Implementing a Dynamic-Linking, Object-Oriented Application Platform on the Macintosh

Elizabeth Brennan, Novell
Mike Russell, Novell

This paper describes a method for implementing a dynamic-linking, object-oriented application platform on the Macintosh. The implementation described is a common, extensible software platform for running network utilities. Our goal was to design a system that enables developers to modify the existing functionality of a utility, and to leverage the functionality of existing utilities when creating new ones. This design is realized by implementing a system that allows developers to inherit or override the functionality of existing utilities. Because the relationships between utility modules are resolved at run time,the utility developer need not have access to the system source code, or even the source to its objects' parent classes. By using this system, the developer of the new utility gains speed and simplicity of implementation. More importantly, the end user gains an important increase in interface consistency.

Introduction

As the number and variety of services available to network users increase, the task of managing and configuring those services becomes more and more complex. When a new service is added to the networking world, developers must create a corresponding utility or set of utilities to administer that service. In the Apple networking world, these utilities appear as Chooser or Control Panel documents, as desk accessories, or as stand-alone applications. Sometimes the administrative functionality of a single network service is distributed over several of these methods. For example, this situation arises when a user wants to submit a file to a network printer that is not the user's default device. The user must first access the Chooser in order to select the printer. Monitoring the progress of the print job on the queue requires the user to open the Print Monitor application. Then, the user must again invoke the Chooser to reset the default printer.

The lack of a good framework in which to run network utilities is as hard on the utility programmer as it is on the user. To make all the administrative functionality of a network service available within a single context, the programmer must currently reimplement the functionality of already existing and well-known utilities, in addition to designing and implementing the new service. The alternative is to create a separate utility that administers only the service's unique functionality; the user must then administer the rest of the service with existing utilities. While this approach allows for quicker delivery of new utilities, the user must deal with a lack of consistency throughout the administrative process with respect to both user interface and system feedback.

An application platform that allows a developer to leverage the functionality and user interface of existing utilities when creating a new one provides him or her with a faster and easier implementation path. More importantly, the new utility runs exactly like the old one, with the addition of a new feature. The user therefore acquires new functionality and power with no loss of interface consistency.

Design

It is possible to design a platform that solves these problems for both user and developer. The key feature of such a platform is the ability it would provide developers to alter the functionality of an existing utility by inheriting or overriding that functionality when creating new utilities. Because the relationships between utility modules are resolved at run time, the utility developer need not have access to the existing utility's source code.

In our design, utilities are implemented as separate code resources called *snap-in* modules. These modules are compiled and linked independent of the application platform, and invoked by the application at run time. The platform, which we call a *utilities console*, consists of three major elements. At the core of the platform is a *base console* that dispatches user and system events to the snap-in modules. We implement the *inheritance mechanism* of the system on top of this core, by providing a set of functions that allow snap-ins to define and create classes and objects at run time. Finally, a set of *core classes* define the framework for a basic GUI environment and provide API-level access to lower-level services and devices. These core classes are detailed in the implementation section below.

Base Console
The base console is a stand-alone Macintosh application. The main event loop of the console functions as it would in any standard Macintosh program. It receives events such as *mouseDowns*, and relays the information to the appropriate snap-in. The main event loop also looks for new snap-ins that may have been introduced to the utilities console environment. In our implementation, that environment is the folder in which the console application resides. A Preferences folder inside the System folder is another possible configuration. When the base console notices that a new snap-in has been dragged into its folder, it adds that snap-in to a list or menu from which the user selects snap-ins to execute.

In addition to user- or system-generated events, the base console may tell a snap-in that another module has sent it a message. The snap-in receiving the message is responsible for handling it. Since the originator of the message is unimportant and one function is executed per message sent, a developer can implement a scripting system to access the functionality of the base console and its snap-ins.

Inheritance Mechanism
The design of our base console, with its associated snap-ins, is not radically different from currently existing Macintosh applications that implement add-on or plug-in modules. However, on top of this dispatching mechanism, we have implemented a true run-time inheritance mechanism. To take advantage of this feature, the utility developer writes code to inform the base console that the new snap-in is a subclass of another snap-in. The programmer then simply defines those new and enhanced functions that differentiate the new class from its parent.

Core Classes
We have defined a set of object classes that form the base for a graphical user interface environment. At the highest level in the class tree is a cleverly named class called “object.” The code that creates an instance of the class “object” resides in the base console. Therefore, the root of our class hierarchy is always available to any snap-in. “object” implements the minimal methods and instance variables necessary for a class, including methods to create and delete instances of a class, a reference count used for allocation control, methods to set and access an object’s name, and methods to access a class’ parent, sibling, and subclass.

Architecture

The key element in our architecture is the run-time binding of code modules into an object-oriented class hierarchy. The base console mechanism implements this binding. The object-oriented features of our system are built on top of a non-object-oriented architecture, in which the base console passes system and user events to the appropriate snap-in modules. We will first explain this non-object-oriented architecture before explaining how to build the object-oriented features into this core.

Console architecture
The base console architecture enables object-oriented programming, but it does not enforce it. A developer can create snap-in modules that receive and send messages through the base console, and process those messages in the classic procedural manner. These procedural snap-ins do not rely on the existence of other snap-ins in a hierarchy to implement part of their functionality. Nor do they allow other snap-ins to leverage their functionality through the inheritance or overloading. Non-object-oriented snap-ins may still send messages to one another, as well as to object-oriented snap-ins, through the base console’s message and event dispatching mechanism.

The principle behind the base console dispatch mechanism is quite simple. When a snap-in is loaded, an entry for that snap-in is added to a table of snap-ins. Each snap-in table entry contains information that the base console needs to manage that snap-in. One of the pieces of information stored is the entry point to the snap-in’s code resource.

The main event loop of the system resides in the base console. The base console sees that active snap-ins in the system get passed those events that apply to them. It is also responsible for handing out idle time to snap-in that have requested that service.

```
BaseConsole.c
/* Base console, main event loop */
main()
{
  …
  for (;;) {
    …
    if (GetNextEvent(everyEvent, &event)) {
      switch (event.what) {
        …
        case activateEvt:
          if ((snap = GetSnapIn ((WindowPeek)
                        event.message)) != 0)
            CallSnapIn (snap, (long) &event,
                          eventCode);

          …
          break;
        …
      }
    }
    else
      SnapIdle ();
    …
  }
  …
}

/*
* Jump to the code entry point for our snap-in.
*/
CallSnapIn ( snap, message, eventCode )
SnapIn *snap;
long message;
short eventCode;
{
  ProcPtr   codePtr;

  if (snap->entry) {
    …
    codePtr = (ProcPtr) *snap->entry;
    (void) (*codePtr)(snap, message, code);
    …
  }
  …
}
```

The base console enables snap-in to snap-in communication by passing messages from one snap-in to another. Besides simple messages, one snap-in may allocate memory it is willing to let other snap-ins access. In these cases, the base console also acts as a memory management monitor; ensuring that no shared memory is trashed while any snap-ins are still using it.

This is the basic architecture which forms the core of our system. The runtime-binding, object-oriented system is built onto this core architecture.

Object-oriented architecture

We first present the terms that describe the object-oriented aspects of the system architecture and implementation. Generally speaking, our terminology is quite standard for object-oriented design, but the meanings of these terms do vary between different OOPs implementations. We provide these definitions for quick reference before getting into the nitty-gritty aspects of the system architecture.

Class	A template or pattern for the creation of one type of object, specifying methods, class variables, and instance variables.
Class Variable	A per-class variable.
Encapsulation	Also known as *Information hiding*. Encapsulation is the design principle that provides access to an object’s services while hiding the implementation of those services and the internal structure of the object’s data from the user.
Inheritance	A class' assumption of its superclass' methods and variables.
Instance	An object, with emphasis on its class membership. Instance is to “citizen” as object is to “person”.
Instance Variable	A per-object variable.
Method	A per-class function.
Object	An allocated group of Instance Variables, Methods, and Class Variables. Objects are created using a specific class as a template.
Polymorphism	The sharing of method names by several classes through inheritance.

Superclass The class from which a particular class receives its initial methods and variables. Superclass is synonymous with parent class. A class is a subclass, or child, of its superclass. Sibling classes share the same superclass.

The base console contains a set of function calls that enable the object-oriented basis of our architecture. When called by a snap-in, these functions create classes and objects, and the methods and variables associated with each. As these functions are called by the snap-ins, a class hierarchy is built into a tree, with each class in a subsidiary relationship to its superclass. Subsidiary classes are said to be subclasses of their superclass. The special class Object is the root class of the tree. It has several built-in methods, including ones for creating a generic object and for destroying a generic object. Since the ability to create an object is built into the root class of the tree, all classes can create instances of themselves.

Another function built into the base console is used to pass messages to appropriate objects in the inheritance tree. This function (referred to as Send in the following section) operates by locating the item to which the message refers in the object's inheritance hierarchy. Instance variables are searched first. If no matching instance variable is found, the class hierarchy is searched for a matching class variable or method. The object's class is searched first, then its superclass, and so on, until object, the root class, has been searched.

Methods and instance variables are referred to by 4 character (32 bit) identifiers. These identifiers may or may not have mnemonic significance. The origin of the message is unimportant. The Send function may have been called by any snap-in within the system. Since all the messages get passed through the base console, it is possible to monitor all of the communication between objects and snap-ins. Also, a scripting mechanism could easily be implemented using this architecture that would allow automated or remote messages to be sent to and received from our class objects. The logical extension of this architecture to encompass AppleEvents is an exciting topic, but unfortunately, it is beyond the scope of this paper.

Implementation

This section provides an example implementation of the environment and architecture we have discussed. This example implementation includes class definitions, methods, class variables, instance variables, inheritance, object creation, and message sending.

A basic class hierarchy

We have implemented our demonstration utilities based on a hierarchy of class objects, where classes can be described as either “view-controllers” or “models” (see [Wittenberg, 1991]). “View-controllers” or “views” are simply classes that implement visual images and controls on the screen. Windows, scroll bars, and icons are examples of familiar objects in the Macintosh world that have been implemented as view-controller classes in the utilities console. In our implementation, “Models” are classes that describe the characteristics and behavior of entities on a network, such as their status, names, net addresses, and so on. *Printer* is an example of a model class, while *laser printer* and *color printer* would be examples of printer model subclasses. Very often a parallel view class is created for model classes. A view class called *ViewPrinter* would know how to draw a printer and display information about its state. A ViewPrinter object would contain a reference to a Printer object and could send messages to the model asking it about its status.

A particular snap-in might create instances of both a model class and one or more parallel view classes. The developer could then modify the behavior of an existing utility at two levels. At the “view” level, he or she could change the visual representation of data by altering or subclassing a particular View class. For example, a utility might display a textual list of printers with more text indicating the status of each printer. A developer could change this view to display the printers as icons which change color according to their status. The same model data is being viewed in the new utility as in the old one, only in a graphical rather that textual format.

At the “model” level, the developer could change the actual data which a particular view represents. For example, a utility which displayed information about users on the network might be made up of a view class which shows icons and status text, and a model class which defines the data that describes a user. It should be possible to create a new model class for printers and use the user-view class to display printer icons and text with minimum changes .

To take advantage of this inheritance mechanism, developers of snap-ins need to know about classes that are defined in other snap-ins . Rather than require snap-in developers to document and distribute their classes and methods, we have created a tool that lets developers gather that information directly from the base console and snap-ins. This tool is a class browser that is implemented as a snap-in. The browser displays the class tree that is currently present in the system, allows a user/developer to select a particular class, and displays the methods and instance variables for that class and all of its ancestors. A sample display of the class browser is included at the end of this paper.

Design decisions
In any implementation of the architecture we have described, the designers of the system have to make certain decisions about just how strictly to enforce the object-oriented paradigm. While the system described here supports a complete object-oriented functionality, it does not enforce it. There are several areas where we permit violations of normal object-oriented discipline. In this light, the next several paragraphs outline some procedures for changing an existing class. The safest procedures are listed first.

• *Subclassing* is the usual and desired way to modify a class. The newly created class inherits methods, class variables, and instance variables of the specified superclass. The developer can then modify or override existing methods and variables, and can add new methods and variables. An example of a subclass is a third party who adds a subclass of the network utility class *Printer* called *Acme Printer*. Subclassing has the advantage of not affecting existing objects, and the minimizes possibility of a destructive collision with another class definition.

• *Adding methods or class variables* to an existing class is a bit riskier than simple subclassing, for two reasons. Any change to a class is instantly propagated to existing subclasses of that class, and to any existing instances of those subclasses. For example, adding a method to an existing class is appropriate if a vendor wishes to replace or augment an existing class by adding another item to a detail display. Use addition of methods or class variables if you are unable to re-define your problem in terms of subclassing. Be aware of the danger of inadvertently changing something that may break a subclass of the modified class.

• *Adding methods that use new instance variables* achieves a new level of danger: objects created before the addition generate a fatal error when they execute the (per class) method that accesses the (per object) instance variable.

For example, consider the addition of the instance variable *myVar* and the method *Increment_myVar* to an already existing class *myClass*. Previously created instances of *myClass* inherit *Increment_myVar*, but not *myVar*. Sending an *Increment_myVar* to one of these objects causes a run-time error.

The rule is simple: add instance variables only at initialization time, or after each instance of the class and its descendants has been destroyed.

Implementation functions
These functions describe the scaffolding on which the real work, classes and methods, are performed. The data type *ident_t* is at this time equivalent to a null-terminated string of which only the first 64 characters are significant. The data type *message_t* is a 4 character (or 32 bit) identifier, that may or may not have mnemonic significance. Unless stated otherwise, these functions return a long error code.

ClassH GetClass(className)
ident_t className;
GetClass() returns a handle to the class named className.

This function is normally used to convert the name of a well-known class into a handle suitable for passing to another function. *GetClass()* may also be used to check for the existence of a class for sequencing purposes during snap-in initialization.

ClassH NewClass(className, superClassName)
ident_t className;
ident_t superClassName;
NewClass() creates a class named *className* whose superclass is *superClassName*. The function returns a handle to the newly created class.

The new class inherits all the methods and variables of the superclass, and is therefore functionally equivalent to it. Subsequent calls to NewMethod, NewVar, and NewClassVar can add to or modify the functionality of the new class.

NewMethod(class, name, userfunc)
ClassH class;
message_t message;
long (*userfunc)();

userfunc(self, ...)
ObjectH self;
NewMethod() installs a method that responds to *message*. When *message* is received via the *Send* mechanism, the function *userfunc* is invoked.

In accordance with normal inheritance rules, the new method takes precedence over any previously existing inherited method responding to the same message in the specified class and its descendants. The old method remains accessible via *SuperSend()*, provided it is indeed defined in a superclass.

Previously existing instances of the class have immediate access to the new method. Usually, this situation is desirable; however, problems occur if the new method uses instance variables not present in an object.

When *Send()* invokes this method, it calls *userfunc()* with a copy of the parameters that were supplied to it. In a similar vein, *Send()* copies and returns the return value of *userfunc()*.

NewVar(class, name, value)
ClassH class;
message_t name;
long value;

NewClassVar(class, name, value)
ClassH class;
message_t name;
long value;
NewVar() creates an instance variable *name* for the supplied *class* with initial value *value*. This instance variable will be incorporated into a template used by the class' *new* method when creating new objects.

In accordance with normal inheritance rules, the newly created variable takes precedence over any previously existing inherited variable of the same name.

Variables created by *NewVar()* are allocated on a per-object basis. Newly created objects belonging to the class will have the new variable, but previously existing instances of the class will be unaffected.

NewClassVar() operates identically to *NewVar()* except that *NewClassVar()* creates a class variable rather than an instance variable. Class variables are shared by all objects belonging to the class. Previously existing instances of the class will have immediate access to the newly created class variable.

Send(object, name, ...)
ObjectH object;
message_t name;

SuperSend(object, name, ...)
ObjectH object;
message_t name;
Send() invokes an object's method or returns the value of a class or instance variable.

Send() operates by locating the item designated by *name* in the object's inheritance hierarchy. The function searches instance variables first. If no instance variable whose name matches *name* is found, *Send()* searches the class hierarchy for a matching class variable or method. It searches the object's class first, then its superclass, and so on, until *object*, the root class, has been searched.

If an instance or class variable matching *name* is found, *Send()* returns its value. This functionality allows a variable to be changed to a method, or vice-versa, without effecting other programs. If *name* matches a method, the method's function is called with the parameters passed to *Send()*, except the *name* parameter. *Send()* then forwards its parameters to the method function, and returns the method function's return value.

Send() also creates an object by invoking the *new* method of the desired class. In this case *Send()* accepts a class handle instead of an object handle.

SuperSend() operates like *Send()*, but invokes the method or variable belonging to *superclass*. This function is used within a method to invoke the functionality of the superclass' method. To preserve the integrity of method inheritance, the programmer must insure that *superclass* is the parent of the method's class, even if method is currently defined in an ancestor of *superclass*.

SetVar(object, name, value)
ObjectH object;
message_t name;
long value;
SetVar() sets the value of a class variable or instance variable. *Object* is a handle to the object whose variable is to be changed. *Value* is assigned to the *name* variable.

SetVar() operates by locating the item designated by *name* in the object's inheritance chain. The function searches instance variables first. If *SetVar()* does not find an instance variable whose name matches name, it searches the class hierarchy for a matching class variable. *SetVar()* searches the object's class first, then its superclass, and so on, until object, the root class, has been searched.

If an instance or class variable matching *name* is found, *SetVar()* sets the its value to *value*.

In keeping with good object-oriented practice, restrict your use of *SetVar()* to the object's own methods. To change a variable from outside an object, create a new method and access it with *Send()*.

Example
At the end of this paper are examples of source code that implement two snap-ins. The code uses the class building and accessing functions described above. The first snap-in displays information about a generic network entity. It defines generic model and view-controller classes and then creates instances of those classes. The second snap-in subclasses those generic classes and creates a new utility that reads and displays information about print queues.

Experiences and Evaluation

There are many random details that we haven't addressed in this paper.

Among the outstanding architectural issues, we have not addressed the issue of *Multiple inheritance*. Also, the *resolution of class hierarchy* problem needs to be resolved. At startup, several snap-in modules may be trying to create classes. Under our current implementation, it might be necessary to re-try the NewClass() call several times until the superclass is created by another snap-in. Admittedly, we need a more orderly way to sequence class creation.

An implementation detail that we find ourselves addressing regularly is *memory management*. Memory management becomes a major issue when you start to seriously think about allocating hundreds of objects in a session. Our objects take up a fixed 58 bytes each; with an additional 12 bytes needed for each method and instance variable within the object. We believe that we can reduce these sizes and are currently investigating this issue.

Another implementation detail which we did not discuss was *performance*. Early measurements of message passing performance has been encouraging. Passing messages through the base console via *Send()*, gives access times on a par with Mac Toolbox calls.

Summary

Our goal was to design a system that enables quick delivery of new network administration utilities having a look and feel consistent with existent utilities. We achieve this goal by implementing a true, runtime binding, object-oriented architecture.

Implementation of this design does not require any special programming languages or development tools. A base console and object-oriented snap-ins can be developed using standard programming environments. Our implementation was produced using Think C (without the Think Class Library) and some assembly language to optimize the *Send()* function's performance.

With the exception of code resources, none of the system we have described relies on Macintosh specific architecture. Any platform that provides a dynamic-linking interface for implementing snap-ins could be used to implement this design.

Finally, there are a growing number of commercial applications that use the snap-in architecture to extend their functionality. As users of several of these applications, we would be delighted to see an inheritance mechanism like the one described here, implemented in these products.

Acknowledgements

The authors would like to recognize their fellow team members' contributions in the design and implementation of this system — Harvey Crisler, Robert Bailey and Mark Wittenberg. We would also like to thank Robin Anderson for her helpful and timely editorial assistance.

This paper was built on the work and contributions of many engineers throughout Novell. Special recognition and thanks goes to Phil Karren, Steve McBride and their team in Provo for laying much of the groundwork for this design, and for their invaluable contributions and feedback.

References

Brad J. Cox, *Object Oriented Programming*. Addison-Wesley, 1986.

Peter Coad and Edward Yourdon, *Object-Oriented Analysis*. Prentice Hall, 1990.

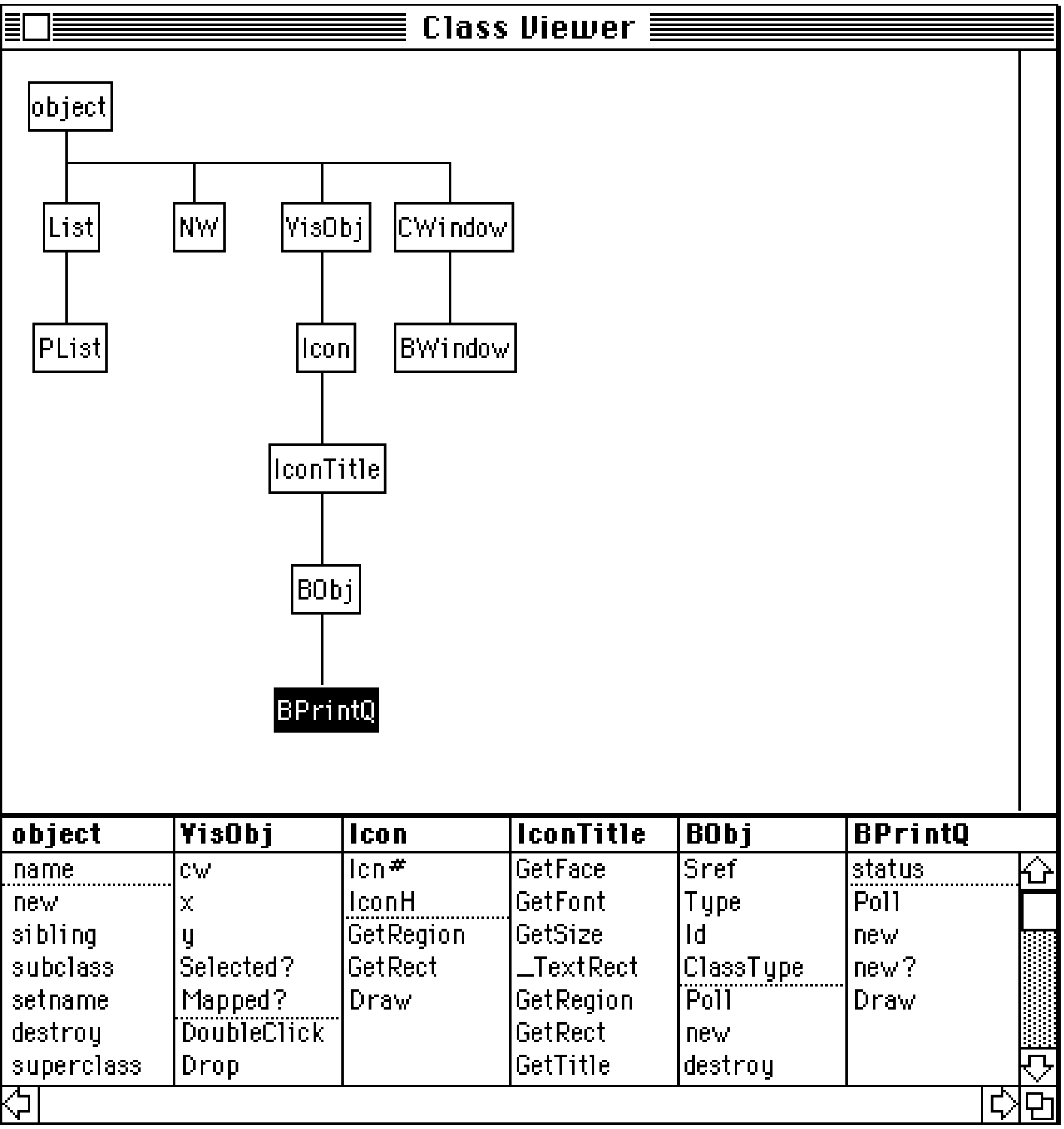
A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

Bjarne Stroustrup, *The C++ Programming Language*. Addison-Wesley, 1986.

D. Wilson, L. Rosenstein and D. Shafer, *C++ Programming with MacApp*. Addison-Wesley, 1990.

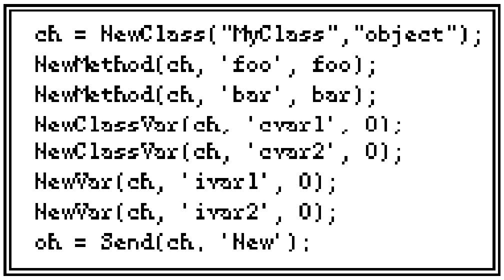
Mark Wittenberg, “Multi-Platform Software Development Using the Model-View-Controller Design Paradigm”. *Macintosh Technical Conference: MacHack™ ‘91 Conference Proceedings*, (June 1991).

The Class Browser

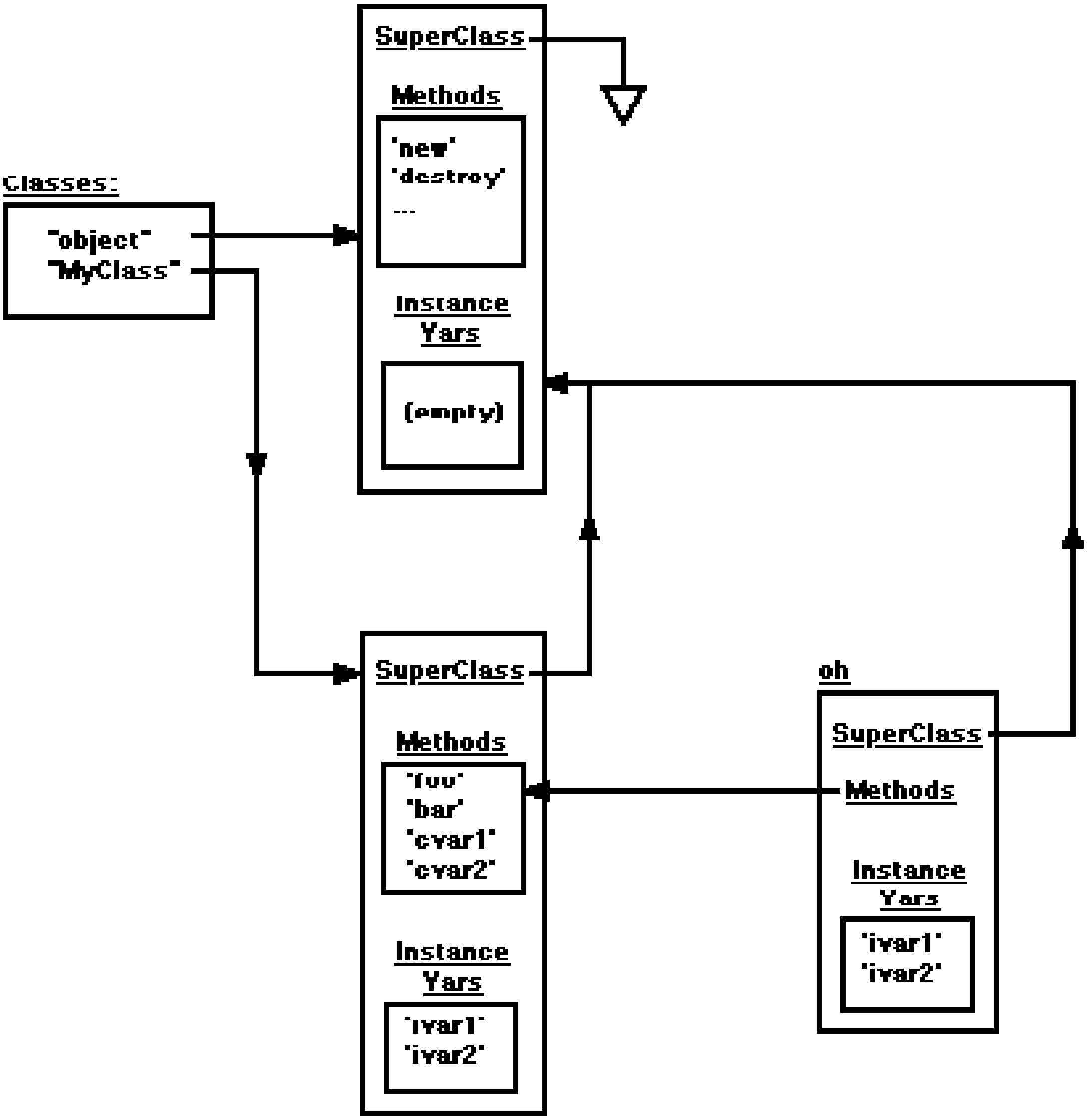


object	VisObj	Icon	IconTitle	BObj	BPrintQ
name	cw	Icn#	GetFace	Sref	status
new	x	IconH	GetFont	Type	Poll
sibling	y	GetRegion	GetSize	Id	new
subclass	Selected?	GetRect	_TextRect	ClassType	new?
setname	Mapped?	Draw	GetRegion	Poll	Draw
destroy	DoubleClick		GetRect	new	
superclass	Drop		GetTitle	destroy	

Message sending chain-of-command



```
ch = NewClass("MyClass","object");
NewMethod(ch, 'foo', foo);
NewMethod(ch, 'bar', bar);
NewClassVar(ch, 'cvar1', 0);
NewClassVar(ch, 'cvar2', 0);
NewVar(ch, 'ivar1', 0);
NewVar(ch, 'ivar2', 0);
oh = Send(ch, 'New');
```



Sample Generic Snap-in

```
/*
*    A snap-in that displays status information about a generic network entity.
*
```

```
*    Model and View/Controller classes are defined to implement this snap-in.
*    In most cases, the classes defined here will be used as parent classes to
*    create more specific network objects.
*
*    3/23/91  by Mike Russell, Novell Inc.
*/

#include    "SnapIn.h"    /* Definition for SnapIn struct */
#include    "NetSnapIn.h" /* Prototypes and definitions for this snap-in */
#include    "Class.h"     /* Definitions for class and object handles.
                Class construction function definitions. */

#include    <SetUpA4.h>   /* this statement must immediately preceed main() */


/* Receives and proccesses messages from the base console.
*/
main(snap, message, code)
SnapIn *snap;
long   message;
short  code;
{
  RememberA0();
  SetUpA4();

  switch(code) {
    case initCode:
      doInit(snap);
      break;
    case activateCode:
    case deactivateCode:
    case eventCode:
      doEvent(snap, (EventRecord *) message);
      break;
    case idleCode:
      doIdle(snap);
      break;
    case gaspCode:
      doGasp(snap);
      break;
  }
  RestoreA4();
}
```

```
/* Initialize snap-in table entry.
* Call initClasses if we're using objects in this snap-in.
*/
void
doInit( SnapIn *snap )
{
    InitBCGlue(snap);   /* sets up address of BCGlue jump table */

    snap->needsLock = true;
    snap->needsIdle = true;
    snap->interval = 20L;

    initClasses(snap);
}

/* Process events received from base console.
* Pass it on to our View/Controller class if it's not a standard event.
*/
void
doEvent( SnapIn *snap, EventRecord *event )
{
    switch(event->what) {
        case keyDown:
        case autoKey:
            KeyService(snap, event);
            break;
        default:
            Send(GetClass("BCWin"), "Event", event);
            break;
    }
}

/* Last chance to clean up before disposing of this snap-in.
*/
void
doGasp( SnapIn *snap )
{
}

void
KeyService( SnapIn *snap, EventRecord *event )
{
    char c = (char)(event->message & 0xff);

    if(event->modifiers & cmdKey) {
        doMenu(snap, FrontWindow(), MenuKey(c));
    }
}

/* end snap-in template */
/* Set up classes used by this snap-in.
*/
void
initClasses(SnapIn *snap)
{
   PL_define();    /* Poll List */
   VIEW_define();  /* Generic View class */
   MODEL_define(); /* Generic Model class */

   initNetObjs();  /* Instantiate and initialize the classes
                      we just defined */
}

/* Poll network objects periodically
*/
void
doIdle(snap)
SnapIn        *snap;
{
   Send(GetClass("PList"), 'Poll');
}
```

Generic Model Class Definitions

```
/*
* Class_Model.c - Definitions for Model Class.
*                This class should be subclassed to create explicit models.
*
*  4/12/91  by Mike Russell, Novell Inc.
*/

#include "Class.h"
#include "Class_List.h"

#define CLASS "Model"        /* bindery model class name */
#define SUPER "object"       /* superclass */


M_define()
{
   ClassH ch;

   if(GetClass(CLASS))    return;

   ch = NewClass(CLASS, SUPER);
   NewVar(ch, 'deps', 0);

   NewMethod(ch, 'new', M_new);      /* create an instance */
   NewMethod(ch, 'dstr', M_destroy); /* destroy an instance */
   NewMethod(ch, 'DpAd', M_depadd);  /* add a dependent object */
   NewMethod(ch, 'DpRm', M_deprm);   /* remove a dependent */
   NewMethod(ch, 'Updt', M_update);  /* send update to each dependent */
}
```

```
/* 'new' - create a model object
*/
static long
M_new(ObjectH self, Ptr xxx)
{
   int icon;
   ObjectH deps = (ObjectH)Send(GetClass("List"), 'new');

   self = (ObjectH)SuperSend(SUPER, self, 'new', icon);
   SetVar(self, 'deps', (long)deps);
   return (long)self;
}

static long
M_destroy(ObjectH self, Ptr xxx)
{
   ObjectH deps = (ObjectH)Send(self, 'deps');
   ObjectH oh;

   /* send '--' to each dependent, and destroy the dependent list */
   /* delete them in reverse order because of the way lists work */
   for_each_item_bwd(deps, oh)
       Send(self, 'DpRm', oh);
   Send(deps, 'dstr');
   SuperSend(SUPER, self, 'dstr');
}

/* add a dependent
*/
static long
M_depadd(ObjectH self, Ptr xxx, ObjectH oh)
{
   ObjectH deps = (ObjectH)Send(self, 'deps');
   Send(deps, 'Append', oh);
   Send(oh, '++');
}

/* remove a dependent
*/
static long
M_deprm(ObjectH self, Ptr xxx, ObjectH oh)
{
   ObjectH deps = (ObjectH)Send(self, 'deps');
   Send(deps, 'Delete', oh);
   Send(oh, '--');
}
```

```
/* send an update to each dependent
*/
static long
M_update(ObjectH self, Ptr xxx, ObjectH oh)
{
  ObjectH deps = (ObjectH)Send(self, 'deps');

  /* send 'update' to each dependent */
  for_each_item(deps, oh)
      Send(oh, 'Updt');
}
```

Generic View Class Definitions

```
/*
* Class_View.c
*    Definitions for View (Window) class.  This class supports generic view
*    behavior, particularly CWindow functions, and support for model/view
*    interraction.  See Parc-Place Systems document "A description of the
*    Model-View-Controller Interface Paradigm in the SmallTalk-80 System,
*    Krasner and Pope, 1988.
*
*    This class may have some use for very generic views, such as those
*    whose contents consist entirely of VisObjs.  But, it is intended
*    for subclassing by specialized classes with explicit model display
*    functions.
*
*    4/17/91  by Mike Russell, Novell Inc.
*/

#define CLASS "View"
#define SUPER "WINDOW"

#include "Class.h"

VIEW_define()
{
  ClassH ch;

  if(GetClass(CLASS))   return;
  if(!GetClass(SUPER))  WINDOW_define(); /* A core class that handles
                                  window stuff */
  ch = NewClass(CLASS, SUPER);
  NewVar(ch, 'VObj', 0);        /* list of visible objects */
  NewVar(ch, 'wp', 0);              /* window pointer */
  NewVar(ch, 'Modl', 0);

  NewMethod(ch, 'new', VIEW_new);     /* create and add self to model list */
  NewMethod(ch, 'dstr', VIEW_destroy); /* destory and remove self from
                              model list */
  NewMethod(ch, 'Updt', VIEW_Update);   /* respond to a change in the model */
}
```

```
/* 'new' - Create a new view window.
*        SuperSend to the parent class sets up the windowptr variable wp.
*/
static long
VIEW_new(ObjectH self, Ptr xxx, ObjectH model)
{
  self = (ObjectH)SuperSend(SUPER, self, 'new', (WindowPtr)0);

  /* Bump model's use count to reflect "Model's" reference.
  */
  SetVar(self, 'Modl', Send(model, "++"));

  /* Add ourselves to the model's list of dependents.
  */
  Send(model, 'DpAd', self);

  {
     /* Set model name as window title.
     */
     WindowPtr wp = (WindowPtr)Send(self, 'wp');
     SetWTitle(wp, pstr((char *)Send(model, 'name')));
  }
  return (long)self;
}

/* 'dstr' = destroy
*  Destroy a view window
*/
static long
VIEW_destroy(ObjectH self, Ptr xxx)
{
  ObjectH model = (ObjectH)Send(self, 'Modl');

  /* remove self from the model's list */
  Send(model, 'DpRm', self);

  /* decrement model's use count */
  Send(model, '--');
  SuperSend(SUPER, self, 'dstr');
}

/* 'Updt' = Update
*  Respond to a model change by invalidating our window
*/
static long
VIEW_Update(ObjectH self, Ptr xxx)
{
  WindowPtr wp = (WindowPtr)Send(self, 'wp');

  SetPort((GrafPtr)wp);
  InvalRect(&((GrafPtr)wp)->portRect);
}
```

Sample Print Queue snap-in

```
/*
*    A snap-in which displays status information about a print Queue.
*
*    Model and View/Controller classes are subclassed to implement this snap-in.
*
```

```
*    4/23/91  by Mike Russell, Novell Inc.
*/

#include  "SnapIn.h"  /* Definition for SnapIn struct */
#include  "PrintQ.h"  /* Prototypes and definitions for this snap-in */
#include  "Class.h"    /* Definitions for class and object handles.
                  Class construction function definitions. */

#include  <SetUpA4.h>  /* this statement must immediately preceed main() */


/*   Receives and proccesses messages from the base console.
*/
main(snap, message, code)
SnapIn      *snap;
long      message;
short       code;
{
    RememberA0();
    SetUpA4();

    switch(code) {
     case initCode:
        doInit(snap);
        break;
     case activateCode:
     case deactivateCode:
     case eventCode:
        doEvent(snap, (EventRecord *) message);
        break;
     case idleCode:
        doIdle(snap);
        break;
     case gaspCode:
        doGasp(snap);
        break;
    }
    RestoreA4();
}

…
… (snap in handling stuff here)
…
```

```
/* Set up classes used by this snap-in.
*/
void
initClasses(SnapIn *snap)
{
      MPQ_define();   /* Print Queue Model */
      VPQ_define();   /* Print Queue View */

      PQInit();    /* create and initialize PQ objects */
}

/* This routine demonstrates creating instances of classes
* and sending messages to and receiving messages from
* the newly created objects.
*/
void
PQInit()
{
   short objID = 0;
   ObjectH visOh, modOh;

   visOh = (ObjectH)Send(GetClass("VPrintQ"), 'new');
   SetPort((GrafPtr)Send(visOh, 'wp'));

   while( (objID = GetNextPrintQ(objID)) != err )
   {
       modOh = (ObjectH)Send(GetClass("MPrintQ"), 'new', objID);
       Send(GetClass("Icon"), 'new', modOh);
   }
}
```

Print Queue Model Class definitions

```
/*
* Class_MPrintQ.c - Definitions for the PrintQ Model Class.
*
* 4/17/91  by Mike Russell, Novell Inc.
*/

#define CLASS "PrintQ"
#define SUPER "Model"

#include "Class.h"

MPQ_define()
{
   ClassH ch;

   if(GetClass(CLASS))   return;
   if(!GetClass(SUPER))  return;

   ch = NewClass(CLASS, SUPER);
   NewVar(ch, 'stat', 1);
   NewVar(ch, 'CTyp', TYPE_PRINT_QUEUE);

   NewVar(ch, 'jobH', 0);
   NewVar(ch, 'Qsts', 0);
   NewVar(ch, 'Njob', 0);
   NewVar(ch, 'Nsrv', 0);
   NewVar(ch, 'Err', 0);

   NewMethod(ch, 'Poll', MPQ_Poll);
}

/* 'Poll' - update the net state, and send self an 'Update' if
*            it changes.  Return true iff state changed.
*/
static long
MPQ_Poll(ObjectH self, Ptr xxx)
{
   UINT32 oqstatus, onjobs, onservers, oerr;
   UINT32 qstatus, njobs, nservers, err;

   oqstatus = Send(self, 'Qsts');
   onjobs  = Send(self, 'Njob');
   onservers  = Send(self, 'Nsrv');
   oerr = Send(self, 'Err');

   refresh(self);

   qstatus = Send(self, 'Qsts');
   njobs  = Send(self, 'Njob');
   nservers  = Send(self, 'Nsrv');
   err = Send(self, 'Err');

   /* if a change occurred, update ourselves,
    * and send a poll to our superclass
    */
   if(qstatus!=oqstatus || njobs!=onjobs || nservers!=onservers || err!=oerr)
   {
       Send(self, 'Updt');
```

```
      SuperSend(SUPER, self,'Poll');
      return true;
   }
  return false;
}
```

Print Queue View Class definitions

```
/*
* Class_VPrintQ.c - definitions for Print Queue Visible object class
*/

#define CLASS "VPrintQ"
#define SUPER "VisObj"

#include "Class.h"


VPQ_define()
{
    ClassH ch;

    if(GetClass(CLASS))         return;
    if(!GetClass(SUPER))        BVI_define();

    ch = NewClass(CLASS, SUPER);
    NewClassVar(ch, 'CTyp", PRINT_QUEUE);
    NewClassVar(ch, 'dVCl", (long)GetClass("ViewPQ"));
    NewVar(ch, 'Icn#', 1006);

    NewMethod(ch, 'Draw', VPQ_Draw);
    NewMethod(ch, 'Updt', VPQ_Update);
}

/* "Draw" - draw a print queue
*/
long
VPQ_Draw(ObjectH self, Ptr xxx)
{
    ObjectH model = (ObjectH)Send(self, 'mmdl');

    if(Send(model, 'Err') || Send(model, 'Qsts') || !Send(model, 'Nsrv'))
     ForeColor(redColor);
    else
    if(Send(model, 'Njob'))
     ForeColor(blueColor);

    SuperSend(SUPER, self, 'Draw');
    ForeColor(blackColor);
}

static long
VPQ_Update(ObjectH self, Ptr xxx)
{
    ObjectH cw = (ObjectH)Send(self, 'cw');

    /* just redraw ourselves */
    SetPort((GrafPtr)Send(cw, 'wp'));
    InvalRect((Rect *)Send(self, 'gRct'));
}
```