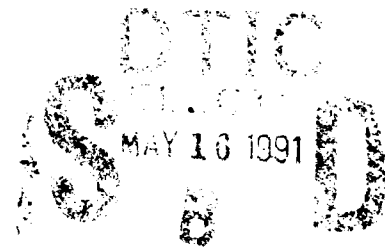


NPS52-90-024

NAVAL POSTGRADUATE SCHOOL

Monterey, California

AD-A235 663



AN INTRODUCTION TO OBJECT-ORIENTED
PROGRAMMING

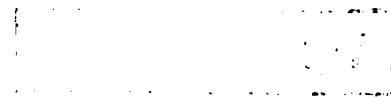
Michael L. Nelson

April 1990

Approved for public release; distribution is unlimited.

Prepared for:

Naval Postgraduate School
Department of Computer Science, Code CS
Monterey, California 93943



91-00006



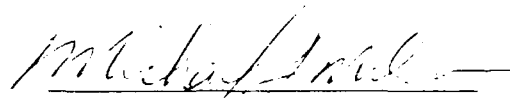
NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral R. W. West, Jr.
Superintendent

Harrison Shull
Provost

This report was prepared in conjunction with research funded by the Naval Postgraduate School Research Council.

Reproduction of all or part of this report is authorized.



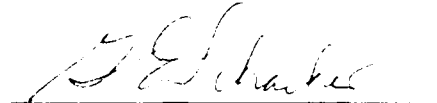
MICHAEL L. NELSON
Assistant Professor
of Computer Science

Reviewed by:



ROBERT B. MCGHEE
Chairman
Department of Computer Science

Released by:



GORDON E. SCHACHER
Dean of Faculty
and Graduate Studies

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED		1b. RESTRICTIVE MARKINGS	
2a. SECURITY CLASSIFICATION AUTHORITY		3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited	
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE		4. MONITORING ORGANIZATION REPORT NUMBER(S) NPS52-90-024	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) NPS52-90-024		5. MONITORING ORGANIZATION REPORT NUMBER(S)	
6a. NAME OF PERFORMING ORGANIZATION Computer Science Dept. Naval Postgraduate School	6b. OFFICE SYMBOL (if applicable) 52	7a. NAME OF MONITORING ORGANIZATION	
6c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943		7b. ADDRESS (City, State, and ZIP Code)	
8a. NAME OF FUNDING SPONSORING ORGANIZATION Naval Postgraduate School	8b. OFFICE SYMBOL (if applicable) CS	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
8c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943		10. SOURCE OF FUNDING NUMBERS	
		PROGRAM ELEMENT NO.	PROJECT NO.
		TASK NO.	WORK UNIT ACCESSION NO.
11. TITLE (Include Security Classification) AN INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING (U)			
12. PERSONAL AUTHOR(S) Michael L. Nelson			
13. TYPE OF REPORT Summary	13b. TIME COVERED FROM TO	14. DATE OF REPORT (Year, Month, Day) April 1990	15. PAGE COUNT 24
16. SUPPLEMENTARY NOTATION			
17. COSATI CODES		18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)	
FIELD	GROUP	SUB-GROUP	
		Object-oriented programming, abstract data types, encapsulation, inheritance, polymorphism, genericity	
19. ABSTRACT (Continue on reverse if necessary and identify by block number)			
<p>Like many new ideas, object-oriented programming (OOP) does not yet have a universally accepted definition. Even the terminology of OOP can vary greatly from one system or language to another.</p> <p>This paper introduces OOP to the newcomer in a language-independent manner. The "underlying theory" of OOP is presented to give the reader the basics necessary to understand the nuances of the various OOP languages that are available. Several OOP languages are briefly considered, as are object-oriented database management systems, object-based programming, and object-oriented design. Various problem areas are explored in detail. This paper should also be of considerable help in making the transition from one OOP language to another.</p>			
20. DISTRIBUTION AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS		21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL Michael L. Nelson		22b. TELEPHONE (Include Area Code) (408) 646-2449	22c. OFFICE SYMBOL 52Ne

TABLE OF CONTENTS

1.	INTRODUCTION	1
2.	WHAT IS OBJECT-ORIENTED PROGRAMMING?	1
2.1	OBJECTS, CLASSES, AND INHERITANCE	2
2.1.1	VARIATIONS ON INHERITANCE	4
2.2	A COMPARISON WITH ABSTRACT DATA TYPES	4
2.3	POLYMORPHISM	4
2.4	COMPOSITE OBJECTS	5
2.5	ABSTRACT AND CONCRETE CLASSES	6
2.6	THE METACLASS	6
2.7	DELEGATION AND PROTOTYPING	7
2.8	CONCURRENT OBJECT-ORIENTED SYSTEMS	7
3.	OBJECT-ORIENTED PROGRAMMING LANGUAGES	7
3.1	"BUILT-IN" LANGUAGES	8
3.1.1	SMALLTALK	8
3.1.2	EIFFEL	9
3.2	"BOLTED-ON" LANGUAGES	9
3.2.1	C-BASED OOP LANGUAGES	9
3.2.2	LISP-BASED OOP LANGUAGES	10
3.2.3	PASCAL-BASED OOP LANGUAGES	10
4.	OBJECT-ORIENTED DATABASE MANAGEMENT SYSTEMS	10
5.	PROBLEM AREAS	11
5.1	SIMPLE TERMINOLOGY DIFFERENCES	11
5.2	CLASS VARIABLES	11
5.3	ENCAPSULATION VS INHERITANCE	12
5.4	SUBCLASSING VS SUBTYPING	13
5.5	MULTIPLE INHERITANCE	13
5.6	GENERICITY	14
6.	OBJECT-BASED PROGRAMMING	15
7.	OBJECT-ORIENTED DESIGN	15
8.	CONCLUSIONS	16
	REFERENCES	17
	INITIAL DISTRIBUTION LIST	20

LIST OF FIGURES

Figure 1	- AN INHERITANCE HIERARCHY	3
Figure 2	- A MULTIPLE INHERITANCE LATTICE	3
Figure 3	- THE CLASS DEFINITION	3

	in For
	A&I <input checked="" type="checkbox"/>
	<input type="checkbox"/>
	<input type="checkbox"/>
	Special

A-1

1. INTRODUCTION

Object-oriented programming (OOP) is still a relatively young field that does not yet have a universally accepted definition that can be called upon. It is somewhat disturbing that the terminology of OOP varies so greatly from one system to another - there are slots and variables, classes and types and flavors, class variables and shared variables, etc. And even when the terminology seems to be the same from one language to the next, there can be some very subtle variations in the implementations and usage.

Something of an object-oriented "Tower of Babel" has been created. And while it is hard enough for someone with OOP experience to understand all of the various dialects and differences, it is even more difficult for the newcomer to understand what is going on.

This paper is intended to introduce OOP to the newcomer. Rather than approach OOP from the viewpoint of a specific language, the basic ideas of OOP (i.e., the underlying theory and a basic vocabulary) are presented. The reader should keep in mind that the actual terminology can vary greatly between languages. However, since these seemingly different approaches can be mapped into the same set of basic ideas, they can also be mapped into one another.

We begin by answering the question of 'what is object-oriented programming?', then take a brief look at several OOP languages and object-oriented database management systems. Various problem areas are then explored in detail, followed by a brief examination of object-based programming and object-oriented design.

2. WHAT IS OBJECT-ORIENTED PROGRAMMING?

As there is no universally accepted definition of OOP, let us begin with several notable quotes on the subject.

"My guess is that object-oriented programming will be in the 1980's what structured programming was in the 1970's. Everyone will be in favor of it. Every manufacturer will promote his products as supporting it. Every manager will pay lip service to it. Every programmer will practice it (differently). And no one will know just what it is." [Rent82]

"Object-oriented is well on its way to becoming the buzzword of the 1980's. Suddenly everybody is using it, but with such a range of radically different meanings that no one seems to know exactly what the other is saying." [Cox86]

"Object-oriented has become a buzzword that implies 'good' programming." [Stro88]

"I have a cat named Trash. In the current political climate, it would seem that if I were trying to sell him (at least to a Computer Scientist), I would not stress that he is gentle to humans and is self-sufficient, living mostly on field mice. Rather, I would argue that he is object-oriented." [King89]

All these quotes entail essentially the same meaning - that OOP is good and that it is currently a hot sales item, but that there is no general agreement as to just what it is.

What is needed is a definition general enough to encompass all of the current views of OOP, yet strong enough to stand up as the basis for the underlying theory of OOP. The following equation [Wegn87] comes very close to fulfilling that need, and is used as a starting point:

object-oriented = objects + classes + inheritance

2.1 OBJECTS, CLASSES, AND INHERITANCE

If this definition of OOP is to be accepted, then obviously the terms contained within it must be defined. In defining the terms class and object, it is easy to get caught in the trap of a circular definition in which an object is defined to be an instance of a class and a class is defined to be a description of similar objects.

This confusion can be avoided by defining an *object* to be a self-contained set of variables which can be manipulated only by a set of methods (procedures) defined exclusively for that purpose. A *class* can then be defined as a description of similar objects, like a template or cookie cutter [Wegn87], or as a factory that produces objects [Cox86].

The variables making up an object can be divided into two sections: class variables and instance variables. A *class variable* is shared in both name and value by all instances of a class, while an *instance variable* is shared in name only by all instances of a class (i.e., each object has its own local version of an instance variable, but all the objects of the class access the same class variable).

The procedures or operations that are defined for the object are called *methods*. A *message* is sent to an object to tell it to perform one of its methods. Ideally, the only way to access any of the variables making up an object is by sending the object a message. In this way, an object is said to be *encapsulated* in that its internal structure may be modified without affecting user-written code which accesses the object (as long as appropriate modifications have also been made to the object's methods).

Inheritance can be defined simply as a code sharing mechanism. It allows a new class to be defined based upon the definition of an existing class without having to copy all of the existing code. A *subclass* inherits all of the variables and methods defined for its *superclass* (including those variables and methods which were inherited by the superclass from some other class).

With single inheritance (usually referred to just as inheritance), a class may have only one superclass. *Multiple inheritance* (MI) allows a class to have several superclasses (i.e., it inherits the definition of each superclass). A *class hierarchy* is a diagram which shows the inheritance relationship between various classes (see Figure 1). Under MI, the inheritance hierarchy is technically called a *lattice*, although it is fairly common to refer to it more simply as a multiple inheritance hierarchy (see Figure 2).

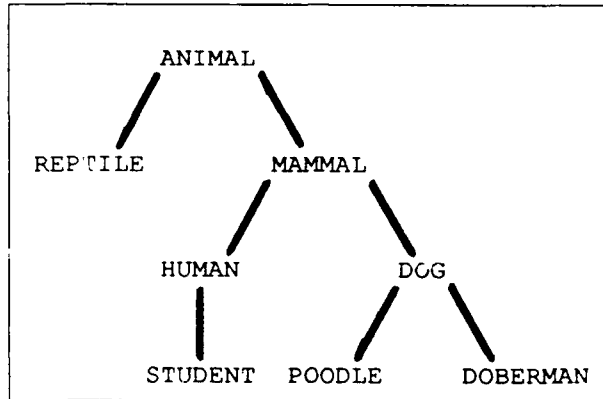


Figure 1 - An Inheritance Hierarchy

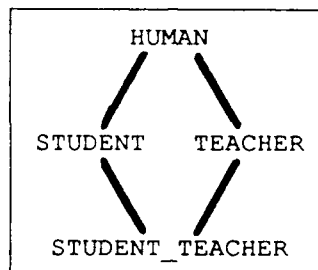


Figure 2 - A Multiple Inheritance Lattice

A very simple and succinct way of representing class definitions, without using any specific language, is now possible (see Figure 3).¹ This model, along with the definition of object-oriented = objects + classes + inheritance [Wegn87], encompasses the underlying theory of OOP. It looks amazingly simple, but that is really all that there is to it. This simplicity should not detract from OOP, rather it should be heralded as one of its central themes: OOP, one of the most powerful software methodologies available today is actually a very simple idea that anyone should be able to understand and put to use.

```

Class <class_name>
  Superclasses:      <superclass_1>, <superclass_2>, ...
  Class Variables:  <class_var_1>, <class_var_2>, ...
  Instance Variables: <inst_var_1>, <inst_var_2>, ...
  Methods:          <method_name_1>, <method_name_2>, ...
  
```

Figure 3 - The Class Definition

¹Figure 3 represents the specification of a class which can easily be extended to a more detailed specification. For example, variables can be typed to model a more strongly typed system. The specification can also be divided into parts such as public, private, and subtype-visible (i.e., who can access the variable or method - anyone can access public variables and methods, only methods defined for the class may access private variables and methods, while both methods defined for the class and subclasses of the class may access subtype-visible variables and methods). However, this simple specification, even without enhancements, captures the fundamental properties of the class definition.

2.1.1 VARIATIONS ON INHERITANCE

It is easy to think of a subclass as "everything that its superclass is, plus some new variables and methods which are defined locally". This is sometimes true, in that it is relatively simple to define new variables and methods in the subclass, which in effect are added to those that are inherited.

However, in most OOP languages, it is also possible to redefine variables and methods which would otherwise have been inherited from the superclass(es). That is, a variable and/or method could be declared in the subclass with the same name as a variable/method that is defined in (or inherited by) the superclass. In this case, the new definition in the subclass simply overrides the definition which would have otherwise been inherited.

It is also possible in some OOP languages to exclude variables and/or methods which would otherwise have been inherited from the superclass(es). That is, the subclass inherits everything from a superclass that is not specifically excluded.

2.2 A COMPARISON WITH ABSTRACT DATA TYPES

An abstract data type (ADT) can be defined as the encapsulation of a data structure along with a set of operators, in which the implementation details of both the data structure and the operators are hidden. This allows the ADT to be referenced with implementation-independent code. Without inheritance, OOP is really nothing more than an ADT system.²

Actually, abstract data types are used in computer systems all the time [OFS84]. The only non-abstract type on the computer is the bit string; all other types have an internal representation (a string of bits), an external representation (typically a string of characters or numbers), and some set of operations to manipulate these data types. Thus, both abstract data types and object-oriented programming can be thought of as ways to allow users to add their own types to those supplied by the system.

2.3 POLYMORPHISM

Polymorphism (sometimes called *operator overloading* or *function overloading*) can be defined as allowing different data types (classes) to have methods (routines) with the same name which may be implemented differently [CW85, Mica88]. We actually use polymorphic routines all the time, even in conventional systems. For instance, we have a plus operation for integers and a plus operation for real numbers - in most cases these operations are implemented differently, but the same symbol ('+') is used for each operation. The compiler and/or run-time environment is "smart enough" to determine which version of the operation to use based upon the arguments (i.e., use the integer plus for integers and the real plus for real numbers).

²OOP has even been defined in terms of abstract data types, as "object-oriented = data abstractions + abstract data types + type inheritance" [Wegn86].

One possible alternative to implementing polymorphism would be to have a single operation defined which would in affect be a "giant" case statement. Every call to that operation would then choose the appropriate case to determine which part of the code to execute (e.g., the plus operation on integers would choose the integer case, etc). However, true polymorphism allows the various operations to be created and tested independently of one another. Adding a new version of the operation has no affect upon the existing code at all.

Simple polymorphism (usually referred to simply as polymorphism) allows different classes to each have their own implementation of an operation. *Multiple polymorphism* allows each class to have several operations with the same name. Once again, the proper operation is chosen based upon the arguments provided.

For example, a class could have several print routines, all doing essentially the same thing, but doing it differently depending upon which printer was chosen. Multiple polymorphism allows a separate print routine to be developed for each printer, independently of all the other print routines. As with simple polymorphism, an alternative would be to have a single routine defined which was again a "giant" case statement, selecting the proper section of code for the chosen printer. Once again, the various versions of the operations may be created and tested independently, and new versions may be added with no affect upon the existing code.

2.4 COMPOSITE OBJECTS

A *composite object* (sometimes called an *aggregate object*) is an object which consists of other objects. That is, the variables (or some of the variables) which make up the object represent other objects.³ If these variables are themselves objects, then they are called *dependent objects*. If these variables are pointers to other objects, then they are called *subobjects*.

A dependent object is completely dependent upon the composite object. It cannot be created unless the composite object is created first. If the composite object is deleted, then all of its dependent objects are also deleted, as there would be no way to refer to them individually.

Subobjects, on the other hand, may exist as stand-alone objects in their own right. Either the subobject or the composite object may be created first. If the composite object is deleted, then a decision must be made about whether or not the subobjects should also be deleted. Subobjects may also be shared between composite objects; this is not possible with dependent objects.

³It could be argued that virtually every object is a composite object. Any variable which makes up an object must itself be of some type (such as real, integer, or a user-defined type). It is possible for an object to have no variables but still respond to some set of messages, and this would be the only type of non-composite object. However, the term *composite object* is usually applied only in those cases where the object is made up of other user-defined objects.

Actually, most objects may be viewed as some type of composite object if no distinction is made between system-supplied types and user-defined classes. If, for instance, an object consists of two integer variables and one real variable, then the object could be viewed as a composite object consisting of three dependent objects (or subobjects) - two of which were integer objects and the other a real object.

Composition may also be compared with inheritance [Cox86, H087, Nels88], choosing one or the other depending upon the relationships involved. Composition can be thought of as a form of part inheritance, while inheritance is more concerned with behavioral inheritance. For example, an automobile is constructed from a body, an engine, tires, seats, etc. It would therefore probably be most appropriate to use some form of composition rather than inheritance. A student_teacher, on the other hand, inherits the characteristics of both a student and a teacher - it is not a student and a teacher "glued" together. Therefore, inheritance is probably the most appropriate construct.

Another alternative also exists - an encapsulated form of inheritance [Nels88, Snyder86], sometimes called *enheritance*. With composition, the methods of the dependent objects (subobjects) are not inherited by the composite object. These messages cannot be sent to the composite object, they must be directed to the appropriate dependent object (subobject). With inheritance, the methods of the superclass(es) are inherited, and these messages can be sent to instances of this new subclass. However, there may be some name conflicts which must be resolved (see the Encapsulation vs Inheritance Section for more information on this). Enheritance can be viewed as a form of composition in which the methods associated with the dependent objects (subobjects) are inherited, or as a form of inheritance in which name conflicts are avoided by keeping separate and distinct variables that just happen to have the same name.

2.5 ABSTRACT AND CONCRETE CLASSES

An *abstract class* is a class which does not have any instances. It generally exists to be used only as an ancestor to other classes which may have instances. A *concrete class* is one which does have instances, although it may also be used as an ancestor to other classes.

This author does not know of any language which provides abstract classes and concrete classes as constructs (i.e., a language in which there is a way to designate a class as abstract so that it may not have any instances). The terms may, however, be used to describe the various classes in a system and be useful in its design and maintenance.

2.6 THE METACLASS

The *metaclass* is a special class provided by some OOP systems which is mainly used to create new user-defined classes. It is possible to take the viewpoint that everything in an object-oriented environment is an object, and that everything is accomplished by sending messages to the objects. With this approach, classes must also be considered to be objects in that messages are sent to them to create new instances (objects) of that class. However, messages must also be sent to create new classes, and these messages are sent to the metaclass. Multiple metaclasses are also possible, with each metaclass setting different parameters/standards for classes which they create.

This approach, however, can lead to an infinite set of classes. The metaclass must be considered an object in its own right, and is therefore created by the metametaclass, which is in turn created by the metametametaclass, etc. Most languages which support the metaclass concept either ignore this problem all together, or they simply decree that the metaclass is a special object provided by the system.

Those languages which do not support the metaclass concept can be considered to be equivalent to this approach. Messages to create new classes are handled by the system, which for all practical purposes is the metaclass.

2.7 PROTOTYPING AND DELEGATION

Prototyping (delegation) [Lieb86, Stei87, SLU89, Wegn87] is probably the most notable variation on OOP. With inheritance, an object is defined by its class. That class may inherit part of its definition from another class (the superclass). With delegation, each object is defined individually (there are no classes). An object *delegates* to its prototype (another object) for any variable or operation which is not defined locally. That is, an object "asks" its prototype to supply any missing information.

In a class-based (inheritance) system, we could think of an object as delegating to its class which in turn delegates to its superclass(es), and so on. Notice, however, that we are only delegating to classes, never to instantiations. In a delegation-based system any object may serve as a prototype.

Technically, these types of languages do not fit within the definition of "object-oriented = objects + classes + inheritance" [Wegn87]. To keep from excluding this form of OOP, it is necessary to modify the definition as follows:

object-oriented = (objects + classes + inheritance) OR
(objects + delegation)

2.8 CONCURRENT OBJECT-ORIENTED SYSTEMS

Concurrent systems are another important aspect of object-oriented programming. Although much interesting research is taking place in this area, it is considered to be beyond the scope of this introductory paper. It should be noted, however, that there are many intriguing possibilities in moving objects into the concurrent world. A good introduction to concurrent OOP systems may be found in [Nels90a] and [TS89].

3. OBJECT-ORIENTED PROGRAMMING LANGUAGES

There are many languages that claim to be object-oriented. Depending upon the definition of OOP that is used, some languages are truly object-oriented, but many are not. In this section, several languages that meet the definition of "object-oriented = objects + classes + inheritance" will be (briefly) reviewed.

Object-oriented features can be added to an existing language, in which case it is referred to as a *bolted-on* language. A language is

considered to be *built-in* if it is designed around the principles of object-oriented programming. Obviously, any OOP language which is designed as an extension or super set of an existing non-OOP language will fall into the bolted-on category.

Depending upon your point of view, a bolted-on language can either be a problem or a blessing. It allows the programmer to mix an object-oriented style of programming with more conventional code. This may be used to produce a more efficient system, but it may also generate a program which is simply a mixed up mixture of old (conventional) and new (OOP) statements.

Bolted-on approaches can also lead to problems of determining who can really design and code an object-oriented program and who cannot. In dealing with super sets, it is possible for someone to write a completely conventional program, run it through the OOP compiler/pre-processor, then proclaim "I wrote a program in an object-oriented language, therefore it is an object-oriented program and I am (sic) an object-oriented programmer".

3.1 "BUILT-IN" LANGUAGES

As previously defined, a built-in language is one which has been designed "from the ground up" as an object-oriented language. Smalltalk is probably the most popular and widely recognized built-in OOP language, and will be the only built-in OOP language reviewed in this section.

3.1.1 SMALLTALK

Smalltalk grew out of Alan Kay's Dynabook project [KG77]. More than just a language, it provides a complete environment. That is, once the Smalltalk environment is entered, it takes care of all of the overhead (operating systems calls, etc). At this time, most available versions of Smalltalk are interpretive, and therefore are not as fast as might be hoped for. However, Smalltalk compilers are being developed.

Smalltalk-80 [GR83] is a product of Xerox PARC. It was initially released as Smalltalk-72, followed by Smalltalk-74 and Smalltalk-76. It is the "original" Smalltalk. Smalltalk-80 is available for a wide variety of machines, but is not available for PC class machines. Smalltalk/V [Digi86, Digi88] is a product of Digital, Inc. It also runs on several machines, including PC class machines.

I have not yet seen a published comparison of Smalltalk-80 and Smalltalk/V - even the sales representatives do not want to make a comparison (they claim that they do not want to "badmouth" the competition). However, from asking several questions of sales representatives of both Xerox PARC and Digital, Inc at the OOPSLA'88 Conference (Oct 1988, San Diego, CA), the following was ascertained:

- (1) they are essentially the "same" language (kind of like different versions of other languages available from different companies);
- (2) Smalltalk-80 provides more tools, a "better, richer" environment, etc; and
- (3) Smalltalk/V is smaller, and therefore can run on smaller machines.

3.1.2 EIFFEL

Eiffel [Meye87, Meye88a, Meye88b, Meye88c] was developed by Bertrand Meyer at Interactive Software Engineering Inc. It is a compiled OOP language, and although it uses C as an intermediate language (i.e., it is "compiled" into C code), it is not bolted-on to the C language. C is used as an intermediate language due to the large number of systems which support it - any system which supports C may also support Eiffel. Eiffel is intended to serve as both a language and environment for designing software that is easily reusable and extendible.

The notion of programming as contracting [Meye88b, Meye88c], in which the relationship between a class and its clients is viewed as a formal agreement (which expresses both party's rights and obligations) is often associated with Eiffel. However, this approach can be viewed simply as a formalization of the idea that it is the external interface which provides users with the only way to access an object. Programming as contracting simply means that: (1) only the result of sending an object a certain message with a certain set of arguments is important - it does not matter how the method is implemented, and (2) this external interface cannot be changed without modifying the original contract. Each side, however, is free to make any changes to their implementations that do not affect the interface itself. As such, programming as contracting could be implemented in virtually any OOP language.

3.2 "BOLTED-ON" LANGUAGES

As previously defined, a bolted-on language is one in which object-oriented principles have been added to an existing language. Many languages fit into this category, with the two most popular being those based on C [KR78] and those based on Lisp [Stee84]. Various C-based and Lisp-based languages will be briefly investigated in this section. OOP systems based on Pascal [Grog78] will also be briefly examined.

3.2.1 C-BASED OOP LANGUAGES

One of the advantages of C-based languages that are true super sets of C is that they maintain C's portability. That is, if a preprocessor (a compiler that compiles the C-based language into "regular" C code) is available, then any machine which supports C also supports the C-based OOP language.

C++ [Stro86, WP88] was developed by Bjarne Stroustrup at Bell Labs. It was originally a C preprocessor, but C++ compilers are now available. The original goals of C++ were to:

- (1) retain C's efficiency (C++ is not faster than C, but it should not be any slower either);
- (2) provide 100% upward compatibility from C (i.e., C++ consists completely of extensions to C, making no modifications to the C language); and
- (3) fix some of C's "weaknesses", specifically providing strong typing (which is available, but not mandatory due to the upward compatibility requirements) and data hiding.

Multiple inheritance was not originally included in C++, but some of the more recent releases include this feature.

Objective-C [Cox86] was developed by Brad J. Cox at Productivity Products International. While the C++ extensions to C were designed to "look like" the original C code, the Objective-C extensions were designed more along the lines of Smalltalk code. Everything in Objective-C that has been added to C is enclosed in brackets ('[...]') so that it stands out in a program listing.

Some of the terminology of Objective-C is quite different from other OOP languages. For instance, Objective-C associates a *factory object* with each class. It is this factory object which is used to create new objects. The term *software-IC* is advanced as an alternate name for a class. This is to emphasize the parallel with the hardware silicon chip, a technology that radically reshaped the way that hardware engineers build systems.

3.2.2 LISP-BASED OOP LANGUAGES

Several Lisp-based languages have been developed. While some of these languages are still in use, the Lisp community seems to be migrating towards CLOS (Common Lisp Object System) [BDGK88, Keen89, Moon89]. This is a proposed standard for ANSI Common Lisp, and has been proposed as a "replacement" for all other Lisp-based OOP languages.

At the ACM Lisp and Functional Programming Conference (summer of 1986), several members of the Lisp community decided that while experimentation should continue, a standard system was needed. An ad hoc committee was started at the conference, which grew into the X3J13 committee for the formal standardization of Common Lisp. The "best" features of the various Lisp-based OOP languages were incorporated into CLOS. CLOS may well become a major OOP language in the future, partly because it is formally a part of Common Lisp.

3.2.3 PASCAL-BASED OOP LANGUAGES

At this point in time, Pascal-based OOP languages have not made much of an impact upon the OOP market. This may change, however, due to the wide acceptance and use of Pascal in other areas as OOP extensions become available.

Turbo Pascal 5.5 [Borl89], a product of Borland, is one such extension. Classes and inheritance have been added, but encapsulation is not enforced by the system. It is, however, definitely a step in the right direction.

4. OBJECT-ORIENTED DATABASE MANAGEMENT SYSTEMS

A *database* (DB) can be defined as a collection of data stored "permanently" in a computer [Ullm82]. The data to be stored in a database is said to be *persistent* in that it survives beyond a single programming session [AH87, CSWZ87]. A *database management system* (DBMS) is the software that allows users to use and maintain the data in the database [Ullm82]. An *object-oriented database management system* (OODBMS) can then be defined as the software that allows users to use and maintain objects (data) in an object-oriented database [Nels90b]. The objects stored in the database are persistent in that they survive beyond a single session [Nels90b].

A complete treatment of object-oriented database management systems is considered to be beyond the scope of this introductory paper. For more information, the reader is referred to [Nels90b].

5. PROBLEM AREAS

If OOP is really all that simple, then why do all of the languages seem to be so different?

5.1 SIMPLE TERMINOLOGY DIFFERENCES

A few differences can be explained simply as a case of using different terms to mean the same thing. For example, a slot can be thought of as another name for a variable (either an instance variable/slot or a class variable/slot), and a flavor is essentially the same as a class. A type may also be used to mean the same thing as a class, but special care is needed here to determine the exact usage (for more information, see the section on subclassing vs subtyping). Unfortunately, other differences are not so simple.

5.2 CLASS VARIABLES

A class variable is one which is shared in both name and value by all objects that are instances of the class. But even this simple definition can be interpreted in two different ways.

Consider, for example, a class variable defined in a superclass (and therefore inherited by every subclass of that superclass). Obviously, all of the objects which are instances of the superclass will share that class variable in both name and value. But what about instances of the subclass? These objects should also all share that class variable in both name and value, but is this the same variable that is shared by the instances of the superclass? This is a question of both implementation and philosophy - if an instance of a superclass modifies the value of a class variable, is the value also modified for instances of the subclass, and vice versa?

The implementation part of this question is fairly easy to resolve - it can be viewed as a copy vs pointer implementation issue. If the subclass receives a pointer to the superclass class variable, then changes will be reflected up and down the hierarchy. If, on the other hand, the subclass receives a copy of the superclass class variable, then changes will be seen within individual classes only.

The philosophical aspects, however, are not that easy to resolve. Rather than argue that one approach is right and the other is wrong, it should only be noted that either one may be useful in certain situations. The problem then becomes: given a specific language and its particular implementation of class variables, how can the other approach be "simulated" if that is what is needed? An additional consideration is whether or not a language should provide both kinds of class variables.

Another problem with the term class variable is its emergence in C++ literature with a completely different usage from the more generally accepted idea. An object is sometimes referred to as a class variable [WP88], and this usage should be halted immediately. It does make some

sense to say that an object is a class variable in that it is represented by a variable and it is an instance of a class. However, this usage should be regarded as entirely inappropriate since the idea of a class variable with a completely different meaning already exists within the OOP world.

5.3 ENCAPSULATION VS INHERITANCE

Encapsulation can be defined as a form of information hiding. As previously discussed, it allows us to make changes to the implementation of a system with minimal effects upon the end-user. Inheritance can be defined as a code sharing mechanism. It allows new classes to be built upon existing classes. While it has been claimed by various sources that either encapsulation or inheritance alone is the central feature of OOP, it is probably more appropriate to agree that both features are necessary to make the claim that a language is truly object-oriented.

Encapsulation without inheritance is actually nothing more than an abstract data type system. As previously discussed, an ADT can be defined as the encapsulation of a data structure along with a set of operators, in which the implementation details of both the data structure and the operators are hidden. This allows the user to reference the ADT with implementation-independent code. Encapsulation is an important part of OOP, but is not sufficient by itself to designate an OOP language.

Inheritance without encapsulation may give the "look and feel" of an OOP language - it is possible to create objects and give the end-user methods to manipulate them, but there is no way to keep the user from accessing the internals of an object. Although "hackers" may prefer this type of system, allowing end-users to access the internals of an object can have the effect of constraining the system to the original implementation, even if a modification may lead to a more efficient or a more powerful system.

If it is agreed that both inheritance and encapsulation are necessary for a language to be considered truly object-oriented, the definition of "object-oriented = objects + classes + inheritance" [Wegn87] can still be used by agreeing that the concept of a class implies some form of encapsulation.⁴

In a typed system (i.e., a system in which variables are defined to be of a certain type or class), encapsulation can be violated by inheritance, and this can cause several problems [Nels88, Snyd86]. As a simple example, consider a Class A with a variable Q defined as an integer. Now say that a Class B is defined to be a subclass of A. If B's methods are allowed to access A's variable Q directly (by name), then A's definition is essentially locked in - if it should become necessary to modify A's definition so that the variable Q becomes a real number, then methods defined for B which expect the variable Q to be an integer will no longer function properly.

⁴As previously mentioned, OOP has been defined in terms of abstract data types as "object-oriented = data abstractions + abstract data types + type inheritance" [Wegn86]. Although encapsulation is still not mentioned per se, the idea of an abstract data type definitely implies some level of encapsulation.

5.4 SUBCLASSING VS SUBTYPING

The terms class and type create much confusion for the newcomer to OOP (and probably for many "oldtimers" as well). The terms superclass and subclass are used in defining inheritance (a subclass inherits from its superclass), and the class of a variable is simply the class that the variable is an instance of. The word type is sometimes used in lieu of class, so that we then have subtypes and supertypes.

The term type, however, is also used to refer to the interface available to an object (i.e., the set of methods it has available). It is then said that two objects are of the same type if they share the same interface, in name only.⁵

Obviously, objects that are instances of the same class are of the same type. If two classes have the same set of methods available (either by chance or by design), then objects that are instances of either class are also said to be of the same type. Two classes are said to have a *common subtype* (sometimes referred to as *conforming* to a common subtype) if some subset of their interfaces are the same. Thus the terms class and type may or may not be used interchangeably.

This problem can be better understood by considering the relationship of OOP to conventional languages. In a conventional language, variables are said to be of a certain type (real, integer, character, etc.) which is provided by the system. One of the goals of OOP is to allow the user to represent any arbitrary object, and then be able to treat this object no differently than a system-supplied object. Indeed, the term class may have been an unfortunate choice as most people tend to think in terms of the type of a variable rather than its class.

The term class, however, seems to have caught on as the predominant term for defining new types (classes) of variables. Since it is generally inappropriate to speak of variables in a conventional system (an integer and a real number, for example) as being of the same type just because they share the same interface, perhaps some alternative term for what is now called subtyping should be found. This could also help to eliminate distinctions between system-supplied data types and user-defined classes.

5.5 MULTIPLE INHERITANCE

Multiple inheritance (MI) is really not too complicated in itself, as it simply allows a class to inherit from two (or more) superclasses. The real problem with MI is in resolving name conflicts - what to do if several variables (or methods) are inherited that have the same name. Much research has been done in this area [Hend86, Nels88, Snyder86, SB86], and there are no easy answers. The implementation in each language which supports MI is slightly different in the way that it handles the name conflicts. The problem itself is amazingly simple though: if the

⁵It makes no difference if the implementations of these methods are completely different. For example if the class `my_integers` has only '+' and '-' operations and the class `my_reals` also has only '+' and '-' operations (implemented differently from the operations of `my_integers`), then objects of these classes are said to be of the same type.

variable (method) being accessed is not defined/maintained locally, where should we look next?

Even though the problem can be stated so simply, it should not be construed that solutions to it are also trivial. Just keep in mind what the problem really is when comparing the various alternatives.

5.6 GENERICITY

Genericity is another term that does not seem to have a universally accepted definition, and it has been used in many different ways within OOP (some of the uses are quite appropriate, but others are rather dubious). Since an exact definition does not exist, consider the following examples.

An ADT is one of the more common examples of genericity. Since the implementation details of the data structure and its operators are hidden, the user can reference the ADT with implementation-independent code. This allows the physical implementation of the ADT to be changed without affecting the user-written code. We can therefore deal with a generic object rather than a specific implementation. OOP itself is representative of this form of genericity, in that objects (as instances of various classes) are a form of an ADT.

Another familiar example of generic code is that of a swap or sort routine that exchanges the values of variables (objects) or sorts a collection of them according to some set of rules. These generic routines are defined and then instantiated as needed for various data types (classes). Any language that allows functions to be passed as parameters provides some capability in this area. The idea of subclass responsibility [Cox86] is similar in that a method can be implemented for a superclass which assumes the presence of another method (to be supplied by the subclass). In general though, the OOP community has done little with the idea of generic routines that can be instantiated for different classes.

A closely related area that helps to confuse the issue is that of polymorphism. As previously discussed, this can be defined as allowing different data types (classes) to have methods (routines) with the same name which may be implemented differently [CW85, Mica88]. Instantiating generic routines for various data types is thus a form of operator overloading - each data type is allowed to have a different (customized) implementation of the generic routine, each one with the same name. There is a fine distinction here though, as generic routines are merely one way of defining polymorphic routines. A routine with the same name as an existing one (which could be generic) could also be defined for a specific data type from "scratch".

The concept of an array (or list or queue, etc.) can also be thought of as generic in that it is possible to have an array of integers or reals or characters (or any other data type). Several OOP languages support this form of genericity. C++ [Stro86, WP88], Objective-C [Cox86], and Eiffel [Meye87, Meye88a], for example, all support some form of a generic class or collection to handle this form of genericity.

One of the features of CLOS [BDGK88, Keen89, Moon89] is the concept of *generic functions*. This is another source of confusion in that these

functions are not generic in the sense that they can be defined and then instantiated for different classes. Rather, declaring a function to be generic causes a special function to be created (or modified, if it already exists) to allow the system to select the appropriate implementation depending upon the arguments supplied to it (i.e., CLOS is building a "giant" case statement for the user). As this appears to be more a form of function overloading than genericity, perhaps a better name would have been 'polymorphic functions'.⁶

Another questionable use of the term generic occurs in some object-oriented database systems. Attention must be paid to version control of objects (i.e., which is the most recent "official" version of the object, and what are the various working versions of the object). The terms generic object [KBCG89] and generic instance [FACC89] have been used to indicate the most recent version of the object and what the next working version will be. However, it seems as though it might be more appropriate to refer to these as 'version-control' objects.

6. OBJECT-BASED PROGRAMMING

Object-based programming can be defined simply as computing with objects. It is often confused with object-oriented programming. Support of objects is a necessary in an object-oriented language, but alone is not sufficient to designate a language as being object-oriented (classes and inheritance are also required in object-oriented languages) [Wegn87].

Any object-oriented language is therefore also an object-based language, but the reverse is not true. Languages such as Ada [Booc87] and Modula [Wirt82] are object-based, but are not object-oriented. Actor languages [Agha86, TS89, Wegn87] are object-based, but are not necessarily object-oriented; the Actor model does not specify that either classes or inheritance are necessary to define the objects, although these features may be included in an Actor language.

7. OBJECT-ORIENTED DESIGN

Software design varies with the way in which the problem being implemented is decomposed. Two of the most common methods are (1) a procedure oriented view in which the problem is decomposed according to the actions to be performed; and, (2) a data centered view in which the problem is decomposed according to the entities involved [Cox86]. Object-oriented design [Cox86, Meye88b, SS86, WW89] is when software is designed using objects, in which both data and procedures are treated together.

This is another area which is often confused with object-oriented programming. However, an object-oriented design could be implemented

⁶A generic function has been defined as one that can work for arguments of many types, generally doing the same kind of work independently of the argument type [CW85]. This may have been the intention of generic functions in CLOS, but there is nothing present in the language to enforce the idea that they do the same kind of work.

in a conventional language (although the implementation would probably be more straightforward in an object-based or object-oriented language). And just because software is implemented in an object-oriented language does not necessarily imply that object-oriented design was employed.

8. CONCLUSIONS

This paper presents a relatively simple definition and model as the basis for the underlying theory of object-oriented programming. Although simple, it encompasses the essence of OOP, making it relatively easy for the newcomer to understand what OOP is all about. It also assails the confusing array of terminology currently in use within the OOP community and makes several recommendations as to what the vocabulary should be.

The basic concepts of OOP have been presented in a language-independent manner. It is hoped that this has given the newcomer the basic knowledge necessary to be able to understand the underlying philosophy of any particular language, and to aid in the rapid transition between the various OOP systems that are available.

REFERENCES

- [Agha86] G. Agha. *Actors: A model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, Mass, 1986.
- [AH87] T. Andrews and C. Harris. "Combining Language and Database Advances in an Object-Oriented Development Environment", *OOPSLA'87 Proceedings*, Oct 1987, Orlando, FL; Special issue of *SIGPLAN Notices*, Vol 22, No 12, Dec 1987, pp 430-440.
- [BDGK88] D.G. Bobrow, L.G. DeMichiel, R.P. Gabriel, S.E. Keene, G. Kiczales, and D.A. Moon. "Common Lisp Object System Specification X3J13 Document 88-002R", special issue of *SIGPLAN Notices*, Vol 23, Sep 1988.
- [Booc87] G. Booch. *Software Engineering With Ada, Second Edition*. Benjamin/Cummings Publishing Co, Menlo Park, CA, 1987.
- [Borl89] Borland International, Inc. *Turbo Pascal 5.5 Object-Oriented Programming Guide*. Borland International, Inc, Scotts Valley, CA, 1989.
- [CSWZ87] M. Caruso, R. Strong, M. Williams, S. Zdonik, and M. Nastos. *Object-Oriented Database Systems*, *OOPSLA'87 Tutorial*, Oct 1987, Orlando, FL.
- [Cox86] B.J. Cox. *Object-Oriented Programming: An Evolutionary Approach*. Addison-Wesley Publishing Co, Reading, Mass, 1986.
- [CW85] L. Cardelli and P. Wegner. "On Understanding Types, Data Abstraction, and Polymorphism", *Computing Surveys*, Vol 17, No 4, Dec 1985, pp 471-522.
- [Digi86] Digitalk, Inc. *Smalltalk/V: Tutorial and Programming Handbook*. Digitalk, Inc, Los Angeles, CA, 1986.
- [Digi88] Digitalk, Inc. *Smalltalk/V 286: Tutorial and Programming Handbook*. Digitalk, Inc, Los Angeles, CA, 1988.
- [FACC89] D.H. Fishman, J. Annevelink, E. Chow, T. Connors, J.W. Davis, W. Hasan, C.G. Hoch, W. Kent, S. Leichner, P. Lyngbaek, B. Mahbod, M.A. Neimat, T. Risch, M.C. Shan, and W.K. Wilkinson. "Overview of the Iris DBMS", in [KL89], pp 219-250.
- [GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley Publishing Co, Reading, Mass, 1983.
- [Grog78] P. Grogono. *Programming in PASCAL*. Addison-Wesley Publishing Co, Reading, Mass, 1978.
- [Hend86] J. Hendler. "Enhancement for Multiple-Inheritance", *SIGPLAN Notices*, Vol 21, No 10, Oct 1986, pp 98-106.
- [HO87] D.C. Halbert and P.D. O'Brien. "Using Types and Inheritance in Object-Oriented Programming", *IEEE Software*, Vol 4, No 5, Sep 1987, pp 71-79.
- [KBCG89] W. Kim, N. Ballou, H-T. Chou, J.F. Garza, and D. Woelk. "Features of the ORION Object-Oriented Database System", in [KL89], pp 251-282.
- [Keen89] S.E. Keene. *Object-Oriented Programming in Common Lisp*. Addison-Wesley Publishing Co, Reading, Mass, 1989.
- [King89] R. King. "My Cat is Object-Oriented", in [KL89], pp 23-30.
- [KL89] W. Kim and F.H. Lochovsky (eds). *Object-Oriented Concepts, Databases, and Applications*. ACM Press/Addison-Wesley Publishing Co, Reading, Mass, 1989.

- [KG77] A. Kay and A. Goldberg. "Personal Dynamic Media", *Computer*, Vol 10, No 3, Mar 1977, pp 31-41.
- [KR78] B.W. Kernighan and D.M. Ritchie. *The C Programming Language*. Prentice-Hall, Inc, Englewood Cliffs, NJ, 1978.
- [Lieb86] H. Lieberman. "Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems", *OOPSLA'86 Proceedings*, Sept-Oct 1986, Portland, OR; special issue of *SIGPLAN Notices*, Vol 21, No 11, Nov 1986, pp 214-223.
- [Meyer87] B. Meyer. "Genericity, Inheritance, and Type Checking", Technical Report TR-EI-8/GH (version 2.1), Interactive Software Engineering, Inc. Jan 1987 (revised version of "Genericity versus Inheritance", *OOPSLA'86 Proceedings*, Sept-Oct 1986, Portland, OR; special issue of *SIGPLAN Notices*, Vol 21, No 11, Nov 1986, pp 391-405).
- [Meyer88a] B. Meyer. "Eiffel: An Introduction", Technical Report TR-EI-3/GI (version 2.1), Interactive Software Engineering, Inc. Jun 1988.
- [Meyer88b] B. Meyer. *Object-oriented Software Construction*. Prentice Hall, Inc, Englewood Cliffs, NJ, 1988.
- [Meyer88c] B. Meyer. "Programming as Contracting", Technical Report TR-EI-12/CO (version 2), Interactive Software Engineering, Inc. Mar 1988.
- [Mica88] J. Micallef. "Encapsulation, Reusability and Extensibility in Object-Oriented Programming Languages", *Journal of Object-Oriented Programming*, Vol 1, No 1, Apr/May 1988, pp 12-35.
- [Moon89] D.A. Moon. "The Common Lisp Object-Oriented Programming Language", in [KL89], pp 49-78.
- [Nels88] M.L. Nelson. "A Relational Object-Oriented Management System and An Encapsulated Object-Oriented Programming System", Ph.D. Dissertation, Department of Computer Science, University of Central Florida, Orlando, FL, Dec 1988.
- [Nels90a] M.L. Nelson. "Concurrent and Distributed Object-Oriented Programming Systems", Naval Postgraduate School, Monterey, CA, Technical Report No NPS52-90-026, May 1990.
- [Nels90b] M.L. Nelson. "Object-Oriented Database Management Systems", Naval Postgraduate School, Monterey, CA, Technical Report No NPS52-90-025, May 1990.
- [OFS84] J. Ong, D. Fogg, and M. Stonebraker. "Implementation of Data Abstraction in the Relational Database System INGRES", *SIGMOD Record*, Vol 14, No 1, Mar 1984.
- [Pete87a] G.E. Peterson (ed). *Tutorial: Object-Oriented Computing, Volume 1: Concepts*. Computer Society Press of the IEEE, Washington, D.C., 1987.
- [Pete87b] G.E. Peterson (ed). *Tutorial: Object-Oriented Computing, Volume 2: Implementations*. Computer Society Press of the IEEE, Washington, D.C., 1987.
- [PW88] L.J. Pinson and R.S. Wiener. *An Introduction to Object-Oriented Programming and Smalltalk*. Addison-Wesley Publishing Co, Reading, Mass, 1988.
- [Rent82] T. Rentsch. "Object-Oriented Programming", *SIGPLAN Notices*, Vol 17, No 9, Sept 1982, pp 51-57.
- [SB86] M. Stefik and D.G. Bobrow. "Object-Oriented Programming: Themes and Variations", *The AI Magazine*, Vol 6, No 4, Winter 1986, pp 40-62.

[SLU89] L.A. Stein, H. Lieberman, and D. Ungar. "A Shared View of Sharing: The Treaty of Orlando", in [KL89], pp 31-48.

[Snyd86] A. Snyder. "Encapsulation and Inheritance in Object-Oriented Programming Languages", *OOPSLA'86 Proceedings*, Sept-Oct 1986, Portland, OR; special issue of *SIGPLAN Notices*, Vol 21, No 11, Nov 1986, pp 38-45.

[SS86] E. Seidewitz and M. Stark. "Towards a General Object-Oriented Software Development Methodology", *Proceedings of the First International Conference on Ada Programming Language Applications for the NASA Space Station*, 1986 (reprinted in [Pete87a]).

[Stee84] G.L. Steel, Jr. *Common Lisp: The Language*. Digital Press, Digital Equipment Corporation, Bedford, Mass, 1984.

[Stei87] L.A. Stein. "Delegation is Inheritance", *OOPSLA'87 Proceedings*, Oct 1987, Orlando, FL; special issue of *SIGPLAN Notices*, Vol 22, No 12, Dec 1987, pp 138-146.

[Stro86] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Co, Reading, Mass, 1986.

[Stro88] B. Stroustrup. "What is Object-Oriented Programming?", *IEEE Software*, Vol 5, No 3, May 1988, pp 10-20.

[SW87] B. Shriver and P. Wegner (eds). *Research Directions in Object-Oriented Programming*. MIT Press, Cambridge, Mass, 1987.

[TS89] C. Tomlinson and M. Scheevel. "Concurrent Object-Oriented Programming Languages", in [KL89], pp 79-126.

[Ullm82] J.D. Ullman. *Principles of Database Systems*. Computer Science Press, Rockville, MD, 1982.

[Wegn86] P. Wegner. "Classification in Object-Oriented Systems", *SIGPLAN Notices*, Vol 21, No 10, Oct 1986, pp 173-182.

[Wegn87] P. Wegner. "Dimensions of Object-Based Language Design", *OOPSLA'87 Proceedings*, Oct 1987, Orlando, FL; special issue of *SIGPLAN Notices*, Vol 22, No 12, Dec 1987, pp 168-182.

[Wirt82] N. Wirth. *Programming in Modula 2*. Springer-Verlag, New York, NY, 1982.

[WP88] R.S. Wiener and L.J. Pinson. *An Introduction to Object-Oriented Programming and C++*. Addison-Wesley Publishing Co, Reading, Mass, 1988.

[WW89] R. Wirfs-Brock and B. Wilkerson. "Object-Oriented Design: A Responsibility Driven Approach", *OOPSLA'89 Proceedings*, Oct 1989, New Orleans, Louisiana; special issue of *SIGPLAN Notices*, Vol 24, No 10, Oct 1989, pp 71-75.