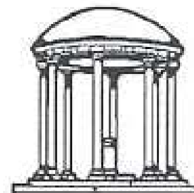
Open Issues in Object-Oriented Programming: Learning Methods, Object Decomposition and Inheritance and Reuse

TR93-016

John Hilgedick

The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175
919-962-1792
jbs@cs.unc.edu



A TextLab/Collaboratory Report

Portions of this work were supported by the National Science Foundation (Grant #IRI-9015443 and by IBM Corporation (SUR Agreement #866).

UNC is an Equal Opportunity/Affirmative Action Institution.

JOHN MATTHEW HILGEDICK. Open Issues in Object-Oriented Programming: Learning Methods, Object Decomposition, and Inheritance and Reuse (Under the direction of Dr. John B. Smith)

ABSTRACT

This thesis presents a survey of the literature of three open issues in object-oriented programming:

1. Learning Methods. The difficulties that beginners face in object-oriented programming are summarized. Techniques for surmounting these problems are presented.
2. Object Decomposition. Decomposition techniques for abstracting and representing a problem domain are placed within four categories according to their method of object formation. The effectiveness of the techniques are compared and contrasted.
3. Inheritance and Reuse. Forms of reuse are identified. Object-oriented mechanisms for obtaining code reuse are presented. Inheritance is shown to be difficult to use effectively. Advantages and disadvantages of single and multiple inheritance are discussed.

ACKNOWLEDGMENTS

I wish to thank a number of individuals who have given their time, effort, and support in making this thesis possible.

I would like to thank Dr. John B. Smith for his guidance in such a difficult task and for his years of support during my graduate career.

I wish to acknowledge the helpful criticism of Dr. James M. Coggins, especially regarding the material on inheritance and reuse.

I would also like to thank Dr. Susanna Schwab for her support and comments in the preparation of this document.

I gratefully acknowledge my parents, Nancy, John, Don, and Elaine, for their encouragement and support of my many quests.

Most importantly, I wish to thank my best friend and wife, Elizabeth, for her years of support, kindness, devotion, and tenacity (as well as her wonderful editing job). I'm a better person because of her.

TABLE OF CONTENTS

List of Figures vi

Introduction 1
A Brief History of Object-Oriented Programming 2
Vocabulary 3

Chapter 1: Learning Methods 6
Relinquishing Process-Oriented Thinking 6
Confusion Between Objects and Classes 8
Encapsulation 8
Message Passing 11
Object Decomposition 11
Inheritance 14

Chapter 2: Object Decomposition 17
Definitions 18
Guidelines for the Evaluation and Development of Object Decomposition 18
Evaluation of Object Decomposition 18
Development of Good Object Decomposition 19
Methods Available for Object Decomposition 20
Guideline Methods 20
Object Oriented Analysis 20
Information Modeling 23
Text Analysis Methods 25
Responsibility-Driven Methods 27
Responsibility-Driven Design 27
CRC Cards 29
Play Acting 30
Structured Analysis Extraction Methods 31
Object Oriented Requirements Specification 31
Abstraction Analysis 33
Object Oriented Structured Design 33
Modified Structured versus Object-Oriented Analysis and Design 35

Chapter 3: Inheritance and Reuse 38
Reuse 38
Reuse of Code 39
Object-Oriented Mechanisms for Code Reuse 39
Delegation 39
Genericity 40
Conformance and Enhancement 40
Inheritance 40
Why Inheritance is Difficult 41
Inadequate Support for Inheritance by Object Decomposition Methods 41
The Conflict Between Inheritance for Definition and Inheritance for Reuse 42
The Conflict Between Inheritance and Data Hiding 43
The Conflict Between Generalization and Specialization 44
Issues Orthogonal to Object Oriented Programming 45
Description of Inheritance Hierarchies 46
Advantages and Disadvantages of Inheritance Hierarchy Types 46

Single Inheritance/Single Tree 46
Single Inheritance/Multiple Tree 48
Multiple Inheritance 49

Conclusion 53
The Future of Structured and Object-Oriented Programming 54

Bibliography 56

LIST OF FIGURES

Figure 1.1: Part-of versus Is-a Relationships 14
Figure 2.1: Object Oriented Analysis Diagrams 21
Figure 2.2: Information Structure Design 25
Figure 2.3: Overview Information Structure Design 25
Figure 2.4: CRC Cards 29
Figure 2.5: Object Oriented Structured Design 34
Figure 3.1: Types of Inheritance 47
Figure 3.2: Multiple Inheritance 50

INTRODUCTION

With the advent of object-oriented languages has come the need to teach or retrain those who hope to use them. There is much disagreement over how this should be done. Additionally, most techniques are anecdotal in nature and their effectiveness is unknown. Identifying the problems that novices encounter is necessary if effective and formal strategies for teaching the principles of object-oriented programming are to be developed. Chapter One summarizes the most difficult problems beginners face in object-oriented programming and reviews techniques for surmounting these difficulties. It also provides a time frame for employing the techniques.

Chapter Two examines the task of object decomposition. Object decomposition is the process of examining a problem domain, abstracting the entities that interact within that domain, and determining how best to encode those abstractions. The considerable advantages of high quality object decomposition warrant an in-depth look at this task. A number of techniques that have been used with a higher rate of success are presented and contrasted. The decomposition techniques are shown to be lacking in formality of transformation and expression.

Chapter Three examines different types of reuse and the object-oriented mechanisms for obtaining code reuse and then examines inheritance in greater detail. Inheritance is shown to be difficult to use because of a number of conflicts, including the conflict between inheritance and encapsulation, between inheriting for definition and reuse, and between specialization and generalization. The chapter closes with an examination of the advantages and disadvantages of single and multiple inheritance.

The conclusion summarizes issues of particular importance to the future of object-oriented programming. These issues demand further attention (and resolution) if object-oriented programming is to deliver on the promises that its proponents have made. It closes with a brief view of what the future may hold for both structured and object-oriented programming.

In the remainder of this section, a brief history of object-oriented programming is presented to provide context for discussion of these issues. Additionally, key terms used in the discussion are defined.

A Brief History of Object-Oriented Programming

During the 1940s and early 1950s, all programming was performed through the use of machine codes. Typically, coding involved the physical setting of wires or switches. These "first generation" machine codes forced the user to think about how the computer performed operations. In the mid-1950s, a number of "second generation" assembly codes began to appear. Through the assignment of mnemonic names to operations, assembly codes helped free the user from thinking solely in terms of wires and switches. In a real sense, assembly codes marked the beginning of a transition from the use of machine language to the development of programming languages with increasing resemblance to human communication.

Although assembly codes offered some freedom for the user, an extensive knowledge of the computer architecture was still required. This prompted the development of "third generation" or "high-level" programming languages. Although these languages also placed emphasis on procedure and little on data structure, languages like FORTRAN, Lisp, Algol, and Cobol allowed the user to think less about how computers performed their operations and more about the operations that were required. Cobol contributed the concept of the record data structure. This allowed the user to define their own data types to augment those predefined in a language. User-defined data types gave greater emphasis to the description of the data used by the processes. It also provided a more effective way to group related data, encouraging modularity and data hiding. This separation of the description of data from the program statements was the beginning of what was to be later known as "data-oriented" programming.

The 1960s and 1970s saw the development of a flurry of specialty languages. A major goal of the developers of these languages was to allow users to think more about what they wanted the computer to do rather than how the computer carried out its operations. These languages are typically referred to as "fourth generation" or declarative languages. One such language, Simula, was designed and implemented in 1962 by Kristen Nygaard and Ole-Johan Dahl [Dahl70]. Simula allowed the user to simulate a system by modeling it as a set of interactive processes, where each process modeled an entity within the real system. In 1967, Simula67 introduced the concept of a class as a method of describing a group of similar entity abstractions. Simula and Simula67 further encouraged modularity by directly supporting the grouping of related data and procedures. During this same period, languages such as Ada and Modula-2 placed greater emphasis on the support of modules. The need to support modularity grew out of a realization that modules generally facilitated the decomposition of a

problem domain into a set of independent sub-components, making programs easier to understand and maintain. Modules contributed heavily to the creation of abstract data types.

The first "object-oriented" language was developed in 1972 by Daniel Ingalls. Ingalls rewrote the software component known as Smalltalk for Alan Kay's Dynabook [Goldberg83]. The language was called "object-oriented" because it was the first system to be completely composed of abstract data types instantiated as objects. The influence of Smalltalk on the rest of the object-oriented world cannot be underestimated. The late 1970s, 1980s, and early 1990s has seen a barrage of object-oriented languages. Flavors [Cannon80, Moon86], Objective-C [Cox84], C++ [Stroustrup84], Eiffel [Meyer88], Modula-3 [Cardelli92], and Actor [Borkoles90] are just a few of them.

With these languages has come a variety of mechanisms to support the definition and creation of abstract data types. The most common mechanism is inheritance. The ability to define abstract data types by reusing those previously defined has introduced a number of difficult and unresolved issues.

For more information, see [Friedman91, Stroustrup88, Shaw84]

Vocabulary

Many of the technical terms found in the object-oriented programming literature are used interchangeably, differently, or erroneously by many authors. This is one of the first difficulties encountered by novices when attempting to learn the principles of object-oriented programming. This section defines key terms used throughout the remainder of this paper.

The term process-oriented programming refers to the style of programming in which the processes that make up a problem domain are the primary components to be abstracted. Additionally, the term is frequently used to describe the actual encoding of process abstractions; it is often used interchangeably (and incorrectly) with structured programming. Structured programming refers to the practice of coding procedures as block-like structures of programming statements. These statements are executed in sequence, with conditional constructs, and without goto statements. Object oriented programming refers to the style of programming where the data and their associated functions are the primary entities to be abstracted. This term is often used to describe the phase of programming in which the actual encoding of entity abstractions is performed.

The process of programming is often divided up into the three somewhat distinct phases of analysis, design, and implementation. Analysis involves examining the problem and its domain and creating a set of abstractions which adequately represent the problem domain. In process-oriented analysis, emphasis is placed on the processes that comprise the problem domain. In object-oriented analysis, the entities that exist, interact, and relate within the domain are emphasized.

The second phase of programming is design. In design, the set of abstractions that were extracted during the analysis phase are refined into a new set of entities which are supported by a programming language. Process oriented design refers to the refinement of process abstractions so that they are supported by a procedural language. Object oriented design refers to the refinement of entity abstractions so that they are supported by an object-oriented language. The distinction between analysis and design can be somewhat diffuse and both are frequently referred to as decomposition. Process decomposition refers to process-oriented analysis and design while object decomposition refers to object-oriented analysis and design.

The third phase of programming is implementation. This refers to encoding the process or entity abstractions produced through analysis and design, in a given programming language.

An entity is anything in the real world that has both state and behavior. Examples of entities include people, automobiles, checking accounts, and the weather. Upon analysis and design, an entity is mapped to a class. A class is an encapsulation, or description of a set of entity abstractions. A class provides a uniform set of instance variables and methods, including a description of how to create new instances of the class. An instance or object is a single instantiation of a class. For a given class, there may be a large number of instances. The terms object and instance are equivalent. Although an object is an instance of a class, the term object is often mistakenly used interchangeably with class in much of the literature.

Instance variables, also known as data members or attributes, embody the state of an object. These variables are common to all instances of a given class. For example, the class House might have the attribute NumberOfRooms. Each instance of class House would have the instance variable NumberOfRooms and its own private value for that variable. Methods, also known as member functions or services, define the behavior of an object. Methods implement the behaviors common to all instances of a given class. For example, the class Frog might have the method Jump. Each instance of class Frog would share the common behavior defined by the method Jump. In order for a method to be invoked, an object must receive a message. Each method

defined for a class is associated with a corresponding message. When an object receives a message, it invokes the corresponding method. A message is similar to a procedure invocation. All interaction between objects occurs through the passing of messages and the subsequent invocation of methods.

The most important contributions of object-oriented programming are the support of encapsulation and inheritance. The principle of encapsulation requires that each component of a program must either embody a problem domain entity or hide a single design decision. The unit of encapsulation for most object-oriented languages is the class. As such, a class encapsulates all state information and operations necessary for instances of that class. The interface for each class is defined to reveal as little as possible about its inner workings [Oxford86]; this is also known as data hiding.

Instances of different classes may respond differently to identical messages. This characteristic is known as polymorphism. For example, instances of class Car, Frog, and Planet might all respond to the message Move but in different ways because each type of object associates a different method for that message. This is necessary because the three objects move differently. Inheritance is a mechanism through which polymorphism can be achieved. It also simplifies the definition of classes similar to one or more classes previously defined by "reusing" those class definitions and augmenting or modifying them as desired. The term reuse is frequently associated with inheritance since one can view the effect of inheritance as reusing the definition of a class for the definition of others.

The definition of these concepts are fairly straightforward, but learning to use them effectively is a challenge.

Chapter One:

LEARNING METHODS

Novices encounter many difficulties when trying to learn object-oriented programming. Some difficulties are conceptual such as relinquishing process-oriented thinking and distinguishing between objects and classes. Other conceptual difficulties are related to encapsulation, message passing, object decomposition, and class relationships. Additionally, there are technical difficulties associated with many of these concepts. Moreover, many of the concepts that underlie these difficulties are related, making it difficult to understand one concept without understanding a number of them.

Identifying these difficulties is necessary if effective techniques for learning object oriented programming are to be developed. Currently, the techniques for overcoming these difficulties are anecdotal in nature and formal study on their efficacy is lacking. The sections that follow each describe one of the difficulties encountered by novices. Techniques for solving these problems and a time frame for employing them are presented from a survey of the literature.

Relinquishing Process-Oriented Thinking

Description. Perhaps the most difficult problem that novice programmers face is shifting focus from a process-oriented to an "entity" oriented approach. There is disagreement regarding the relationship between object-oriented and process-oriented programming. Some view object-oriented programming as a revolutionary advance requiring a complete mental retraining [Beck89, Gibson91], while others view it as a natural evolution or extension of process-oriented programming requiring only minor technical adjustments [Wybolt90, DeNatale90, Moo91]. Despite these differences, it is clear that object-oriented programming requires a different focus than process-oriented programming.

In process-oriented programming, the primary focus is on the processes which comprise a problem domain. Passive data are passed back and forth by processes that manipulate them. In order to manipulate data properly, processes must know the

representation of the data. Processes may interact with many types of data, therefore a great deal of information on the representation of those data may need to be available to a process, and thus available to the programmer of the process.

In object-oriented programming, greater emphasis is placed on the data. Instead of directly manipulating the data, processes send requests to the data objects which then, conceptually, "manipulate themselves". A class, therefore, includes a user-defined data type and the procedures that operate upon it. Thus, users of a class need only require knowledge about the messages an object can receive and need never know the details of the data representation.

The idea that "objects" know how to manipulate themselves and actively interact with other objects is confusing to many programmers. Some have labeled this problem "relinquishing global control thinking" and believe that it is one of the first and most difficult concepts to be learned [Beck89]. Others have argued that it is so difficult to overcome, that individuals with little or no programming experience have less difficulty in overcoming this problem than more experienced programmers because they have not yet become indoctrinated in the process-oriented approach [McKenna88]. There is, however, only anecdotal evidence to support this claim.

Potential Solutions. To overcome process-oriented thinking, many experts advise a submersion technique [Beck89, Gibson91]. The novice is introduced to object-oriented programming concepts through a purely object-oriented language like Smalltalk and allowed to struggle until the concepts are absorbed. It is virtually impossible to do anything in Smalltalk until some understanding is reached of many of the basic concepts of object-oriented programming.

Others argue that this sink-or-swim technique is both unnecessary and impractical. Many companies that employ large numbers of programmers often find it more effective to introduce object-oriented concepts in a more evolutionary, rather than revolutionary, manner [Wybolt90, DeNatale90, Moo91]. Novices are introduced to object-oriented concepts through a hybrid language such as C++ that supports both process-oriented and object-oriented programming. This allows them to gradually adopt the new concepts of object-oriented programming. Purists argue that this approach is not as effective in transmitting the necessary understanding of basic concepts simply because the user is not forced to confront the basic issues of object-oriented programming. Because hybrid languages allow the user to continue to program using structured programming techniques, it is argued that novices have no reason, or find it difficult, to come to grips with the concepts of object-oriented programming [Henderson91, Gibson91, Beck88].

Still others assert that the best way to learn object-oriented programming is not through the use of a language at all. They claim that any language, pure or hybrid, colors the novice's thinking about what object-oriented programming is and how it can best be done. They argue that object-oriented programming is not defined by a particular language but is a way of examining a problem. They concluded that language independent settings are best [OShea86, Lieberherr89].

Time Frame for Learning. Most experienced trainers find that perceiving the difference between processes and entities in a problem domain, and comprehending the idea of objects manipulating themselves can take from a few weeks to a few months [Meyer93, Whiting93]. Of the techniques described above, none has been shown to be superior in either regard.

Confusion Between Objects and Classes

Description. Class and object are two distinct concepts. Novices often treat class and object as interchangeable concepts. In fact, the literature often uses the words erroneously. This confusion can result in difficulties understanding other concepts of object-oriented programming.

Potential Solutions. When introducing the concepts of classes and objects, it is often helpful with experienced programmers to use the analogy of types and variables. The type of a variable provides information to both the user and the compiler about the structure and behavior of the variable. For a given type, there may be a number of instances (variables) of that type, each following the rules for that type. In object-oriented programming, a class is used to describe the structure and behavior of instances (objects) of that class. For a given class, there may be a number of instances (objects) of that class. All instances (objects) of a given class have the same attributes and the same methods. A class, therefore, can be viewed as a description of the attributes and methods an instance of that class will have.

Time Frame for Learning. Learning the distinction between class and object is elusive for many. Different trainers indicate that it takes a few moments to a few hours. Meyer believes that much of the popular literature is so confusing that it can take months for a reader to overcome this problem [Meyer93]. Certainly for experienced programmers, the analogy above should shorten the learning period.

Encapsulation

Description. The encapsulation of an entity refers to the combination of state information needed to express the entity with the operations needed to support that

information. The mechanisms for encapsulation are perhaps the most important contribution that object-oriented programming has to offer. Frequently, novices have difficulty understanding what procedures should be combined with data, and determining which classes should contain the data.

One function of encapsulation is to isolate modules that interact with an encapsulated object from obtaining knowledge about its implementation. If knowledge about an implementation is gained and relied upon by other modules and that implementation changes, then that knowledge becomes invalid and errors result.

When attempting to encapsulate an entity, novices often make three errors. Frequently, there is a failure to properly encapsulate all of the necessary components of an entity, giving access to inappropriate information. These components may exist at varying levels of abstraction. A component may be of a lower level and involve instance variables or methods defined for the internal maintenance of an object. For example, a user attempting to encapsulate a stack might implement the stack as an array, and provide a method for updating the stack pointer. The manipulation of a stack pointer is the sole responsibility of the stack and no other class should be given access to such a method. If such access is allowed, other modules and objects may directly manipulate the stack pointer and compromise the integrity of the stack.

Other design decisions that can be encapsulated improperly are of a higher level. A user might encapsulate an entity and improperly disclose the "philosophy" of the object. Such disclosure should always be explicit. Had the user wished to disclose this information, it might be done more properly by incorporating it into the class name or by providing a method which returns the desired information. For example, a user wishing to encapsulate a SortedCollection may improperly disclose that the class uses a particular algorithm to sort its members. Another class, requiring the same algorithm (possibly because of space or time limitations), might employ the SortedCollection to sort its members. Because there is no explicit indication of the algorithm used by SortedCollection, other classes may behave unpredictably if the algorithm is changed at a later time.

A third problem is that novices are often unsure how general or specific to make an encapsulation. Generalization and specialization are competing goals and finding a balance is often very difficult. Objects should be general enough to be reused in future projects, but not so general as to make their intended use incomprehensible or inefficient.

Potential Solutions. Many argue that the presentation of object-oriented programming as an extension or evolution of process-oriented programming makes the

concept of encapsulation much easier to comprehend. Beginning with the principles underlying the concept of a module often provides a good transition point between process-oriented and object-oriented programming. In process-oriented programming, modules are used to model processes, but are frequently used to model entities as well. Modules were developed in an attempt to divide large programs into smaller components. The use of these smaller components offers two distinct advantages, particularly when they abstract single design decisions. First, modules facilitate conceptual understanding. Secondly, modules can be more thorough tested because they have fewer input values than a system that has not been divided into sub-components. This makes the component easier to test for errors and increases reliability [Dijkstra68].

Parnas offers a number of reasons why the abstraction of data representation is superior to the abstraction of processes [Parnas72a]. He argues that data representations are far more likely to change than the processes that act upon them. Thus, each time a data representation changes, every module that interacts with that data must be modified to incorporate the change. Given that data representations change with relative frequency, corresponding procedure modifications would be needed often. The greater the number of modules and processes that interact with the data, the more likely that an update will not be performed properly.

The use of abstracted data representations results in a much different scenario. When a data representation changes, only the module (class) that implements that representation needs to be modified. Modules only have knowledge about the interfaces of other modules, and such knowledge is much less likely to change. This insulates other processes and modules from requiring modification whenever data representation changes.

Parnas proposes that a reasonable course of action is to begin by generating a list of difficult design decisions and design decisions that are likely to change. It is further recommended that each module be designed to hide decisions from the other modules.

Time Frame for Learning. The advantages of encapsulation have been understood by structured programmers for many years. Meyer argues that inexperienced programmers can understand the need for encapsulation within a few minutes [Meyer93]. Many state, however, that both structured programmers and novices will require a number of weeks of experience with a few small projects before learning how to effectively employ the encapsulation mechanisms that object-oriented languages provide [Meyer93, Whiting93].

Message Passing

Description. Objects are encapsulations of entity abstractions and respond to requests made by other objects via message passing. When an object receives a message, it invokes the method associated with that message. These messages are formally specified and comprise the interface to the object. The interface to an object is the only information that other objects should have to know about. As specified in the section entitled Relinquishing Process-Oriented Thinking, this method of communication is often difficult for novices to conceptualize.

A more technical problem that novices often have with message passing is a concern about performance penalties [Wilson88, Parnas72a]. Encapsulation introduces a new level of indirection. Processes can no longer directly manipulate data. Instead, an object must be sent a message to perform the desired operation. Upon receiving such a message, the object may invoke multiple methods upon itself in order to perform the desired action. Concern about performance penalties can make the user grant other objects direct access to instance variables and methods used for internal support. Such access is almost invariably a mistake because it violates encapsulation.

Potential Solutions. The literature provides very little information that addresses the conceptual difficulties of message passing. A set of diagrams, developed by Cunningham and Beck, have been used to teach over 100 students the concept of message passing [Cunningham86]. Cunningham and Beck attribute their success to the omission of "object state" and "message sequence" from their technique. This omission, they argue, facilitates understanding of the relationships between classes as indicated by the messages.

When performance is a primary concern, more efficient languages may be the only solution. This chapter does not compare the efficiency or speed of process and object-oriented languages.

Time Frame for Learning. Both Meyer and Whiting believe that the concept of message passing can be learned within a few minutes to a few hours [Meyer93, Whiting93]. Whiting believes that more experienced structured programmers may require additional time for learning the concept of message passing than their inexperienced counterparts. There is no formal evidence to support this claim.

Object Decomposition

Description. When attempting object decomposition, novices frequently encounter a number of difficulties. Users who have not relinquished process-oriented

thinking, often transform procedural abstractions directly into objects [Henderson91, OShea86]. In other words, an attempt is made to turn a process into an object. Consequently, the processes that act on the data, instead of the data themselves, become the objects. While this is possible, it is almost always a bad idea. Parnas argues that the data representations are most likely to change and should be encapsulated [Parnas72a]. There are certain application areas, however, where processes change at least as often as the data representations on which they act. These include experimental, computational, and utility processes. In these cases, it may be more appropriate to abstract the processes themselves [Coggins93].

Novices may be tempted to use this idea as an excuse to map all procedural abstractions to objects. Given the fundamental importance of overcoming process-oriented thinking, it may be prudent to avoid introducing the concept of procedure abstraction until after object-oriented thinking has been mastered.

When attempting to define a new class, novices have been observed grouping entities by state attributes of lesser importance while ignoring more important attributes or behaviors. Gibson illustrates with the following exercise. On a sheet of paper, list the features of an apple. Upon completing this list, record a second list of features of an imitation apple. Most individuals who complete this exercise, find the two lists are virtually identical except that the second list might include "inedible" or "for display only". Although the roles of two objects may be quite different, novices may group them due to their similar state [Gibson91]. A simpler example is that of a house and a person. Both a house and a person have an address and, therefore, share some common state. Given the degree of dissimilarity between the two objects, it is probably not appropriate to try to represent them in one class. Experts recognize the necessity of examining those qualities that distinguish an entity from others when creating a potential class.

The lack of tools to help programmers perform analysis and design hinders object decomposition. Currently, implementation resources include editors, compilers, linkers, debuggers and class browsers. While these tools are helpful during the implementation phase of a software system, access to tools that help with analysis and design are needed. Such tools could potentially prevent the acceptance of conflicting requirements, ensure the completeness of encapsulation, and verify that design decisions can be implemented with a specified programming language. The availability of such tools is limited.

Novices are usually introduced to programming concepts through a popular language. In the case of object-oriented programming, the language is typically

Smalltalk, C++, or Eiffel, while in the case of process-oriented programming it is usually C, FORTRAN, or BASIC. Novices are rarely introduced to programming through training for process or object-oriented analysis and design [Coggins90a]. Most universities and corporations trust that the user will learn these skills through experience. Gibson argues that a pattern similar to what occurred during the early years of structured programming is reemerging. The novice is first introduced to a language and over time realizes that his or her implementation strategy is flawed as a result of poor design. The novice seeks out and adopts techniques to help with design and, after time, realizes that a poor understanding of the original problem is responsible for design flaws. Gibson believes that this pattern of implementation, design, and analysis is extremely common and is not confined to novices [Gibson91].

Potential Solutions. There are a number of techniques available for performing object-oriented analysis and design. These are described in the following chapter. Only a few of them are identified as being suited for use by novices. One such method has been proposed by Beck and Cunningham [Beck89], who report successfully teaching object-oriented thinking, analysis, and design to over 100 students using CRC cards and an exercise consisting of "what-if" scenarios. Through this exercise, novices are allowed to gradually build on their knowledge of a given problem domain. During this process, analysis and design questions are raised. The instructor provides the students with assistance in these decisions. CRC cards have also been used successfully in industry [Coplien91]. This technique will be discussed more thoroughly in the next chapter.

Another apparently useful technique is that of play acting. In play acting, students "become" objects and attempt to interact with one another until they are able to identify their responsibilities and relationships. This technique seems particularly useful for demonstrating that instances of a class have the same methods and attributes (but possibly different values for those attributes). Additionally, it appears to help students determine what different types of relationships among objects are possible [Pugh88, Whiting90, Rosson89]. This technique will be discussed more thoroughly in the next chapter.

Wilson has successfully utilized a third technique [Wilson88]. The instructor first designs a small program using structured analysis and design methods. Next, the students are guided in rewriting the program using object-oriented analysis and design methods.

Inheritance

Description. Novices often attempt to use inheritance (is-a) relationships when client-of, server-of, or part-of relationships might be more appropriate [Pugh88]. For example, a novice might mistakenly create a class TextEditor by inheriting from class MemoryManager. An instance of class TextEditor might contact an instance of class MemoryManager and request a needed amount of memory. In this example, a TextEditor might require essential services from a MemoryManager, but that doesn't mean that a TextEditor is a MemoryManager so inheritance is not appropriate. Given this context, a client-of relationship is more suitable.

Instances of two different classes may also have a part-of relationship. For example, an instance of class Car might have an attribute (state variable) that is an instance of class Engine. Class Engine should not inherit from class Car because engines do not behave like cars. An Engine is a part of a car and inheritance is not warranted in this case.

The literature appears to be at least partially responsible for the confusion regarding the possible types of relationships among objects. This is particularly true in many articles that attempt to justify the use of multiple inheritance [Meyer88, Booch91a]. One of the more well known examples of this is the food, fruit, spice, apple, cinnamon, apple pie example (figure 1.1).

In this example, the use of multiple inheritance is promoted in an attempt to demonstrate that an apple pie can inherit the qualities of both apple and cinnamon, saving the

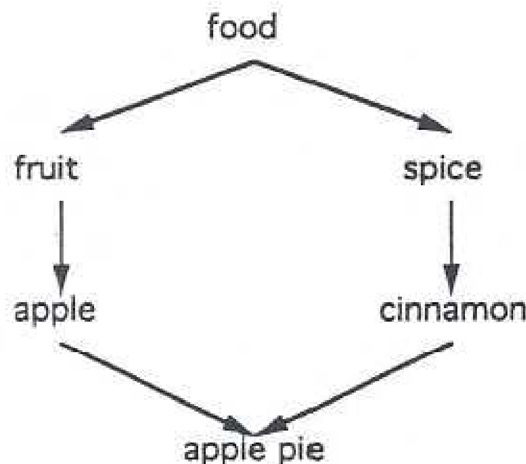


Figure 1.1

Apple Pie

user from creating duplicate code. The problem with this example and with most similar examples is that an apple pie is neither a fruit, an apple, a spice, or cinnamon. An apple pie contains those items but that does not mean it should inherit from them. Certainly, a part-of relationship would be more appropriate.

Once novices become reasonably adept at constructing inheritance relationships, they begin to construct larger, more complex inheritance hierarchies. Since analysis and design are usually iterative, the novice begins to introduce new attributes and methods into the class hierarchy. Because it is often difficult to decide where to place new attributes and methods, novices often place attributes and methods in an inappropriate class within the hierarchy. This can result in poor correspondence between the abstraction and the different levels of the hierarchy [OShea86]. Eventually, an incomprehensible hierarchy is produced. Poor abstractions are often the result of not understanding how general or specific a given class should be [Sandberg86].

A related problem involves the technical difficulties that exist in large and complex hierarchies. Often, hierarchies are so immense that it is difficult for the novice to determine which classes are the most appropriate to use [OShea86, Beck88, Linton87]. Future tools for examining class libraries may help alleviate this problem [Fontana91, deChampeaux91, OShea86]. Although Meyer asserts that this complaint is exaggerated [Meyer90], most would agree that it is important to understand the layers of abstraction in a library before attempting to use it, and that such tools are necessary to help understand the abstractions a library encapsulates [Beck88].

An issue related more to reuse than inheritance is the reluctance by both novices and experts to reuse code written by other individuals. This resistance to "outside" code is likely the result of many programmers' desire to control the software product as much as possible [McKenna88, Fontana91, Griss91]. This problem is perpetuated by many managers who will not purchase software from outside sources because they believe it is easier and/or cheaper to write in-house. Most do not realize how difficult it is to develop truly good software [Griss91]. Fear may be the primary reason that reuse is not done on a larger scale.

Potential Solutions. Techniques for learning object-oriented programming are remarkably silent on the subject of inheritance. Shafer claims that design for reuse is difficult, and even more difficult to teach [Pugh88]. This argument is generally supported by the literature. It may be possible to use Wilson's method of teaching by example, to convey the meaning and use of inheritance to novices. This technique,

however, requires the availability of a qualified teacher. Until more techniques are available, novices without teachers will likely resign themselves to the study of existing class hierarchies as a means of learning inheritance. A good set of class browsers or a rich environment such as Smalltalk's would be helpful in this effort.

Time Frame for Learning. Many claim that the concept and purpose of inheritance can be learned in a few hours to a few days [Meyer93]. Others claims that the proper use of inheritance can take months or years of experience. During this time, he states, a user will develop a personal philosophy on how to employ inheritance based on the advantages and disadvantages of the mechanism [Whiting93].

Summary

While there is a great deal of advice on the best ways for beginners to overcome the difficulties of object-oriented programming, there is little formal study. A universally accepted recommendation is that novices should determine how experts learn these concepts and try to mimic them [Gibson91]. The difficulty in applying this "technique" is that experts cannot always explain how they learned the necessary concepts. Often, experts can only describe the products of their learning but not the underlying process of their learning.

Because languages are taught more frequently than programming concepts, there is more data available for gauging the length of time necessary to learn a particular programming language. Many claim that Smalltalk can be learned in a couple of days. Such claims do not take into account the more difficult aspects of how to use encapsulation and inheritance effectively. Some claim that the learning curve for C++ is approximately 6 months while Jossman states that it is one year [Jossman90]. These claims appear to conform with the literature.

Beck concludes that inertia, lack of experience and a lack of teaching tools and methodologies are preventing object-oriented programming from being effectively disseminated [Beck86]. While time will undoubtedly solve the problems of inertia and inexperience, it is more difficult to predict how the lack of tools and methodologies will be resolved, and how their absence will affect the abilities of those using object-oriented languages.

Chapter Two:

OBJECT DECOMPOSITION

Frequently, an attempt to map a problem domain to a set of abstractions is made under the false assumption that the problem domain is understood at a level necessary for implementation. As the implementation progresses, unexpected dependencies in the problem domain are discovered and the original abstractions are modified to handle these "exceptions". Upon completion, the abstractions poorly represent the problem domain. The class library is virtually incomprehensible, cannot be reused, and ultimately is discarded.

Object decomposition is an iterative process. Since refinement of the model of the problem domain is unavoidable, one must expect the abstractions that represent it and the implementation of the abstractions to change. Relationships and interactions evolve in this manner [Fontana91]. If a class library is to withstand the demands of future needs, abstractions must be designed and implemented with change in mind.

The problem of evolving classes and their relationships is further compounded by the lack of easy to use formal analysis and design techniques. Many programmers and managers view programming as an art, and conclude that it is unnecessary to teach the fundamentals of analysis and design to students and employees. Many individuals also assume that other programmers will understand a problem domain as they do and, therefore, decompose a problem domain as they would. Additionally, the same problem domain is frequently broken down into different abstractions by different individuals. As a programmer gains experience with object-oriented programming, the preferred style of decomposition may change. The difference between a poor system and an exceptional one depends upon how the abstractions are developed and related. The process of determining how a set of abstractions relate and interact leads to further refinement of the original set of abstractions.

This chapter presents a set of criteria for guiding both the process of object decomposition and the evaluation of completed decompositions. It then reviews and compares a number of methods for the decomposition of problem domains. It concludes with a brief discussion on the advantages and disadvantages of using

techniques that transform structured analysis and design results to object oriented design specifications.

There are many arguments over the effectiveness of these methods due to their informality and failure to address such basic issues as inheritance and maintenance. Formality can be expressed on two levels. The formality of the heuristic used to decompose a problem domain is important because it can aid users in deriving a superior set of classes with which to represent the problem domain. The formality of the expression of the discovered classes is important in relaying information to clients and in the implementation of analysis and design results. This chapter shows that most object-oriented decomposition methods lack both qualities.

Definitions

Object Decomposition. Object decomposition refers to a three part process. First, the problem and its domain are examined. Next, a set of entities which adequately represent the problem domain are extracted. Finally, the set of extracted entities are refined into a new set of entities which are supported by a programming language.

Object-Oriented Analysis. Object oriented analysis refers to the first two parts of the object decomposition process. No thought is given to the implementation of the abstractions. As stated by Coad and Yourdon, analysis is "the process of extracting the 'needs' of a system - what the system must do to satisfy the client, not how the system will be implemented" [Coad91a].

Object-Oriented Design Object oriented design refers to the third part of the object decomposition process. Coad and Yourdon write, "design is the practice of taking a specification of externally observable behavior and adding details needed for actual computer system implementation" [Coad91b].

Guidelines for the Evaluation and Development of Object Decomposition

Two topics are discussed in this section. First, a number of criteria are identified by which the quality of object decomposition can be evaluated. Next, guidelines for the development of good object decomposition are outlined.

Evaluation of Object Decomposition

There are many characteristics of a good software product. However, object-oriented programming addresses only characteristics related to modularity. The

modularity of a system can be evaluated using five criteria which are as follows: modular decomposability, modular composability, modular understandability, modular continuity, and modular protection [Meyer88].

Modular Decomposability. The decomposition of the problem domain decomposes the problem into several subproblems whose solutions may then be pursued separately (divide and conquer).

Modular Composability. The decomposition of the problem domain produces software elements which may be freely combined to produce new systems, possibly in an environment quite different from the one in which they were initially developed.

Modular Understandability. The decomposition of the problem domain produces software elements which can be understood individually by the user.

Modular Continuity. A small change in a problem specification results in a change of only a few modules.

Modular Protection. The effect of an abnormal condition occurring at run-time in a module will remain confined to that module, or at least will propagate to only a few neighboring modules.

Development of Good Object Decomposition

There are a number of guidelines for the development of good object decomposition. Some characteristics can be checked during the development phase. These characteristics include the following: information hiding, understandability, completeness, few couplings, weak couplings, explicit couplings, sufficiency, and primitiveness.

Information Hiding. All information about a module should be private to the module unless it is specifically needed and declared public [Meyer88, Parnas72a, Stroustrup88].

Understandability. Modules must correspond to syntactic units in the language used [Meyer88].

Completeness. The interface of the module is general enough to be used by any client [Booch91a, Stroustrup88].

Few Couplings. Every module should communicate with as few others as possible [Meyer88].

Weak Couplings. If any two modules communicate at all, they should exchange as little information as possible [Meyer88, Booch91a, Parnas72a].

Explicit Couplings. Whenever two modules A and B communicate, it must be obvious from the text of A and B [Meyer88].

Sufficiency. The module captures enough characteristics of the abstraction to permit meaningful and efficient interaction [Booch91a].

Primitiveness. High level operations may be better accomplished through a set of lower level operations (this must be balanced with efficiency) [Booch91a].

Methods Available for Object Decomposition

There are a variety of methods available to aid the user in object decomposition. Four groups of methods are discussed; guideline methods, text analysis methods, responsibility driven methods, and structured analysis/extraction methods. The first three groups are primarily based on object-oriented techniques, while the last group is composed of modified process-oriented techniques.

Guideline Methods

Guideline methods typically provide a list of the kind of objects the user should look for within the problem domain. In addition, they often provide guidance for testing the validity of the candidate objects. Two examples of guideline methods are object-oriented analysis and information modeling.

Object Oriented Analysis. This method was developed by Coad and Yourdon. It is presented as a technique which builds upon information modeling [Fichman92], with the addition of the concepts of inheritance, methods, and messages [deChampeaux92]. Object decomposition is described in terms of five phases: class & object discovery, structure formation, subject description, attribute formation, and method formation. For class and object discovery, the user is given a set of concepts for which the problem domain must be examined. The types of relationships among classes are determined during the structure formation phase. During the subject description phase, the discovered classes are segregated into groups of related classes. The purpose of this phase is not to determine relationships among classes, but to help the user group potentially large numbers of classes into a smaller set of subjects. This helps to provide an overview of the abstractions of the problem domain. The final two phases involve the identification of attributes and methods, respectively. A different diagram is used to express results at each of the five phases of object decomposition (figure 2.1).

In the first phase, users discover classes within the problem domain by looking for structures, other systems, devices, things or events remembered, roles played, operational procedures, sites, and organizational units. Structures are defined as either

generalization-specialization (is-a) or whole-part (part-of) relationships. Since Coad and Yourdon believe this to be the best area in which to search for potential classes, it is discussed in greater detail in the second of the five phases. Other systems are those entities outside the problem domain that interact with entities within the problem domain. This ambiguous definition of other systems suggests that the technique does not provide much guidance in determining the boundaries of a problem domain.

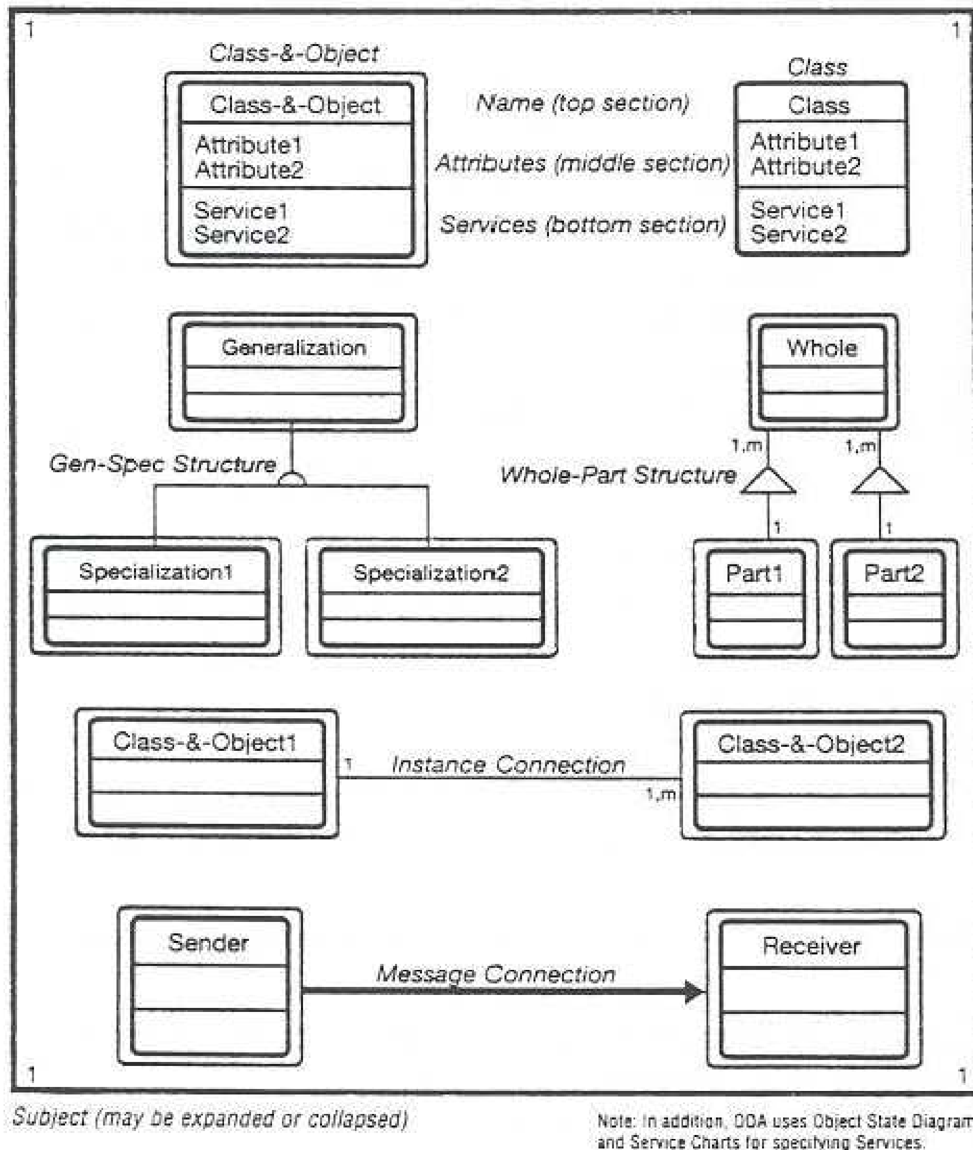


Figure 2.1

Object Oriented Analysis Diagrams [Coad91a]

Devices are defined as any system within the problem domain and are, consequently, poorly defined. Things or events remembered are records of transactions within the system or records of historical events within the system. Roles played are the responsibilities that different components within the domain have. Operational procedures are responsibilities that the examined system must provide over time. Sites are the physical locations of the entities of the problem domain. Finally, organizational units are the structural components that people impose upon themselves and other entities.

The authors provide a list of hints on how to find classes within the problem domain. These include observing the problem domain first hand and actively listening to people who work within it. Interviews can help to determine exactly how the domain functions. Reading about the problem domain is necessary. Checking results obtained by others modeling the same or similar domains is also invaluable; this is also an almost completely ignored form of reuse. Further candidate classes may reveal themselves when the user begins to prototype.

Once a set of candidate classes is obtained, it must be tested for validity. The authors provide the following questions for testing potential classes:

- Do instances of this class need to be referred to over time?
- Does the class provide a behavior necessary to the problem domain and the solution?
- Does the candidate class have multiple attributes?
- Is there more than one instance of the class?
- Are the attributes and methods applicable for all instances of the class?
- Does the candidate class fulfill an actual requirement of the system?

According to this method, a candidate class that passes these tests is probably valid.

In the structure formation phase, the user examines the set of classes for is-a and part-of relationships. The technique does not offer much guidance as to how this task might be accomplished. The user examines each class in turn to determine if one class is a specialization of another. For determining part-of relationships, the authors recommend looking for assembly-part, container-contents, and collection-member relationships between classes. With this information, a set of class hierarchies emerges.

During the subject description phase, the root class of each structure is identified, and each class not in a structure is identified. The identified classes are designated as subjects. Coad and Yourdon describe a subject as "a mechanism for guiding a reader (analyst, problem domain expert, manager, client) through a large, complex model. Subjects are also helpful for organizing work packages on larger projects based upon initial object-oriented analysis investigations" [Coad91a].

During the fourth and fifth phases, attributes and methods are identified. The authors suggest the examination of previous object-oriented analysis results but provide limited guidance in how the validity of an attribute or method may be tested. It is then determined whether instances of a class behave differently according to the value of their attributes. If so, additional methods may be required. Method discovery is accomplished primarily through the determination of the necessary services that a class must provide for both itself and other classes, and the necessary services that the class receives from other classes.

This technique has both strengths and weaknesses. Although the five phases are accomplished primarily through a set of informal hints and tests, it is a well thought-out and relatively coherent technique. Additionally, the object diagrams provided in each of the five phases do lend themselves towards greater formality than most methods. For further information about this technique, the reader is referred to the work of Coad and Yourdon [Coad91a, Coad91b].

Information Modeling. Similar to object-oriented analysis, information modeling as developed by Shlaer and Mellor [Shlaer88] provides a set of concepts which can be used in the examination of a problem domain. It also provides a set of questions for testing the validity of a candidate class and a set of diagrams to aid in the formalization of analysis results. It also attempts to address the issue of inheritance with a set of entity-relationship diagrams. Finally, it provides greater support in attribute discovery.

Information modeling provides several concepts to aid with object formation; these include tangible things, roles, incidents, interactions, and specification objects. Tangible things are capable of being touched. Roles are functions that define a particular object. Incidents are used to represent an occurrence or a specific event. Interactions, or interaction objects, generally have a "transaction" or "contract" quality, and relate objects in the model. Specification objects have the quality of a standard or a definition of other objects.

Upon obtaining a set of candidate classes, a written description of each class is made. The description consists of a name and a short, informative statement. This

statement allows one to determine whether a particular real world entity is an instance of the class as conceptualized. Once descriptions have been written for each class, they are tested for validity using a series of tests. The uniformity test requires each instance of a class to have the same characteristics and to be subject to the same rules. The more-than-a-name test states that if a candidate class has no attributes and no characteristics other than its name, it is probably not a class but an attribute. The OR test states that if the class description uses the word "or" in a significant way, then it is probably not a class but a group of classes. Finally, the more-than-a-list test states that if the class description is merely a list of all the specific instances, it is probably not a class.

Once the user has written descriptions of each candidate class, the process of attribute definition is initiated. The Shlaer and Mellor method of object decomposition is more helpful than the Coad and Yourdon method in the creation of attributes. Unlike many of the methods presented in this chapter, this technique places great emphasis on what should be and should not be an attribute. For example, the authors state that attributes should be mutually independent in order to prevent conflicting internal state values. They should be complete enough to express all possible internal states of the class. They should also capture separate aspects of the object abstraction. Each attribute, like each class, is given a description. This description consists of a short statement of purpose for the attribute and a specification of the domain or range of values that a given attribute might have. The domain of an attribute can be specified by enumeration, citation, acceptance rules, range, "same as", or using a special "see above" explanation.

Once all attribute descriptions have been written, the user begins describing the relationships of the classes to one another through the use of graphic and text diagrams. These diagrams, in order of use, are: Information Structure Design (an entity-relationship diagram with attributes included - figure 2.2), Overview Information Structure Design (another entity-relationship diagram without attributes - figure 2.3), Object Specification Document (all class and attribute descriptions), Relationship Specification Document (indicates all relationships), and Summary Specifications (indicates all attributes, methods, and relationships).

Like Coad and Yourdon, Shlaer and Mellor provide a set of diagrams in an attempt to formalize a description of the resulting class structure, but offer little guidance for the actual determination of class structure. Further, there is little guidance for method discovery.

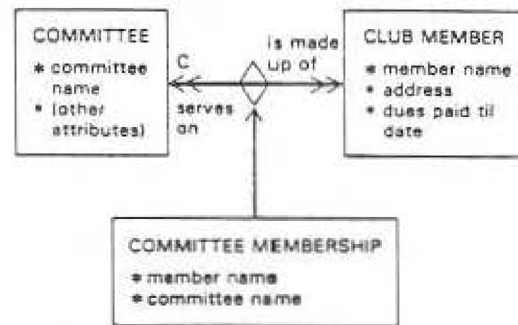


Figure 2.2

Information Structure Design [Shlaer88]

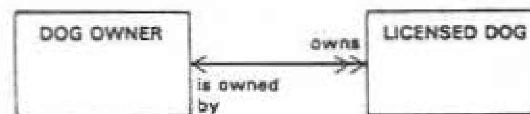


Figure 2.3

Overview Information Structure Design [Shlaer88]

Text Analysis Methods

Unlike the guidance methods presented above, text analysis methods require the user to examine text descriptions of both the problem domain and a solution strategy. Nouns and verbs within the text descriptions are identified and mapped to objects and methods. A number of similar techniques are available. Rather than present each one individually, they are presented as a composite with attributes unique to a particular technique noted.

This method, first proposed by Abbott [Abbott83], requires the user to state the problem in clear, transcribed English. The user then reads the textual description of the problem and extracts candidate objects and methods from the nouns and verbs used in the problem description. This technique was minimally expanded by Booch for Ada users [Booch91a] while Sincovec and Wiener added the use of modular design charts [Sincovec84].

This kind of technique typically consists of three iterative phases. In the first phase, an informal textual description of the problem domain and the solution strategy

is developed. The description of the solution strategy used to solve the problem must use the same terminology as that found in the description of the problem domain. Upon completing the two informal descriptions, the textual description of the solution strategy is formalized. During this second phase, the user identifies data types, objects, operators, and control constructs within the text description. Common nouns become candidate classes. Proper nouns and direct references become candidates for instances of a class. Verbs, predicates, or descriptive expressions become candidates for methods. Adjectives become candidates for attributes. Control structures are often implied by the structure of the English description. During the identification of these elements, the original text descriptions may be refined. The final phase consists of the segregation of the discovered classes, attributes, and methods into two groups: those that formalize the problem domain and those that formalize the solution strategy. The elements placed into the latter group constitute the classes to be implemented.

As Abbott and Booch point out, this is really an iterative process [Booch83]. Neither believes that the process of formalization in the second phase is so mechanical as to make the transformations automatic. As the user formalizes the classes, methods, and attributes discovered through this process, additional items emerge. Often, these newly discovered items have no corresponding noun, verb, or adjective in either the problem description or the solution strategy. Furthermore, it is often difficult to formalize some language beyond a certain point. Abbott concedes that some transformations can be difficult.

An advantage of this technique is that users can determine how formal to make the problem and strategy descriptions. For example, Abbott asks the user to distinguish between common nouns, proper nouns, direct references, mass nouns, and noun phrases. The user is given formal guidance on handling verbs, attributes, predicates, and descriptive expressions. Although this technique is initially intimidating, most users are adequately adept at the identification of nouns, verbs, and adjectives, to use it effectively. Moreover, users can decide the level of formality they wish to use. One potential disadvantage of this flexibility occurs when group members and clients disagree as to how formal to make descriptions and transformations.

Another potential problem with this technique involves the relationship between language and thought. Since language and thought are intertwined, the words chosen to describe a problem domain are often a good way to communicate a problem to others. The words, therefore, can be extremely valuable as class, method, and attribute names. Frequently, however, conceptual problems expressed in words do not closely

parallel the original problem and dependence on a specific description may preclude a superior solution based on a different intuitive foundation.

An additional difficulty with this technique involves the imprecision of human language [Ladden89]. Many of the definitions of common nouns and verbs are difficult to articulate. Additionally, nouns are often applied to concepts that are not fully formed. Nouns are used to express such intangible entities as mass, units of measure, and other elusive concepts. Henderson-Sellers and Constantine [Henderson91] offer the example of TextEditor, a noun whose semantics encompass both an abstract noun and an operation. While techniques are provided for distinguishing between these different types of nouns, confusion about how to handle even the most common cases occurs readily. Ladden argues that this technique "inherently lacks rigor due to the impreciseness of the English language."

Unlike the guideline methods presented in the previous section, this technique provides a more formal method of object discovery. Rather than suggesting a set of concepts by which a problem domain may be examined, a set of formal transformations in finding classes, attributes, and methods through the process of identifying the components of sentences, are provided. This ability makes it noteworthy and it is unfortunate that the entities to which this formal set of transformation are applied are so unavoidably ambiguous. Although Sincovec and Wiener have added the use of diagrams in an effort to further formalize the technique, they do not appear to address the issue of inheritance. This may be due to the authors' emphasis on the Ada programming language.

Responsibility-Driven Methods

This group of methods is the most poorly defined. All of them decompose a problem domain according to the responsibilities of entities within it. Three response-driven methods are discussed below: responsibility-driven design, CRC cards, and play acting.

Responsibility-Driven Design. Of the methods described in this section, Wirfs-Brock's Responsibility-Driven Design is the most formal [Wirfs90a]. In this technique, the problem domain is examined for client/server relationships, and the client and server entities are transformed into objects. It is argued that this approach maximizes information hiding and encapsulation by deferring implementation issues until after a model of classes and their interactions is in place. It is also asserted that by focusing on object behavior, rather than object structure, rework is minimized. The

process is divided into two phases: the exploration phase and the hierarchy construction phase.

The exploration phase involves three steps. First, the user determines an initial set of candidate classes and attributes, by extracting the nouns and adjectives within a text description. Although these elements are found through text analysis, it is not the most noteworthy characteristic of this technique for object discovery. In step two, responsibilities within the system are assigned to classes that contain related information. Discovered responsibilities are evenly distributed across all classes within the system. In step three, the responsibilities for each class are examined, and the classes which are needed for collaboration in order to fulfill each responsibility are determined. A card is used to describe the responsibilities and collaborators for each class. These cards were originally proposed by Cunningham and Beck [Beck89] and are discussed below.

The hierarchy construction phase also involves three steps. First, the methods are placed as high as possible within the hierarchy. "Contracts" are then developed between classes. A contract is an agreement between two classes and specifies what types of requests a client may make and how the server should respond to each request. The concept of a contract has its roots in the client/server model and is helpful in specifying the actions for which an entity is responsible. It is also helpful in determining what information a given entity shares. Secondly, a collaboration graph for the system of classes is created. The collaboration graph is used to identify those groups of classes that interact in a frequent and complex manner. These groups of classes are designated as subsystems. Classes within a subsystem should be strongly interdependent and support a small and strongly cohesive set of responsibilities. The concept of a subsystem is similar to Coad and Yourdon's Subjects concept, and can be helpful in making a system with a large number of classes more comprehensible. The collaboration graph is also used to identify areas where encapsulation is violated and in determining relationships that are unnecessarily complex. Finally, design specifications for classes, subsystems and contracts are written and the signatures for all methods are specified.

A weakness of this method is that the user is provided very little guidance for its use. One potential advantage to this technique is that it may provide for more comprehensible abstractions. Specifically, it emphasizes the dynamic relationships of classes rather than the static relationships that many object decomposition techniques emphasize [Fichman92] and this emphasis may result in more comprehensible abstractions. However, this conclusion is highly speculative.

CRC Cards. Class, Responsibility, and Collaboration (CRC) Cards were developed by Cunningham and Beck [Beck89, Cunningham86] in an attempt to teach the basic concepts of object-oriented analysis and design to novices. This method consists of three steps. First, the user determines a couple of the "obvious" entities of the problem domain using any available method. In the second step, a CRC card is created for each entity by dividing a 4 by 6 inch card into three regions (figure 2.4).

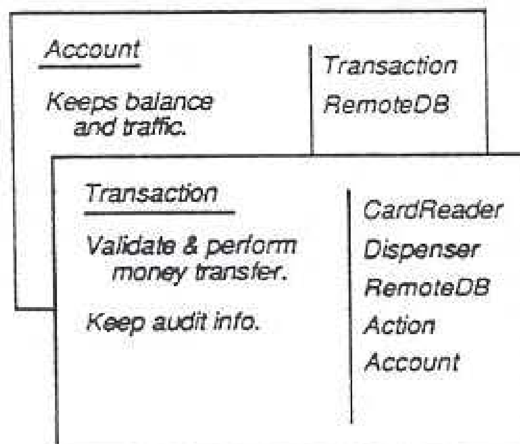


Figure 2.4
CRC Cards [Beck89]

The first region is designated for a class name. The user must determine an appropriate name for the class although the method provides no assistance in this task. The responsibilities of a class, stated in short verb phrases, are listed in the second region. Collaborators are listed in the third region. Collaborators are other classes of objects with which the given class of objects must interact in order to accomplish their responsibilities.

Upon completing the cards, the user presents a number of "what-if" scenarios related to the problem domain. These scenarios involve the classes already created, as represented by their corresponding CRC cards. When a scenario calls for a responsibility that is not covered by one of the classes already created, that responsibility must either be added to one of the existing classes, or a new class, and new CRC card, must be created that can address the new responsibility. In this manner, the decomposition of a problem domain progresses from the knowns to the unknowns.

One of the benefits of this method is that a user can quickly determine whether a class is becoming too "cluttered" with responsibilities and collaborators by examining the lists on a given card. If an object interacts "too much" with other objects, it is probably due to poor design [Beck89]. Excessive numbers of responsibilities may be suggestive of a poorly defined class, in which case it must be broken into components with the responsibilities distributed among them.

Another advantage is that the cards can be held by the user as physical entities. The authors of this method argue that this makes objects more "real" to a novice and facilitates thought about responsibility and collaboration. The authors demonstrate this ability in the following:

> We had videotaped experienced designers working out a problem [..]. Our cameras' placement made cards and the designers' hands visible but not the writing on the cards. Viewers of the tape had no trouble following the development and often asked that the tape be stopped so that they could express their opinions. The most telling moments came when a viewer's explanation required that he point to a blurry card in the frozen image of the screen [Beck89].

Another advantage of this method is that the cards can be placed on a table or wall within spatial relation to each another to indicate is-a, part-of, client-of, and server-of relationships. As refinement progresses, cards can easily be added, updated, or discarded. The authors have stated that the ability to see cards spatially has revealed holes within the decomposition. They report designers repeatedly referring to cards they intend to write by pointing to where they will put them when completed.

Although this method does not claim to be a complete methodology, it has been used successfully by a number of individuals and has been incorporated into other, more complete techniques, such as Wirfs-Brock's responsibility-driven approach. Unlike the diagrams used by other methods, CRC cards are not static and more brainstorming may result. Because the method provides little support for inheritance or for the generation of the scenarios used in the third step, its use as a tool to be incorporated into other, more comprehensive methods may prove the most beneficial.

Play Acting Play acting involves members of an object decomposition team pretending to be instances of classes which they are attempting to refine. A person attempting to understand the purpose or responsibility of a class acts out a scenario highlighting the responsibility in question with another member. For example, a group member attempting to determine a responsibility of a TextEditor may turn to a member and say, "OK, I'm a text editor and you are the disk management module. If I give you

a tag identifier, am I responsible for the format of that tag, or are you? Or maybe the tag should be responsible for itself." The person addressed might then respond, "I think that should be the tag's responsibility. I shouldn't know anything about tag formats."

In this exchange of words, people frequently gesture with hands to portray the passing of information from one object (person) to another object (another person). Frequently, the people involved will address the other people as the class with which they want the other person to be identified. In this manner, the person becomes an instance of a class.

The method of play acting is the most informal of all the responsibility-driven methods. It provides no mechanism for object discovery, no notation for expressing a system of classes, and no facilities for inheritance. With a few modifications, if could probably be used for process-oriented decomposition. However, play acting has demonstrated its usefulness. It can be very helpful in the refinement of classes already discovered. Additionally, the behavior of play acting has been observed frequently in group members of object decomposition teams. Like CRC cards, this technique is probably most helpful when incorporated into other methods.

Structured Analysis Extraction Methods

Structured analysis extraction methods all involve two distinct phases. In the first phase, structured analysis and design techniques are applied to a problem domain. In the second phase, the results of the first phase are transformed in order to extract the information necessary to create an object-oriented design specification.

There are a great number of techniques of this style available. This section examines a few of them. The techniques reviewed are Object Oriented Requirements Specification Method [Bailin89], Object Oriented Design [Seidewitz86], and Object Oriented Structured Design Method [Wasserman89]. These methods provide a good cross-section for this style of methodology. For an insightful discussion on the integration of structured and object-oriented analysis and design techniques, see [Henderson91].

Object Oriented Requirements Specification. This method, first proposed by Bailin [Bailin89, Ward89], depends on structured analysis and design techniques to create a set of data-flow diagrams describing the problem domain. Classes are extracted from the data-flow diagrams and used to construct an entity-relationship diagram. This entity-relationship diagram is used to indicate classes and their relationships and is helpful in finding opportunities for inheritance although it does not

provide any formal assistance in this task. Although the methodology offers little concerning the discovery of classes, it does provide a mechanism for refining classes.

This method consists of seven iterative steps. The seven steps are divided into two phases; the first phase (steps 1 - 3) determines an initial set of candidate classes; the second phase (steps 4 - 7) refines those candidate classes and adds additional classes when appropriate.

In the first step, key problem domain classes are identified by creating dataflow diagrams and extracting the nouns from the process names. These names are used as candidate classes. In step two, active classes are distinguished from passive classes. Active classes possess operations that describe system requirements. Passive classes possess operations which can be deferred until the design phase. This is an important but somewhat ambiguous distinction. During step three, dataflows are established between active classes by constructing an object-dataflow diagram (called Entity Dataflow Diagrams) where active classes becomes process nodes and passive classes become either dataflows or data stores.

In the second phase, classes and their methods are decomposed into sub-classes and sub-methods. A class is composed of sub-classes if the methods of a given class define a set of classes. New object data-flow diagrams are recursively created for each sub-class identified. In step five, the object data-flow diagrams are examined to verify that all methods are addressed by existing classes. If not, new classes may be required. In step six, all methods that are performed by or on new classes are identified. Classes may be redesignated as active or passive at this point. In step seven, all classes are relegated to a specific application domain. Domains are collections of collaborating classes. These individual domains are then represented by an entity-relationship diagram that can be used to help specify classes and their interactions and for giving an overview of the system.

This method offers little guidance for the discovery of an initial set of classes. It does, however, provide two tests which the user can employ to help refine classes. The first determines whether every known functional requirement is met by one of the entities. The second determines whether the internal state of the system is adequately represented by the states of the entities. The manner by which this information is determined is not explained.

One disadvantage to this system is that it fails to effectively incorporate the concept of inheritance. The entity-relationship diagrams produced in step seven can be helpful even though the method does not provide any guidance for this process.

Abstraction Analysis. This technique, proposed by Seidewitz and Stark [Seidewitz86], uses abstraction analysis to transform structured analysis results into a diagram from which classes, objects, and their relationships can be extracted. Like Bailin's Object Oriented Requirements Specification method, this technique begins with the description of a problem domain as a set of data-flow diagrams. Entities within the diagram are identified and transformed into objects through the process of abstraction analysis.

Abstraction analysis consists of three phases. During the first phase, the data-flow diagram is examined for a "central entity". A central entity is defined as the abstraction that best represents the system being decomposed. To locate the central entity, the user examines the data-flow diagram for a set of processes and data stores that are "the most abstract".

In the second phase, entities that support the central entity are identified by following the data flows away from the central entity. The processes and data flows that emanate from the central entity are then grouped together in supporting entities. The data-flows emanating from the supporting entities are then examined in a similar manner. This process is repeated until all processes and data stores are associated with an entity. When all entities have been determined, they are placed within an entity graph.

In the third phase, entities are transformed into classes. These classes are further refined by dividing or combining the entities identified into a set of classes that balance the levels of abstraction with complexity of design. This method provides no formal mechanism for achieving this refinement. It is also during this phase that the operations provided and used by each class are defined. These operations are identified by examining the processes that were incorporated into those entities found during the second phase.

This technique, like Bailin's method described above, provides a more formal mechanism for object discovery and refinement. Unfortunately, this method abandons its formality during the crucial first and second phases when the processes and data stores are combined into central and supporting entities. Like many object-oriented techniques, this method simply provides the user with a set of guidelines for determining the types of abstractions that are best suited to object-oriented programming.

Object Oriented Structured Design. It was the intention of Wasserman, Pircher, and Muller to create a design methodology that could be used for both structured and object-oriented programming [Wasserman89]. The result was not a mechanism for

performing decomposition but a set of notations for expressing decompositions. A series of object-oriented structure charts are used to express objects, classes, methods, instantiation, exception handling, inheritance, and concurrency.

Using Structured Design as a base, this method begins with a diagram created as the end result of structured analysis and design. This diagram is used to indicate modules and their interconnections as well as their calling structures and parameter passing. Classes are indicated similarly to processes. To differentiate classes from other processes within the diagram, the rectangles representing classes are modified to indicate the methods defining a class's interface and its instance variables (figure 2.5a). Instantiation is indicated in a similar fashion. Thick arrows can be placed between modules to indicate visibility, a term intended to mean that the compilation of one module depends upon the compilation of another module.

Inheritance relationships are indicated with thick arrows that emanate from the superclass and point to the subclass. In order to distinguish inheritance from visibility, the superclass's rectangle is dashed (figure 2.5b).

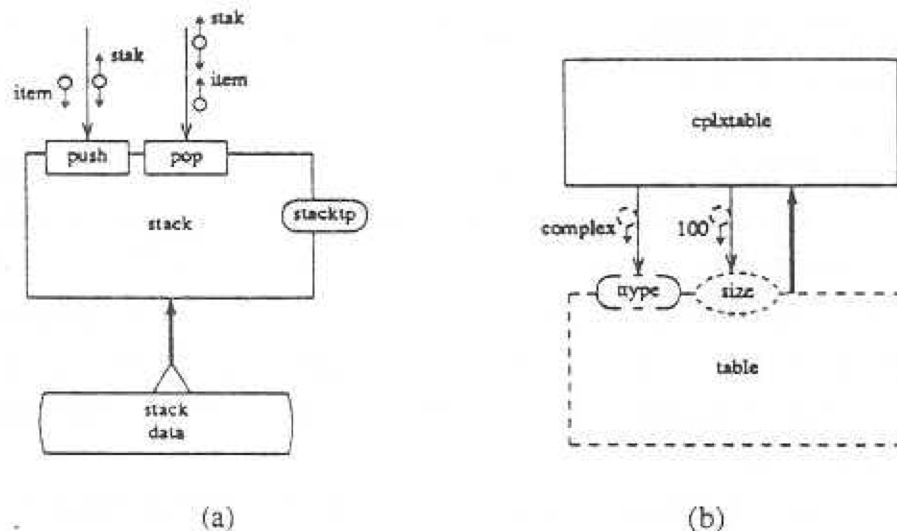


Figure 2.5

Object Oriented Structured Design Example [Wasserman89]

This "method" provides a set of formal notations for describing different styles of decomposition, but it provides no mechanism for object discovery or refinement. Similar to the CRC cards described previously, this notation is probably best used as a tool for other analysis and design methods.

Modified Structured versus Object-Oriented Analysis and Design

The advantages of the modified structured techniques are easier to understand than their potential disadvantages. One advantage is that they require little or no retraining of those individuals who already use them. Another advantage is that many of them are quite formal and can be used to validate system requirements [deChampeaux91]. This is important in large projects, where different individuals are responsible for the development of components that must be integrated into a larger system. It is also important for systems whose correctness and reliability is of a more critical nature (ie. life support systems, surgical guidance systems, elevator controls, heavy equipment control systems, etc.). Formality is also important when presenting analysis results to clients in order to verify that the clients' or reviewers' needs have been met [Liskov75].

The potential disadvantages, however, are numerous. Some argue that object-oriented programming is a fundamentally different programming strategy [Gibson91, Henderson91]. Given that structured techniques emphasize processes and the interactions among processes, they may not provide the necessary information for object-oriented programming [Booch86, deChampeaux90]. Some believe that the use of structured techniques will further discourage the user from addressing the technical and conceptual difficulties presented in Chapter 2. These problems include a tendency to map procedural abstractions directly into object abstractions [OShea86] and the tendency to focus on data rather than on processes [Beck89]. Additionally, it is argued that most structured techniques do not address the issues of data abstraction and information hiding and are not responsive to changes in the problem space [Booch86]. Moreover, few of the modified structured techniques address inheritance. Based on these weaknesses, many conclude that object-oriented code produced with structured analysis and design techniques does not adequately employ inheritance or allow for rapid development and will not be easily modifiable or comprehensible [Gibson91].

It has also been argued that object-oriented programming more closely mirrors the actual world [Booch86, Goldberg83] and that object-oriented techniques are, therefore, more intuitive, easier to learn, and more responsive to changes in the problem domain. Rosson, however, points out that no formal analysis supports these claims [Rosson89]. Additionally, it is argued that object-oriented techniques better address the issues of encapsulation and data hiding, making code produced with these techniques easier to understand and simplifying maintenance.

On the other hand, many have argued that purely object-oriented analysis and design techniques are rare and that those that do exist are seriously lacking in formality

[Wilson88]. Most have no formal notation, do not cover the entire life-cycle, are not as rigorous or comprehensive as their more structured counterparts [deChampeaux91, Ladden89]. Due to this lack of formality, it is impossible to validate systems against requirements. Industry standards are lacking on object-oriented analysis and design techniques and are, therefore, likely to change. Surprisingly few object-oriented techniques address inheritance [Griss91]. In addition, most object-oriented methodologies have little to say about reuse and usually assume a clean slate when beginning an analysis or design activity [Griss91]. Because of this, potential reuse of analysis and design results is lost.

Another group of individuals believe that structured and object-oriented techniques can be integrated. Because there are so many different structured analysis and design techniques, it is impossible to make general statements. Individual techniques must be examined [Ward89]. Henderson and Constantine suggest using strict object-oriented techniques to determine classes and their relationships and then developing the methods for those classes using structured techniques [Henderson91]. Seidewitz and Stark suggest that structured analysis should first be used to develop the specification of a system and then abstraction analysis can be applied to make the transition to an object-oriented design [Seidewitz86].

Summary

Many claim that the informality of object oriented decomposition methods prevents them from being effective for large projects and that structured analysis and design techniques should be used. There are two levels of formality being stressed in these arguments. The first is a formality of notation for expressing the results of design. The second level is the formality of the mechanism for decomposing a problem domain into a set of classes. While it is true that all of the object oriented methods for transforming the problem domain are informal, claims that process-oriented analysis and design techniques are more formal are weak. Of those process-oriented techniques investigated, all fail to provide an algorithm for decomposing a domain into a set of process abstractions. What heuristics they do provide are little more than advice and tests which can be applied to a candidate decomposition. This failure may not be acknowledged due to the familiarity of process-oriented programming. Most programmers are so indoctrinated in structured programming that they forget that decomposition is inherently ambiguous and that a set of rules cannot, in itself, claim to reduce a problem to a superior set of abstractions.

On the other hand, the structured analysis and design techniques do appear to be superior in their formality of expression. Structured programming has existed for many years and has the benefit of numerous formal notations. It seems obvious that the lack of such formal tools for object-oriented programming will be solved over time. Indeed, the notation provided by Wasserman, Pircher, and Muller seems to be a step in this direction.

The fact that none of the object-oriented techniques presented in this chapter adequately addresses inheritance or the issue of maintenance is, however, a real failing.

Chapter Three:

INHERITANCE AND REUSE

Inheritance is a powerful mechanism that supoorts polymorphism. It also provides the ability to define abstract data types in terms of other data types and places them within a hierarchical structure. This chapter shows that inheritance can be difficult to use effectively due to conceptual and technical problems. It also shows that the relationship between inheritance and reuse is ambiguous.

Open issues related to inheritance and reuse are discussed in this chapter. First, several potential candidates for reuse are identified. Next, a number of techniques related to object-oriented programming that help users reuse code are discussed. These include delegation, genericity, conformance, enhancement, and inheritance. Inheritance is then discussed in greater detail, with particular emphasis on the issues that can make its effective use difficult. Finally, the advantages and disadvantages of single and multiple inheritance are outlined.

Reuse

According to Jones, only 15 percent of new code serves an original purpose [Jones84]. Many individuals initially approach object-oriented programming with the expectation that it will result in more reuse of code and shorten the development cycle. While it is true that object-oriented programming has this potential, there is no guarantee that more reuse will occur in the long term. Furthermore, object-oriented programming does not address many other types of reuse. Reuse can also include code templates and libraries. Cutting and pasting segments of code from one place to another can be done, although many argue that this form of reuse propagates errors [Wybolt90]. Others argue that the design of class interfaces should be reused as they are often more important than the classes, themselves [Griss91, Yakemovic90]. There are many products of analysis and design that are potential candidates for reuse. These include specifications, requirements, and diagrams. Additionally, the transference of personnel from project to project is a form of reuse [Meyer87]. The term reuse, in the

object-oriented programming world, refers to a very limited form of its more general meaning and only addresses the reuse of code [Lewis91].

Reuse of Code

There is a great deal of evidence that object-oriented programming and reuse increase productivity. Lewis, Henry, Kafura, and Schulman [Lewis91] conducted a formal study and found that object-oriented programming substantially improved productivity. The authors conclude that a significant portion of this productivity was due to reuse and that object-oriented programming provides for greater reuse than process-oriented programming. Wybolt claims that new software products created through the mechanism of inheritance allowed for a code savings of 85 percent over process-oriented solutions [Wybolt90]. Such claims are common although others claim that these levels of reuse are rare but can be duplicated if the practice of reuse is encouraged [Woodfield87].

Object-Oriented Mechanisms for Code Reuse

Code reuse has been suggested as a way of obtaining faster prototyping through less work. The following mechanisms provide greater flexibility in the creation of encapsulated entities based upon those previously defined and in implementing the concept of polymorphism. Both of these provisions can lead to greater code reuse.

Delegation

Delegation is used in both class-based and non-class-based object-oriented languages. In most languages that support delegation, objects are defined by stating how they differ from prototype objects. The prototype objects encapsulate default behavior. When an object receives a message, it either performs the requested action or delegates a prototype object to accept the request. Delegation differs from inheritance in that determining which prototype should accept a request is a dynamic process, whereas inheritance is static. For example, if an object is defined from several prototypes, each of which possesses a method named methodX, an object can decide dynamically which prototype should receive the request. No such mechanism is available in inheritance. Some claim that delegation can simulate inheritance by disallowing dynamic delegation and that inheritance cannot be made to simulate delegation, thus making delegation the more powerful construct [Booch91a,

Lieberman86, Wegner87, Blair89]. While more flexible than inheritance, the use of delegation may result in objects whose behavior is more difficult to predict and control.

Genericity

While genericity is not strictly an object-oriented technique, it is frequently used in conjunction with object-oriented programming languages. Many assert that it provides better reusability and extendibility. Genericity provides for the creation of procedures and methods that can operate on several different types of objects. This is accomplished by specifying the "type" of the objects as a parameter to the procedure or method, allowing it to operate correctly on the other parameters and, thus, simulate polymorphism. A proposed strength of genericity is that it seeks to remove the burden of type checking, binding, and behavior sharing from inheritance [Blair89]. It is often used to make untyped languages more robust and strongly typed languages more flexible. Meyer argues that since genericity can be simulated by inheritance but not vice-versa, inheritance is a more powerful mechanism [Meyer86].

Conformance and Enhancement

Conformance is a mechanism used in some class-based programming languages to implicitly derive a class hierarchy and, through enhancement, provide polymorphism. Under this system, classes that "conform" to other classes can reuse methods defined by those classes. A set of rules is used to indicate whether one class conforms to another. A class P conforms to class Q if P provides all of the methods defined by Q. Each method argument and return value defined under P must also conform to those arguments and return values defined under the equivalent method of Q. Finally, the arguments and return values of Q must conform to those of P.

While conformance does not provide code sharing, the additional mechanism of enhancement allows conforming classes to indicate which class contains the shared code. When used in combination, these mechanisms impose a class hierarchy and provide polymorphism. For more information on these mechanisms, readers are directed to the work of Black, Hutchinson, Levy, Carter [Black87], and Horn [Horn87].

Inheritance

Inheritance is the most popular mechanism for polymorphism and code reuse in class-based object-oriented programming languages. It supports the creation of classes through the "modification" of other pre-existing classes. Modifications can be

expansions and/or contractions of pre-existing classes. This allows users to "borrow" attributes and methods from superclasses and to modify those attributes and methods to suit the needs of the newly defined class without affecting the superclasses.

The hierarchy created through the mechanism of inheritance provides polymorphism because it helps to ensure that subclasses of a particular class can respond to a particular message defined under that class. This stems from the fact that classes inherit methods defined under superclasses. While a class is allowed to redefine a message, most object-oriented programming languages do not allow a class to ignore an inherited message.

Both polymorphism and inheritance have been simulated using structured programming languages. This is generally done by abstract data types keeping track of arrays of pointers to functions. When an abstract data type receives a message, the array of pointers associated with it is examined. If the appropriate function resides in the array, it is invoked. If the function does not exist, a pointer is followed to an abstract data type that represents a parent of that type and its array of function pointers is examined. Because this technique is prone to error, few people use it. The mechanism of inheritance greatly simplifies this style of programming by removing from the programmer any responsibility to keep track of function and superclass pointers.

Why Inheritance is Difficult

Although inheritance has been used successfully, it still has not provided the level of reuse that proponents originally claimed. Many novices find inheritance to be a difficult mechanism to use effectively. There are a number of reasons why inheritance is difficult. These include the following: inadequate support for inheritance by object decomposition methods, the conflict between inheriting for definition and inheriting for reuse, the conflict between inheritance and data hiding, the conflict between generalization and specialization, and a number of other issues orthogonal to object-oriented programming. Each of these issues will be discussed in the following sections.

Inadequate Support for Inheritance by Object Decomposition Methods

Most techniques for object decomposition do not address inheritance. Those methods that do provide mechanisms for discovering or expressing inheritance relationships are inadequate. Because of this, programmers often attempt to ascertain

inheritance relationships using informal and ambiguous methods. This can result in hierarchies that are difficult to understand and reuse.

The Conflict Between Inheritance for Definition and Inheritance for Reuse

Two common reasons to use inheritance are at times in conflict [Wirfs90a]. For example, a user may define a new class by inheriting from another class to reuse the methods defined under the pre-existing class. The user inherits with less regard to "program structure", differences in class responsibilities, or levels of abstraction within the inherited classes. Reuse of code is the overriding factor in determining from which classes to inherit. This form of inheritance is more frequently found in languages that support multiple inheritance but is also found in single inheritance examples. It is often referred to as inheriting for reuse. Proponents of this type of inheritance claim that it maximizes reuse [Meyer88]. Some point out that this form of inheritance may encourage subclasses that may not be compatible with ancestors and may change method arguments making inherited methods incorrectly typed [Johnson89]. Still others argue that the end result of inheritance for reuse is a class hierarchy that is impossible to understand, maintain, or reuse. Others assert that inheritance for reuse is difficult because it is unclear which behaviors of a class are "essential" and should not be redefined among subclasses [Stemple89].

Inheritance is most frequently used to indicate an is-a relationship between a new class and a pre-existing class. Here, the purpose is to inherit the attributes and methods from a given class to indicate similarity between the two classes [Korson90]. This is often referred to as inheriting for definition. One viewpoint argues that coherent and well structured class hierarchies are the primary factor behind maximum reuse [Coggins93]. They argue that the primary benefit of inheritance is not reuse, but modularity and understandability [Micallef88], which is more likely to arise from inheritance for definition. Coggins states that the chief contribution of object-oriented programming is not reuse, but raising the level of abstraction implemented in libraries to a level where clients can effectively contribute to the design of the software they need [Coggins90a]. One implication of this view is that undue emphasis is placed on the reuse of code.

Unfortunately, both of these techniques are frequently used in conjunction, in spite of the fact that their principles are contradictory. The conflict between the technical goal of maximizing reuse and the conceptual goal of a creating a conceptually intuitive hierarchy produces class hierarchies that are difficult to comprehend. Eriksson

suggests splitting the mechanism of inheritance into two mechanisms; one supporting inheritance for definition, and the other supporting inheritance for reuse [Eriksson90].

<u>The Conflict Between Inheritance and Data Hiding</u>

The greatest difficulty in the effective use of inheritance likely arises from the conflict between inheritance and encapsulation. Through data hiding, encapsulation helps create a system that is easier to understand and maintain. Data hiding minimizes dependencies among classes. Inheritance allows descendent classes to rely on the implementation of ancestor classes. Many object-oriented programming languages allow descendent classes to directly reference instance variables within ancestor classes, call private operations on ancestor classes, and refer directly to superclasses of their parent classes [Liskov87]. These abilities are what gives inheritance its power. Many feel it is also a liability. According to Booch [Booch91a],

> There is a very real tension between inheritance and encapsulation. To a large degree, the use of inheritance exposes some of the secrets of an inherited class. Practically, this means that to understand the meaning of a particular class, you must study all of its superclasses, sometimes including their inside views.

An is-a relationship is a "contract" between an ancestor and a descendant. It restricts the kind of changes the designer of an ancestor class can make. Renaming, removing, or reinterpreting an instance variable or method can adversely affect descendants [Micallef88]. Many argue that as classes evolve [Scharenberg91], making such changes becomes more difficult. As conceptual modifications are made to classes, the is-a relationships between descendants and ancestors may be altered, causing unexpected behavior in descendent classes [Stemple89]. For those whose primary goal is inheritance for reuse, there is the additional issue of implementation evolution. When the reuse of a particular implementation is the primary goal for inheriting from a class, descendent classes may behave unpredictably when that implementation is altered.

Others argue that inheritance compromises encapsulation by scattering related functions among ancestors and descendants, creating unnatural dependencies [Wybolt90]. A number of rules have been proposed to help minimize the dependencies between ancestors and descendants. These rules generally treat ancestors and descendants as unrelated classes in an attempt to insulate descendants from the evolution of ancestors. One of the more common sets of rules is known as the Law of Demeter. The stronger version of this law states that, for all classes C and all methods M attached to C, all objects to which C sends a message must be instances of the argument classes of M (including C) or the "explicit" instance variables of C. This prevents classes from directly referencing inherited instance variables. Instead,

inherited instance variables must be accessed through methods provided by ancestors. Thus, if an instance variable name in an ancestor is changed or it is removed to determine the value dynamically, descendant classes would not be affected. For more information, refer to [Lieberherr88, Lieberherr89, Sakkinen88, Snyder86].

The Conflict Between Generalization and Specialization

Class hierarchies typically have more general classes towards the top of the hierarchy and more specific classes towards the bottom. In the construction of a hierarchy, the designer must frequently balance immediate usage needs of a class, which are usually more specific, with the more general need of ensuring the ease of reuse of that class at a future time. Associated to this conflict is the conflict between inheriting a few "large" classes and many "smaller" classes. While more general classes and abstract classes may be important in defining interfaces for descendants and easier to reuse, there potential for reuse may be small when compared to larger, more specific classes. This is used as an argument by some for top-down design. The opposing argument is that more specific classes are harder to reuse and that bottom-up design should be used to ensure smaller, more general, and more reusable classes. It is unknown which technique has greater potential for reuse.

Determining how general or specific to make a class can be difficult. Beck points out that a minor modification to an ancestor class may eliminate the need for many descendent classes but it may also alter the levels of abstraction for a hierarchy and make it more difficult to understand [Beck88]. Meyer argues that it is impossible to detect commonalities early enough to move them to the proper classes within the hierarchy and that developers must resign themselves to occasional reorganizations of the hierarchy [Meyer90].

In an effort to resolve this issue, some have emphasized the distinction between abstract classes that are defined solely to be subclassed but are not instantiated, and classes that are instantiated but never subclassed. Others advocate abstract classes that are not instantiated, but do not insist that instantiated classes are not subclassed. These general classes are used as "contracts" between derived classes and users [Rao93]. Subclasses of these abstract classes must fulfill all requirements defined under the abstract classes. Many have argued that abstract classes can be more easily reused and are often more valuable than the implementation underlying these contracts [Griss91].

Issues Orthogonal to Object Oriented Programming

When object decomposition and inheritance have been done "properly", or at least consistently, there are still a number of issues that can negatively impact the use of inheritance. These issues were originally raised in the structured programming world and remain unresolved. Most object-oriented methodologies never address these issues. These include the error handling, argument validation, composite object handling, control and communication, group and compound operations, and a number of "stylistic" issues.

Error Handling. Error handling refers to how a particular object or process responds to errors. There are a variety of ways that an error can be handled and no single way has emerged as the clearly superior method. Although a standard, non-resumptive exception mechanism currently exists in C++, few languages have addressed the issue of error handling. This can seriously limit the collaboration between classes and the amount of reuse. Classes may provide for similar levels of abstraction but employ incompatible techniques for error handling, making it difficult to combine them.

Argument Validation. Similar to error handling, argument validation requires cooperation between objects within a system. If objects handle argument validation differently, class interaction may become difficult to integrate and prone to error.

Composite Object Handling. Composite objects refers to the combination of a number of objects into one. Group and compound operation difficulties arise when classes abstract a domain at different levels. In other words, classes in one library may embody larger concepts, while classes in another library may break down those larger concepts into smaller sub-components. While neither decomposition is incorrect, the different levels of abstraction can cause incompatibilities in how operations are performed upon these composite objects.

Control and Communication. This refers to the general problems involved in determining which objects within a system are in control and how communication is to be established between objects defined in different libraries.

Stylistic Differences. Stylistic differences include the level of functionality that classes may provide, optimizations for speed, ease of use, code or data space requirements, and debugging facilities.

For further information on these issues, the reader is referred to the works of Berlin [Berlin90] and Cohen [Cohen91].

Description of Inheritance Hierarchies

There are different types of inheritance hierarchies supported by object-oriented programming languages. These include single inheritance/single tree, single inheritance/multiple tree, multiple inheritance/single tree, and multiple inheritance/multiple tree. These last two forms of inheritance will be discussed together.

Single inheritance/single tree is a form of inheritance where all classes are descendants of a single root class and each class has only one parent class (figure 3.1a). The class hierarchy provided by Smalltalk is an example of a single inheritance/single tree. In single inheritance/multiple tree inheritance, the user is allowed to have more than one inheritance tree. In this situation, each class still inherits from only one superclass but more than one tree exists, each with its own root class. The trees remain disjoint (see figure 3.1b). Multiple inheritance allows different trees to become intertwined by allowing classes to inherit from more than one parent class (figure 3.1c).

Many object-oriented languages support all three styles of inheritance. It is up to the user to decide which form of inheritance to employ. Knowing the potential costs and benefits of all three styles will allow the user to make an informed choice when deciding which style and language to use.

Advantages and Disadvantages of Inheritance Hierarchy Types

Single Inheritance/Single Tree. The primary advantages of single inheritance are both its technical and conceptual simplicity. A programmer investigating a particular class can easily determine a class's ancestors, making it easier to determine its inherited behavior and definition. This simplicity extends to the definition of objects. Technically, single inheritance provides a more stable initial set of attributes and methods with which to define an object. It is not necessary for the user to search through a variety of inheritance hierarchies in order to determine the origin of a particular method or attribute. Conceptually, single inheritance makes understanding the purpose of the object easier. The user is freed from frequently confusing cases in which classes are defined from the combination of conceptually disjoint classes.

The most obvious disadvantage of single inheritance/single tree environments is that it inhibits the use of hierarchies developed by others. Another problem is that conceptually unrelated entities may be forced into unwanted relationships. In other words, classes that encapsulate unrelated entities must be placed within the same class hierarchy. Since an inheritance hierarchy implies an is-a relationship between classes

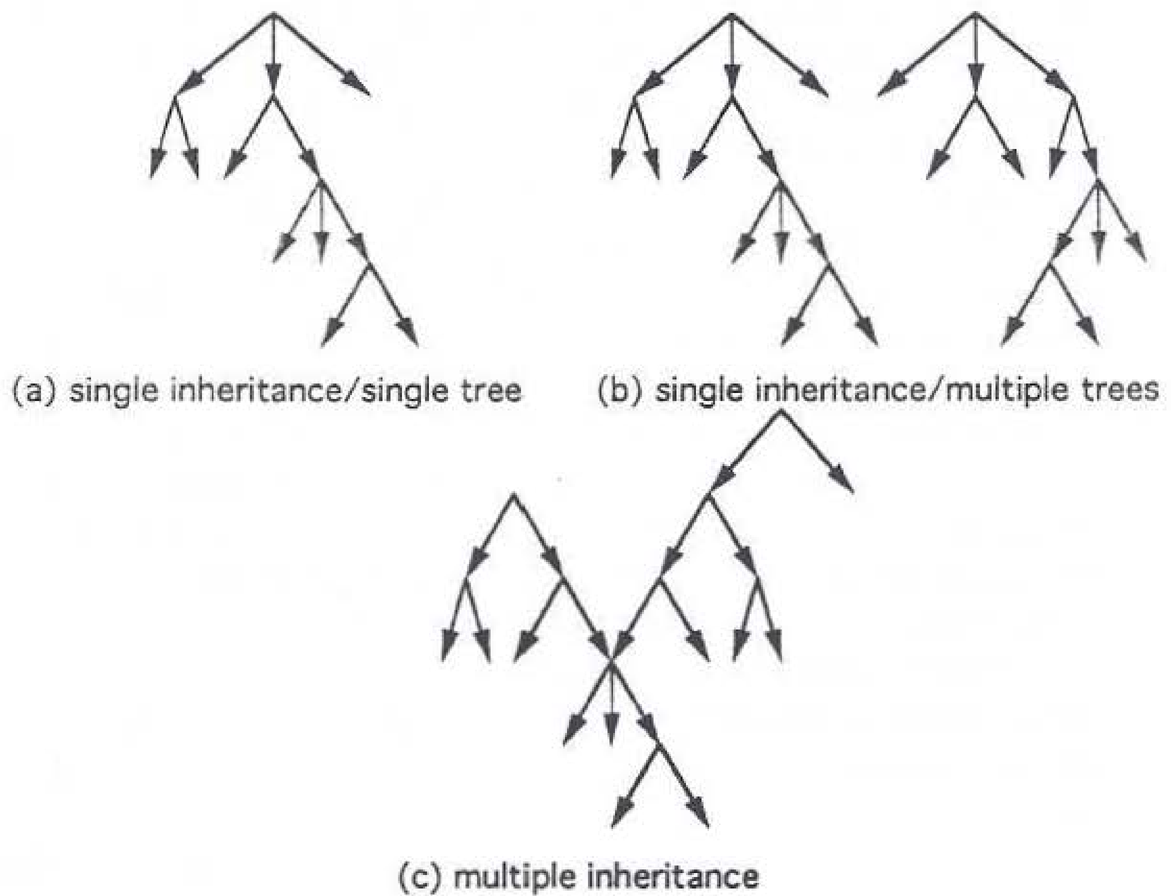


Figure 3.1
Types of Inheritance

within it, the relationship implied by the hierarchy may not be valid. If the user is forced to place the classes within a single tree and does not remember that there is no conceptual relationship between the classes, an unwanted relationship may develop as the hierarchy evolves. Although Smalltalk provides some of the best tools for class browsing of any object-oriented language, it is often difficult to discover dependencies within the hierarchy.

Placing large numbers of classes within one tree also results in the creation of libraries that are too large. If all classes must be placed within one tree, the library will become increasingly large as it develops from project to project, thus making navigation of the hierarchy more difficult and use of the library more expensive in computer and user effort.

Another problem caused by placing too many classes in one tree is the inability to separate applications and tools from the environment in which they were developed. The author and his colleagues encountered this difficulty while developing a client/server project in Smalltalk. It was decided to create the client and server modules within the same hierarchy (image). This was done with the belief that once debugging was completed, the two modules could be pulled apart and placed within their own hierarchies in order to decrease the size of the hierarchy. While developing the two modules within the same hierarchy served to speed development time, it made the separation of the modules nearly impossible. It was extremely difficult to determine how the code was divided between the two modules. The "tightness" of a single-tree hierarchy further magnified the problems brought about by our poor engineering.

Some members of our group complained that it was often difficult to distinguish between classes developed to support the client/server modules and classes developed for the tools we had constructed. Others argued that the system was engulfed by excessive amounts of unrelated code. The removal of unrelated code was not possible because we did not know whether code was being used by the environment itself. The developers became concerned that the removal or modification of code would break the system. These phenomena have also been reported by Wirfs-Brock and Wilkerson [Wirfs88].

Another disadvantage involves the employment of reuse. Advocates of multiple inheritance argue that single inheritance cannot sufficiently employ reuse. They argue that single inheritance is not sufficiently flexible for class definition and that much reuse that could be obtained using multiple inheritance is lost.

Single Inheritance/Multiple Tree. One advantage of single inheritance/multiple trees is that the user is free to incorporate externally-supplied libraries with his/her own libraries. Moreover, given that not all classes are descended from the same class, it is easier to avoid the situation described in the previous section, where the user is forced to place all, potentially unrelated, classes together in one tree. With this form of inheritance, conceptually related abstractions can be placed in their own inheritance tree. Thus, inheritance can be employed exclusively to represent valid is-a relationships rather than to impose a structure on unrelated classes.

An additional technical advantage of single inheritance/multiple trees is that individuals on a development team can be responsible for different inheritance trees. This can eliminate much of the concern that changes made by an individual programmer may inadvertently result in changes in the definitions of other classes.

One disadvantage of multiple trees is that determining relationships between classes is more difficult. With multiple trees, a user cannot assume common ancestry for different classes. The behaviors and interactions of classes that belong to different trees may be more difficult to determine.

Multiple Inheritance. One advantage of multiple inheritance is increased flexibility. Multiple inheritance does not constrain inheritance to one parent class. This may be beneficial if the user desires to inherit a number of attributes and methods from different, even unrelated, classes. A user can inherit a particular attribute or method from any number of pre-existing classes and avoid the duplication of the code necessary to implement the attribute or method, within the newly defined class. In this manner, multiple inheritance can facilitate the reuse of code [Linton87, Borning82]. Moreover, others argue that it allows the user to remove a rigid, hierarchical view of the world that single inheritance imposes [Wirfs88, Meyer88].

Although multiple inheritance is frequently used for inheritance for reuse, it can be used to inherit for definition. This can occur if a number of related entities can be divided into two conceptually overlapping groups. Coad and Yourdon give the example of planes (figure 3.2). In this example, the class Plane is subclassed into two groups; type of craft and use of craft. Type of craft can be either jet or propeller, while use of craft can be either civilian or military [Coad91a]. Another proposed advantage is that multiple inheritance encourages modularity by allowing classes to be built from a variety of component classes [Moon86].

There are a number of disadvantages of multiple inheritance. One of the most difficult problems is that of additional complexity. This may occur in a variety of ways. First, there is the technical problem involving inheritance from two or more classes which have a method or instance variable name conflict [Carre90]. These clashes occur whenever a descendent class is defined as inheriting from at least two classes, where each has a method or instance variable with the same name. For example, a class C might inherit from the classes A and B, each of which has a method entitled methodX. When class C is defined, the user must determine which methodX should be inherited.

This problem is typically handled from a technical, rather than conceptual, standpoint by most object-oriented programming languages. Some languages force the user to rename at least one of the offending methods or instance variables. This necessitates the modification of descendent classes even though their interfaces may not have changed [Meyer88, Micallef88]. When attributes or methods are renamed, it can

become difficult to find particular behaviors or attributes, because the user is forced to create new names that are less intuitive than the original names.

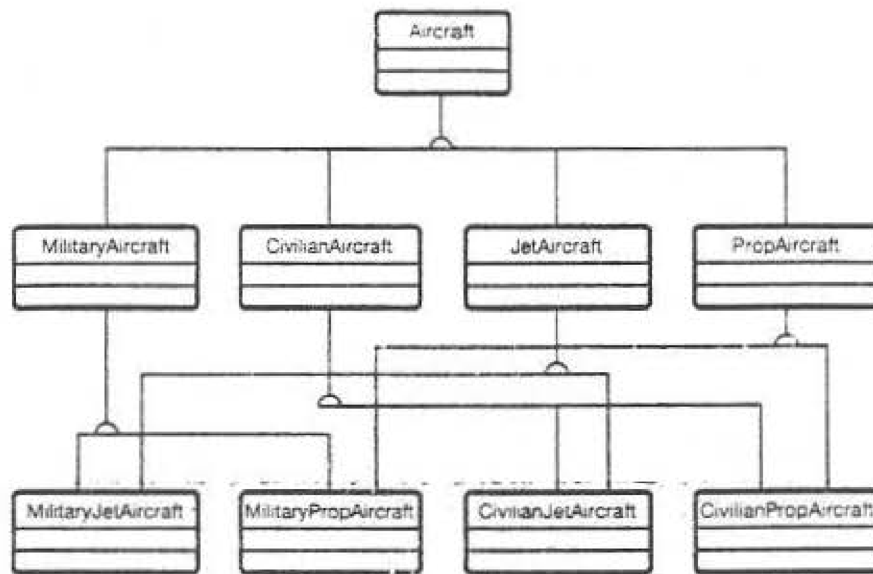


Figure 3.2
Multiple Inheritance [Coad91a]

Some languages require the user to specify the order in which ancestor classes are inherited so that the appropriate method is determined. With this method, classes may behave differently if the specified order of inheritance in changed in any ancestor classes, even though the descendent class' interface has not changed. This method has the additional problem that it does not support situations where two or more ancestor classes have multiple clashes and the descendent class requires default methods or instance variables from one ancestor and some from another ancestor. Users must also remember the order in which classes are inherited, which may not be intuitive and, some argue, violates encapsulation.

A third group of languages require the user to specify at invocation time which of the offending methods should be used. If evolution occurs, all three methods force the user to change the interface to one or more classes. Changing the interface of a class often requires changes to other classes that interact with it. Thus, multiple inheritance can require the user to spend additional time on code maintenance and may make clean encapsulation more difficult to achieve.

Problems also arise when multiple paths exist between two classes. This is known as repeated inheritance and causes different problems for different languages. Such cases generally require language specific solutions. Because the problems and solutions of repeated inheritance are language-specific, this chapter does not address this issue. For more information, refer to [Cargill90, Meyer88, Booch91a].

Another difficulty involves definition. When writing or maintaining code, it is important to understand what a class represents. Multiple inheritance can make this task more difficult. Does the proposed abstraction differ from all other entities to the extent that it must be described by two potentially unrelated classes? What are the implications of using multiple inheritance to borrow behavior from potentially unrelated classes? How can this affect our comprehension of a particular class?

The most serious disadvantage of multiple inheritance is shared by single inheritance. Given that every class within a hierarchy depends directly on all ancestors for definition and implementation, any modification of ancestors may change the behavior and definition of all descendants. With multiple inheritance, this problem is magnified due to the exponential growth in the number of inheritance paths between ancestors and descendants, any one of which may affect descendent classes. Since classes may be defined much further down the inheritance hierarchy from a modified ancestor class, the user may remain unaware of the modifications.

Cargill points out that the use of multiple inheritance is fairly easy to avoid [Cargill90]. When the inheritance of methods or attributes from a number of parent classes is desired, one can inherit directly from one parent class and place within the new class attributes whose values are instances of each of the other "parent" classes. Thus, the new class becomes a composite class. Access to the methods and attributes of the additional "parent" classes can be obtained through the use of the attributes in the newly created class.

If the use of multiple inheritance is for definition rather than for reuse, the technique outlined above may create a class that is less comprehensible than a class created using multiple inheritance. For example, if a user wishes to create a class Cyborg by inheriting from class Man and class Robot, placing an instance of class Robot inside an instance of class Man could be very confusing because it sets up a part-of relationship when an is-a relationship is more conceptually satisfying.

Multiple inheritance is, by definition, more flexible than single inheritance. It allows users to freely inherit from any class. Many argue that it carries immense potential for misuse [Henderson91], and some argue that most examples of multiple inheritance in the literature are contrived [Erikkson90] or indicate an error in analysis

and design [Cargill90]. Users must give careful consideration to the use of multiple inheritance, weighing costs against benefits for each situation.

Summary

Classes are the mechanism that support encapsulation. The need for encapsulation has grown out of many years of structured programming experience. As such, the goals and advantages of encapsulation are well-understood. Inheritance is a mechanism that supports polymorphism. Like encapsulation, polymorphism has also grown out of a programming need. It allows newly created abstract data types to reuse unmodified code. Without polymorphism, code often requires large sets of conditional statements that test the type of an object and call the appropriate function. Every time an abstract data type is created, a new conditional must be created.

The mechanism of inheritance, however, has not grown out of a well-understood programming need. Although it supports polymorphism, inheritance also supports the definition of abstract data types in terms of other data types and places them within a hierarchy, allowing the reuse of code defined by ancestors. While this ability is powerful, there appears to be no consensus on what fundamental problem of programming it was created to solve. What problem does this power address? What are the implications of defining one class in terms of others and placing classes within a hierarchy of shared code? It is as if inheritance were some strange item discovered in the attic and for which everyone is now claiming to know the proper use.

Unlike encapsulation and polymorphism, the goals and advantages of inheritance are unclear. Because of this, it is explored from the opposite direction. The end results of its use are examined first and then an attempt to surmise its functions are made. This confusion can be seen in the arguments over single and multiple inheritance. Most of these arguments consist primarily of end-result advantages and disadvantages of the mechanisms rather than the theory underlying them.

Through all this confusion over the principles behind inheritance comes those grasping the banner of code reuse. Reuse of code is an extremely important goal but using inheritance strictly for code reuse may be like using a shotgun to hunt flies. While it can be used, it may not be effective without proper guidelines. Additionally, it provides a great deal more power than the reuse of code. Understanding this power and learning how best to apply it are interesting challenges.

CONCLUSION

In discussing the issues of learning object-oriented concepts, object decomposition, and inheritance and reuse, a number of questions have arisen.

The first issue was vocabulary. There are a multitude of terms in the literature used to describe similar concepts. Additionally, many of them are used to describe multiple concepts. This makes deeper discussion of some issues difficult.

Another issue was learning the basic concepts of object-oriented programming. Learning from experts seems to be a universally accepted idea even if no one knows how to do it at this time. It seems apparent that some sort of language independent setting can be helpful, especially in helping the user distinguish between languages and language principles. It is also important that academia and industry take a more aggressive role in teaching the why of programming rather than just the how of programming as they have traditionally done.

There is a great need for evaluation of the effectiveness of object-oriented analysis and design techniques. Why are these techniques so informal? Does this tell us something about the nature of programming? Why don't more of them properly address the issue of inheritance? Why don't more of them cover the software life cycle? How can these informal techniques possibly be used on projects involving 10, 50, or 200 people? Can structured techniques be integrated with object-oriented techniques? Should they? If object-oriented programming is supposed to provide for easier reuse, why aren't there more object-based libraries available? If they are available, why aren't more people using them? Methodologies that cover the entire life-cycle, especially the most lengthy, expensive, and overlooked phase of maintenance, must be developed to aid developers in hierarchy construction and maintenance. Object oriented analysis and design cannot have a significant impact on software construction until these issues are resolved.

It has been claimed that the two most important contributions of object-oriented programming are encapsulation and inheritance. Encapsulation is a principle whose usefulness is easily grasped even if its effective use is more elusive. Inheritance, on the other hand, is not a principle. It is a mechanism that is used to aid in the definition of entity abstractions and in the reuse of code. These goals are often contradictory.

Mechanisms like delegation and genericity that share some of the same goals as inheritance should also be examined more thoroughly for their usefulness and power. The desire to inherit for reuse and definition has not been resolved. Are these two different mechanisms? Is one better than the other?

The type of reuse provided by inheritance and other mechanisms in most object-oriented languages is only one kind. Is it really fair to expect object-oriented programming to help in the quest for reuse when most of the techniques available are the same techniques that have been available (and ignored) for decades? It seems unlikely. Neither a philosophy (like object-orientation) nor a language mechanism (like inheritance) can overcome its poor use by a programmer.

Finally, the questions surrounding multiple inheritance are numerous, with passionate proponents on both sides of the issue. Whether or not multiple inheritance is eventually judged good or bad seems moot. It is here now and is being used by many. Perhaps more research should be done on how to use it effectively in order to limit its disadvantages.

The Future of Structured and Object-Oriented Programming

Object oriented programming attempts to solve many of the problems discovered in structured programming. It provides mechanisms superior to most structured programming languages for higher-level concerns such as encapsulation and reuse. It does not, however, address any of the lower level issues that have plagued structured programming since its inception. Error handling and validation of arguments are examples of lower-level design decisions that are not handled well by either method. Because it does not address these issues, object-oriented programming, by itself, will not provide the levels of reuse that many have claimed.

Some have suggested that the future of programming languages resides in logic programming, functional programming, knowledge-based systems, multi-paradigm systems, and many others. The competing concerns of efficiency, understandability, and flexibility will most likely prevent any one language or methodology from replacing all of the others. As the computing community expands and becomes more integrated through networks, the need for tools that support distributed, asynchronous computing becomes more important. Object oriented programming languages are notoriously weak for dealing with asynchronous communication. While some research is being done to overcome this weakness, standards remain distant.

As lower-level issues are addressed and the needs for integration, efficiency, understandability, and flexibility are understood, object-oriented languages will most

likely splinter into numerous groups, each with their own set of solutions to the problems at hand. The concepts that underlie object-oriented programming will undoubtedly become incorporated into future programming methodologies.

Bibliography

[Abbott83] Abbott, R. J. Program design by informal English descriptions. Communications of the ACM, 26(11), November 1983, 882-894.

[Antebi90] Antebi, M. Issues in teaching C++. Journal of Object Oriented Programming, November 1990, 11-21.

[Bailin89] Bailin, S. C. An object-oriented requirements specification method. Communications of the ACM, 32(5), May 1989, 608-623.

[Beck86] Beck, K. comments made by K. Beck during panel session Panel: the learnability of object-oriented programming systems. OOPSLA conference proceedings, September 1986, pp. 502-506.

[Beck88] Beck, K. Raghaven, R., LaLonde, W. R., and Weinreb, D. Panel: experiences with reusability. OOPSLA conference proceedings, September 1988, pp. 502-506.

[Beck89] Beck, K., and Cunningham, W. A laboratory for teaching object-oriented thinking. OOPSLA conference proceedings, October 1989, pp. 1-6.

[Berlin90] Berlin, L. When objects collide: Experiences with reusing multiple class hierarchies. ECOOP/OOPSLA conference proceedings, October 1990, pp. 181-193.

[Black87] Black A., Hutchinson, N., Jul, E., Levy, H., and Carter L. Distribution and abstract types in Emerald. IEEE Transactions on Software Engineering, SE-13 (1), January 1987.

[Blair89] Blair, G. S., Gallagher, J. J., and Malik, J. Genericity vs inheritance vs delegation vs conformance vs.... Journal of Object Oriented Programming, September/October 1989, 11-17.

[Booch83] Booch, G. Software engineering with Ada. Benjamin Cummings, 1983.

[Booch86] Booch, G. Object-oriented development. IEEE Transactions on Software Engineering, 12(2), February 1986, 5-15.

[Booch91a] Booch, G. Object-oriented design with applications. Benjamin/Cummings, 1991.

[Booch91b] Booch, G., and Vilot, M. Object-oriented analysis and design. The C++ Report, 3(8), September 1991, 7-10.

[Borkoles90] Borkoles, J. A flare for the dramatic - Actor 3.0. London University, UK, EXE UK 5(6), November 1990, pp 20,23-24,26.

[Borning82] Borning, A. H., and Ingalls, D. H. H. Multiple inheritance in Smalltalk-80. Proceedings of the National Conference on Artificial Intelligence, 1982, pp. 234-237.

[Cannon80] Cannon H. I. Flavors. Cambridge, MA: MIT Artificial Intelligence Laboratory Technical Report, 1980.

[Cardelli92] Cardelli, L., Donahue, J., Glassman, L., Jordon, M., Kalsow, B., and Nelson, G. Modula-3 language definition. SIGPLAN Notices, 27(8), August 1992, 15-42.

[Cargill90] Cargill, T. A. Does C++ really need multiple inheritance? 1990 USENIX C++ conference proceedings, 1990, pp. 315-323.

[Carre90] Carre, B. The point of view notion for multiple inheritance. ECOOP/OOPSLA conference proceedings, 1990, pp. 312-321.

[Coad91a] Coad, P., and Yourdon, E. Object oriented analysis. Yourdon Press, 1991.

[Coad91b] Coad, P., and Yourdon, E. Object oriented design. Yourdon Press, 1991.

[Coggins90a] Coggins, J. M. Designing C++ libraries. 1990 USENIX C++ conference proceedings, 1990, pp. 25-35.

[Coggins90b] Coggins, J. M. Letters to the editor. C++ Journal, Fall 1990, 54-60.

[Coggins93] Coggins, J. M. Practical principals for library design: Selecting the right abstractions is the key. C++ Report, February 1993, 38-42.

[Cohen91] Cohen, B., Hahn, D., and Soiffer, N. Pragmatic issues in the implementation of flexible libraries for C++. USENIX C++ conference proceedings, 1991, pp. 193-202.

[Coplien91] Coplien, J. Experience with CRC Cards in AT&T. C++ Report, 3(8), September 1991, 1-6.

[Cox84] Cox, B. J. Message/object programming: An evolutionary change in programming technology. IEEE Software, 1(1), January 1984, 50-69.

[Cunningham86] Cunningham, W., and Beck, K. A diagram for object-oriented programs. OOPSLA conference proceedings, 1986, pp. 361-367.

[Dahl70] Dahl, O. J., Dijkstra, E. W., and Hoare, C. A. R. Structured programming. New York: Academic Press, 1972.

[deChampeaux90] de Champeaux, D., Constantine, L., Jacobson, I., Mellor, S., Ward, P., and Yourdon, E. Panel: Structured analysis and object oriented analysis. ECOOP/OOPSLA conference proceedings, October 1990, pp. 135-139.

[deChampeaux91] de Champeaux, D., America, P., Coleman, D., Duke, R., Lea, D., and Leavens, G. Formal techniques for OO software development. OOPSLA conference proceedings, 1991, pp. 166-170.

[deChampeaux92] de Champeaux, D., and Faure, P. A comparative study of object-oriented analysis methods. Journal of Object Oriented Programming, March/April 1992, 21-33.

[DeNatale90] DeNatale, R., Lalonde, J., Leathers, B., and Philips, R. Panel: OOP in the Real World. OOPSLA/ECOOP conference proceedings, 1990, 29-34.

[Dijkstra68] Dijkstra, E. W. The structure of the THE multiprogramming system. Communications of the ACM, May 1968, 341-346.

[Eriksson90] Eriksson, M. A correct example of multiple inheritance. SIGPLAN Notices, 25(7), July 1990, 7-10.

[Fichman92] Fichman, R. G., and Kemerer, C. F. Object-oriented and conventional analysis and design methodologies: Comparison and critique. IEEE Computer, October 1992, 22-39.

[Fontana91] Fontana, M., and Neath, M. Checked out and long overdue: Experiences in the design of a C++ class library. USENIX C++ conference proceedings, 1991, pp. 179-191.

[Friedman91] Friedman, L. Comparative programming languages. Prentice Hall, 1991.

[Gibson91] Gibson, E. Flattening the learning curve: Educating object-oriented developers. Journal of Object Oriented Programming, February 1991, 24-29.

[Goldberg83] Goldberg, A., and Robson, D. Smalltalk-80: The language and its implementation, Addison-Wesley, 1983.

[Griss91] Griss, M. L., Adams, S. S., Baetjer, H. Jr., Cox, B. J., and Goldberg, A. The economics of software reuse. OOPSLA conference proceedings, 1991, pp. 264-270.

[Harrison89] Harrison, W. H., Shilling, J. J., and Sweeney, P. F. Good news, bad news: Experiences building a software development environment using the object-oriented paradigm. OOPSLA conference proceedings, 1989, pp. 85-89.

[Henderson91] Henderson-Sellers B., and Constantine, L. L. Object-oriented development and functional decomposition. Journal of Object Oriented Programming, January 1991, 11-16.

[Horn87] Horn, C. Conformance, genericity, inheritance, and enhancement. ECOOP/OOPSLA conference proceedings, June 1987.

[Johnson89] Johnson, R. comments made by R. Johnson during panel session. Panel: Inheritance: Can we have our cake and eat it, too? OOPSLA conference proceedings, October 1989, pp. 486-490.

[Jones84] Jones, T. C. Reusability in programming: A survey of the state of the art. IEEE Transactions on Software Engineering, September 1984.

[Jossman90] Jossman, P. R., Schiebel, E. N., and Shank, J. C. Climbing the C++ learning curve. USENIX C++ conference proceedings, 1990, pp. 11-23.

[Kempf87] Kempf, R., and Stelzner, M. Teaching object-oriented programming with the KEE system. OOPSLA conference proceedings, 1987, pp. 11-25.

[Korson90] Korson, T. and McGregor, J. D. Understanding object-oriented: A unifying paradigm. Communications of the ACM, 33(9), September 1990, 40-60.

[Korson91] Korson, T., Hazeltine, N., Hilgenberg, T., Philip, R., and Taylor, D. Managing the transition to object oriented technology, OOPSLA conference proceedings, 1991, pp. 355-358.

[Ladden89] Ladden, R. M. A survey of issues to be considered in the development of an object-oriented development methodology for Ada. Ada Letters, 9(2), March/April 1989, 78-88.

[Lee91] Lee, S., and Carver, D. L. Object-oriented analysis and specification: A knowledge base approach. Journal of Object Oriented Programming, January 1991, 35-43.

[Lewis91] Lewis, J. A., Henry, S. M., Kafura, D. G., and Schulman, R. S. An empirical study of the object-oriented paradigm and software reuse. OOPSLA conference proceedings, 1991, pp. 184-196.

[Lieberherr88] Lieberherr, K. J., Holland, I., and Riel, A. Object-oriented programming: An objective sense of style. OOPSLA conference proceedings, September 1988, pp. 323-334.

[Lieberherr89] Lieberherr, K. J., and Riel, A. J. Contributions to teaching object-oriented design and programming. OOPSLA conference proceedings, 1989, pp. 11-22.

[Lieberman86] Lieberman, H. Using prototypical objects to implement shared behavior in object oriented systems. OOPSLA conference proceedings, September 1986, pp. 214-223.

[Linton87] Linton, M. A., and Calder, P. R. The design and implementation of InterViews. Usenix C++ Papers, 1987, 256-267.

[Liskov75] Liskov, B. H., and Zilles, S. N. Specification techniques for data abstraction. IEEE Transactions on Software Engineering, March 1975, 7-19.

[Liskov87] Liskov, B. H. Data abstraction and hierarchy, OOPSLA conference proceedings, October 1987.

[McKenna88] McKenna, J. comments made by J. McKenna during panel session, Teaching OOP, OOPSLA conference proceedings, September 1988, pp. 386-387.

[Meyer86] Meyer, B. Genericity versus inheritance. OOPSLA conference proceedings, September 1986, pp. 391-405.

[Meyer87] Meyer, B. Reusability: The case for object-oriented design. IEEE Software, March 1987, 50-64.

[Meyer88] Meyer, B. Object-oriented software construction. Prentice Hall, 1988.

[Meyer90] Meyer, B. Tools for the new culture: Lessons from the design of the Eiffel libraries. Communications of the ACM, 33(9), September 1990, 69-88.

[Meyer93] Meyer, B. Personal correspondence, 1993.

[Micallef88] Micallef, J. Encapsulation, reusability and extensibility in object-oriented programming languages. Journal of Object Oriented Programming, April/May 1988, 12-36.

[Moo91] Moo, B. comments made by B. Moo during panel session. Issues in moving from C to C++. OOPSLA conference proceedings, 1991, pp. 163-165.

[Moon86] Moon, D. A. Object-oriented programming with Flavors. OOPSLA conference proceedings, 1986, pp. 1-8.

[OShea86] O'Shea, T. , Beck, K., Halbert, D., and Schmucker, K. Panel: the learnability of object-oriented programming systems. OOPSLA conference proceedings, September 1986, pp. 502-506.

[Oxford86] Dictionary of computing. Oxford University Press, 1986.

[Parnas72a] Parnas, D. L. On the criteria to be used in decomposing systems into modules. Communications of the ACM, 15(12), December 1972, 1053-1058.

[Parnas72b] Parnas, D. L. A technique for software module specification with examples. Communications of the ACM, 15(5), May 1972, 330-336.

[Pugh88] Pugh, J. comments made by J. Pugh during panel session. Teaching OOP. OOPSLA conference proceedings, September 1988, pp. 386-387.

[Rao93] Rao, B. R. C++ and the OOP paradigm. McGraw-Hill, 1993.

[Rosson89] Rosson, M. B., and Gold, E. Problem-solution mapping in object-oriented design. OOPSLA conference proceedings, October 1989, pp. 7-10.

[Sakkinen88] Sakkinen, M. Comments on "the law of Demeter" and C++. SIGPLAN Notices, 23(12), December 1988, 38-44.

[Sandberg86] Sandberg, D. W. Smalltalk and exploratory programming. SIGPLAN Notices, 23(10), 1986, 85-92.

[Saunders89] Saunders, J. H. A survey of object-oriented programming languages. Journal of Object Oriented Programming, March/April 1989, 5-11.

[Scharenberg91] Scharenberg, M. E., and Dunsmore, H. E. Evolution of classes and objects during object-oriented design and programming. Journal of Object Oriented Programming, January 1991, 30-34.

[Sedbrook90] Sedbrook, T. Improvisational exercises for hatching objects. Journal of Object Oriented Programming, November 1990, 40-42.

[Seidewitz86] Seidewitz, E., and Stark, M. 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.

[Shaw84] Shaw, M. Abstraction techniques in modern programming languages. IEEE Software, October 1984, 10-26.

[Shilling89] Shilling, J. J., and Sweeney, P. F. Three steps to views: Extending the object-oriented paradigm. OOPSLA conference proceedings, 1989, pp. 353-361.

[Shlaer88] Shlaer, S., and Mellot, S. J. Object oriented systems analysis: Modeling the world in data. Yourdon Press Computing Series, Englewood Cliffs, NJ, 1988.

[Snyder86] Snyder, A. Encapsulation and inheritance in object-oriented programming languages. OOPSLA conference proceedings, September 1986.

[Sincovec84] Sincovec, R. F., and Wiener, R. S. Modular software construction and object-oriented design using Ada. Journal of Pascal, Ada & Modula-2, March/April 1984, 30-36.

[Stemple89] Stemple, D. comments made by D. Stemple during panel session. Panel: Inheritance: Can we have our cake and eat it, too? OOPSLA conference proceedings, October 1989, pp. 486-490.

[Stroustrup84] Stroustrup, B. Data abstraction in C. AT&T BLTJ, 63(8), October 1984, 1701-1732.

[Stroustrup88] Stroustrup, B. What is object-oriented programming? IEEE Software, May 1988, 10-20.

[Ward89] Ward, P. T. How to integrate object orientation with structured analysis and design. IEEE Software, March 1989, 74-82.

[Wasserman89] Wasserman, A. I., Pircher, P. A., and Muller, R. J. An object-oriented structured design method for code generation. ACM SIGSOFT, Software Engineering Notes, 14(1), January 1989, 32-55.

[Wegner87] Wegner, P. Dimensions of object-based language design. Special Issue of SIGPLAN Notices, 22(12), October 1987.

[Whiting90] Whiting, M. A., and DeVaney, D. M. Workshop: Finding the object. OOPSLA/ECOOP conference proceedings, October 1990, pp. 99-107.

[Whiting93] Whiting, M. A. Personal correspondence, 1993.

[Wilson88] Wilson, D. A. comments made by D. A. Wilson during panel session. Teaching OOP. OOPSLA conference proceedings, September 1988, pp. 386-387.

[Wirfs88] Wirfs-Brock, A., and Wilkerson, B. An overview of modular Smalltalk. OOPSLA conference proceedings, September 1988, pp. 123-134.

[Wirfs89] Wirfs-Brock, R., and Wilkerson, B. Object-oriented design: A responsibility-driven approach. OOPSLA conference proceedings, 1989, pp. 71-75.

[Wirfs90a] Wirfs-Brock R., Wilkerson, B., and Weiner, L. Designing object oriented software. Englewood Cliffs, New Jersey, Prentice-Hall, 1990.

[Wirfs90b] Wirfs-Brock, R., and Johnson, R. E. Surveying current research in object-oriented design. Communications of the ACM, 33(9), September 1990, 104-124.

[Wirfs90c] Wirfs-Brock, A. Vlissades, J., Cunningham, W., Johnson, R.,and Bollette, L. Panel: Designing reusable designs: Experiences designing object oriented frameworks. OOPSLA/ECOOP conference proceedings, 1990, pp. 19-24.

[Woodfield87] Woodfield, S., Embley, D. W., and Scott, D. T. Can programmers reuse software? IEEE Software, July 1987, 52-59.

[Wybolt90] Wybolt, N. Experiences with C++ and object-oriented software development. USENIX C++ conference proceedings, 1990, pp. 1-9.

[Yakemovic90] Yakemovic, K. C. B. The bottom line: Using OOP in a commercial environment. OOPSLA/ECOOP conference proceedings, 1990, pp. 93-97.