Problems in Object-Oriented Software Reuse

David Taenzer, Murthy Ganti, Sunil Podar

U S WEST Advanced Technologies
6200 S. Quebec St.
Englewood, CO 80111

ABSTRACT

This paper discusses problems in object-oriented software reuse and tools that can be used to help alleviate these problems. Two approaches to software reuse in object-oriented programming environments are presented: *construction* and *subclassing*. Previous papers on inheritance have centered on maintenance issues and the problems of subclasses directly referencing instance variables of their superclasses. We have identified another inter-class dependency which seems to cause more problems during the creation of new subclasses. This issue, which we call the *yoyo* problem, is related to objects sending themselves messages which may cause the execution of methods up and down the class hierarchy. Tools we have built to help with these problems are described.

1. INTRODUCTION

This paper describes some problems we have faced using subclassing as a mechanism for software reuse in object-oriented programming and tools that we feel help solve some of these problems. Our specific experiences are with the Objective-C language, but we believe that these issues are generally applicable.

We review two basic approaches to software reuse in object-oriented systems, *construction* and *inheritance*, and explain the advantages and disadvantages of each. Our experiences indicate that reuse is more straight forward using the construction approach. This observation has not been documented in the literature. Object-oriented programming is a very useful and promising technique, but there are many open issues that need to be addressed. We hope this paper will generate more interest regarding practical issues in object-oriented programming and software reuse.

2. OBJECT-ORIENTED SOFTWARE REUSE

Object-oriented programming is advertised as being much more productive than conventional programming [Cox86]. The main reason cited is the reusability of the code. The major features of object-oriented languages which enhance reusability are *encapsulation* and *inheritance*.

Objective-C is a registered trademark of The Stepstone Corporation.

Encapsulation means that objects can be treated as black boxes. All that must be known about an object is its protocol or interface, namely, what messages it responds to and what they mean in terms of the object's behavior.

Inheritance refers to the ability to describe new objects or classes of objects by specifying how they differ from other objects. In theory, this can be done without understanding the implementation details of the parent object. A user should be able to define new classes via subclassing by simply understanding the protocol of the parent class.

These two features lead to two different styles for the reuse of objects: *construction* and *subclassing*. Construction is the approach of defining new classes which create instances of other classes and use them in their standard form. This style of programming relies only on the encapsulation features of the object-oriented approach. The subclassing approach means that new classes of objects are defined by inheriting and modifying the behavior of an existing class. Subclassing uses both the encapsulation and inheritance features of object-oriented languages.

2.1. An Example

A simple example is building a string oriented symbol table. This is a list of strings (keys), each of which has a corresponding value (also a string in this simple case). This can be built in Objective-C by reusing a library class called *Dictionary*. Dictionary objects maintain a list of key/value pairs where both the key and value are objects. The library also has a *String* class for character strings.

The basic behavior we would like the symbol table class objects to have are:

```
lookup: aKey
add: aKey value: aValue
```

The *lookup:* message is used to get the value string for a key string. It returns the value or *NULL* (a special value which cannot be a string) if the key is not in the table. Symbols are added with the *add:value:* method which takes the key and value strings as parameters.

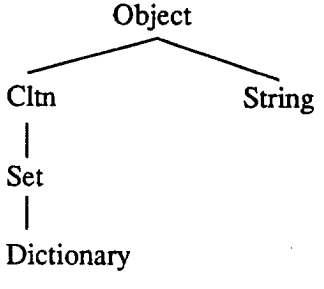

Figure 1: Class Hierarchy for String and Dictionary

As shown in Figure 1, the *Dictionary* class is a subclass of the *Set* class which, in turn, is a subclass of the *Cltn* class. The Cltn class supports collections of objects. The Set class supports collections with no duplicate members. The Dictionary class is a set of associations (key/value pairs). The associations are modeled with *Association* objects. Although the Dictionary class only defines one new class method and seven new instance methods, because it is a subclass of Set, its protocol is quite complex. Dictionary objects respond to ninety-nine messages and the Dictionary class object (called the *Factory Object* in Objective-C) responds to sixteen messages.

2.1.1. The Construction Approach

We can define a new class (SymTab) for symbol table objects via either construction or subclassing. The construction approach would be to define a new class as a subclass of the *Object* class (the root of the class hierarchy) with an instance variable that contains the object identifier of a Dictionary object. The SymTab class must refine the *new* class method (which is used to create new objects) and the *free* class method (which is used to free objects). The code looks like:

```
////////////////////////////////////////////////////////////////////////
// SymTab:  symbol table class based on the construction approach
////////////////////////////////////////////////////////////////////////

@requires String, Dictionary; // uses the String and Dictionary classes

= SymTab : Object (Demo, Primitive)    // SymTab a subclass of Object
{
   id i_dictionary; // Dictionary object which stores the symbol table
}


+ new
{
   // Creates SymTab object by sending "new" message to superclass.
   // Creates i_dictionary instance variable by sending "new" message
   // to the Dictionary class.  Returns the SymTab object.

   self = [super new];
   i_dictionary = [Dictionary new];
   return self;
}


- free
{
   // Frees Dictionary object before sending its superclass the "free"
   // message to free the SymTab object.

   [i_dictionary free];
   [super free];
   return self;
}
```

```
- (char *) lookup: (char *) symbol
{
   // Convert key to a String object and get value via the atKey: method
   // in the Dictionary.  Returns value object converted to a C string.

   return [[i_dictionary atKey: [String str: symbol]] str];
}

- (char *) add: (char *) symbol value: (char *) value
{
   // Convert key and value to String objects and add to Dictionary

   return [[i_dictionary atKey: [String str: symbol] put: [String str: value]] str];
}
=:
```

Objective-C is a superset of the C language with the addition of messages which are put into square brackets. Comments start with "//". The class definition line starts with "=". Class method definition lines start with a "+" and instance method definition lines start with a "-".

The *new* method creates the Dictionary object (by sending the *new* message to the Dictionary class object) and stores the new object identifier in the instance variable for the new SymTab object. The *free* method must send the *free* message to the Dictionary object so that both the SymTab and Dictionary objects are freed.

The *lookup:* method converts the key into a String object and then sends this object to the Dictionary object for this SymTab object. It converts the value to a C string (via the "str" message) before returning it.

The *add:value:* method converts the key and value strings into String objects and then sends these to the Dictionary object. It uses the *atKey:put:* method in the Dictionary object, which will return the old value object for the key (if there was one). It returns the old value as a C string.

2.1.2. The Subclassing Approach

The SymTab class can also be defined via subclassing. In this case, SymTab is defined as a subclass of Dictionary rather than Object. It does not have to define a *new* or *free* method. The *lookup:* and *add:value:* methods are similar to the construction approach, but after the strings are converted into String objects, the SymTab object send messages to itself to access the Dictionary methods, rather than sending the messages to another

object. The code looks like:

```
////////////////////////////////////////////////////////////////////////
// SymTab:  symbol table class based on the subclassing approach
////////////////////////////////////////////////////////////////////////

@requires String;  // uses the String class

= SymTab : Dictionary (Demo, Primitive) // SymTab is a subclass of Dictionary
{
        // no extra instance variables
}


- (char *) lookup: (char *) symbol
{
    // Convert key to String and sends itself the atKey: Dictionary
    // message. Returns value object converted to a C string.

    return [[self atKey: [String str: symbol]] str];
}


- (char *) add: (char *) symbol value: (char *) value
{
    // Convert key and value to String objects and sends them to itself
    // via the atKey:put: message to add the objects to the Dictionary.

    return [[self atKey: [String str: symbol] put: [String str: value]] str];
}
=:
```

This code is not perfect because it does not free the String object which represents the key. Some similar languages (like Smalltalk) use automatic garbage collection so that the programmer does not have to worry about freeing objects.

In this example, both the construction and subclassing approaches are reusing the Dictionary class and slightly modifying its behavior. The subclassing approach requires less code since there is no need for special methods for creating and freeing the symbol table objects.

Another advantage of the subclassing approach is that the new class will automatically inherit a wide range of useful behavior. For example, Dictionary objects know how to print themselves out for debugging. They respond to messages which will send a message to all the objects in the Dictionary. They respond to messages which will return the number of objects in the Dictionary, a list of keys, a list of values, etc. The SymTab class which is based on construction could implement any of these by sending the corresponding messages to the Dictionary object (the instance variable), but this requires explicit coding.

The advantage of the construction approach is that some of the behavior inherited from the superclass may not be appropriate for the subclass. The protocol for the subclass will therefore be much simpler using the construction approach. This reduces the complexity of the class and makes it easier to reuse. In this example, the SymTab class defines two new messages. In the construction approach these are added to the 46 messages that are defined in the Object class. In the subclassing approach these two new messages are added to the 99 messages that Dictionary objects respond to. The protocol for SymTab objects is twice as large (101 messages vs. 48) when subclassing is used instead of construction.

Looking at this issue of construction versus subclassing from the perspective of a strongly typed language, subclassing should be used when the new class is a subtype of the old class. This is true if objects of the new class could be used wherever objects of the old class are valid. This criteria also makes sense in weakly typed languages like Objective-C. The issue is whether objects of the new class should respond to the complete protocol of the old class.

3. PROBLEMS IN OBJECT-ORIENTED REUSE

Most object-oriented programs use a combination of the construction and subclassing styles of software reuse. Each approach is applicable in certain cases. In our experience, construction is much easier than subclassing. When you are working on very simple problems (such as the example in the previous section), both approaches seem to work well. As you progress to more complex tasks which are reusing more complex classes, construction remains relatively easy while the subclassing approach can become extremely difficult. The difficulty of subclassing seems directly related to the complexity of the superclass being reused.

There are several reasons for this, but all are related to encapsulation. In our work to this point we have been able to use the construction approach without breaking the encapsulation of the classes we are using. We have not had to look at the code for the classes when we use this approach. The documentation on the message protocols seems sufficient in most cases. The major reason for looking at the implementation of a class would be to evaluate space/time performance tradeoffs if there are several classes available which can solve a particular problem. This type of information is sometimes available in the documentation, which makes looking at the code unnecessary.

This has not been our experience with subclassing. We have been forced into looking at the code for the classes we wish to subclass, and usually the code for their superclasses as well. This radically increases the complexity of the task.

One of the major reasons for these problems, is that complex behavior is often implemented by a set of methods in a class. Specifically, many methods cause objects to send themselves messages to implement parts of the required behavior. Objective-C also supports the Smalltalk feature of letting a method refine its inherited behavior using special *super* messages. The combination of *self* and *super* messages makes it very difficult to follow the flow of control in many classes.

For example, an object in many classes is created with the *new* message. The method for this often sends itself the *initialize* message. The *initialize* method is responsible for initializing the instance variables in the new object. Many classes in the hierarchy define only the *initialize* method and inherit the *new* method. This makes it hard to understand when this *initialize* method will be used. In this case, you must find the *new* method in the superclass (or its super-superclass or its super-super-superclass, etc.) and discover that it sends itself the *initialize* message. Furthermore, in writing an initialize method, you have to remember to send the initialize message to *super*.

A related problem can occur if a class redefines the *new* method and does not send itself the *initialize* message. In this case, the initialize method in the superclasses will never be executed. This will destroy the behavior that the new class expects to inherit. The *new* method may be implemented several levels up the hierarchy.

In general we have run into the situation where there is a fairly complex control tree where the nodes in the tree are messages. For example, let's say we have five classes, **C1** to **C5** and four methods, **A**, **B**, **C** and **D**. The classes form a linear subclass hierarchy (**C2** is a subclass of **C1**, **C3** is a subclass of **C2**, etc.). Each class implements or refines some of the four methods:

Class	Method A	Method B	Method C	Method D
C1	**implements** sends *self B & C*			
C2		**implements** sends *self D*	**implements**	**implements**
C3	**refines** sends *super A*	**refines** sends *super B*		
C4	**refines** sends *super A*		**refines** sends *super C*	
C5				**refines** sends *super D*

A class *implements* a message if it defines a method for the message which does not use inherited behavior. A class *refines* a method if it defines a method for a message and then sends the message to *super* which executes the inherited method. In this case, class **C1** is an abstract class that defines the **A** method which sends itself the **B** and **C** messages. These methods are implemented in the subclasses of **C1**. The **C2** working implements methods **B**, **C** and **D**. Method **B** sends itself the **D** message. The other classes simply refine their inherited behavior.

If we look at the messages that an object of class **C5** sends itself in response to an **A** message, the overall structure looks like:

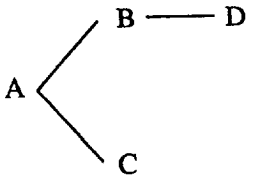

Figure 2: The Message Control Tree for the A Method in Class C5

Method **A** is implemented by methods **B** and **C**. Method **B** is implemented with method **D**. This seems simple enough, but if we look in detail at the control flow of messages that an object of class **C5** sends itself in response to the **A** message, it looks like:

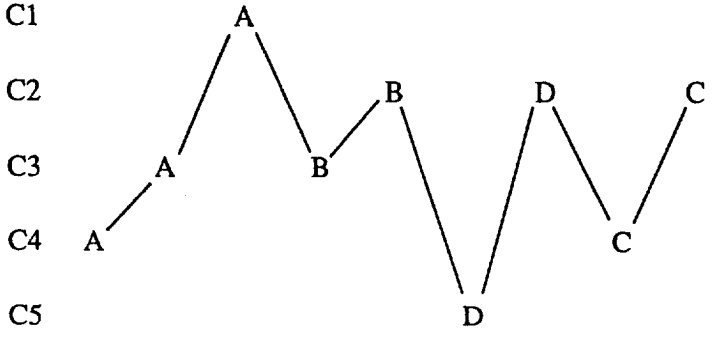

Figure 3: Control Flow Trace for Message **A** in Class C5

When an object of class **C5** receives the **A** message, the method in the **C4** class in executed. This method sends itself the **super** A message which causes the **A** method in the **C3** class to be executed. This method also refines its inherited behavior, so the **A** method in the **C1** class is executed. This method sends itself the **B** and **C** messages. When the **B** message is sent, it is interpreted from the standpoint of *self* (the **C5** class). Since **C5** does not implement **B**, the hierarchy is searched for a superclass which does respond to this message. In this case, the **B** method in the **C3** class is executed. It sends the **B** message to *super* which causes the **B** method in **C2** to be executed. This sends itself the **D** message and we go back down to class **C5**.

In a conventional language the nodes in the control tree (as shown in Figure 2) would be subroutines. There would be exactly one subroutine for each node in the tree. Object-oriented languages support *polymorphism* which means that the implementation for a

message (i.e., the method) is dependent on the class of the object which receives the message; the shape of the message control tree (for a particular message) can vary from one class to another in the class hierarchy. In our example, the message control tree for message **A** in class **C1** does not include method **D**.

The combination of polymorphism and method refinement (methods which use their inherited behavior) makes it very difficult to understand the behavior of the lower level classes and how they work. The problem is that a class several levels down in the hierarchy may need to implement only one or two of the methods. In our example, **C5** only refines method **D**. The important behavior (from the standpoint of construction) may only be *A*. It is very difficult to visualize the entire message tree and understand what a particular method is supposed to do.

3.1. The Yoyo Problem

Often we get the feeling of riding a yoyo when we try to understand one these message trees. This is because in Objective-C (and Smalltalk) the object *self* remains the same during the execution of a message. Every time a method sends itself a message, the interpretation of that message is evaluated from the standpoint of the original class (the class of the object). This is like a yoyo going down to the bottom of its string. If the original class does not implement a method for the message, the hierarchy is searched (going up the superclass chain) looking for a class which does implement the message. This is like the yoyo going back up. *Super* messages also cause evaluation to go up the class hierarchy.

The example presented in the previous section may seem contrived, but it is actually much simpler than many methods we have examined in the past few months. The Objective-C library has many levels, and some classes have seven or eight superclasses instead of the two used in this example. These leaf classes often define only a few methods but inherit more than three hundred methods from their superclasses! This leads to tremendous complexity in analyzing their behavior, how they get their behavior and what should be done in a subclass to modify their behavior. Figure 4 shows a diagram of the execution of the **origin:extent:** message in the **StdSysLayer** class in the Objective C hierarchy.

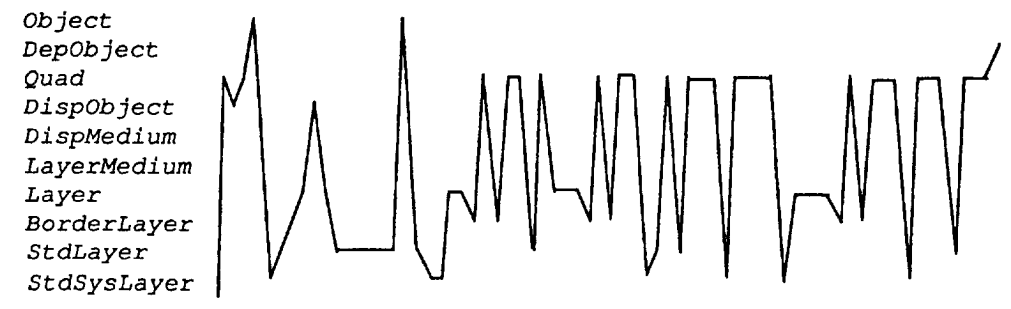

Figure 4: Control Flow Trace for SysSysLayer origin:extent:

Another potential problem with inheritance and encapsulation in Objective-C is that each class inherits the instance variables of its superclasses and the right to access them. This can lead to readability problems in the code since the definition for the variable may be several classes up in the class hierarchy. In practice, this has not been as severe a problem as methods for two reasons. First, there are far fewer instance variables than instance methods. A leaf class in the library may have three hundred and fifty instance methods, but only ten to twenty instance variables.

The second reason is that instance variable definition is additive in Objective-C. Each class inherits instance variables and can add more to the list, but classes cannot change the type of inherited instance variables. James Alexander has suggested applying a similar approach to methods where possible [Ale88].

3.2. Discussion

Several papers in the literature have discussed the potential conflict between encapsulation and inheritance [Sny86], [Mic88]. The papers have been concerned about the fact that if a subclass has access to the internal structure of a superclass (its instance variables in Objective-C), it may be difficult, if not impossible, to change the superclass implementation without breaking the subclasses.

We have discovered that a more important inter-class implementation dependency is the message control tree. A writer of a new subclass must understand how its superclasses are implemented in order to understand what messages an object sends itself and which of these methods should be refined. This not only makes it hard to create new subclasses, it also make the subclasses dependent on the implementation of their superclasses. This leads to greater difficulty in modifying the implementation of classes in the middle of the class hierarchy.

Some languages use "delegation" instead of inheritance [Lie86]. In these systems, an object is also defined by its differences from an existing object. When an object receives a message that it does not understand, it *delegates* it by sending the message to

another object or set of objects. The approach may help solve the *yoyo* problem although we have no personal experience to back up that speculation. We also don't know if the delegation approach introduces other problems which are just as serious.

There seem to be several easy solutions to parts of the reuse problems we have encountered. The instance variable problems seem the easiest to resolve. We use a special naming convention to make it easier to distinguish instance variables from other variables (local variables in methods, class variables, globals, etc.). We have also adopted a coding style of not directly using inherited instance variables, but instead using messaging to access them. This is done by having an object send itself a message to *get* or *set* the value of an inherited instance variable.

The method (*yoyo*) problem seems harder to solve. One solution is to try to reduce the complexity of the message control trees. Another is to clearly document them, indicating exactly what each method should be doing in the context of the overall object behavior.

Snyder [Sny86] points out that each class has two interface protocols: the one use by other classes (via construction reuse) and the additional methods which are intended for use by subclasses. The *C++* language [Str86] makes this distinction explicit via its *public*, *private*, or *protected* classifications of methods and instance variables. This should make it easier to understand how to make subclasses.

4. TOOLS FOR SOFTWARE REUSE

The construction approach to software reuse relies on the external protocol of the classes being reused. Tools which let the user examine these interfaces, such as the Smalltalk browser [Gol84], are very effective in aiding this type of reuse. We have built a similar browser for Objective-C which supports a simple pattern match query mechanism for classes, methods and instance variables. This tool has been very helpful in our reuse of both library classes and classes that we have developed.

Solving the *yoyo* problem requires tools that help the programmer understand the control structure of class or set of classes. This can be done using either static or dynamic analysis. Static approaches look at the code for the methods in the classes to determine the control flow while dynamic analysis looks at the messages that are sent by a running system.

There have been several papers on doing dynamic analysis of control flow in object-oriented systems. Cunningham and Beck [CuB86] describe a technique for making diagrams of messages used in an object-oriented computation. Kleyn and Gingrich [KlG88] have developed a systems for visualizing the dynamic behavior of object-oriented programs. The major problem with dynamic analysis is that the number of messages sent can be very large even for apparently simple object behaviors.

The system described by Cunningham and Beck makes message diagrams using a modified version of the Smalltalk debugger. Human intervention is required to determine which parts of the message trace are important and should be included in the diagrams and which messages should be excluded. The tool is really intended as a

documentation aid which lets an expert explain the messages used by objects in a system.

The GraphTrace system developed by Kleyn and Gingrich displays the dynamic behavior of object-oriented systems and shows how the messages relate to a structural view of the system. Animation is used to help visualize the messages and the objects receiving them. These tools are used for understanding complex object systems rather than just explaining them. The main problem with the dynamic approaches is that there is an enormous amount of data which must be visualized and it only shows control paths which were used during the execution being viewed. Other possible control paths are not available.

Static analysis is very limited in typeless systems like Objective-C and Smalltalk. When a message is sent to *self* or *super*, it is possible to follow the control flow based on knowledge of the inheritance hierarchy. When messages are sent to other objects, the type of the receiving object can not, in general, be automatically determined by static analysis and it therefore not possible to show the complete control tree. Strongly typed languages, such as C++, Trellis/Owl and Eiffel could support more in depth analysis.

In many cases the ability to follow the message control flow inside a single class is very useful. We have developed a system for Objective-C which does this and can produce diagrams similar to the figures used in the last example. This has proved very useful in understanding classes which use method decomposition (*self* messages) and method refinement (*super* messages).

An extension to this system which we are considering is to allow the user to make queries about the control graph. This would answer questions like, "if I send the X message to an object of class C, which methods in that class (or its superclasses) may be invoked?". This is basically going down the control flow tree. Another type of query would be "which messages could I send to an object of class C which would cause the invocation of method X (in class C or one of its superclasses)?". These are basically queries on the transitive closure on the complete message control graph for a class. These queries would allow the user to understand the connections between the external protocol for a class and its internal methods and should make it much easier to understand the implications of refining a method in a subclass.

5. CONCLUSIONS

We have only been using Objective-C for a about a year. At this point it appears that the construction approach to building new classes works very well. We are able to use existing classes without breaking the "encapsulation barrier" of the classes. Specifically, we find that we can use the library classes without having to understand their implementation details.

The Objective-C library classes are very powerful and support complex behavior in a typeless way. This weakly typed approach seems very useful when constructing new classes. For example, dynamic arrays are supported by a single class. This class *(OrdCltn)* can be used for arrays of any type of object. We were able to switch between using simple arrays and sorted arrays *(SortCltn)* by simply implementing a comparison

method in the objects being sorted in the array.

The reuse of classes via construction seems simple and much more powerful than the use of library subroutines in conventional languages like C. This means that object-oriented code is much easier to reuse than conventional code and this should lead to greater software productivity in object-oriented environments.

Reuse through subclassing, on the other hand, seems much more difficult. The *yoyo* problem, in particular, seems like it may be a general problem with inheritance based languages that support the concept of an object sending itself messages. In particular, we suspect that Smalltalk users encounter similar problems. The *yoyo* problem may get even worse when multiple inheritance is added to these languages, depending on the algorithms used.

Both methods of object-oriented reuse are useful in practice. Subclassing makes more sense when the new class wants to inherit the complete protocol of the existing class. Construction makes more sense when the new class is not a specialization of the old class and therefore should not accept its complete protocol.

The tools we have built to browse Objective-C class hierarchies and to display message control trees have proved very useful. We feel that extensions which support graph queries make it much easier to understand the design of classes which we are trying to subclass. Object-oriented software reuse in much more powerful than reuse in traditional languages. Classes are often extended and reused in ways that were not anticipated by the designer of the classes. Powerful tools are required to make this type of reuse more feasible.

References

[Ale88] Alexander, J. H., "About Behaviorism: Rational Uses of Inheritance", Technical Report ST 04-01, U S WEST Advanced Technologies, March 1988.

[Cox86] Cox, B. J., *Object-Oriented Programming: An Evolutionary Approach*, Addison Wesley, 1986.

[CuB86] Cunningham, W. and K. Beck, "A Diagram for Object-Oriented Programs", *Proc. Conf. Object Oriented Programming Systems, Languages and Applications*, Portland, OR, Sep. 1986, 361-367.

[Gol84] Goldberg, A., *Smalltalk-80: The Interactive Programming Environment*, Addison Wesley, Reading, MA, 1984.

[KlG88] Kleyn, M. F. and P. C. Gingrich, "GraphTrace - Understanding Object-Oriented Systems Using Concurrently Animated Views", *Proc. Conf. Object Oriented Programming Systems, Languages and Applications*, San Diego, CA, Sep. 1988, 191-205

[Lie86] Lieberman, H., "Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems", *Proc. Conf. Object Oriented Programming Systems, Languages and Applications*, Portland, OR, Sep. 1986, 214-223.

[Mic88] Micallef, J., "Encapsulation, Reusability and Extensibility in Object-Oriented Programming Languages", *Journal of Object-Oriented Programming*, April/May 1988, 12-36.

[Sny86] Snyder, A., "Encapsulation and Inheritance in Object-Oriented Programming Languages", *Proc. Conf. Object Oriented Programming Systems, Languages and Applications*, Portland, OR, Sep. 1986, 38-45.

[Str86] Stroustrup, B., "An Overview of C++", *SIGPLAN Notices*, Oct. 1986.