Object Oriented Programming in Fifth

Cliff Click and Paul Snow
P.O. Box 10162
College Station, Texas 77840

May 17, 1986

Abstract

Object orientated programming is fundamentally different from traditional languages. Instead of passing data structures to functions, the programmer passes messages (function names) to objects (data structures). The objects determine what function actually gets executed. Our system was modeled after SmallTalk and is implemented in a **Fifth**, a Forth based programming environment.

Introduction

Object oriented programming is a state of mind. It's nature is very different from traditional languages. Instead of setting up data structures and passing them to functions, the programmer defines objects and their behavior, and passes the object messages. Unlike C an object's behavior is bound to the object. An object cannot be made to "behave" in ways it was not "meant" to.

An object is an entity with size, contents and behavior. A description of it's contents and some of its methods (possible behaviors) come from the class the object belongs to. Each method is associated to a particular message, and describes what an object does in response to a particular message. When the programmer sends a message to an object, the associated method is executed. It is the object's responsibility to understand and interpret the message. If the message is not one the object's class understand, the object's super class (the class that is the parent of the object's class) messages are checked. Inheritance continues up a tree structure until the message is understood, or the root is reached.

We defined the basic elements of our system in terms of **Fifth** data structures. Some the data structures we designed are objects, classes, and methods. We then discuss some of the routines needed to get this system off the ground.

We modeled our design after SmallTalk. SmallTalk is complete implementation of an object orientated environment. Unfortunately, it only runs on minicomputers. SmallTalk versions running on micro computers have problems with processing power. Our version takes a few shortcuts in the interest of speed.

To implement this system, we used our programming environment, **Fifth**. **Fifth** has a number of features essential to the object orientated approach that make it a better starting point than a standard **Forth** system. **Fifth** has tree structured scoping, memory management, and a large memory model.

Objects

An object is a complete entity. It can be considered to be a collection of related information. This information may be data, code, or other objects. It has attributes like size, contents and inheritance of behavior. If a change to an object is desired, the object is requested to make the change. Or in other words, an object changes itself; it is never changed by another object. Objects are referenced by a single value, an object pointer.

An object has a class. The class describes the layout of fields within the object and the messages the object understands.

An object has a reference count. This is a count of the number of times an object is referenced (has a pointer to it) in the system. When this count falls to zero no more references to the object exist, and the object can be deleted; the programmer does not have to explicitly free objects (e.g. the C function `free()`).

Messages and Methods

To change an object or to make it exhibit some behavior, the programmer sends the object a message. Every message an object understands has a corresponding method. The method is the code the message invokes. The object interprets the message and returns a result (an object) based on the method and any parameters. Contrast this to more traditional languages where data structures are passed to functions. In object oriented programming, the object determines what function will interpret the message. In traditional languages the programmer picks the functions, and it is his responsibility to insure the data structures match the function's expected inputs and outputs.

A couple of examples would be in order at this point. The following expression sends a + to 3 with a parameter of 5. (Notice that even simple integers are objects and are passed messages.)

```
    3 + 5
```

In this expression the message + and the parameter 5 are passed to the integer object 3. All integers understand the message + . With the parameter 5, the object 8 (an integer) is returned.

Integers understand one kind of addition, floating point another, and strings yet another. The next example illustrates how a string might interpret the message + .

```
        '3' + '5'
```

In this code the message + and the parameter '5' are passed to the string object '3'. The object '35' (a string) is returned.

The message + had two very different behaviors, depending on the object that received the message. This is a powerful tool, as it lets the same piece of code work on different objects.

Classes and Method Inheritance

Every object belongs to a class. A class holds the description of an object (the organization of data within an object) and the messages the object understands.

Classes have an inheritance tree (i.e. classes have sub-classes which have sub-classes which have...). The parent of a class is called it's super class. Objects that belong to a class use that class's messages and methods. The objects also inherent the messages and methods of their classes' super class. This inheritance continues up to the root object, *Object*. This allows type independent code.

For example, all objects of class *Integer* understand the messages + and * as add and multiply. All *Real* objects understand these messages too. *Integers* and *Reals* are both subclasses of the class *Numbers*. The class *Numbers* understand the message `square` to mean: `number * number`. This passes the message * and the parameter `number` to the object `number`. The object interprets the message * . The same code squares both real numbers and integers; it is type independent.

All classes understand the message `new`. Sending this message to a class returns a new instance of this class.

Data Structures

To implement SmallTalk in **Fifth**, we first designed the basic data structures that make up the system. These are objects, classes, method tables, and methods.

An object in our object orientated language is a standard **Fifth** module with some fields predefined. The first 4 bytes hold the object's class. The next 2 bytes hold the object's reference count (the reference count is used in the memory management scheme).

A class is an object that also has fields to describe instances (objects that belong to the class). The fields are:

- SuperClass — This classes' parent class.
- Text — The text object which describes this class.
- Method Table — The object which holds the table of methods understood by this class.
- Instance Variable Names — A list of names describing the fields found in an instance of this class.
- Class Variable Names — A list of names describing variables shared by all instances of this class.

A method table is an object which has a list of names and method objects.

A method is an object which has some code. The method's code pulls the parameters and the receiver (the object which received the message that invoked this method) off the stack and places a return object back on the stack.

Some needed functions

A convenient representation of names and messages is needed. All messages in the system exist in a table of messages (names). During execution, a message is represented by a 4 byte pointer into this table.

When a message is sent to an object, the proper method must be found. First the object's class is found. Then the class's method table is scanned for a match with the message. If a match is found, the method corresponding with the message is executed. If there is not a match, the class's SuperClass field is fetched. This class is searched as before. This is repeated until the SuperClass field is null — which occurs in the root object *Object*. A 'Message not understood' error occurs if the message is not found.

A store routine that understands the reference count is needed. When a new value is stored in an object it overwrites the existing value. The overwritten object's reference count is lowered, and the stored object's reference count is incremented. If an object's count falls to zero, it is deleted. **Fifth**'s memory management simplifies the implementation of reference counting immensely.

A "compiler" that understands the language syntax is also needed. The compiler converts a class description to a data area with fields defined appropriately for the class. The compiler then needs to set up the method table, and the methods. Within the methods the compiler needs to generate code.

The Data Stack

A key part of this system is how parameters and objects are passed to methods, and how the methods return results. Unlike SmallTalk, all objects and parameters are passed on the same stack.

Before invoking a method all parameters needed by the method are placed on the stack. Then the object receiving the message is placed on the stack. After the message is looked up and a method found, the method is executed.

The method reserves storage on the return stack for temporary variables. Then it executes whatever code the compiler placed in the method. Before the method exits, all temporary storage, the invoking object, and parameters are removed from the stack. The result object is placed on the return stack, then the method returns.

This simple **Forth**-like method of passing parameters reduces some of SmallTalk's overhead. In addition, reference counting is not done with objects on the stack. This is possible because a stack object is both created and destroyed during the execution of the method. The net number of references is zero. The only "gotcha" to this shortcut is if the last reference to an object is destroyed while the object exists on the stack.

Conclusion

To simplify SmallTalk's parameter passing, all parameters are passed on the data stack. A number of complex procedures are needed to build this system, including procedures for compilation, reference counting, method lookups, and name tokenizing.

A number of basic data structures need to be defined. These data structures define the basic design of the system. Some defined data structures are objects, classes, method tables, and methods.

Objects within this system have a class. The object's class defines the fields the object has and the methods the object understands. The class also defines a hierarchy of classes. The object uses messages and methods from any class in this hierarchy.

To make an object exhibit behavior it is sent a message. The message is found in the method tables of the object's class and it's super classes.

Objects themselves have size, contents, a class, and a reference count. The reference count defines the number of times the object is referenced in the system. When this count falls to zero the object is deleted.

An object oriented programming environment has advantages over traditional languages and environments in ease of use and understand. An object orientated programming environment is not beyond the capabilities of a good **Forth** implementation.