Making Products Using Object-Oriented Programming

John Uebbing - Hewlett-Packard Laboratories (Chair)
Jim Waite - Hewlett-Packard Lake Stevens Instrument Division
Tom Lanning - AT&T
Ragu Raghavan - Mentor Graphics Corporation
Carl Nelson - Carl Nelson and Associates
Alen Schiller - Wild Leitz
Hamid Eghbalnia - Artecon Inc.

John Uebbing

In putting together this panel, I had the opportunity of talking with several companies making products using object-oriented programming. There are many such companies, and there are more products under development than have been released. Based on talking with these companies, and Hewlett-Packard's experience, the following conclusions can be drawn.

Large systems have special problems. Cross reference by large numbers of different objects from different classes can lead to "spaghetti structure." There is a need to put classes into major groups that are well architected and supported with special documents giving the philosophy as well as cookbook examples of how classes in the group are used. It is wise to try to integrate large systems early, to identify performance and other usability problems. Intelligent memory and storage management is not well developed and has significantly hurt the performance of large CAD systems that a number of major CAD houses have built.

Some of the best advantages of object-oriented programming for products occur when the product area has a special match to object-oriented programming. These areas include:

- Computer Aided Design (including CASE)
- Compound Documents
- Vector and Matrix Test Results
- Interfaces to disparate pieces of hardware
- Windowed User Interfaces
- Any product where somewhat disparate elements must be handled in a similar way

As in any software project, it is important not to become entranced with the technology, but to focus on solving the customer's problems.

There is good evidence that, within reason, the object-oriented programming language itself is not nearly so important to success as the surrounding environment of tools and libraries. The tools include interactive interpreters, memory and database management, dynamic linkers, performance analyzers, message tracers, and documentation systems. Systems like Lisp and Smalltalk are more mature in many respects. However, the lack of interfaces to conventional programming languages and their appetite for machine resources makes them unattractive for competitive end user products. Languages like C++, Objective-C and MacApp need more tools and development. Tools for the rational design of classes and their inheritance are in their infancy. Designing for code reuse shows promise, but is hard to do. It requires well designed classes that are well documented and easy to learn. It should be a goal of the project and requires good engineering management.

Jim Waite

Lake Stevens Instrument Division has used the object-oriented methodology in software development for over three years. In the summer of 1985, we moved from our own homegrown object-oriented technology and adopted Objective-C as a platform for future software products, for both host workstation and firmware environments. We have released one major product in this

time, a workstation based virtual instrument and measurement interface for the HP3565 multichannel Dynamic Signal Analyzer. The modular nature of this hardware lends itself to object-oriented software design.

The software consists of 312 classes, 6044 methods, and 100,474 NCSL. The project effort took 39 engineer-years for a productivity rate of 11 lines per engineer per day. We expect higher productivity in projects that leverage existing code by inheritance. An open question concerns how to measure this productivity, since inheritance results in more functionality per line of code than conventional programming methods.

The division has made a large investment in developing tools for implementing large software projects in Objective-C. One of the most useful contributions we have made is a series of scripts that manage source code control, makefile dependencies, and the process of building an executable. Code metrics are based on class documentation and headers; these metrics prove useful in predicting the project lifecycle. Also, we have combined efforts with other HP divisions to add memory management to Objective-C, as well as a runtime message interpreter and a dynamic loader.

We now have a manageable process in place from which to do development, but our object-oriented design process needs to improve. We have had problems applying standard structured analysis techniques (data and control flow) to classes and methods. Messaging is a concept that doesn't fit well with these tools. We don't have a good graphical way of describing the system, especially one with 300 classes.

Another difficulty is that maintainance of such a large system of classes is difficult. This is particularly true for instances of highly "visible" classes, i.e., those that are known by many other objects in the system. This problem would be alleviated somewhat by the use of graphical techniques to document the system. In addition, we have learned to avoid the use of hidden dependencies between objects in the system.

We agree that an object-oriented view of a software system more precisely models the problem space of the system. Beginning at the highest level of abstraction, classes denote the fundamental players, while methods are operations to be performed on them. Critically, the high level of modularity that develops from this approach to object-oriented software design must not sacrifice efficiency. We've found it is too easy to use so many little objects that the system spends a lot of time allocating and deallocating memory, resulting in lower than expected performance.

Tom Lanning

A team of software developers at AT&T Bell Laboratories has created six versions of a CENTREX management system since 1985. The systems were designed using object-oriented techniques and were implemented in C++. Object-oriented techniques were originally used to improve software reuse and software quality. After several years of experience, we think that object-oriented design does improve software reuse and software quality in two major ways, maintainability and extensibility.

Maintainability was improved by inheritance and encapsulation. Inheritance reduced the amount of code to be maintained without increasing that code's complexity. Encapsulation improved our ability to quickly locate code. Extensibility was also improved by inheritance: it usually allowed us to add new functionality without altering working software. This led to improved confidence and shorter unit-test and integration-test intervals. Encapsulation insured that, if tested software was modified, the effective scope of our change was usually well understood.

Although object-oriented design did provide increased software reuse, we think wasteful duplication will continue until environments are created that allow a large number of developers to easily share, locate, and enhance a large common set of classes.

Some people claim large software development projects will notice large productivity improvements if object-oriented designs are applied. However, it is still difficult to predict the behavior of complex systems, object-oriented or not, since modules have complicated hidden

interdependencies. Much as "independent encapsulated" biological objects interact to create dynamic interconnected biological environments, software objects interact in a common interconnected computing environment. (Some examples include use of resources such as processors, memory, disk storage, and operating system resources). Programming complex systems with object-oriented languages will be an imprecise activity until object-oriented languages and their environments can present this interconnection information to a system designer.

Ragu Raghavan

Mentor Graphics has been developing and marketing workstation based CAE/CAD software since 1981. The current generation system features a variety of applications built on top of a large (almost 1,000,000 lines of Apollo Pascal) shared software library.

The library has been evolving to meet the needs of a growing number of applications. This evolution is becoming increasingly more painful. The solution to this problem is to re-architect the entire system using an object-oriented language with inheritance.

C++ was chosen as the language of the future. It provides very good object-orientation. It is fully typed and type-checked at compile time. Its performance/efficiency is comparable to that of Pascal or C.

A team of experienced developers spent a year re-architecting key portions of the library in C++. The areas chosen for initial development were memory management, the user interface management system, the graphic subsystem and the design data management system.

Based on the results of this investigation, the decision to go with C++ has been reaffirmed. A strategy has been put in place for the large scale infusion of object-oriented programming technology into the company. The first step was an intensive in-house training course in C++ and object-oriented design.

The nucleus of a new object-oriented library is already in place. The conversion from Apollo Pascal to C++ will follow a phased approach. The viability of mixing Pascal and C++ has been demonstrated. Thus, each application group has the option of either converting over to C++ in one shot or gradually evolving the code from Apollo Pascal to C++.

The need for good software engineering tools is accentuated when moving over to an object-oriented language with inheritance. A C++ class browser was the first tool developed. Another tool developed was a progressive disclosure text editor. Fortunately, the symbolic source code debugger on the Apollo worked OK on C++.

Software design and coding standards were put in place even before any prototype code was written. This decision has proved to be extremely valuable.

Carl Nelson

Our applications are small by comparison to many products written with object-oriented programming techniques. The Navigation Application was only a prototype. It has a graphic and geographic window orientation and was used to prove market feasibility. Our SCSI disk formatter and driver installer has a simple dialog interface. The objects are used to support differing hardware characteristics. It is sold to hard disk drive manufacturers and do-it-yourself disk builders for the Mac. Archive and Restore are disk archiving programs that are marketed to disk drive manufacturers and private label software sellers. Restore used 25% of original Archive code. We started with 70% of the original code and rewrote 60-70%.

Product Statistics:

Program	Lines	Bytes	Units	Months	People
- Navigation Application	8039	274133	-	5	1
- SCSI Disk Formatter	7600	207967	-	5	1
- Archive	14658	358662	10	8	1.5
- Restore	7942	183048	8	2	1
- MacApp Itself	32111	824195			
UMacApp	12558	335358			
Object Support	4355	100727			
UList	645	16749			
UAppleTalk	3101	87201			
UDialog	2146	57764			
UPrinting	2528	69899			
UTEView	2178	55837			
Debugging Support	3451	77800			
Resource Support	1149	22860			

MacApp is a big win for small to medium applications that have one or more of these characteristics: 1) Use the standard Mac user interface, 2) Have a concept of views that represent flat pieces of paper that display a view of your data, 3) Read, write, and display data that resides in one or more documents, 4) Have simple dialogs, 5) The interface is uncertain or will change.

It is a good prototyping tool for the non-standard or large application. MacApp or any object-oriented language is just another tool to get the job done. Good judgement when designing an application for a machine outweighs any advantage of an object-oriented programming language. Standard data structures and algorithms that map onto a machine's architecture are best done in standard ways. Using an object-oriented programming framework to organize these data structures in a high level way is where the real win lies. Application speed should not be a concern. You should speed it up after you build the prototype and bottlenecks are found. MacApp also serves as a common framework and notation from which to describe the pieces of the application. This works fine as long as the complexity of a piece does not exceed a threshold of understandability. A rule of thumb is 2-3 minutes of conversation at a white board.

In doing object-oriented design, an obvious thing to do is to make the objects correspond to things in the application, such as:

UmyApp	- Application object
UmnDoc	- Document
UmyCmnd	- Command objects
UCommon	- Common place to store stuff
UmyFrames	- Frame overrides
UmyMainView	- Main view in program
UmyFirstView	- A support View
UmySecondView	- Another support View
UmyWorkObject	- objects that hold data or work in the application

Pascal has some shortcomings as a language onto which to graft object-oriented concepts. The *unit* structure for separate compilation causes problems with circular references, mainly as a result of strongly typed data structures. The solution is to use "spaghetti inheritance" and type coercion. The types are changed to accommodate strict inheritance, and coercion is used to make simple ancestor types into a descendant type.

Object reorganization during the development of a project takes time, but the benefit is the review of code and object design for possible repartitioning. In MacApp, the *unit* implementation is combined with the interface. Separation of the definition from implementation, as C++, Modula-2 or TML Pascal seem to have done, would solve some of these problems.

Alen Schiller

GIS is a geographic information system that stores information associated with graphics. It differs from System 9 in that the user can ask questions about the graphics and images displayed in terms of spatial references as well as by attribute references: for example, "show all houses within one mile of this electrical substation." The databases are currently on the order of 20-100 megabytes; the long term goal is 1-5 gigabyte databases.

IMPLEMENTATION

Why we looked at object-oriented programming:

- Dissatisfied with time frame needed to put an idea into code and demonstrate it.
- Traditional software lifecycle approach wasn't working.
- Traditional approach made code hard to change and maintain.

We phased it in: Picked areas of implementation where failure or performance problems would have minimum effect on the project.

Implementation Areas (Initial):

1) Human Interface: (Menu Handler)
   - Requirements were complex and often conflicting.
2) Plot Task:
   - Fairly standalone, complex and must support many plotters whose requirements were similar but had specific differences.
   - Implemented this part by using GKS metafile output and creating classes and objects for each type of entry in GKS metafile.

Areas of implementation (Later on):

1) Database
   - Extendible hashing
   - Range trees
   - Cache management
2) Application Language
   - Intent was to build a language that users could use to access the system without having to know C or the internals.
   - Interactive and interpreted with an on line debugger. After the user has got it working it should be compiled and run to maximize speed.
   - Current Status: Development stage using the vendor's product rather than developing our own. This is an *important* point. We felt it was more important to develop the "objects" rather than concentrate on the environment.
3) Human Interface: Devices, Windows, etc.
   - Extended menu handler to incorporate other devices, windows.
   - Rather unique in that our product *must* support input from multiple devices including alphanumeric screens and 16 button "puck" or mouse.

PROBLEMS

1) Learning Curve
   - Productivity initially goes down
   - Pays for itself in the long run
   - Vendor also has a good training course

2) C and Object-Oriented World

- In the Unix world, C can be somewhat of a "religious" language.
- Difficult to persuade people that the object-oriented approach is "necessary" or "better."
- Need a "phased" non-pressured approach to introduce object-oriented C.

3) Code: Looks suspiciously like C code

- First object-oriented C code looks like regular C procedure calls, except they are called methods. Similar to transplanting a Fortran programmer to a C environment. You get C code that looks suspiciously like Fortran. The old learning curve again.

4) Debugger:

- Need a debugger that understands objects. If the object language you decide to use extends the basic language, be sure you have some reasonable way for your programmers to debug their resultant code. This debugger should understand the language extensions and yet allow you the same abilities as your system supplied debugger. In our case, our vendor supplied us with a product that not only allows us to debug object code but also to write and debug regular C code.

5) With Vendor Code: "The Bug"

- If you make the commitment to use a particular vendor product, you better check to see what kind of support you get for any bugs you find.
- One of the reasons that we chose Objective-C over C++ was that at the time of evaluation (over 2 years ago), Objective-C was a supported product and C++ was not.
- You *are* going to find bugs in the software supplied to you. Vendors don't have any more magical ways of developing and debugging software than you do. The important thing is to make sure that you can get immediate support or work-arounds. If the company does not have a hot line number or people dedicated to customer software support you might want to give it a second thought.
- In our specific case we went to the extent of visiting the vendors offices and talking to the people in software development.

6) NIH: Not Invented Here

- You'll hear this *a lot* from the people on the panel. *Object-oriented programming doesn't solve this problem.*

7) Collections

- Same object in different collections. What happens when you start freeing collections. Memory Management a *severe* problem.

8) Over a Network

- Our environment is Sun workstations connected over Ethernet with NFS. How to share or transfer objects over the net? This is a non-trivial problem and we have not satisfactorily solved it. The problem is aggravated by the fact that when objects are created they have a pointer as a unique identifier. This is totally useless over a net.

9) Fuzzy Requirements

- Many of our system requirements were not clearly spelled out. They changed over time. Our product sells internationally. It was not uncommon for the European market to have a need that was exactly counter to what is needed by the North American market.
- It appears that using object-oriented programming does help in a limited way in trying to develop a product in an environment where the requirements conflict.

10) Prototyping

- Seems to be quite good here but mainly because of vendor's foundation classes.

Hamid Eghbalnia

During development of applications using object-oriented technology for the past several years, a number of issues have emerged. In retrospect, most issues are not unexpected topics, but are of a fundamental nature. Focusing on these fundamental issues should provide a smoother path during product development. If one point is to be stressed, that point must be that object-oriented languages are mainly a toolbox intended to provide a more natural representation of the problem domain.

One must distinguish between object-oriented design, implementation, and languages. The act of design in terms of objects or object modeling can take place independent of the language. This object-based model can be implemented using traditional programming languages. Object-oriented languages are effective toolboxes to bridge the gap between the object-based design and implementation. Viewing object-oriented languages as toolboxes will emphasize the need for training in order to use the tool appropriately.

The toolbox view will emphasize another aspect as well. As toolboxes become larger, finding the right tool among the many tools available becomes more time consuming. Therefore, techniques for organizing toolboxes become critical - otherwise, the toolbox is rendered ineffective. The sheer number of objects and methods in a large system can overwhelm the memory of any programmer before overwhelming machine memory.

Management of memory and the storage environment is an important system issue. This is despite advances in hardware and boards with more real memory for less money. The persistent storage of many revisions of objects and the management of this environment requires more than "plugging in another memory board." It is true that this is not specific to object-oriented environments. However, it is also true that these environments tend to use more memory and are more resource intensive and may therefore need a special focus in order to mature effectively.