated message router we had built using NEXTSTEP."

Michael Manning can be reached at mmanning@its.com

Please write or call for further information.

WorldClock from ITS



Just $45
...and you've got all the time in the world...
800 394-4487

■

INFORMATION TECHNOLOGY SOLUTIONS, INC
400 WEST ERIE, SUITE 301, CHICAGO, IL 60610

Information Technology Solutions

400 West Erie, Suite 301
Chicago, IL 60610
(312) 587-2000
info@its.com

PagerKit • StringSurgeon • Yahv.app (Header Viewer)

Insight Software

(503) 222-2425
info@insight.com

ImageView • ScannerKit

Itasca Systems, Inc.

(612) 851-3155
info@itasca.com

ITASCA NEXTSTEP Client • ITASCA ODBMS

Joe Barello Consulting, Inc.

(212) 580-8366
(212) 580-1857 FAX
info@jbc.com

Tab Palette

Kapiti Limited

(071) 587-0033
(071) 735-3765 FAX

FIST: Complete Dealing Room System

KCW Consulting

(703) 938-4152
curt@kcwc.com

PhoneTones

Lamb Software Design

41-22 735.96.03
lamb@lsd.ch

LSDDistMatrix

Liveware Corporation

(303) 484-7607
info@liveware.com

LockOut Object Set

Look Glass Design, Inc.

(604) 739-3131
(604) 739-3008 FAX

LDGCreditCardAuthorization • LGDModem • LDGSerialPort

Metaresearch, Inc

(503) 238-5728
(503) 232-6323 FAX
info@metaresearch.com

Color Digital Eye Objects • SoundWorks Objects

Mouthing Flowers.

(206) 325-7870

slugg@mouthers.wa.com

Nightshade Software

(403) 492-9343
nightshade@niagara.ucs.ualb erta.ca

FilteredFields • GraphMe • NiftyButton

Objective Technologies, Inc.

Suite 1502
7 Dey Street
New York, NY10007
(800) 3-OBJECT
(212) 227-6767
(212) 227-3567 FAX
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
(703) 273-8170 FAX
info@rdr.com

RDRGadgets • RDRImageView • RDRSelector • RDRSound • RDRSwitchView

The Stepstone Corporation

(203) 426-1875
hotline@stepstone.com

ICpak 101

Stream Technologies, Inc.

+358 0 4357 7340
info@sti.fi

Object Store

Target Development

(800) 444-5435
(717) 898-9190
objects@target.com

Link View Palette• Retriever Palette

Trillium Sound Research, Inc.

(800) L-ORATOR
(403) 284-9278
manzara@cpsc.ucalgary.ca

Text-to-Speech Kit

Uptime Object Factory, Inc.

(+41) 55 12 42 29
(+41) 1 932 4923 FAX
info@uptime.ch

FilterKit • Generic Search Facility for DBKit • Nikita: Advanced NetInfo Kit

Versant Object Technology

(415) 329-7542
geoff@osc.com

Versant ODBMS

VNPSoftware

(617) 661-4292
(617) 864-6768 FAX
info@vnp.com

AccessKit • UIBinder Palette

Workstation AG

+41 91 505094
rgi@wag.chation

DBDragger • FuncEdit • Knob

ZGDV Darmstadt

+49 06151 293863
essmann@igd.fhg.de

Realtime Voice and Video Communication

ZippyTech

(412) 421-9588
ztech@well.sf.ca.us
PO Box 322
Homestead PA 15120

InetObjects Base Collection & Protocol Collection

We would be happy to list your company -- please let us know how to list your company and what objects are available from your company. Filling out the object submission form on the next page will also help us to provide a complete list of resources for the object based NeXT developer.

In future issues an important part of this publication will be a descriptive list of tools and objects that are available to NeXT programmers. We will be trying to review two or three objects or programming tools each month. If you would like to have your object or tool reviewed send a copy, with all associated literature and documentation to:

Ted Shelton, Editor
Object-Based Computing
c/o I T Solutions
400 West Erie, Suite 301
Chicago, IL 60610

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

Object - Based Computing

InetObjects

from Zippytech

Pittsburgh based **Zippytech** began shipping the InetObjects Base Collection version 1.0 in July of this year. InetObjects is a collection of TCP/IP networking objects for NEXTSTEP. The object kit is available for both NeXT machines and Intel machines running NEXTSTEP.

According to S. D. Cooper, head of Zippytech, "This object collection is the only one that provides seamless integration of TCP/IP networking and the NEXTSTEP Appkit. It is now much easier for programmers to construct network applications in a heterogenous TCP/IP environment."

The InetObjects Base Collection provides classes for TCP/IP and UDP/IP sessions, header files, online documentation, and source code for a NEXTSTEP application that uses the InetObjects object collection.

Zippytech programmers reported that it was more painful to install NEXTSTEP on their Intel box than it was to port InetObjects for it.

The InetObjects Base Collection sells for $100 with substantial student discounts. Cooper: "Our pricing is structured so that developers can try the objects with minimal up-front costs. Our educational discounts also make the InetObjects Base Collection ideal for students wanting to learn network programming."

Zippytech is a privately held company that develops objectware for heterogenous network environments.

For more information, contact

Zippytech (ztech@well.sf.ca.us).
PO Box 322
Homestead PA 15120
412-421-9588

Object Catalog

ObjectWare for NEXTSTEP

Objects, palettes, and other tools for NeXT Developers:

ABComputers

(401) 521-2829
(401) 521-2829 FAX
rca@cs.brown.edu

ProFuse Rule

Anderson Financial Systems, Inc.

(800) 237-8723
(215) 653-0911
(215) 653-0711 FAX
kits@afs.com

AFSApplication • AFSButton • AFSEventManager • AFSFindPanel • AFSForm • AFSHelpPanel • AFSLookupsPanel • AFSMatrix • AFSMouseCalcPanel • AFSReportPanel • AFSScanPanel • AFSTextField • AFS3DButton/Graph • AFSWindow,AFSPanel • TradeKit

Archetype, Inc.

(617) 890-7544
(617) 890-3661 FAX
info@architype.prospect.com

Document Engine

BenaTong

(614) 276-7859
(614) 276-7859 FAX
benatong@count0.uucp or chuck@kiwi.swhs.ohio-state.edu

Serial Solutions

Black Market Technologies, Inc.

(718) 522-5090
(718) 522-5090 FAX
info@bmt.gun.com

GridPalette • Multicell

Conextions, Inc.

(508) 689-3570
(508) 689-2450 FAX
info@conextions.com

3270Builder • 3270Palette • 3270Toolkit • 5250Palette • 5250Toolkit

Dept. of Radiology Ohio State University Hospital

(614) 447-9194
mitroo@magnus.acs.ohio-state.edu

ImageScrollView/MiniView

Digital Composition Systems, Inc.

(415) 673-5322
gary@dreyfuss.portal.com

SpreadSheetVue

Digital Tool Works

(617) 742-4057
lexcube!equation@bu.edu

Equation

Doberman Systems

(801) 944-4329
doberman!mike@esunix.sim.es.com

Simulation Kit

Elysia, Inc.

(+33) 1 47 49 61 96
(+33) 1 47 14 99 09 FAX
info@elysia.fdn.org

ImageView Palette • ScanKit

Eye Research Institute

(416) 369-6478
(416) 369-526 FAX
info@eric.on.ca

Calera Network OCR Toolkit

Harvard Toolworks, Inc.

(508) 772-4420
(508) 772-4603 FAX
info@magdalen.dmc.com

DBCalendar & DBText Adaptor

Hot Software

(617) 252-0088
(616) 876-8901 FAX
info@hot.com

BarCodeKit • SerialPortKit

Hutchison Ave. Software Corp.

(514) 499-2067
(514) 845-5236 FAX
darcy@solutions.ca

NewsKit • QuoteKit

Object - Based Computing

■

```
{
 return [self only];
}

 // INSTANCE METHODS
- init // Only initialize self if
    the single instance isn't set.
    This ensures that the object is
    only initialized once.
{
 if
    (!theOnlySolitaryObjectInstance
    ) {
  [super init];
  theOnlySolitaryObjectInstance =
    self;
 }
 return
    theOnlySolitaryObjectInstance;
}

@end
```

This class will never have more than a single instance. Because objects loaded from a ".nib" file are allocated via the class method +allocFromZone:, it can even be safely instantiated in a number of separate ".nib" files with the assurance that they will all be the SAME instance. In the Inspector panel example described earlier, each of the separate ".nib" files for each type of window might have an icon corresponding to a solitary InspectorController instance which could be used to invoke the Inspector Panel just as FontManager can be used in invoke an applications Font Panel.

In cases where it is appropriate, simply replacing the static declaration for the single instance variable with a global could provide a global similar to NXApp. Alternately, we can extend this concept to provide a root class for solitary objects. The incorporation of a hash table as shown below will yield a class from which subclasses will also always be solitary.

```
@implementation SolitaryRoot //
    class with solitary descendants
    Hash table used to store
    instances of this class and all
    its subclasses. Every subclass
    of this class will be
    constrained to have only a
    single instance.
static id instanceTable; // Use
    class initialization to create
    hash table, which is essentially
    a private class variable.
{
 [super initialize];
 instanceTable = [[HashTable alloc]
    initKeyDesc:"@" valueDesc:"@"];
 return self;
}

 // CLASS METHODS

+ allocFromZone:(NXZone*)aZone //
    Always check the hash table to
    see if an entry already exists
    for this class. We use the class
    id as a search key into the
    table.
{
id tInstance;

 if (tInstance = [instanceTable
    valueForKey:self])
  return tInstance;
 else
  return [super
    allocFromZone:aZone];
}

+ only // Find or create the single
    instance which is associated
    with this class.
{
id tInstance;

 if (tInstance = [instanceTable
    valueForKey:self])
  return tInstance;
 else
  return [[self alloc] init];
}

+ new // Alternate method name for
    above.
{
 return [self only];
}

 // INSTANCE METHODS

- init // Initialize and insert the
    a new instance into the table if
    it is not already there. (Be
    wary if subclasses are to use a
    different designated
    initializer.)
{
id tInstance;

 if ((tInstance = [instanceTable
    valueForKey:self]) == NULL) {
  [super init];

  [instanceTable insertKey:[self
    class] value:self];

  tInstance = self;

 }

 return tInstance;

}

@end
```

This technique provides a convenient and reliable method for creating solitary classes. It also aids in reducing confusion by making it easy to identify solitary classes within a class hierarchy, since they can always be subclasses of SolitaryRoot.

The methods described above for creating and managing single instance object classes are simple, powerful, and surprisingly counter-intuitive. Using solitary objects, you can avoid unnecessary pointers, facilitate communication between objects within your application, and model application-wide data and processes more naturally. You can use them in Interface Builder to make connections which would otherwise be impossible. Although some might object to these single instance classes because they blur the distinction between classes and instances, they can greatly simplify a design, making the code more readable and easier to maintain, and that, after all, is the basic goal of all object-oriented techniques.

■

Joe Barello Consulting

NeXT Computer

Software Design

Training

System Administration

4043A 23rd Street
San Francisco, CA 94114
415.647.6398
joeba@jbc.com

which are naturally global in scope. A typical application, for instance, might have a number of open windows containing different types of information but only a single inspector panel which would display information in response to actions in any windows. Rather than using many outlets to one object or employing a global variable, an object class which by its very nature allows only a single instance of itself is a simpler and more robust solution. I call such classes solitary objects.

The AppKit contains a number of solitary object classes including OpenPanel, SavePanel, and FontManager. Imagine how frustrating and confusing it would be to have to provide a separate outlet to the FontManager for every object which needed to alter fonts. Imagine how annoying it would be if the icon for the FontManager presented in Interface Builder appeared in only one ".nib" file per application. (Remember that its not possible to connect objects in different ".nib" files using Interface Builder except through File's Owner.) Although the AppKit provides an excellent framework for connecting multiple object instances via the outlet and target/action constructs, it doesn't provide a convenient mechanism for handling events and data which are application-wide in nature.

Many programmers resort to techniques such as including a larger number of controller-type outlets in the Files owner class for each of their ".nib" sections, or routing all application-wide commands through an all-knowing, overseeing controller class, or even resorting to subclassing the Application object. Through the use of class methods, solitary object classes provide a more robust and readable alternative.

Solitary objects like the AppKit's Font Manager class are quite easy to create. Consider the implementation below.

```
@implementation SolitaryObject //
    class with single instance
// This static variable will hold
    the single instance of this
    class. (Set by -init method.)

static id
    theOnlySolitaryObjectInstance =
    NULL;

// CLASS METHODS

+ allocFromZone:(NXZone*)aZone //
    Skip allocation if an instance
    already exists (even -
    loadNibSection::: will be
    fooled by this)
{
 if (theOnlySolitaryObjectInstance)
  return
    theOnlySolitaryObjectInstance;
 else
  return [super
    allocFromZone:aZone];
}

+ only // Both +only or +new may be
    used to access the single
    instance. I prefer +only.
{
 if (theOnlySolitaryObjectInstance)
  return
    theOnlySolitaryObjectInstance;
 else
  return [[self alloc] init];
}

+ new // Alias for +only
```

IT Solutions Developers =

NEXTSTEP Development

Black & White

INFORMATION
TECHNOLOGY
SOLUTIONS

We know NEXTSTEP on INTEL

Information Technology Solutions, Inc.
400 West Erie, Suite 301
Chicago, IL 60610
(312) 587 -2000 FAX: 312.587.2012

NeXTMail: info@its.com

simplest form, this method reads like this:

```
- become:(Class)newClass
{
id newObject;

 newObject = [[newClass alloc]
    init];
 [self free];
 return newObject;
}
```

and it would be used like this:

```
 id myObject = [[myClass alloc]
    init];

 // Some intervening code

 myObject = [myObject
    become:otherClass]; // myObject
    gets replaced.
```

The version of this method that I use more often does not simply replace an object with another object, but will copy as much of the original as possible into the new object. The way I decide how much to copy, is to look at the inheritance of the original and the new classes, and note where they diverge.

```
- become:(Class)newClass
{
id newObject;
int index = 0;
List *mySuperClasses,
    *newSuperClasses;
Class myClass = [self class];
Class commonAncestorClass;

 if (newClass == myClass) // NoOp if
    these match.
  return self;

 newObject = [[newClass alloc]
    init]; // Make the new Object
 mySuperClasses = [self
    superClasses]; // Get the
    superclasses of each
 newSuperClasses = [newObject
    superClasses]; // class.
 for (index = 0; [mySuperClasses
    objectAt:index] ==
    [newSuperClasses
    objectAt:index]; index++)
 {
  commonAncestorClass =
    [mySuperClasses
    objectAt:index];
  }
 memcpy (self, newObject,
    sizeof(commonAncestorClass-
    >instance_size);
 newObject->isa = newClass;

 [mySuperClasses free]; // Tidy up?
 [newSuperClasses free];
 [self free];
 return newObject;
}
```

This is a bit more elaborate. First I create the new Object, and then, using the -superClasses method described above, I get lists of the ancestors of each object. The for loop steps through the lists until they diverge, and then the call to memcpy copies that much of the memory of the original object into the new object. The statement: newObject->isa = newClass; fixes the one thing that we didn't want memcpy() to change.

If we consider the case of changing one view subclass into another view subclass, then we can see that it will usually make sense to copy the View instance variables into the new Object. This is similar to what happens when a CustomView is loaded from a .nib file. The CustomView object resides in the file, and when it is copied out of the nib, it creates an instance of the View subclass which it supposed to represent, copies its own instance variables (sizing information, etc.) into the new View, adds the new View to the view hierarchy in its window, and then destroys (frees) itself.

I hope you have found the techniques in this article interesting and useful.

A number of helpful programming techniques and objects are available from ITS for a nominal fee. Send email to ***info@its.com*** *for more information.*

Solitary Objects

James Herre

In our current age of structured, modular, and object-oriented programming, outmoded concepts like global variables are usually banished altogether, yet there is often a need for both data and functions

■

Adding Methods to Object

John Randolph, Dolphin Technologies, Inc.

When I began writing software using Objective-C, one of the features which most impressed me was the ability to add methods to an existing class, by writing a category of that class. This article describes several methods I have added to the root class, Object. Some of these methods are useful in debugging, and others are of general interest in understanding the workings of the Objective-C runtime system.

These methods are collected in a category called (ITSObjectExtensions). The first method is -inheritance which reports the names of all the classes from which the receiver inherits. For example, when self is a View, the following statement:

```
NXAtom tempString = [self
    inheritance];
```

results in tempString containing: "View:-Responder:Object"

```
- (NXAtom) inheritance
{
char *workString;
 if ([self class] == [Object class])
  return NXUniqueString([[self
    class] name]);

 workString = alloca(MAXPATHLEN);

 sprintf(workString,"%s:%s",[[self
    class] name], (char *)[[self
    superclass] inheritance]);

 return NXUniqueString(workString);
}
```

There is one pitfall which I hit when writing this method. Originally the next to the last line read:

```
 sprintf(workString,"%s:%s",[[self
    class] name], (char *)[super
    inheritance]);
```

This caused the compiler to complain, since this method was being added to Object, and Object has no super! Object does, however, have a method which returns the receiver's superclass. The call to [self superclass] looks like it would still be a problem at run time, but when the receiver of the method is Object, execution will never reach that line anyway.

The next method, -superClasses, is a good example of recursion through the class hierarchy. This method gets the superclasses of the receiver, and then appends the class of the receiver, unless the receiver of the message is Object. This propagates up the hierarchy until it reaches Object, at which point this method creates a List, and adds the class, Object to that list.

```
- (List *) superClasses  // To
    iterate is human, to recurse,
    divine!
{
 if ([self class] == [Object class])
  return [[[list alloc] init]
    addObject:[self class]];

 return [[[self superclass]
    superClasses] addObject:[self
    class]];
}
```

The method, -subclasses returns a list of all of the classes which are direct subclasses of the receiver class. This method is more elaborate since it involves stepping through the Objective-C runtime system's hashtable of all of the classes.

```
- (List *) subclasses
{
NXHashTable *classTable;
NXHashState hashState;
Class myClass, thisClass;
id subclassList = [[List alloc]
    init];

 myClass = [self class];
 classTable = objc_getClasses();
 //First, get ALL the classes.

 for (hashState =
    NXInitHashState(classTable);
    NXNextHashState(classTable,&has
    hState,(void **)&thisClass);)
  if ([thisClass superclass] ==
    myClass)
   [subclassList
    addObjectIfAbsent:thisClass];

 if ([subclassList count])
  return subclassList; // Return the
    list if there are any subclasses

 [subclassList free]; // Otherwise,
    Kill the list, and return nil.
 return (List *) Nil;
}
```

What the runtime system gives us is a hashtable, which we can step through using NeXT's hashtable functions. The for loop initializes a hashstate for the classTable, and iterates through the table with NXNextHashState(), putting each class in the table into thisClass.

Since thisClass is a class object, we can send it a superclass message, and compare the result to myClass. If we wanted this method to result in a list of all of the subclasses of the receiver, then we could replace the test ([thisClass superclass] == myClass) in the body of the loop with ([thisClass isKindOf:myClass]).

The logic of the method, -hasSubClasses, is similar to that of -subclasses, but it only steps through the class list until any match is found.

```
- (BOOL) hasSubclasses // Like the
    loop above, but quits when any
    subclass is found.
{
BOOL foundSubclass = NO;
NXHashState hashState;
NXHashTable *classTable =
    objc_getClasses();
Class thisClass,myClass = [self
    class];

 for (hashState =
    NXInitHashState(classTable);
    NXNextHashState(classTable,&has
    hState,(void **)&thisClass);)
   if ([thisClass superclass] ==
    myClass)
   {
    foundSubclass = YES;
    break;
   }
 return foundSubclass;
}
```

The final method in this article, illustrates a technique which I've used on many occasions. This technique involves having an object of one class act as a stand-in for another object, replacing itself when called upon to do so. In its

Object - Based Computing

Published often by
INFORMATION TECHNOLOGY SOLUTIONS INC.

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

AUGUST, 1993
ISSUE #4

VOLUME 1, NUMBER 4, AUGUST 1993

editorsDesk

One Year Anniversary

It is both a pleasure and an embarrassment to be celebrating our first anniversary this month. A pleasure because I am happy to have been delivering this publication to our readers for the past year and an embarrassment to be publishing only our fourth issue in that year.

We have entirely given up the idea of publishing *OBC* on a monthly basis, at least for right now. As you will see on the subscription form we now refer to subscriptions as "12 issues" rather than "Annual."

We will continue to publish as frequently as time and material permits and I hope to work back toward a 12 issue per year schedule. In the meantime we will focus on providing valuable, high quality information to you - NEXTSTEP programmers worldwide.

Ah, NEXTSTEP not NeXT... It's beginning to trip off the fingertips as I type (as long as I can remember to capitalize the "e"). Many changes have occurred since our last issue, not the least of which has been NeXT's decision to stop manufacturing their famous black hardware.

While many of us, I'm sure, were sad to see the NeXTstations disappear, most of us agree that NEXTSTEP now has a much better chance to become a market force in desktop operating systems.

The decision faced by a corporation to purchase a $700 software license vs. a $7,000 piece of hardware should push many companies the rest of the way into NEXTSTEP.

Already we have seen many companies that considered the NeXT computer to be too risky for their IS department take a serious interest in NEXTSTEP on Intel. Add that to the "Object Enterprise" alliance between NeXT and Hewlett Packard and NeXT may actually deliver on their 1993 NeXTWORLD slogan -- "The alternative to the Microsoft Monopoly"

At this year's Object World, however, NeXT had not yet proven itself to the marketplace. Despite good attendance at both my speech to attendees on Real-World applications of NEXTSTEP and Brett Bachman's (VP Product Marketing, NeXT) similar speech, NeXT received very little attention from the industry.

While having their own booth on the show floor helped, no one in the Hewlett Packard booth knew anything about a NeXT-HP alliance. So much for the first round. Clearly a lot of work is still to be done if NEXTSTEP is going to succeed.

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.

Dan McCreary is rewriting many of the sections of his book on Object Oriented programming to reflect 3.0 NEXTSTEP. Thus the fourth section of Dan's continuing series will not be appearing in this issue.

Other articles in this issue include an overview of PagerNX, a complete paging kit for NEXTSTEP; a description of a new TCP/IP networking object from **ZippyTech**, and two articles on programming in Objective-C. Enjoy!

contents