Dynamic Encapsulation of C++ Objects for Distributed Object-Oriented Systems

A Thesis Proposal

Frank Barrus
shaggy@csh.rit.edu

Rochester Institute of Technology
March 15, 1996

ABSTRACT

A fully-dynamic object should support the lookup of methods and attributes without requiring prior knowledge of the exact data type or implementation of the object, and should allow dynamically typed objects to be passed as parameters to methods, even between address spaces or networked machines, without requiring knowledge of the structure of the object.

Existing fully-dynamic object-oriented languages do not have the speed of static systems, and static object-oriented languages, such as C++, lose a lot of flexibility, and the potential for higher-level abstractions, by not supporting fully-dynamic objects. A method for creating dynamic objects to encapsulate existing C++ objects is presented, which allows for both the speed of static C++ objects when class information is known, and the flexibility of a dynamic system.

A utility, DECO (Dynamic Encapsulator of C++ Objects) is described for automatically generating the interfaces to the new objects with only trivial extensions needed to existing static C++ code.

Approved by:

____
Professor Andrew T. Kitchen

____
Professor Warren R. Carithers

____
Professor Peter G. Anderson

1. Table of Contents

2. Introduction and Background

C++ provides a framework in which programs can be developed using a subset of an object-oriented model, but some important aspects missing from C++ include fully-dynamic data types and dynamic method selection.

Communication between objects in different address spaces, whether on the same machine or on networked machines, is possible using existing RPC methods, but the programs on each side are both required to have a full understanding of the objects being passed across the network, by sharing common header files.

Truly dynamic objects, whose methods and attributes can be looked up at runtime, and whose data types can be dynamically checked, would allow different machines to communicate without the need for previous knowledge of implementation information. This also allows different versions of objects to be as compatible as possible, given the features available. As long as the expected functionality, in terms of methods and attributes, is available, objects remain compatible, regardless of changes in the underlying representation, even when they're moved between address spaces.

2.1. Problem Statement

Since some form of specification is needed for the new dynamic interface, and existing IDL's did not meet the needs of a truly dynamic system, a new form of IDL had to be written. I call this DECO (Dynamic Encapsulator of C++ Objects), and it can be used both as an IDL and a pre-processor to the C++ compiler, so that dynamic C++ can be seen as an extension to the C++ language.

By providing all of the dynamic extensions through this new language extension, rather than just through specialized header files, libraries, and interfaces that the user is required to write, greater future compatibility is gained as well. Changes can be made to the implementation of the object system, in terms of how an object is stored, how methods are dynamically accessed, etc, without affecting programs that use the system, since the language extensions allow specification of the dynamic features in a higher level form, and DECO takes care of generating the implementation-specific details.

2.2. Previous Work

2.2.1. Shag/OS v0.34

This was the first version of an object-oriented operating system that I wrote, in C, and was the core of many of my initial ideas. It used a lot of ugly macros and pre-processing to accomplish what C++ can now do cleanly.

2.2.2. Shag/OS v1.0

This is my independent study project for writing a distributed operating system in C++, based on the previous design, but making it more flexible, and following the C++ object model, but with dynamic extensions. While developing this, I attempted to do everything within the C++ language, without a separate IDL or pre-processor, but eventually the need for one became clear, as I realized that most of what I was struggling with, syntactically, could be done cleanly with a little help from some language extensions. Since dynamic object-oriented communication is the glue that holds the entire operating system together, the development of the operating system will serve as a testbed for my proposed object framework, and for the DECO pre-processor, since all system objects will be written using it.

2.2.3. CParse

I wrote a C/C++ lexical analyzer, with CPP support built-in, as a conveniently useable C++ class. It is based around a state machine and is designed to be very fast as well as flexible. It handles recognition and conversion to internal form of all the standard constant data types, tokenizing of all operators and keywords, processing CPP directives, and it can even regenerate C++ code from tokens, including indentation levels, and line breaks. This is used by DECO to handle parsing the dynamic C++ code, integrating dynamic extensions into static C++ code, and simply regenerating static C++ code in the output.

2.2.4. DECO

Development is partially complete on the DECO program, which converts dynamic C++ to static C++. This, of course, is where the majority of the work will be done for this thesis (in addition to documenting it), as well as in trying some variations on the implementation of the object model (which will require modifications to DECO to test them).

2.3. Theoretical and Conceptual Development

Many changes have been made in this area, and some will probably continue to be made in the future, as the implementation shows variations that are either necessary, or more sensible or efficient. What follows is a brief summary of the currently proposed object-oriented framework.

First of all, everything is an object, including types and classes. Types and Classes are just specialized forms of objects, but by keeping them consistent, the means of accessing remote classes, object, and types are all consistent as well.

Types and classes are not the same thing. Two objects could both be of type ’String’ and thus act like a string, and support string operations, but yet not be of the same classes, since they might have different forms of storage and implementation. Also, two object could both be of class ’int64’ and both be stored as 64-bit numbers, and yet one could be of type ’Time’ and another of type ’Size’ and their meanings are incompatible even though their storage type and implementation are the same. Thus, the exact datatype of an object is a combination of its type and its implementation. Conversions, without changing the format of the data, are possible from one type to another, but not from one implementation to another without creating new data. For instance, one can convert ’Time_nsec’ to ’Time_sec’ by simply dividing the data by 1000000000 (and possibly rounding), since both types use ’int64’ as their class, but to convert ’Time_nsec’ to ’String’ involves two types with different underlying implementation classes, and therefore the data representation must be changed to show the time number in character string format.

To avoid some confusion, it might be useful to think of 'type' as 'abstract type', since it just has the meaning of the object and what it represents, without any care for the implementation. A 'class' represents the implementation of the object, and is more or less similar to a standard C++ class. And finally, a 'datatype' is the embodiment of both an (abstract) 'type' and a 'class'.

A dynamic object, 'obj', has pointers to a datatype object, the actual data representation, and a reference count which is used for garbage collection. If the data representation is constant and small enough, it can be contained within the class itself. The size of two pointers is reserved for inline data, since neither the data pointer nor the reference count are needed if the data is self-contained within the object descriptor. On 32-bit systems, this means that up to 8 bytes of inline data are available. This also means that constant strings up to 8 characters in length can be stored directly in the object descriptor, eliminating the need to pass their character representations separately from their pointers.

A datatype object, 'obj_datatype' has pointers to both a class object and a type object.

A class object, 'obj_class' has a pointer to the which should change which every change to the implementation that alters the format of the stored data. This name must stay unique in order to locate matching class implementations for a given data representation.

A type object, 'obj_type' which has the name of the type, which must be a unique name representing only types that are guaranteed to have the same meaning.

All objects (types, classes, datatypes), can have superobjects (which would represent superclasses or supertypes when used accordingly) as well as forwarders. Forwarders are used to intercept the normal behavior of an object, whereas a superobject is fallen back upon only when the object itself doesn't implement a given behavior.

Communication between addresss spaces or across networks is handled by creating proxy objects that actually perform the communication, and contain local state information on how to reach the other end of the connection. Proxy objects convert data that must be passed across networks by calling the 'flat' method of the object to be passed, which converts the object to a standard format.

2.4. Glossary

Note that these definitions are specific to the DECO object framework, and may differ somewhat from the standard meanings that these words have, so a careful perusal of them would be recommended.

attribute: any of a number of uniquely named values that an object has, on which get and set operations can be performed. Attributes are similar to C++ instance variables within an object, however there is not necessarily a direct correlation, since the code to get or set an attribute can be overriden to do things other than simply get or set a variable.

class: the implementation classification of an an object. Specifically, with DECO, the class is a special object which contains a pointer to the code necessary to implement the features of an object. The class of an object represents its implementation. (contrast to *type*)

class descriptor: a special descriptor used for classes. With DECO, it has C++ type 'obj_class'

datatype: the exact type used for the storage and implementation of an object. It consists of both the *class* of an object, which specifies how the object is stored and implemented, and the *type* of the object, which specifies the meaning of the information held within.

datatype descriptor: a special descriptor used for datatypes. With DECO, it has C++ type 'obj_datatype'.

DECO (Dynamic Encapsulator of C++ Objects): the utility for converting dynamic C++ code to standard C++ code. It can be used either as an IDL, by converting special dynamic header files (which usually end in .dh) to standard header files (.h), or as a special pre-processor to C++, thus extending the language.

descriptor: a special C++ class which is used to represent dynamic objects. It contains a pointer to a datatype, which describes the implementation and interface of the object, a pointer to the data, which can be contained within the descriptor itself if it is small enough and constant, and a pointer to usage information for the data, such as a reference count, which can be used for garbage collection of the data.

feature: in reference to an object, a feature represents some visible part of the object, either a method, or an attribute.

forwarder: an object which can be attached to another one, and whose methods and attributes override the object to which it is attached.

IDL (Interface Description Language): a pseudo-language that is used to describe complex interfaces in simpler, more general, terms. The IDL program converts IDL code into native code for another language, automatically building all the complexities of the interface, and hiding those details from the programmer. As an example, an IDL for RPC's would generate functions automatically that handle all the details of passing data across the network in a standard format.

method: an operation that an object can perform. A method is analogous to a member function in C++. However, with DECO, a method is a dynamic entity, which can be called by knowing only its name and no details about the underlying object or where the code resides. In contrast, a member function requires that the calling code have full knowledge (either a static address or a virtual table pointer) of the location of the function. Multiple methods can exist with the same name, but taking different types of arguments. Methods with the same name in different classes are considered compatible if the functions they perform have the same meaning.

object: an entity composed of data and a means of operating upon that data. An object generally has features, such as methods and attributes, for operating upon its data. In standard C++, an object is the instance of a class, and consists of a block of memory that has been reserved, and functions for operating upon the data in that memory. With DECO, dynamic objects are C++ objects encapsulated by object descriptors.

object descriptor: a descriptor used for generic objects. With DECO, it has C++ type 'obj'.

OCF (Object Call Frame): a data structure used to send a method information about the call that was made, including the method and parameters passed, as well as any other implementation-specific information for optimization. It can also be used to keep links for unravelling the call chain when errors or exceptions occur.

pre-processor: a program which takes source files and performs initial conversions on them, such as macro substitution or the generation of additional code, and produces output which is suitable for use as input to a compiler or assembler.

proxy object: an object that acts as a local representation of an object on a remote site (or in another address space on the same machine). The proxy object acts in every way like the remote object, so that programs don't need to know where an object actually resides in order to operate on it.

RPC (Remote Procedure Call): a standard method for calling functions on other machines across the network, and converting all of the parameters into compatible data types that can be transferred to another address space. The interface for defining these is usally built with an IDL.

selector: an identifier used to locate a feature (method or attribute). In source code, selectors are arbitrary-length strings, and at the moment they are used as such when finding a feature, however in the future optimizations may be done to use special unique identifiers or other methods to speed the search process.

superclass: a class that represents a more generalized form of another class. When deriving a class from a superclass, the new class inherits all of the features of the superclass, unless they are overriden. With DECO, a pointer is kept to the superclass, so that nothing is actually duplicated. In essence, superclasses are represented as the superobjects of a class, since classes are objects as well.

superobject: an object that is attached to another one and whose features are used if the initial object does not have the desired feature. Classes with superclasses automatically generate objects with superobjects to follow the model.

supertype: a type that represents a more generalized form of another type. These are represented as the superobjects of the type, since types are also objects.

type: a general classification of the meaning of an object. A type does not specify how an object is stored or implemented, only what the data within it represents. If two objects have the same type, then they have the same subset of features, represented by the type, and they behave in the same fashion. A type is analogous to the interface available to an object. (contrast to *class*)

type descriptor: a special descriptor used for type information. In DECO, it has C++ type 'obj_type'

3. Project Description

3.1. Functional Specification

3.1.1. Functions Performed

The DECO program will take C++ source files that have been enhanced with dynamic C++ extensions (DECO extensions) and convert them to standard C++ code. Since the syntax is the same as standard C++ except for the extensions, and since all non-extended code is passed through unchanged, semantically, it can be used either as a separate pre-processor front-end to a C++ compiler, or as an IDL. When used as an IDL, it is possible to specify lines that are passed through without pre-processing, so that control is available concerning whether lines normally handled by the C pre-processor, such as "#include", are executed before DECO, or during normal inclusion of the resulting header file.

3.1.2. Limitations and Restrictions

The newest versions of C++ compilers are needed, because the code requires many recent additions to the draft C++ standard. Under Unix, it may only work with GCC 2.7.2 right now, until other compilers have more of the standard adopted.

3.1.3. User Inputs

3.1.3.1. Syntax

New keywords and built-in types have been added to support dynamic extensions to the C++ language. They are as follows:

dynamic:

> When placed within a class definition, this implies that the variables and functions declared within this scope are to be both public (public: is automatically placed in the resulting code) and dynamic (interfaces are generated to turn them into dynamic attributes and methods). This allows calls with dynamic parameters, but does not make the class itself into a dynamic object.

objdef *static_type dynamic_type*

> This takes a given static class type, which is assumed to have dynamic features within it, and creates a dynamic type by which it can be known. The convention is to use "obj_" followed by the static type name. (i.e.: objdef Blorp obj_blorp;) The syntax is similar to 'typedef' because the function is similar, except that the two types are not quite equivalent. A dynamic object is a reference to the static object, and has run-time type information. As such, it can be automatically converted to type 'obj'.

obj

> This is a new built-in type, which is understood to represent an instance of any class. It has both class and instance information available, and can be converted to its specific static type when the implementation is locally available. The size of given implementation. The actual instance data can reside either within the 'obj' data structure itself, or somewhere else, but this does not change the way in which features are accessed, and when converted to a known static type, it is treated as if indirection is necessary to access the data).
> When an object of type 'obj' is called directly as a function with one argument, it creates an OCF with the object descriptor and the argument, which is treated as a method selector. (i.e.: extern obj io; OCF io_write = io(CSTR("write"));)

OCF

This is the type of the Object Call Frame, and it can be used by methods that need access to not just variable types, but also variable numbers of parameters, or even the original object called, and the method.

An OCF can be called as a function, since it already has information about an object and method. (i.e.: extern OCF io_write; io_write(CSTR("hi\n"));)

Combining this with creating an OCF from an obj, a fully dynamic call be be performed in one step. (i.e.: extern obj io; io(CSTR("write"))(CSTR("boo!\n"));) However, in most cases, it is best to define an interface class that handles calling the methods when their names are known ahead of time, to keep the syntax compatible with static C++. (i.e.: io.write(CSTR("boo!\n"));)

CSTR(*"constant string"*)

This is a macro which converts a constant string into a datatype known as CStr, which has both an address and a length. This way, there is no overhead for calling strlen() to find the length of the string, since sizeof() can find it at compile time for static constant strings. This would be preferable to do this with templates, but there appears to be no way to do so.

Any place where CSTR is used, the code would still function with just the string, and no macro, but would have the added overhead, both in size and execution speed, of having to dynamically determine the length of the static string. Therefore, it is best to use it whenever possible.

3.1.3.2. Examples

Here is a simple string class, some of whose functions have been made dynamic:

```
class String {
private:
        char *ptr;
        size_t len;
public:
        String() { ptr = 0; len = 0; }
        String(char *str);
dynamic:
        String& append(char *str);
        String& append(String &str);
        const char* data() { return ptr; }
};
```

If a descriptor 'str' of type 'obj_String' has been declared and refers to this string class, it can then be called using standard C++ notation:

```
        str.append("something");
        char *cp = str.data();
```

or with dynamic C++ notation:

```
        str(CSTR("append"))("something");
                or:
        static obj_Selector s_append(CSTR("append"));
        str(s_append)("something");
                or:
        static obj_Selector s_append(CSTR("append"));
        OCF str_append(str, s_append);        // refers to object and method
        str_append("something");
```

A completely generic descriptor must be called with the dynamic notation, since method names are not known. However, it can be cast to an interface description object to use the standard C++ notation in the calling code.

Here is a simple object that supports reading and writing, and is implemented using the standard Unix filesystem.

```
class File {
private:
        int _fd;
public:
        File(int fd) : _fd(fd) {}
dynamic:
        int write(char *buf, int len) const
                { return ::write(_fd, buf, len); }
        int write(const CStr &buf) const
                { return write(&buf[0], buf.size()); }
        int write(obj o) const;
};
```

Code that is functionally equivalent to the following is generated by DECO for calling 'write' with any arbitrary number of parameters:

```
        obj write(OCF param) {
                switch(param.count()) {
                case 1: return write(param[1], param[2]);
                default: return Err_BadNParam;
                }
        }
```

Also, if 'write(obj)' had not been specifically declared, code equivalent to the following would be generated for calling write with one parameter of any type:

```
        obj write(obj o1) {
                if((obj_CStr p1 = o1))
                        return write((CStr)p1);
                else
                        return Err_BadType;
        }
```

Return values do not have to be explicitly converted, because implicit conversions occur from built-in data types to their corresponding dynamic object types, and also for user-defined dynamic objects. However, no such conversion will occur for user-defined objects that do not have a 'dynamic:' qualification within them, as it would be inappropriate to attempt to convert an unknown type.

That solves the problem of completely dynamic calls to be found, but a dynamic access method is needed for lookup of all methods based on their selector, so the following code is also generated:

```
obj _call(OCF ocf) {
static obj_Selector s_write(CSTR("write"));
        if(ocf.method() == s_write)
                return write(ocf);
        else
                return Err_BadMethod;
}
```

In order to directly access the object, a special form of object descriptor can be created which insures that it points to an object of type 'File', so that all methods can be called directly, and the internals of type 'File' can be accessed (to the extent that the standard security of C++ will allow).

This is done using the 'objdef' command which is followed by the name of a dynamic class and the name to call the new object descriptor class. For example:

```
objdef File obj_File;
```

3.1.4. User Outputs

The output generated by DECO is C++ code, consisting of the original, non-dynamic C++ code, along with the automatically generated interfaces for the dynamic objects.

Diagnostic messages are issued for errors in the dynamic object specification. All errors in standard C++ code, however, must be caught by the C++ compiler, to simplify DECO and insure that it is not dependent on any one specific version of the C++ compiler.

3.1.5. System Files

In addition to standard header files and libraries, the following files are utilized by DECO:

deco	The DECO pre-processor itself (binary executable).
object.h	Definitions of all data structures used by the DECO object framework, as well as interfaces for standard system objects. This file is automatically generated by using DECO to pre-process 'object.dh'.

3.2. Implementation Tools

A C++ compiler is needed that conforms to most of the specifications of the C++ language in the April 1995 (or later) working paper [X3J16 1995]. For GNU g++ users, this means that GCC 2.7.2 or later should be installed.

3.2.1. Deliverable Items

- Object Framework Specification

  the complete generalized description of the structure of dynamically encapsulated C++ objects, along with the means of accessing the objects. This includes information on the C++ header files that make an implementation useable. This does not include exact details about the exact implementation contained in those header files, only on what types of entities are expected to be there. Programs written based on this specification should be compatible with any implementation.

- DECO

  the actual pre-processor program that generates interfaces to dynamic objects.

- Formal DECO Specification

  the documentation for DECO, including the syntax of the input files, example input files, samples of generated output code, and the exact details of the implementation of the object framework for this version of DECO.

- Test Objects

  these will be used for testing DECO, and the methodologies of the object framework. These will probably include the source code to many objects from Shag/OS v1.0, since the operating system utilizes DECO.

- Thesis

  this is the complete, final thesis, which will contain the formal specification for the object framework, the specification and documentation for DECO, and examples using many of the test objects.

3.3. Qualifications

3.3.1. Books and Articles Read

I have done an abundant amount of reading on many topics related to operating systems, distributed operating systems, object-oriented methodologies, etc, including books, papers, and articles in journals, and I am still compiling a list of ones that are in some way related to this project. The ones that I had easily available to me still, or that I had recently read, are listed in the "References" page at the end of this document. However, as I find more of them, I will add them as references to my actual Thesis, as sources I have used for ideas over the years.

Among some of the relevant materials from long ago that are not yet listed are papers related to such topics as:

- Ameoba
- Clouds
- Plan 9: The early papers
- Oberon

I have also done some reading on (and have a basic understand of) Lisp, Smalltalk, Eiffel, and Python, and I've been keeping up with all the latest information on the standardization of C++ and its libraries, through books, papers, and newsgroups.

In addition, I attended the OSF Research Institute's 1995 Spring Symposium, "The Operating System Balancing Act", and have a large collection of papers that were presented at the two day conference. The relevant ones will be added to my references after I have looked through them again.

3.3.2. Programs Written

3.3.2.1. Operating Systems

- ???, JakeDev (1990)

  a project at Computer Science House to develop a small kernel for Jake, an old, unused, VAX 11/780. It included simple memory management, device drivers, and multitasking.

- Fubar, OS Lab (1991)

  my group's Operating Systems Lab project. I wrote the memory management, device-independent I/O, and serial port device driver.

- Shag/OS 0.34, Independent Study Project (1992)

  this was the first incarnation of Shag/OS, an object-oriented microkernel, in C.

- Shag/OS 1.0, Independent Study Project (1995)

  this is the revised version of Shag/OS, in C++, and is an ongoing project.

3.3.2.2. Language/Parsing

- OOPL (1988)

  an 'Object Oriented Programming Language' for writing interactive fiction. It was an interpreted language, and all builtin keywords were single punctuation characters, so that all words could be used by the programmer.

- Scripted Telnet (1992)

  a Telnet program with a simple scripting language attached for handling customized logins, menus, etc. This was written for ISC and is used as the front-end to access many services, such as SIS (Student Information System).

- Language Processors (1993)

  I wrote an assembler and a simple compiler

- Compiler Construction (1994)

  I wrote several more variants of compilers

3.3.3. Previous Investigations and Projects

- Shag/OS 1.0 - described earlier
- CParse - described earlier
- OO Research

  I have done a lot of research into many varieties of object-oriented programming techniques, and implementation techniques, and have experimenting with coding many different implementations to see what sort of complications each one had.

4. References and Related Material

[Agesen 1994] O. Agesen, ‘‘Type Inference of SELF: Analysis of Objects with Dynamic and Multiple Inheritance,’’ *Software- Practice and Experience*, vol. 25, no. 9, September 1995.

[Aho et al. 1988] A. Aho, R. Sethi, J. Ullman, *Compilers: Principles, Techniques, and Tools,* p. Addison-Wesley, Reading, MA, , 1988.

[Beck 1990] L. Beck, *Systems Software,* 2nd ed., p. Addison-Wesley, Reading, MA, , 1990.

[Chang and Tseng 1995] W. Chang, C. Tseng, ‘‘Supporting Distributed Objects in FIFO-based Message-Passing Systems,’’ *Journal of Object-Oriented Programming*, p. 56-64, February 1995.

[Chen and Lee 1996] J. Chen, S. Lee, ‘‘The Necessary and Sufficient Conditions of Type-Safe Polymorphism,’’ *Journal of Object-Oriented Programming*, p. 33-42, February 1996.

[Daniels and Cook 1993] J. Daniels, S. Cook, ‘‘Strategies for Sharing Objects in Distributed Systems,’’ *Journal of Object-Oriented Programming*, p. 27-36, January 1993.

[Davis 1992] S. Davis, ‘‘C++ Objects that Change their Types,’’ *Journal of Object-Oriented Programming*, p. 27-32, July/August 1992.

[Evered et al. 1995] M. Evered, A. Schmolitzky, M. Kolling, ‘‘A Flexible Object Invocation Language based on Object-Oriented Language Definition,’’ *The Computer Journal*, vol. 38, no. 3, p. 181-191, 1995.

[Ford et al. 1995] B. Ford, M. Hibler, J. Lepreau, ‘‘Using Annotated Interface Definitions to Optimize RPC,’’ , Department of Computer Science, University of Utah, March 31, 1995.

[Grossman 1993] M. Grossman, ‘‘Object I/O and Runtime Type Information Via Automatic Code Generation in C++,’’ *Journal of Object-Oriented Programming*, p. 34-42, July-August 1993.

[Haddad 1995] H. Haddad, K. George, ‘‘A Survey of Method Binding and Implementation Selection in Object-Oriented Programming Languages,’’ *Journal of Object-Oriented Programming*, p. 28-41, October 1995.

[Huang and Chen 1992] S. Huang, D. Chen, ‘‘Efficient Algorithms for Method Dispatch in Object-Orientented Programming Systems,’’ *Journal of Object-Oriented Programming*, p. 43-54, September 1992.

[Kent 1991] W. Kent, ‘‘A Rigorous Model of Object Reference, Identity, and Existence,’’ *Journal of Object-Oriented Programming*, p. 28-36, June 1991.

[Koenig 1995a] A. Koenig, ‘‘Variations on a Handle Theme,’’ *Journal of Object-Oriented Programming*, p. 77-80, October 1995.

[Koenig 1995b] A. Koenig, ‘‘Another Handle Variation,’’ *Journal of Object-Oriented Programming*, p. 61-63, November-December 1995.

[Lewine 1992] D. Lewine, *POSIX Programmer's Guide,* p. O'Reilly & Associates, Sebastopol, CA, , 1992.

[Nascimento and Dollimore 1992] C. Nascimento, J. Dollimore, ‘‘Behavior Maintenance of Migrating Objects in a Distributed Object-Oriented Environment,’’ *Journal of Object-Oriented Programming*, p. 25-33, September 1992.

[Partridge 1994] C. Partridge, ‘‘Modelling the Real World: Are Classes Abstractions or Objects?,’’ *Journal of Object-Oriented Programming*, p. 39-45, November-December 1994.

[Plauger 1995] P. Plauger, *The Draft Standard C++ Library,* p. Prentice Hall, Englewood Cliffs, NJ, , 1995.

[Pratt 1984] T. Pratt, *Programming Languages: Design and Implementation,* 2nd ed., p. Prentice Hall, Englewood Cliffs, NJ, , 1984.

[Presotto et al. 1991] D. Presotto, R. Pike, K. Thompson, H. Trickey, ‘‘Plan 9, A Distributed System,’’ *Proceedings of the Spring 1991 EurOpen Conference*, vol. 2nd ed., p. 43-50, May 1991.

[Silberschatz et al. 1991] A. Silberschatz, J. Peterson, P. Galvin, *Operating System Concepts,* 3rd ed., p. Addison-Wesley, Reading, MA, , 1990.

[Stroustrup 1993] B. Stroustrup, *The C++ Programming Language,* 2nd ed., , Addison-Wesley, Reading, MA, 1993.

[Stroustrup 1995] B. Stroustrup, *The Design and Evolution of C++,* , Addison-Wesley, Reading, MA, 1995.

[Tanenbaum 1995] A. Tanenbaum, *Distributed Operating Systems,* , Prentice Hall, Englewood Cliffs, NJ, 1995.

[Voss 1994] R. Voss, ‘‘Time Invariant Virtual Member Function Dispatching for C++ Evolvable Classes,’’ *Journal of Object-Oriented Programming*, p. 23-33, November-December 1994.

[Wieringa 1995] R. Wieringa, ‘‘Combining Static and Dynamic Modelling Methods: A Comparison of Four Methods,’’ *The Computer Journal*, vol. 38, no. 1, p. 17-30, 1995.

[Wilde 1991] M. Wilde, ‘‘A Dynamic C-Based Object-Oriented System for Unix,’’ *IEEE Software*, p. 73-85, May, 1991.

[Wirth 1995] N. Wirth, ‘‘A Plea for Lean Software,’’ *Computer*, vol. 28, no. 2, p. 64-68, IEEE Computer Society, February 1995.

[X3J16 1995] A. Koenig, ‘‘Working Paper for Draft Proposed International Standard for Information Systems-- Programming Language C++, (X3J16/95),’’ , AT&T Bell Laboratories, Murray Hill, NJ , April, 1995.