A Model for Implementing an Object-Oriented Design without Language Extensions

Jennifer Hamilton
IBM C++ Compiler Development
1150 Eglinton Ave. E
Toronto, Ontario M3C 1H7
jenniferh@vnet.ibm.com

Abstract: This paper proposes a means of implementing an object-oriented design in programming languages that do not directly support the object-oriented paradigm, without requiring language extensions. The model supports information hiding, dynamic binding, polymorphism and single inheritance through a typeless, dynamic approach similar to that of Smalltalk. Effecient dynamic method binding is achieved through direct lookup method tables constructed using an incremental graph-colouring algorithm. The methodology can be applied to any language that supports data and procedure pointers and dynamic memory allocation.

Keywords: Object-oriented programming, message sending, method search, coloring,, dynamic binding, inheritance, polymorphism, structured programming

Introduction

Object-oriented programming concepts such as information hiding, polymorphism, and inheritance can greatly improve the reusability and extensibility of software. These strengths have contributed to the increasing popularity of programming languages such as Smalltalk and C++ that support the object-oriented programming paradigm. However, for a variety of reasons, many programmers are unable to take advantage of these new programming languages, and must continue to use languages that do not support object-oriented programming. Some use object-oriented design techniques and attempt to retrofit such designs into their current programming language.

This paper proposes a means of implementing an object-oriented design in programming languages that do not directly support the object-oriented paradigm, without requiring language extensions. The model supports information hiding, dynamic binding, polymorphism and single inheritance through a typeless, dynamic approach similar to that of Smalltalk. The model allows methods and classes to be added or removed without requiring recompilation of code that does not depend upon the interface change. Message routing is supported through a general library routine that dynamically determines the target method at runtime. Each class has a method table associated with it that is used in conjunction with a selector index to determine the target method through a single array index into this table. The method table for each class is constructed dynamically at runtime, using a graph-coloring algorithm to minimize space consumption. The majority of the work involved in creating and manipulating objects is handled by routines to handle message routing and construct the method tables, so that the interface for invoking a method on an object is very simple, in accordance with the design goals of abstraction. The methodology can be used with any language that supports data and procedure pointers and dynamic memory allocation.

The model is described from two perspectives: object consumer and object producer. The object consumer view uses objects, without regard for implementation details, while the object producer view defines and implements a class and its methods.

Object Consumer

From the consumer perspective, there are four main artifacts: 1) objects 2) selectors 3) the messager and 4) classes. An object is represented by a pointer. A message is sent to an object using the following convention:

```
call messager(object, selector_address, parameters)
```

messager is the dispatcher that handles dynamic binding, mapping the given selector to the appropriate method for the target object. The selectors are global variables, where the variable name identifies the message to be sent to

an object. Note that the address of the selector, not its value, is passed. This requirement is discussed in more detail under Method Routing. The selectors are polymorphic: a given selector can be supplied to the messager for any type of object accepting that message. For example, a single variable such as **m_display** would be defined and supplied to the messager as a selector for any type of object that accepts the message display. The means of defining the selector values is discussed under "The Messager Model".

With a model such as Smalltalk's, a class is an object to which messages such as **new** can be sent to create instances. In this model, classes are not treated identically to objects, but they are conceptually the same. The main difference is in the interface to a class. Rather than calling the messager to send a message to a class, each class has single entry point that is called directly as follows:

call class_ep(selector_string, parameters)

The class will determine the appropriate action to take based on the selector_string, which is a string representing the message rather than a variable as for the messager. For example, to create a new object of a given class, the entry point for that class is called with the selector string "new". This would allocate and return an object of the given class.

Once an object is created, it can be sent messages through the messager as discussed above. Because the model does not support a garbage collection mechanism, the **delete** message must be sent to its class to delet an object.

Object Producer

Objects and classes are created dynamically as requested in the program. The first time an object is created for a given class, that class will be be initialized, which mainly involves constructing the method table for the class. The method table is an array containing function pointers, one for each method supported by the class, including methods inherited from superclasses. This allows method binding to be performed with a single direct array access, using the selector as the index. The method table is constructed using an incremental graph coloring algorithm in order to reduce the space requirements of method tables across the program.

Object Creation

An object is created by sending the message **new** to the target class entry point, which will allocate, initialize, and return a pointer to the created object. Each object is represented by a structure consisting of an integer that provides the number of classes in the hierarchy for the object's class, followed by an array of instance pointers, one per class, as shown in Figure 1. The terminal class in the hierarchy appears first, followed by each subsequent superclass, such that the topmost superclass in the hierarchy is the last element in the array.

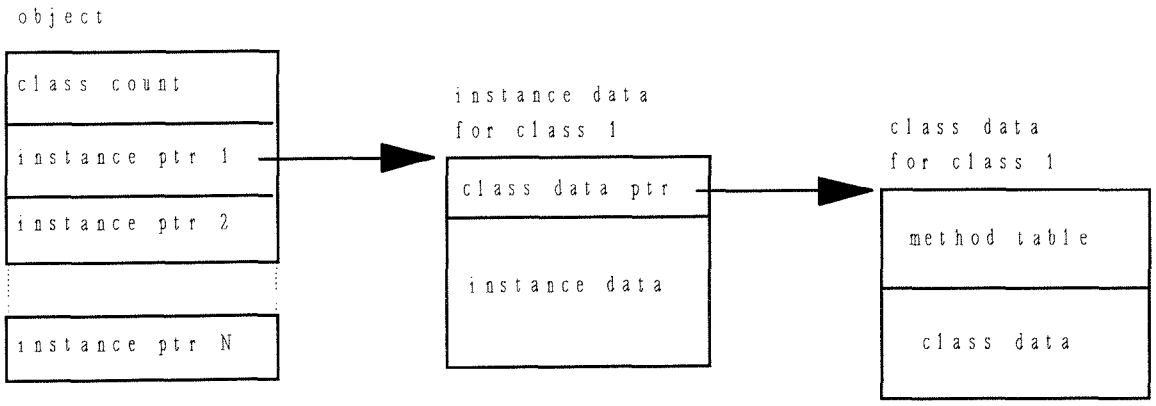

Figure 1 Object Representation

The instance pointer array is used to access the appropriate instance information for the target method. Each pointer provides addressability to the instance data introduced by a particular class within the hierarchy. To create and

initialize this structure appropriately, each class must support a message, **createSuperInstance**, that will set the superclass instance data pointer for the object. The algorithm for **createSuperInstance** is:

```
if no parent class
    allocate an object with entries for the given class_count
    set the class count in the object to class_count
else if parent
    send createSuperInstance to parent class with class_count+1
allocate and initialize instance data for the class
set instance data pointer at index given by class_count
return object to caller
```

When a **new** message is sent, the receiving class will send the **createSuperInstance** message to itself with a class count of 1. Each class will call its parent class, incrementing the class count until the topmost class in the hierarchy is reached. This class will be responsible for allocating the object structure and setting the class count variable within it. Once the object has been allocated (either directly by createSuperInstance for the current class or as returned from a parent **createSuperInstance** that has allocated the object), the instance data for the class is allocated and the appropriate instance data pointer element is set, using the class_count as an index into the instance information array. Upon return to **new**, the resulting object will be returned to the caller.

The instance information for an object consists of a pointer to associated class information, followed by any instance data introduced. The class information consists of a common class header followed by any class data (ie. data that is shared by all instances of that class) for the specific class. The common class header contains information that is maintained by all classes, such as the class method routing table.

Class Initialization

Each class is responsible for initializing its class information. Class initialization is performed in the implementation of the **new** message: the class checks to see if it has been initialized yet, and if not, initialization takes place. To perform class initialization, each class must support a message, **initialize**, that allocates and initializes its own class data and also calls **initialize** for its direct parent class. The algorithm for **new** is:

```
if class has not been initialized
    invoke initialize against self        /* initialize self and parents */
send createSuperInstance to self          /* create new class instance */
return created object
```

Class initialization mainly involves initializing the parent class and assigning the method table entries to the appropriate target method implementations. As part of class initialization, a new class must provide a list of the selectors that it implements, along with the list for each of its parents, to the **selectorAssignment** routine, which assigns selector index values to the selector variables. The details of this assignment are discussed below under "The Messager Model". **selectorAssignment** expects a linked list of elements, where the first element in the list is the selector list for the new class, and each subsequent element is that of the direct parent for the previous class. The entries in the linked list are of the structure shown in Figure 2.

The first member in the structure points to the selector list for the direct parent class. This list is returned from the **initialize** message sent to the parent superclass. Next is an array of selector elements, where each entry is a pair consisting of the selector variable address and a pointer to the target method, one for each selector that is implemented by the class. For any parent class messages to be disabled by this class rather than inherited, a selector entry which contains a null for the target method is supplied. The last element will contain a null pointer indicating the end of the structure. The selector variables are the global selector variables discussed earlier under "Object Consumer". The addresses of the selectors are used to uniquely identify each selector to **selectorAssignment**. Note that the information required to construct each structure is known at compiletime, so the structure can be declared statically within the class code.

The algorithm for class initialization then is:

```
if class has already been initialized
   return address of selector list
send initialize to parent class
assign returned list to superclass link in selector list
call selectorAssignment with selector list
allocate and initialize method routing table using size from selectorAssignment
classIndex = 1
for each selector implemented by this class
   /* assign methods implemented by this class */
   update method table entry with classIndex and target method address
for each entry in the linked list of superclass selectors
   increment classIndex by 1
   for each selector in the superclass list
      /* assign methods inherited from parent class */
      if a method table entry is not yet assigned for the selector
         update method entry with classIndex and target method
for each selector disabled by this class
   clear the corresponding method table entry
return address of selector list
```

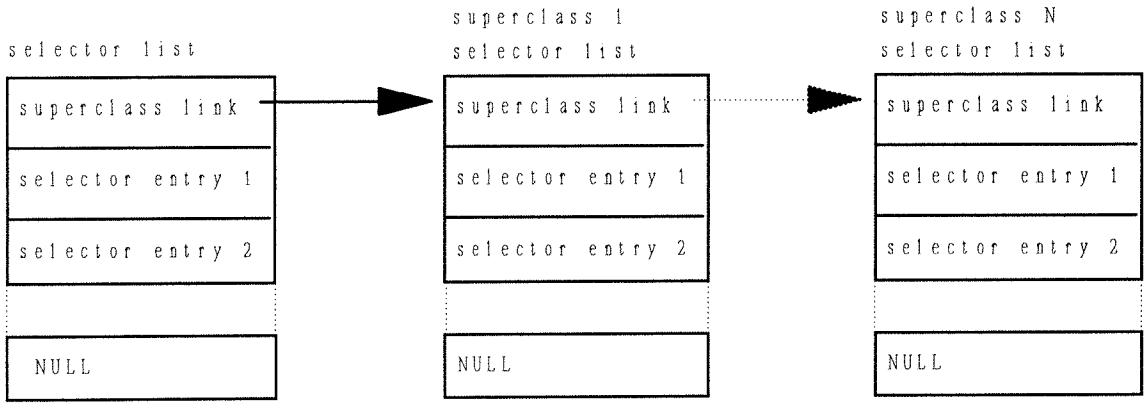

Figure 2 Selector List Structure

If a class has already been initialized (by another subclass), **initialize** just returns the address of its selector list structure, otherwise it sends **initialize** to its direct parent class and attaches the returned selector list to its own selector list structure through the supererclass link. Then, **selectorAssignment** is invoked to assign colors to the selectors. Upon return from **selectorAssignment**, the method table is allocated and initialized with procedure pointers to the target methods. The method table is stored in the class data, shown in Figure 1, and has the following format:

```
sizeOfTable
methodEntry1
methodEntry2
methodEntryn
```

The **sizeOfTable** element is equal to the maximum selector value used by the class, which will be returned by **selectorAssignment**. The remainder of the method table is an array of method elements, where each element is a pair consisting of a class index and a procedure pointer to the appropriate method. The size of the array is equal to **sizeOfTable**. The class index identifies the class corresponding to a given method, by relative position within the class hierarchy. For each selector implemented by the new class, the class index for that method will be 1. For each selector that is implemented by a superclass, the class index will be the relative position in the inheritance hierarchy of that implementing superclass, where the direct parent of the new class is 2 etc. The use of the class index is discussed later under Class Index.

A class can override a message implemented by a superclass by providing an entry in the selector list for the current class and setting the method table entry for that selector to the implementing method. A class inherits a method from a parent class by setting the method table entry for a selector to the method address supplied in the selector list for a parent class. A class can disable a parent class message by clearing the method table entry for that selector.

Each entry in the method table will either be (0,null), indicating that the class does not accept a message that corresponds to that selector, or it will contain a class index and procedure pointer to the implementing method that corresponds to the index value of the given selector. For example, if the class accepted the message **display**, and the global selector variable **m_display** had been assigned the index value 4 by **selectorAssignment**, the 4th element in the array would be the address of the method to invoke when the **display** message is sent to objects of this class. If this message were implemented by the current class, the class index would be 1, and if it were implemented by the superclass of the superclass of the current class, the class index would be 3.

The Messager Model

As discussed earlier, a message is sent to an object using the following convention:

call messager(object, selector_address, parameters)

The messager maps the object type and selector to a target method implementation. The selector identifies the target method to be invoked for the object. For each possible selector used, a global variable must be defined that is used within the object consumer code to specify a message, and by the messager to determine the target method to call. This implies that a variable for each selector must be defined prior to compiling any object consumer or producer code that uses or accepts that selector.

The messager model described here is based on that of Objective-C, described in [COX91]. In the Objective-C model, each possible selector represents a unique integer value. Cox describes several alternatives for handling the mapping from the selector and object pair to the target method. The slowest approach follows that of Smalltalk, where a list of selectors maintained for each class is searched for the given selector. If a match is not found, the search continues in the selector list for the parent class, until either a match is found or the top of the class hierarchy is reached. This approach requires little storage to implement, but can be prohibitively slow, especially with deep class hierarchies.

A much faster approach, but very expensive from a storage perspective, is to maintain a method table for each class that is indexed using the selector number. There would be a table entry for each unique selector value, so the amount of storage required is proportional to the number of classes times the number of possible selectors times the size of each entry. If there were 200 possible selectors and 30 different classes, 600 table entries would be required. The approach used by Cox is to create a cache that stores selector/class to implementation mappings, which is indexed by hashing the selector and class codes. If the resulting cache entry is valid, it is used; otherwise, the slow search described above must be used.

The model described here uses a variation on this approach, proposed in [DIX89] and further refined in [AND92]. It involves using graph-coloring techniques to assign values to each selector such that all selectors for a given class have unique values, or colors, but that the same value can be used for other selectors. The goal is to enable the fast lookup of using the selector to index a method table in each class, while reducing the storage overhead of such a method by minimizing the number of unique selector values.

The approach is to represent each selector as a node in the graph, where an edge between two selectors indicates that there is an object that responds to both selectors. The problem is then equivalent to that of coloring the graph such that no adjacent nodes are assigned the same color. For static languages such as C++, the entire graph can be constructed and colored at compile-time. In this model, however, the class hierarchy is created incrementally at runtime, so the incremental coloring algorithm described in [AND92] is used instead.

When a class is activated, it will supply a list of the addresses of the selector variables it responds to, including those inherited from parent classes, to the coloring routine, **selectorAssignment**. The coloring routine will maintain

an NxM table, where N is the maximum color number and M is the number of classes in the system. Each time a new class is added, a new column is added to the table. The elements in the table represent the selectors used by each class, corresponding to the selector color value. The algorithm used by the coloring routine is as follows:

```
    for each selector in the set of selectors for the superclasses of the given class
        record its color in the row for the given class
    for each selector in the set of selectors for the given class
        if the selector is not colored
(1)         color it with the smallest unused color and record its color
        else if the selector is colored by a superclass of the given class
(2)         do nothing
        else
(3)         reassign selector to smallest unused color in all terminal classes introducing that selector
```

The reason that the addresses of the selectors are supplied is that the coloring algorithm must be able to 1) uniquely identify each selector and 2) easily update the selector value. Using the variable address guarantees a unique compiler-assigned value for each selector that also provides direct accessability to the storage for that variable.

As an example of color assignments, if class A supplied the selector list (a,b), followed by subclass C of A, with list (c,d), followed by subclass E of A with list (e,f), the result would be the colouring table shown on the left side of Figure 3. The color for a is 0, b is 1 etc. If a subsequent subclass G of E were added with list(c,g), G would inherit methods (a,b,e,f) from its parent. This would require reassignment of the selector for c, (case 3 in the graph coloring algorithm), because it is already assigned the value 2, which collides with that of e. So c would be reassigned the lowest unused color in all the terminal classes that introduce c, which, in this case would be 4, and g would be assigned the color 5. The color assignment would be as shown in the right side of Figure 3.

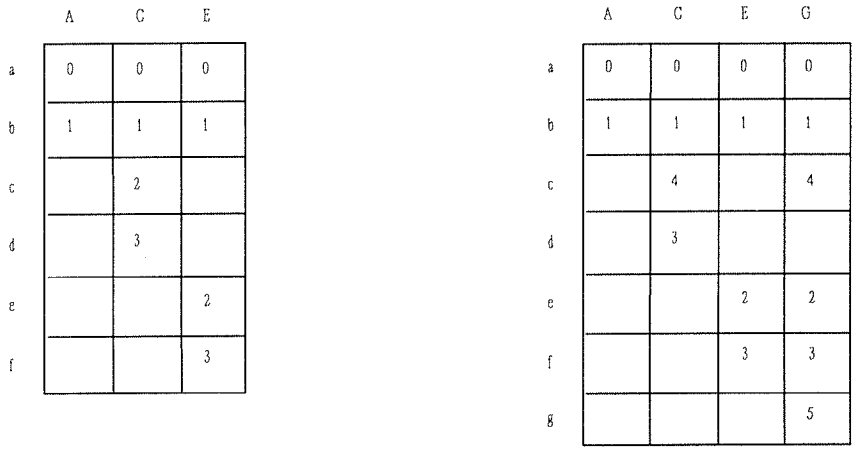

Figure 3 Graph Coloring Tables

Note that reassigning the color for a selector also requires that for each class using that color, the method pointer in its method table must be moved to the new element. The method table for a class may also need to be resized if the new color is greater than the current method table size. This would be the case for class C in the example above: introducing class G caused the value for selector c to be changed from 2 to 4.

Method Binding

When the messager is invoked, it uses the selector specified as an index to lookup the method pointer in the method table for the target object type. The method table is easily retrieved from the object by accessing the class data via the first instance pointer element in the object (see Figure 1). If the selector is valid (the index is within the bounds of the method table size and the designated element is set), the target method is invoked via the procedure pointer with the parameters that were supplied to the messager, along with the target object and the selector address.

If the selector is invalid, an error is returned to the caller.

As noted in [DIX89], "the use of a particular color by more than one selector means that is very likely that an invalid selector will have an alias. The object will accept the selector and do the aliased method. Such a system would be unusable. To detect such error every method call would pass its unique selector code as an argument. Each method would check this argument against a copy of the selector to which it responds". In this model, the selector address is used to indicate the intended selector, the reason the address rather than the value of the selector is supplied to the messager. A method must check the passed selector address against the address of the selector that it implements to verify that the correct method has been invoked. If the selector is incorrect, an error is returned to the caller.

Class Index

Because this is completely a runtime model, the method implementation has no static information available to determine which element within a given object's instance information array is its corresponding instance data. This is a particular problem if a method implementation is inherited from a superclass. This is handled by passing an additional parameter to the method, a class index, which indicates the appropriate instance information element to access. When a method is invoked against an object, the messager will look up the selector in the method table and invoke the indicated method implementation and supply the associated class index. The method code can then use the class index to obtain addressability to the appropriate class and instance data via the instance pointer. A method is thus called from the messager as follows:

```
call method_pp(object, selector_address, class_index, parameters)
```

The class index also solves the problem of how to handle calls to superclass methods, in particular, calls to a superclass method from within a superclass method. This problem has two aspects: the first is to determine the appropriate method to call, and the second is to notify the target method of where its instance data is within the object. Two special messagers are introduced for handling this scenario: **message_super** and **message_self**. The messagers provide a means for invoking a method in the direct superclass or the same class, respectively, of the current class. Both messagers have the same interface:

```
call messager_name(object, selector_address, class_index, parameters)
```

Both messagers use the class index to determine the appropriate target method to call, and pass this class index to the target method. **message_super** increments the class index by one, uses the index to access the method table for the superclass via the instance information pointer, and invokes the appropriate method in the method table for that class, passing the incremented index through to the method. **message_self** uses and passes the class index as is.

One could simply define a single additional messager, such as message_to, and have the method code pass either the current index (for self) or the current index incremented by one (for super). However, this would introduce a dependency within all methods on the details of the object structure. Having two separate messagers means that a method is simply supplied with an index that it uses to access its instance data and pass to the other messagers, but no additional information regarding that index is necessary.

Parameter Passing

When a method has been invoked, there must be a mechanism to determine the number of parameters passed. How this is achieved depends upon the language support available. If the language does not support a means of determining the number of parameters passed, it may be necessary to pass an additional parameter on the call to the messager. Two possibilities are a terminal value at the end of the parameter list or a parameter count preceding the parameter list. The terminal value approach requires the ability to specify a unique value that can be checked against each parameter to determine the end of the list, such as a null pointer passed by value in a call-by-reference model. The parameter count method requires the caller to accurately supply the exact parameter count, which is more error-prone than the terminal value.

Conclusion

The model presented here allows an object-oriented design to be implemented in a language that does not directly support object-oriented programming. The model itself follows the goals of object-oriented design. Each class is responsible only for handling itself and knowing about its direct superclass, while a superclass required no information about any of its subclasses. New classes can be added without requiring code changes within existing classes. The object consumer model is quite simple and reasonably efficient, as is method implementation. The main overhead and details of the model are limited to object creation and class initialization. The model can be implemented relatively easily in most structured programming languages.

References

[AND92] Andre, P. and Royer, J. "Optimizing Method Search with Lookup Caches and Incremental Coloring," *OOPSLA 1992 Proceedings*. New York, NY: ACM, 1992.

[COX91] Cox, B. and Novobiliski, A. *Object-Oriented Programming: An Evolutionary Approach* 2nd ed. Reading, Mass: Addison-Wesley, 1991.

[DIX89] Dixon, R., McKee T.; Schweizer, P.; and Vaughan, M. "A Fast Method Dispatcher for Compiled Languages with Multiple Inheritance," *OOPSLA 1989 Proceedings*. New York, NY: ACM, 1989.

[GOL83] Goldberg, A. and Robson, D. *Smalltalk-80: The Language and its Implementation*. Reading, Mass: Addison-Wesley, 1983.