Introduction to Object Oriented Approaches

Steven M. Lewis, Ph.D.

RhinoDiagnostics Corp.

Introduction

In the past several years object oriented programming has become extremely popular. It is important for the Forth community to understand this approach and the reasons for its popularity. This introduction defines some of the critical concepts in object oriented programming and some of the reasons for its popularity. It discusses the basic methods and vocabulary of object oriented programming. Finally it discusses some options for implementation of objects within Forth. Several general principles must be realized from the outset.

A Paradigm

First, object oriented programming is not a language or a specific set of programming tools, rather it is a paradigm. A paradigm is a new way of viewing the world, in this case the entire concept of a program. As Thomas Kuhn **[Kuh62]** points out in his book *The Nature of Scientific Revolutions*, when a paradigm shift occurs, it may force people to rethink almost everything they are doing. Programmers developing object oriented software will eventually find that their code looks nothing like what they have written before. Object oriented Forth code will grow to be as dissimilar from conventional Forth as Forth is from C or PASCAL.

The popularity of object oriented programming is the result of trends in computer design over the past several decades. The speed and memory of 'typical' machines has roughly been doubling every two years or less. Today a 'typical' machine might be a 33 MHz '386 class machine with at least 8 MB of memory. In a few years we expect systems to be '586 class with at least 32 MB memory. While machines have been getting faster, larger and smarter, programmers stopped getting smarter at some time in the late Pleistocene. As a result, the balance between the three resources under a programmer's control: the computer's time, the computer's memory and the programmer's time has shifted. Compared to earlier approaches, object oriented programs consume more memory, usually take more time to compile and more time to run. The tradeoff is that good object oriented programs tend to be easier to write and to have fewer errors than conventional programs. Thus, the system's resources are used to allow programmers to write better, 'safer' software.

Concepts

The important ideas in object oriented programming may be represented as a collection of concepts. Many of the important gains may be realized by systems written in conventional languages such as C or Forth which use ideas and approaches espoused by object oriented programming. These benefits come from careful adherence to specific rules. The advantages are similar to those obtained writing structured code in FORTRAN by eschewing GOTO statements. More object oriented coding forces changes in the style and the language. Thus, object oriented

software represents a continuum. This discussion seeks to outline the major steps and the decisions in approaching object oriented coding.

Structures

The first important concept is the notion of collections of data. This corresponds to a C struct, Pascal record or ADA packet. There is no direct equivalent in Forth, but the tools exist to easily implement such approaches. I will term such a collection as a STRUCTURE. A structure is a group of data items which can be passed to execution units (C functions or Forth words) as a packet. Consider, for example, a structure describing a person's bank account to an ATM. ATM operations may need access to the person's name, current balance, credit history, credit line and an encrypted form of the PIN. Rather than passing each of these items to a routine, it is faster, safer and cleaner to create a structure containing all the information and pass in a reference to that structure. I will talk about each of the terms in a STRUCTURE as a FIELD. Thus, for example, in a bank account, the account number, PIN, balance and holder name will all be fields within the Account structure. Most high level words take not single arguments but rather organized collections of data. Grouping of data makes most functions cleaner since passing a data group guarantees that all required data is present. While this ability to group data is not sufficient for object oriented programming it is a vital prerequisite. Older languages, such as FORTRAN and BASIC, lack the ability to define data groupings. Forth lacks this ability in the core word set but it is easy to add.

Data Typing

Once structs are introduced into a language, argument typing becomes important. In classical Forth relatively few explicit data types are used. Typically, Forth programs deal with cells, addresses (which are treated as cells), doubles, characters and floats. With structs, a large variety of other types are introduced. Clearly it is an error to pass the address of a RECTANGLE structure to a routine expecting a COLOR structure. Traditional languages such as C (or Forth) lack a way to tell the programmer when a routine is being called with the wrong structure. ANSI C offers type checking. Typically type checks are performed by the compiler which will flag as an error, an attempt to pass the address of one type of structure to a routine expecting another. Smalltalk can detect these errors at runtime (and pays a significant performance penalty for this ability). In theory, Forth stack comments can be used to test the use of routines. However, in practice, type safety will require major changes in the Forth language and to Forth compilers. While type safety is not critical to object oriented programming, the proliferation of data types in object oriented systems make tools to help the developer test proper use of types become increasingly critical.

Data Hiding

Once data can be grouped, another important property is the notion of data hiding. Once one has defined a grouping of data and a number of functions to operate on that group, it is apparent programmers shouldn't know the internal data. Suppose a developer is charged with developing a complex structure, say a window on a graphic screen. He could give other developers a list of functions (Forth words) which operate on a Window. There might be a word to create a Window, a word to destroy one. Other words would hide, show, move and resize windows. In a typical GUI a couple of dozen functions describe everything that one can do with a Window. Suppose we say that those few dozen functions represent the ONLY possible uses of a Window. Then, if the only thing we can do with a Window structure is to create it, pass it to functions which understand a Window and later destroy it, then there is no need to understand the internal data within that structure since there is no meaningful operation we can perform with that data. This ignorance is a real advantage. If there are multiple programmers using the structure, they can

write their code given only the set of words which can use the structure. Furthermore, the programmer or group charged with maintaining the code operating on the structure can modify the internals of that structure secure in the knowledge that as long as all publicly announced routines operate correctly no code will be broken by changes in the structure.

Data hiding breaks a program into two sections. There is a public declaration of a structure's functionality which are the calling arguments (stack diagrams) of the routine. These are published and may be used by any developer. There is also private information about the precise field of a structure which is only available to the programmers responsible for maintaining the code associated with that module. This arrangement allows a complex problem to be broken up into a collection of independent modules, each maintained separately and, in theory, supported by separate teams. The ADA language makes extensive use of the concept. At this stage of this discussion, developers have created a structure and a collection of functions operating on that structure. In principle, any language which supports structures may be used to write structures in this manner.

Objects

Previously we viewed data as a collection of application specific data elements. Later, with date hiding, we began to associate a limited collection of functions with a particular data structure. The structure and the functions associated with it can be treated as a unit. The next step is to make the structure representing the data include the relevant functions. Imagine a Forth structure including an array of code field addresses (CFA's). Instead of calling for a function by name one can write code asking for the fifth function in the array. Once the collection of functions is grouped with the data we have an Object. An Object has two parts: a collection of data and a collection of functions which can operate on that data. What advantages accrue from placing functions with the data? First, if the only operations possible on an object are to execute code accessed through the Object, one guarantees that the Object will not be misused. It becomes impossible (well at least more difficult) to pass an Object to a word expecting different data. Second, by tightly associating functions with data, there is an assurance that the functions cannot be altered without also altering all calls to those functions.

Another important change in coding style is that as functions are associated with a structure, the concept of a structure is broken into two terms. A CLASS describes the fields of an instance of a structure. It tells the type and name of each field and the type and name of each function associated with the structure. An instance is a specific structure created by the CLASS. The relationship between CLASS and INSTANCE is the same as that between a Forth defining word and the word produced by the defining word. In essentially all Forth implementations of objects, a class is merely a special form of a defining word. Thus, for example, a class might by an IBM PC with a 386 processor. An instance might be the computer on John's desk with a specific serial number.

Grouping the functions within the Object without other modifications, such as inheritance and late binding, represents largely a semantic difference from earlier approaches. One has moved from a structure and a collection of functions to the structure and functions embedded within the structure. Although the calling sequence is slightly different, and safer since it is harder to misuse an object, the coding functionality and style are similar.

Functions associated with objects are called methods. In object oriented compilers there is a way to associate a method with a name. By convention in this discussion I will end the names of methods with a : , so a method called Print will be Print:. If one has a Window object named foo, one may invoke a Resize: method to resize the Window foo. In all object oriented systems one either writes Resize: foo or foo Resize: to accomplish this. Object oriented systems differ in their view of what happens when the text Resize: foo is encountered. In all cases the text Resize:

is used to locate a Window-Resize function which is executed. This process is called binding. Binding associates an internal function with a call to that function. However, object oriented systems may differ in two respects.

The first is when the binding takes place. Systems which use early binding look up the function when the code is compiled and compile it. Systems which use late binding look up the function when the code is executed. These seemingly minor differences make major differences in the power and style of coding. A second difference is the relative roles of the object and the compiler in binding. In heavily object oriented systems, the object plays a major role in interpreting and binding. In object oriented systems more heavily based on compilation such as C++, the compiler plays a much more major role in binding, limiting the role of the object. Both approaches have been separately used in Forth.

Classes

Typically, a class may be thought of as a description of a structure and the code for all of that classes' methods. One important function of a class is to create an instance of that class. An instance is a specific object representing an object created to fit the class description. Consider a class describing a Person. The class will describe data fields such as NAME, AGE, SOCIAL_SECURITY — The class Person describes the structure of any Person. An instance of Person will be a structure with NAME set to "John Smith", AGE to 34 — Usually a class will have a number of instances.

A major new programming approach is inheritance. The idea of inheritance is to build new classes of objects from existing classes. Usually new classes add new data, new methods or both. For example, one might build a class MilitaryPerson as a specialization of Person. A MilitaryPerson might add fields for RANK, SERIAL_NUMBER and SUPERIOR. New methods might be KP: and court-martial: which lack meaning outside the military. A major concept in inheritance is that a MilitaryPerson is a specialization of Person. Any routine which can use a Person can also use a MilitaryPerson. All of the data in the class person is also present in MilitaryPerson and all of Person's methods may be applied as well. Inheritance represents a major gain since we have to write very little code to build the class MilitaryPerson. The only code we need to write is where a MilitaryPerson behaves different from ordinary Persons. As a specialized class of Person, Military Person is a SUBCLASS of the class Person. Person is called the SUPERCLASS of MilitaryPerson. Another name for the class hierarchy is the IS_A hierarchy, i.e. a MilitaryPerson is a Person.

Hierarchies

Object oriented systems represent hierarchies of classes. A root class will declare some general behaviors and subsequent classes will specialize the behaviors. One might, for example construct a hierarchy of class for:

PhysicalObject - things with size, weight, position

Toy - physical object used in games

Ball - spherical toys

Billiard Ball

Cue Ball

In developing the hierarchy we define properties once at the appropriate level. Because a Cue Ball is a Toy we know it is used in games. This fact is encoded once while developing the Toy class. Because Cue Ball is a Ball we know it is round and rolls because this information is in the Ball methods and applies to all balls. Finally, Cue Ball has special behaviors such as being the only ball which can be hit with the cue and penalties when it is sunk.

Inheritance is an extremely powerful aid to programming because it allows the developer to specify only those behaviors which cause a new object to be different from a known object. Furthermore, developers can build ABSTRACT objects to allow many objects to share code. For example, a developer may be describing the behavior of Billiard Balls, Soccer Balls and Baseballs. In order to facilitate code reuse, he develops the ABSTRACT CLASS, Ball. He will never actually create an instance of Ball, only a specific specialization, say Soccer Ball. However, the fact that a ball is round and rolls is common to all balls and may be placed in the abstract class.

Another major innovation in object oriented programming is the idea of late binding. Late binding is the idea that if a group of objects share a method with the same name (but usually different function) the actual code executed depends on the type of object sent the message when the program is run. The developer can write code without knowing what type of object will ultimately execute the code. For example, consider a function like Print. In Forth when we want to print a number we say . , when we want to print a DOUBLE, we write D. , for a float we say F. . If there is a complex structure such as a STUDENT, we will write a special word such a PRINT_STUDENT. In writing code we need to know what is on the stack and what word will print that data. In an object oriented system, we can define a method PRINT. Sending Print: to an object representing a float will perform F. , sending it to a STUDENT object will print name, classes and grades, sending Print: to a CHAPTER object might print a chapter in a book. For all objects supporting that method we can simply send the message Print:.

Polymorphisms

This ability to use same method, in this case Print:, to execute different code for different objects is called POLYMORPHISM. An essential feature of polymorphism is that the code executed by one class may be replaced by different code in a subclass. This process is call OVERRIDING the method. Overriding may entirely replace the code for the method or merely extend it with the newer code calling the older code as part of its execution.

Finally, the relationships among objects is a recurring theme. The functioning of any object of significant complexity can depend on that object's relationship to other objects. If, for example, one is building an object to represent a course, then the Course object will make reference to an Instructor, a collection of Students, one or more Textbooks, and perhaps a collection of Exams. Each of these will represent separate objects. Much of the development of a system comes from defining the nature of the relationships between objects. Some objects, Exams for example, are dependent on the Course object. If Computer Science 342 ceases to exist, the exams for that course would have no reason for independent existence. Other objects such as the Instructor will continue to exist in most systems. The designer of object oriented systems spends a large fraction of his time worrying about relationships among objects. Some relationships are one to one, such as the relationship between a Student and an Exam (Students take an exam only once.) whereas others are many to one, such as the relationships between a course and the students (Students may take a number of courses and Courses contain a number of students.). A designer has to worry about whether a course has a single Instructor or may have several Instructors.

Coding

Once relationships are defined, much of the coding can be written in terms of relationships. One can, for example write the pseudo code to print a grade sheet as:

```
ThisCourse Name: Print:
Instructor Name: Print:
ForEach Student
```

ThisStudent Name: Print:

For each Exam

ThisStudent Print: Grade Print:

Here Print: is a method to print an object, string, number or a complete object. Name: gets the name of an object.

Methods in polymorphic coding are almost always implemented as late binding if the system supports the concept. The power of late binding is the ability for an developer to apply the same operation to a group of objects of indeterminate class including classes which may have been created long after the code was written. This style is used extensively in most object oriented projects.

Implementations

Any object oriented system has to address the issue of polymorphism, that is that the same method, Print: for example, will execute different code depending on the object it is associated with. Object oriented implementations differ in the way this code is selected. Those differences involve three issues: first, when the selection is made, whether at compile time (Early binding) or run time; second, the distribution of responsibility with making the selection between the method word and the instance word; and, third, the selection mechanism. In Forth there are a number of possible object implementations. Almost all implementations make a class a defining word which relates instances in the DOES portion. There is much more flexibility in the implementation of instances and methods.

A method is invoked by pairing an instance (or a reference to an instance) with a method, i.e. by writing Print: FOO or FOO Print: . The actual code executed may be associated with either the method or the instance. Similarly, the responsibility for selecting the proper code to execute may be assigned to code within the method or the instance. In all cases, Instance has the responsibility for leaving its address on the stack at run time. Different approaches have been used to allocate responsibility for selection of the code.

In most cases I will assume that the code that is ultimately executed when a method is invoked is passed the address of the affected instance on the top of the stack with any other argument lower on the stack. In the case of Print:, the instance is the only argument. Suppose George is an object of class Person, SomePerson is a reference to an object of class Person which will be set at run time and Print: is a method defined in Person. The following discussion deals with the case when an object is paired with a method, for example, Print: GEORGE . It is a separate, interesting problem to deal with the case when an object such as GEORGE is executed without a method. Most implementations will have the object leave its address on the stack in this case. Determining that no method has been passed to an object is a non-trivial problem which I will not address here.

Early Binding Approaches

Early binding almost always makes either the method, the instance, or both, an immediate word. This is because computations to select which of several implementations of a specific method need to be performed at compile time.

Immediate Instances - Executable Methods - Vocabulary Classes

A common approach to early binding is to associate a hierarchy of vocabularies with a class. Person, the class, is a vocabulary with methods such as Print: defined within that vocabulary. In this case George can be an immediate word with the following actions. First, George compiles code to place his address on the stack at run time. Second, George looks up the next word in the

Person vocabulary with the search proceeding to higher class vocabularies if needed. If find succeeds, the resultant word, the method, is compiled. At run time, George's address is placed on the stack and consumed by the Print: method.

The syntax is:

George Print:

with the immediate instance preceding the method.

SomePerson must also be immediate and must act in much the same manner as a known instance. Pountain uses this general approach in his work **[Pou87]**.

Immediate Instances - Methods tokens - methods in class

A second approach is to make the method a token which is associated with a piece of code contained within a structure created with the class. In this case the method is simply a token used in searching for the appropriate code. Because a method is just a token, methods may be global and need not be associated with any piece of code. Also, the developer has control over the search process instead of relying on the standard Forth vocabulary search. The syntax is similar to that described above.

Immediate Methods - Immediate Instances

In this approach, both the method and the instance are immediate words. The method reads the next word which must be the name of an object or appropriate class. The method looks up the instance in the current vocabulary and in some manner, there are a variety of ways this might be accomplished, finds the appropriate code for that class of object. The code to place the instance's address on the stack at run time is compiled along with the appropriate code to execute. References to objects of a particular class must be detected and handled in the same manner as the original object. The code may be located through the class of the instance or may be held in a complex structure within the method itself.

Approaches to Late Binding

In late binding, the type of object is not known until run time. At run time an object must determine whether it is being sent a method and if so what code to execute. As described above, there are several approaches. These may be separated into those placing intelligence in the method and those placing intelligence in the instance.

Method is a Token

In this approach the method places a number on either the parameter or a separate methods stack. The object reads that number and uses it to determine the appropriate code to execute. The most common implementation builds a table of code addresses for each class. The most efficient approach is to make the Method hold an index into the appropriate entry in the table. Thus, Print: might hold the number 11. For all classes supporting the method Print:, Print: is the 11th entry in the table. This makes late binding an extremely fast operation, one fetch + offset for the table, one fetch + offset for the code and an execute. Every class has its own copy of the table. The size of the table for each class is proportional to the number of methods applicable to that class. Methods are overridden by placing the address of the new routine in the appropriate slot in the table. The method is used to retrieve the appropriate code from the table and execute it. This approach requires the every class to generate a table large enough to hold the addresses of all methods an object in the class might use. Because a single table is used for all instances of a class, there is no penalty for having a number of instances. Tables will get large as the number of classes grows. The most direct version of this approach requires method names to be globally

unique although more sophisticated versions can allow different class trees to share a method name. In this approach Print: FOO will translate to something :

11 METHODPUSH	(Place 11 on the method stack)
FOO	Put address of FOO on stack
DUP TABLE@	Get Foo's jump table
METHODPOP SWAP	Get the Method index (11) from method stack
METHODCODE@	Get 11th item in the method table
EXECUTE	Do it

Here, the methods are pushed onto a method stack to simplify parameter stack manipulations. This is the approach used by Boron, **[Lew87]**.

Method is an Identifier

An alternative approach is to treat the method as an identifier and to search a list of possible methods for the identifier, code pair. Each class has a list of those methods defined for that class. The list for the current class is searched and if the method is not found, then the superclass list is searched. Because the first entry encountered is executed, methods can be overridden by adding a new identifier, code pair. This approach minimizes the amount of memory used since method addresses are not replicated in all classes but require an expensive search at run time. This is advantageous in memory-limited systems where the number of methods is small and run time performance is not critical. The run time penalty imposed will be roughly proportional to half the number of methods defined, since on average half of all methods will have to be tested.

Method is a Patch

Dreams **[Bro94]** presents yet a third approach which uses a context switch to patch the CFA's of all affected words at run time. This approach allows code to look more like generic Forth than object oriented code. It imposes a run time penalty since every time a new class is invoked all methods must be patched. The run time penalty when objects of differing classes are invoked will be roughly proportional to the number of methods defined, since all methods must be patched. However, patching need only be done when there is a context shift between two classes (Dreams).

Instance Holds a Token

A second approach to late binding is to place an instance within the Method references to all possible implementations, typically as a table or list. The instance then holds a key to access this table and find the appropriate code. The same basic approaches discussed above are available. The method may hold a table of all classes and the class token may be an index into this table. Alternatively, a list of class, method pairs may be searched for the proper class. Both of these approaches present some difficulty in implementation. Because the size of the table for a method must be the number of classes, as new classes are defined, the tables for all methods must grow. Contrast this with the previous approach where the methods are part of the class and as new methods are added, they are only applicable to the new classes in which they are defined.

Searching class, method pairs also presents some difficulties since there is no guarantee that the class will be present in the table and it is possible that the entry is for some ancestor class. Then each entry must be tested to see if it represents the target class or any ancestor class. On the order of the number of implementations times the number of ancestors needs to be tested at run time. If both of these number are small, this approach may still prove more efficient than searching all methods.

Comparison of Various Approaches

Three criteria are important in considering various approaches to object oriented design. These are run time performance, memory requirements, and independence. The first two are straightforward. There is a natural tradeoff between time consuming activities at run time such as searching and patching, and memory requirements for multiple jump tables. The designer has to make decisions about the resource requirements he wishes to impose and select an appropriate tradeoff. Independence is a more difficult requirement to define. As an object oriented program grows, there is no way for the developer of the higher, base classes to know how many sub classes will be generated, what new methods these classes will define and how many times existing methods will be overridden. Whatever approach is used should be prepared to deal with systems that are potentially quite large. There are several pitfalls to consider early in the design. Fixed size tables will invariably be outgrown. When tables are generated, there has to be a means to guarantee that the possible number of entries is limited only by system memory. Also, once a table has been generated, a way must exist to either extend the table as new methods and classes are defined or to guarantee that no extension is required. Globally unique name spaces are another problem. In large projects employing multiple developers, it is extremely difficult for all developers to keep track of all names employed. Schemes, such as vocabularies, can partition name spaces, but it is still important to guarantee that the correct name space is searched at run time.

Correspondences between Conventional Terms and Brown's Terms in Dreams

Dreams **[Bro92]** is an unusual implementation in that all Forth words are treated as methods of some root class which Brown terms Reality. All variables are treated as fields within this vast root, Reality. Dreams differ from objects in that the instances of objects alter the execution context only to process a single method. Dreams alter the execution context for a relatively large number of instructions.

Dream - A dream is roughly analogous to a class. The CFA values set at initialization may be seen as methods overridden by the Dream. Dreams are both classes and instances, the difference between the template nature of a class and its implementation is not distinct in dreams.

Essence - If a Dream is an object, the essence of a dream is a pointer to that object, that is, the address of the data which defines the object.

Fantasy - A Vision, see below, is a collection of Dreams. Each overrides certain words and declares certain local variables. A Dream (class) which is the union of these is a Fantasy. This corresponds roughly to a subclass in conventional system.

Imagine - Override, this changes the meaning of a word in a specific context.

Melieu - The currently active object. The concept has little meaning in conventional object oriented programming since objects are active only for a single method.

Ponder - execute within a context of a dream. Since all words are treated like methods, pondering is equivalent to executing a method of the dream object.

Relapse - This corresponds to a applying the copy operator to an instance to produce a new, identical instance.

Reality - A root class, but encompassing the entire base Forth system.

Regress - Call superclass method.

Thought or Message - Method or Forth word. The distinction is unclear but a thought seems to be the word, Print: for example, and a Message is the code executed within a specific environment.

Understanding - This is a concept not usually encountered in object oriented.

Vision- Vision is a nested set of dreams and corresponds to a class hierarchy such as the PhysicalObject - Toy - Ball hierarchy discussed above. Unlike the usual hierarchies in which the order is fixed, visions may represent arbitrary nesting.

Conclusions

There are a number of options involved in the development of an object oriented system. Forth gives the developer the freedom to select from a number of different approaches depending on the nature of the problem. In systems which make small use of objects, any approach is adequate. When a large fraction of the code is object oriented, the words that handle objects and methods become as important as words that handle the stack. Developers must carefully weigh alternative implementation for reliability, speed and resource utilization as choices here are critical to successful application development.

Dreams gives an interesting new view of the problem.

References

[Bro94] Brown, R.J., 1994, " Dreams: A Message Passing Object Oriented System for Forth", *J. Forth Appl. and Research*, 6(4) p 275

[Pou87] Pountain, R., 1987, *Object Oriented Forth Implementation of Data Structures*, Academic Press, London, England.

[Lew87] Lewis, W. M., 1987, "BORON: An Object Oriented Forth Extension", *Rochester Forth Conference Proceedings.*

[Kuh62] Kuhn, T., 1962, *The Structure of Scientific Revolutions*, University of Chicago Press.