NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

THE SUITABILITY OF AN OBJECT-ORIENTED LANGUAGE
FOR PROTOTYPING AND ABSTRACT DATA TYPES

by

Michael O. Rowell
III

June 1988

Thesis Advisor: C. Thomas Wu

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

. REPORT SECURITY CLASSIFICATION Unclassified	1b RESTRICTIVE MARKINGS
. SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; Distribution is unlimited
. DECLASSIFICATION / DOWNGRADING SCHEDULE	
PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
NAME OF PERFORMING ORGANIZATION Naval Postgraduate School 6b. OFFICE SYMBOL (*If applicable*) Code 52	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
ADDRESS (*City, State, and ZIP Code*) Monterey, California 93943-5000	7b. ADDRESS (*City, State, and ZIP Code*) Monterey, California 93943-5000
NAME OF FUNDING / SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (*If applicable*)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
ADDRESS (*City, State, and ZIP Code*)	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO / TASK NO / WORK UNIT ACCESSION NO.

TITLE (*Include Security Classification*)
HE SUITABILITY OF AN OBJECT-ORIENTED LANGUAGE FOR PROTOTYPING AND ABSTRACT DATA TYPES

PERSONAL AUTHOR(S)
owell, Michael O.

TYPE OF REPORT	13b TIME COVERED	14. DATE OF REPORT (*Year, Month, Day*)	15 PAGE COUNT
aster's Thesis	FROM _____ TO _____	1988 June	136

SUPPLEMENTARY NOTATION
he views expressed in this thesis are those of the author and do not reflect the official olicy or position of the Department of Defense or the U.S. Government.

COSATI CODES			18 SUBJECT TERMS (*Continue on reverse if necessary and identify by block number*)
FIELD	GROUP	SUB-GROUP	Object-oriented language; Abstract data types; AVL tree

ABSTRACT (*Continue on reverse if necessary and identify by block number*)

ACTOR, an object-oriented follow-on language to Smalltalk, is used in an implementation f a prototype video store management package and in the implementation of a tree abstract ata type package. The language has high power, familiar syntax, portability, windowing nd extensive development tools which make it an excellent choice for prototyping. The ntuitive nature of the language and the close connection between modeling physical and ogical entities are aptly demonstrated by detailed discussion of the design of the video anagement package. Implementation of binary search tree and AVL tree abstract data types emonstrate that coding of constructs which are procedural in nature does not exploit the trengths of an object-oriented language such as ACTOR.

20 DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22 NAME OF RESPONSIBLE INDIVIDUAL Prof. C. Thomas Wu	22b TELEPHONE (*Include Area Code*) (408) 646-3391 22c OFFICE SYMBOL Code 52Wq

D FORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

Approved for public release; distribution is unlimited.

THE SUITABILITY OF AN OBJECT-ORIENTED LANGUAGE FOR PROTOTYPING AND ABSTRACT DATA TYPES

by

Michael O. Rowell
Captain, United States Marine Corps
B.S., Michigan State University, 1978

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

June 1988

ABSTRACT

ACTOR, an object-oriented follow-on language to Smalltalk, is used in an implementation of a prototype video store management package and in the implementation of a tree abstract data type package. The language has high power, familiar syntax, portability, windowing and extensive development tools which make it an excellent choice for prototyping. The intuitive nature of the language and the close connection between modeling physical and logical entities are aptly demonstrated by detailed discussion of the design of the video management package. Implementation of binary search tree and AVL tree abstract data types demonstrate that coding of constructs which are procedural in nature does not exploit the strengths of an object-oriented language such as *ACTOR*.

TABLE OF CONTENTS

LIST OF FIGURES

I. THE RESEARCH PROBLEM

A. INTRODUCTION

Computers are intricate tools built by man for his ultimate benefit. During the course of their evolution however, they have become tools which force their users to operate exclusively on the computer's terms. These terms, although well suited to machines, can be quite unnatural for humans. Often, a person wishes to model a real world activity with a computer program. This task should be feasible without having to resort to the peculiar and unfriendly logic typical of computer modeling applications. When computer technology was immature, its users were primarily limited to those with intimate knowledge of the internal workings of both the hardware and the software which they employed. These individuals as a group understood the reasons for the contorted way of doing business with these strange and demanding pieces of high technology.

Today, sophisticated computer users have been joined by many individuals who wish to use the power of computer technology without having to understand any of the details of the action taking place behind the scene. This is a reasonable desire. Why should a human be forced to think in the same manner that allows most high-level programming languages to accomplish their tasks? Isn't there a programming environment which is a natural extension of the way the human mind models everyday situations in the course of problem solving? The research contained in this thesis is the result of a desire to explore the utility of a computer language which departs from the procedural orientation of traditional languages. Specifically, the study focuses on an approach to programming which is known as the object-oriented paradigm. Languages of this type are members of the fifth generation of computer programming languages and,

as such, are the result of much evolution [Ref. 1]. The commercial implementation of this object-oriented approach used for the programming done in this thesis is the language *ACTOR* [1].

B. BACKGROUND

One of the original motivating forces behind the development of object-oriented languages was the desire to develop a simulation and graphics-oriented programming environment which would make computers accessible to nonspecialists. It was realized that persons who did not possess a background in programming languages benefited from a richly interactive environment making use of graphics. Non-expert users would also find programming an easier task if there was a close correspondence between the physical entities which were to be modeled and the logical entities which actually performed the modeling within the realm of the programming language. With these ideas as goals, object-oriented programming was first implemented in the language Smalltalk in 1972 at the Xerox Palo Alto Research Center. Smalltalk took some of its most important fundamental concepts from the language Simula, which was designed in the 1960's for performing simulations. Many of the features of Smalltalk's graphically oriented interface were based on the language LOGO designed at MIT. [Ref. 1]

Just what is it that makes Smalltalk and other object-oriented languages unique? Their basis is the creation and management of *objects.* The characteristic which makes these languages well suited to simulation and modeling applications is that these objects are designed to closely resemble the physical objects which are to be modeled. It is not necessary for the programmer to force himself to think in terms of data structures and commands, but rather he can develop the program logic more in terms of the attributes of

---

[1]ACTOR is a registered trademark of the Whitewater Group, Inc.

these objects and the instructions to which they respond. In effect, an object is a package consisting of relevant data together with the instructions which act upon those data. When a real world object receives an "instruction" to perform a specific action, the ability to understand the instruction and carry it out are encapsulated within the object. An illustration of this is an automobile "knowing" exactly how to respond when an instruction is received via steering wheel movement. The "knowledge" of how to respond to specific instructions is built into the automobile by the designer. In a similar way, an object in Smalltalk "knows" how to correctly respond to an instruction sent to it by calling upon the knowledge contained in special *methods* which are an integral part of the object.

The linking of data and procedures into objects is a significant departure from procedural languages which generally treat data and the procedures that operate on the data as two separate things. There are important benefits which arise from this encapsulation of data and methods into one entity. One of the main advantages of this arrangement is the reduction of program debugging and maintenance time because of information hiding. MacLennan [Ref. 1] defines the incorporation of the information hiding principle as a language which permits modules designed so that (1) the user has all of the information needed to use the module correctly, and nothing more; (2) the implementor has all of the information needed to implement the module correctly, and nothing more. The methods which an object uses to perform tasks in response to instructions are completely encapsulated within the object. The user knows the name of the method and what the end result will be, but he knows nothing of the internal details. How the method accomplishes what it guarantees is hidden from the user. They only know the proper way to invoke the method and the form in which to expect the returned results. The beauty of this design feature is that it allows the internal workings of an object to be changed at will by the implementor. As long as the changes are hidden from

the user and the ways that the user communicates with the object remain unmodified, the external behavior of the object does not change.

Another important benefit of object-oriented languages accrues from the relationship between different types of objects. Objects can be thought of as specific instantiations of general *classes* of objects which define exactly what properties that object possesses and how it responds to various instructions or *messages*. These messages are much like procedure calls in a procedural language except that they are more general: the same message can be sent to objects of many different classes. The response to the message depends upon the details of the method contained in the class to which the object belongs. Classes have a very specific hierarchical relationship with one another which defines the behavior of objects within the class. All of the characteristic behavior of an ancestor class is inherited by any class which has a descendant relationship with it. In other words, objects of a descendant class need only contain methods needed *in addition* to those inherited from the ancestor class. All of the methods of the parent class are automatically available to the descendant class. No redefinition of methods is required. This saves significantly on the amount of code in an application and is a good example of the abstraction principle: avoid requiring something to be stated more than once; factor out the recurring pattern [Ref. 1]. Thus it can be seen that an object-oriented language makes good use of data abstraction. An *ACTOR* object, by nature, is quite an autonomous entity and is most useful when allowed to manage its own behavior via its internal data and methods.

C. RESEARCH OBJECTIVES

The main objective of this research is to characterize the effectiveness of an object-oriented language when used for prototyping and the development of abstract data types. The *ACTOR* language was chosen for the implementation portion of the research

because it is well documented, well-supported, portable and has syntax which can be readily mastered. In using *ACTOR* to accomplish the main objective of the research, the strengths of the language are exploited and areas where the general paradigm has limitations are exposed. The goal throughout the implementation of the prototype and abstract data types is to depart from the traditional procedural style of design and development of programs and to demonstrate that the object-oriented concept is a unique, useful and powerful approach to programming.

D. INVESTIGATIVE QUESTIONS

Specifically, this research attempts to answer two investigative questions:

1) How natural is the connection between a group of interrelated physical objects and the logical objects which are developed to model them in the *ACTOR* environment? In other words, is an object-oriented language a good choice for a modeling or simulation task to be performed by someone without significant programming experience?

2) Is an object-oriented language such as *ACTOR* a suitable tool to use in the development of abstract data types? Specifically, can its strengths be exploited in the development of a set of classes designed to model the behavior of both ordinary and height balanced binary search trees?

These questions are answered through the implementation and examination of *ACTOR* applications which serve to illustrate the strengths and limitations of the object-oriented paradigm.

E. ORGANIZATION

The remainder of this thesis is divided into four chapters. Chapter II is an introduction to the object-oriented paradigm through the illustration of the *ACTOR* language. The background and motivation of this type of programming are presented and the salient features of the language are discussed. *ACTOR* is also compared with procedural languages with which the reader may be more familiar.

Chapter III presents the author's experiences in developing a high-level application in *ACTOR*. The goal of this portion of the research is to explore the feasibility of rapidly prototyping a package which could be used to manage a video store. Note that rapid prototyping involves the development of a software package which exhibits some, but not necessarily all, of the desired capability of a final version of an application. Prototypes are developed to demonstrate the feasibility of a concept and to reduce the risks involved with certain approaches to the implementation of the application [Ref. 2]. Techniques peculiar to the object-oriented design philosophy and the exploitation of the strengths of the approach are discussed. Particular attention is paid to the modeling of the application. The completed package is also compared with a functionally similar version of the application written in Pascal.

Chapter IV describes the implementation of a binary tree package in *ACTOR*. The aim of this section is to study the practicality of using an object oriented language such as *ACTOR* for implementing a low-level family of abstract data types. Development of a binary search tree type and an AVL, or height-balanced, tree type are detailed. The topic of AVL trees can be found discussed in depth in Standish [Ref. 3]. Their development and use are then analyzed.

Chapter V contains the author's conclusions as a result of the work performed and described in Chapters III and IV. It also contains recommendations for future applications in the object-oriented area of programming. The computer code developed during the course of this thesis work is contained in Appendices A through C.

II. THE ACTOR LANGUAGE

A. THE OBJECT-ORIENTED PARADIGM

Chapter I briefly introduced some of the more important concepts of object-oriented languages in general. This chapter illustrates and expands those concepts by examining the specific details of *ACTOR*. There are two important concepts which should be kept in mind throughout this chapter. The first is that in *ACTOR*, virtually *everything* is an object [Ref.4:p. 31]. This includes classes and the language application package itself. Secondly, every action which occurs in *ACTOR* is the result of sending a message to an object, which responds to that message by executing a method [Ref.4:p. 31]. *ACTOR* objects have an active nature. They encompass not only data but the capacity for various manipulations of their data. These concepts should become clearer as the reader gains a more thorough understanding of the language philosophy and characteristics. [Ref. 4]

In general, there are three characteristics which a language must possess in order to be considered object-oriented. It must exhibit information hiding through the use of encapsulation of data and instructions, dynamic binding and inheritance. Some languages which have one or two of these characteristics have been improperly referred to as object-oriented. However, to be a true member of this language category, all three characteristics must be present. [Ref. 5]

1. Encapsulation

An object-oriented language encapsulates both data and instructions, or methods, within one entity, an object. The amount and diversity of information which is held as a unit in the object package is much greater than that held in a procedural data structure. Code which consists of object manipulations via messages results in a package of autonomous entities which are responsible for their own internal management. This is

incorporation of information hiding. The information hiding principle, as previously defined in Chapter I, leads to the reduction of the interdependencies between various portions of a software package. Neither the user nor the implementor of a package have complete details: the user lacks the knowledge of the implementation details while the implementor lacks the details of the context in which the package is to be used [Ref. 1]. The result of this limit on information distribution is that code is more reliable and more maintainable. Object-oriented languages accomplish information hiding by dictating that objects hold the state of computation. An object uses both the details of its methods and the contents of its internal named variables, *instance variables*, to determine its behavior upon the receipt of a message. These variables are discussed in more detail later in this chapter. All of the information relevant to this behavior is contained within the object. The object may be relied on to respond consistently upon receipt of a given message. The design of most object-oriented languages, including *ACTOR*, discourages the internal manipulation of an object's details by the user. If the implementor is careful in writing their code, the internal implementation details of an object can be changed by the implementor without adverse effects on the user's code. This characteristic helps to promote greater maintainability of object-oriented code.

2. Dynamic Binding

Programmers experienced with procedural languages are often dismayed to discover that type declarations are not required when declaring the names of variables in an object-oriented language. Although it may seem that such languages are not enforcing type checking, in reality, they are. *ACTOR* and all true object-oriented languages use what is known as *dynamic binding*. This means that no objects in the language are typed and any object can be bound to any name [Ref. 1]. The actual type checking occurs when a message is sent to an object bound to a particular name. If that object is capable of responding to the message, it does so and the message is considered to be legal.

Otherwise, the object does not respond, the message is illegal and dynamic type checking has been enforced. It is important to realize that dynamic checking does not constitute weak typing. Weak typing is the allowance of one type where another is expected [Ref. 1]. *ACTOR* is able to perform type checking as described at run time instead of compile time and still perform strong typing. In other words, type abstraction is enforced, but it is done at a time and in a manner which is transparent to the user. An important benefit of dynamic typing is the flexibility it offers as a result of being able to pass a message to any object which has the ability to understand the message. This feature is known as *polymorphism* since the same message can elicit a different response depending on the class of the receiver [Ref. 6]. Polymorphism facilitates the rapid building and expansion of an object-oriented software package because it can eliminate much of the control structure which tends to clutter and complicate procedural code.

3. Inheritance

One of the main reasons for the power of an object-oriented language is the third characteristic property of such languages, inheritance. Inheritance, as previously discussed in Chapter I, refers to the relationship between the classes of the language. Classes are arranged hierarchically in what is known as the *class tree*. The most generic classes are located near the top of the tree while those with more detailed features and specialized functions are located further down the tree hierarchy [Ref.4:pp. 118-119]. An example of a portion of the *ACTOR* class tree is given in Subsection D of this chapter as Figure 2. Each class in *ACTOR* has only one direct ancestor, an arrangement known as *single inheritance* [Ref.4:p. 39]. All objects inherit both methods and instance variables from their ancestors. If an object is sent a message invoking a method which it defines internally, this method is used just as expected. But what happens if an object is sent a message which references a method not defined in the class of the object? A search is

begun up the class's ancestral path looking for the method in question. The *ACTOR* system will search the immediate ancestor of the class and then the ancestor of that class and so on until either the method is found or the top of the tree is reached. If the method has not been located at this point, an error will be generated. Instance variables are also inherited from an ancestor class to its descendants. The importance of this feature is that a descendant class only need define instance variables which are required in addition to those that are already defined by each of the ancestors of the descendant class. Of course, care must be taken to ensure that instance variables or methods are not inherited from an ancestor when they may represent inappropriate properties or behavior in the descendant. The avoidance of inappropriate inheritance is best accomplished by defining a new class as a descendant of the most general predefined class which offers the basic features required by the new class.

B. THE ACTOR PHILOSOPHY

The designers of *ACTOR* made a number of specific decisions about the language environment. The language runs within the Microsoft Windows[2] programming environment and is specifically limited to the 640 kilobyte memory boundary accessible by DOS. Because of these features, it is tailored for usage on an IBM compatible PC with at least 640 kilobytes of memory and a hard disk. Due to these memory space and hardware limitations, *ACTOR* must be designed to run efficiently. To utilize and reclaim memory space in an efficient manner, *ACTOR* uses both an incremental dynamic memory garbage collector and a static memory garbage collector. The dynamic garbage collector manages space allocated to temporary objects and its operation is transparent to the user. The static garbage collector reclaims space used when methods are recompiled by the

---

[2] Microsoft Windows is a registered trademark of the Microsoft Corporation.

user. It must be specifically invoked by the user as required to maintain sufficient space for the compilation of methods. [Ref. 4]

C. LANGUAGE CHARACTERISTICS

1. User Interaction

Now that the characteristics of object-oriented languages in general have been explored, the focus shifts to the specifics of the *ACTOR* environment. *ACTOR* is a highly interactive language which provides immediate and continuous feedback to the user. To accomplish this, it incorporates three primary tools known as the *browser*, the *inspector* and the *debugger* [Ref. 4]. The browser is the primary viewing mechanism available in *ACTOR*. It allows the programmer to examine, edit and add to the *ACTOR* source code. It is a specialized file editor which manipulates the source files which contain the details of the *ACTOR* classes, keeping the programmer apprised of the changes made to the source code package. The inspector facilitates the checking of any object's instance variables or component elements and the identification of the object's class at any point in the programming task. The *ACTOR* debugger is an interactive correction device which not only gives diagnostic error information but also allows the user to examine the site of the error, correct it and resume execution at the point where the error was detected.

With the advent of the Macintosh computer and numerous window managing software packages, users have come to expect such environments to provide certain features. Among these are pull-down menus, popup windows, iconic logic, dialogs and the management of graphics. Microsoft Windows provides all of these expected features and, since *ACTOR* resides within the Microsoft Windows environment, it uses predefined classes to perform such functions as windowing, displaying text and repainting the display screen in appropriate ways. The interaction between Microsoft Windows and *ACTOR* is a very natural one. The message passing paradigm of an object-oriented

language is easily extended to interact with a windowing operating system. Microsoft Windows uses messages of a predefined type which are sent directly to windows to cause the occurrence of a particular event. *ACTOR* defines windows as objects which of course are capable of receiving and responding to messages using the methods to which they have access. *ACTOR* window objects are also designed to successfully receive and comply with messages sent by the Microsoft Windows system. The effect of this is that *ACTOR* window objects are event driven entities, inactive until a message arrives directing that the object perform a particular operation on itself or cause other objects to perform appropriate actions [Ref.5:p. 2.6.1]. The event-driven nature of object-oriented programming provides program control built into the language and it models very closely the behavior of the windowing environment.

2. Messages

Messages make up a large portion of *ACTOR* code and accomplish the majority of the work of the language. To invoke a particular method of an object, a message of particular format is sent to the object. The message name, called the *selector*, corresponds exactly to the name of the method in the receiving object. This receiving object is known as the *receiver* [Ref.4:p. 137]. The most common syntax used for *ACTOR* messages puts the selector first followed by parentheses containing the receiver and any optional arguments to the message. Here is a typical example of this message syntax:

```
printString(aTW, title)
```

In this example, *aTW* is the receiver and *title* is the only argument to the message. Common *ACTOR* convention is to begin method names, message names and variable names with small letters and, if the names consist of more that one word, to concatenate

the words and begin the second and later words with capital letters. Variable names should always be as descriptive as possible and not misleading. Using a familiarity with the *ACTOR* classes, it can be surmised that aTW represents a Text Window object. The number of arguments in the message must always match the number of arguments in the method being invoked.

It is tempting to think of the *ACTOR* message as analogous to a Pascal procedure call since the syntax of the two is similar. There are important differences, however. First, the leftmost expression within the parentheses of the *ACTOR* message is the object that the message is being sent to, not an argument to a procedure. Second and more important, an *ACTOR* message is an instruction to an object to perform a task upon itself, not the invocation of a function. Third, all messages have a return value associated with them. This value may be ignored in some cases, but in others it must be specifically dealt with.

There are two special cases where the format given in the previous example is not used. One case is the use of infix format. *ACTOR* allows common infix format to be used for such things as arithmetic operations and the concatenation of strings. Hence, both of the following are legal *ACTOR* expressions:

```
67 + 41          "a + string"
```

It should be noted that the first of these expressions is interpreted as "send to the integer object 41 a + message with 67 as the only argument". Another exception to the standard format of messages involves the use of *early binding* [Ref.4:p. 315]. Recall that the type of the receiver of an *ACTOR* message is not determined until run time. There are several situations where it is desirable to specify the class of the receiver when the message is

written, before it is executed. The syntax used for this case is very similar to a procedural type declaration:

aTW : TextWindow

A discussion of uses of this concept together with examples is deferred until Chapter III.

3. Methods

A method in *ACTOR* may be thought of in the same way that a subroutine or procedure is in a procedural language. Figure 1 shows an example of a short method. In the figure, the keyword *Def* signifies the start of the definition of a method. Immediately following the Def is the method name. The word *self* inside of the parentheses serves as a placeholder for the receiver of the message. In the method definition, self stands for an instance of the class to which the method belongs. The keyword self must be present in the header of all method definitions. It is followed by any arguments passed to the method in the invoking message, a " | " and any local variables which are used within the scope of the method. The " { " and " } " parentheses define the limits of the block which

---

```
Def addCust(self,name | cust)
{
cust := new(Customer);
buildCust(cust);
add(customers,name,cust);
^name;
} !!
```

Figure 1. An Actor Method

---

constitutes the method. The " ^ " character, if present in the body of the method, is followed by the value to be returned to the caller of the method. If there is no occurrence of " ^ " in the body of the method, the current value of self is returned to the caller. Any number of returns can be used within a message at different points and in different branches of the logic. When a return command is encountered, the remainder of the method is not executed. The double exclamation mark at the end of the method is the *chunk* mark. *ACTOR* uses this as a delimiter in its source files for purposes of loading files into the browser and inspector [Ref.4:p. 224].

The *new* method, seen in the previous figure, is worthy of special attention. This method is the normal manner in which to bring about a new instantiation of a particular class. The new method is unique in that it is a message sent to a *class* which in effect tells that class to make and initialize a fresh copy of itself. Recall that in the case of all other messages, the receiver is an object. This is why the new method is referred to as a *class method* and all other methods, since they are sent to objects, are referred to as *object methods*. The default behavior of the new message is available to all classes. However, when the user writes specialized or custom classes, there are two situations where it is appropriate to redefine the new method within the custom class. The first situation is where the new method must take a different number of arguments than the new method in the ancestor class. The second case which requires a redefinition of the new method is where it is desired to have the object initialized to a value other than nil. Examples of these cases can be seen in Chapter III. [Ref.5:pp. 2.3.1-2.3.5]

4. Variables

ACTOR employs three types of variables: global, instance and temporary variables. Global variables in *ACTOR* behave just as in other languages. They are system variables and are visible to the application as a whole. The well known dangers of side effects associated with global variables apply to *ACTOR* globals. It is convention to

capitalize the names of all global variables. Instance variables have been previously introduced. They are named packets of data that are carried around with each instance of a particular class [Ref.4:p. 542]. As everything else in *ACTOR*, they are themselves objects which can belong to any class. Instance variables are recognized by reference to their names within the scope of the method to which they belong. Outside of this scope, they may be referenced by giving the name of the object which they are a part of followed by a " . " and then the name of the instance variable, e.g., Customer.name. Care must be used in using this form of reference to avoid side effects similar to those associated with global variables. Instance variables can also be used to hold objects of any class desired. Even though *ACTOR* classes can inherit behavior from only one ancestor, the functionality of other classes can also be incorporated by letting an instance variable hold an object of that class [Ref.5:pp. 2.4.3-2.4.4]. Temporary variables are of use only during the execution of a particular method and can be of two types. The first type are arguments which are passed via message to the method. These arguments are passed by value. This means that a local copy of the variable is made for the use of the method so that any manipulations performed on it within the method only apply locally. The second type of temporary variables are those which are defined within the method only, to the right of the " | " in the definition header line.

Of special note is the treatment of the boolean values true and false in *ACTOR*. The *only* object in the system which is logically false is the object *nil*. The nil object is an instance of the class NilClass. *Everything* else beside nil in *ACTOR* is logically true. This includes the number zero and the class NilClass. When an object is first created and not yet assigned a value, it has the value of nil or false. [Ref.4:p. 55]

5. Syntax

ACTOR syntax was purposely designed to bear strong resemblance to both C and Pascal syntax. The difference between upper and lower case in names is important in *ACTOR*, as discussed in the previous section on variables. The language contains most of the familiar control structures including if and if/else conditionals, assignment statements using the ":=" operator, select or case statements, an enumeration construct and a generic loop which may be structured to behave as a while-do type or a repeat-until type of loop. There is a special *do* method which is available to all collection objects. The details of the Collection class are explained later in this chapter, but for now it can be thought of as analogous to the Pascal array or record type. The do method has the following syntax:

```
do(AnArray,
    { using(element)  print(element);
    });
```

The effect of this statement is to map the instructions between the "using" and the "});" across the elements of the Collection. The syntax for *ACTOR* loops is as follows:

```
loop <statement list>
while <expression>
[begin]
<statement list>
endLoop;
```

The effect of a while-do loop is obtained by leaving the first <statement list> out and the repeat-until logic can be effected by leaving out the second <statement list> and the "begin" construct [Ref. 4].

D. CLASSES IN ACTOR

1. Overview

When first installed on a host machine, the *ACTOR* system has nearly 80 predefined classes [Ref.4:pp. 118-119]. The top few layers of this class hierarchy are depicted in Figure 2. The most important feature to notice about this class tree is that

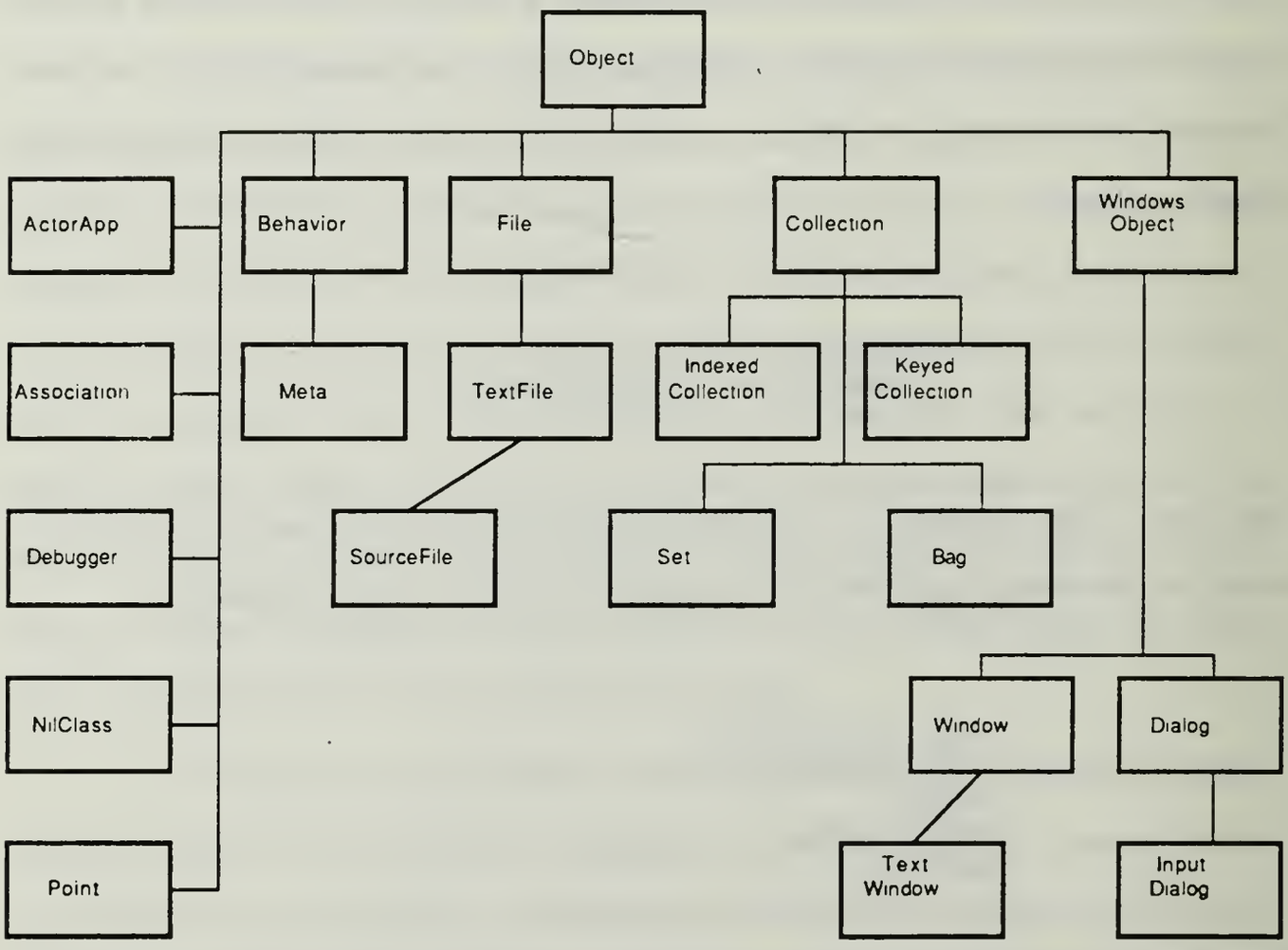

Figure 2. The Actor Tree of Predefined Classes

every class is a descendant of the Object class. Since the Object class contains elements which form the basis of every object in *ACTOR* and everything in the language is an object, it is natural to find that all classes descend from the Object class. This is the technique used to ensure that all classes have access to the most basic methods needed by all objects. The classes examined in the following sections do not include all of the classes in the tree but do focus on the primary ones used for the implementation portion of this thesis.

The philosophy of the *ACTOR* implementation on an individual machine is somewhat different than conventional languages. The predefined classes constitute only the beginning skeleton of the *ACTOR image*. The image consists of all predefined classes which come with the language as distributed by the vendor and all modifications made by the user. *ACTOR* applications are constructed by defining new classes and adding to those already in existence. When the user is satisfied that their new modifications and additions have resulted in a workably correct system, a *snapshot* is taken. This serves to freeze the current language image, however modified, as the image to be used as the language master [Ref.4:p. 78]. There are both flexibility and inherent danger involved in this approach to the maintenance of the current state of the language implementation. The language can be quickly and easily tailored to meet the needs of the user. However, unforeseen side effects can occur if a snapshot is taken of a system which has been modified without being fully debugged. To remedy this mistake, the user has a number of alternatives. The simplest is to remove the offending methods and take a snapshot to redefine the system without the recently introduced errors. Another way to correct the ill advised snapshot is afforded by the automatic maintenance of a backup copy of the system baseline as it existed at the previous snapshot. This version of the system can easily be made the current version and work can start again from that intermediate point.

In the worst case, the user can always start over with the basic package as originally distributed.

2. Collection Classes

An *ACTOR* collection is an object which contains a group of other objects. The Collection class descends directly from the Object class, as can be seen from Figure 3 [Ref.4:p. 118]. The Collection class itself is just a formal class. This means that it is unlikely that objects of type Collection will be created since the primary purpose of the class is to act as a unifying ancestor for all of its various descendants. Some of these descendants can also be seen in Figure 3. The classes contained in this portion of the *ACTOR* class tree are very powerful and useful for building complex objects. All descendants of the Collection class have two types of data: *named* and *indexed*. Named data are held in the instance variables of the object while indexed data are held in the elements of the collection themselves [Ref.4:p. 156].

a. The Array Class

The Array class, as a descendant of the Indexed Collection class, is probably the most familiar to the reader with procedural language experience. In *ACTOR*, an array object is created with a new message which specifies the desired size of the array, e.g., new(Array,10). Once an array has been created, it is of fixed size and the number of elements which it contains cannot be increased. The property which makes *ACTOR* arrays different from those in other languages is their ability to hold objects of any class. The classes can even be mixed within the same array, allowing, for example, a single array to contain objects from the Real, Integer, String, Text Window and Array classes. *ACTOR* arrays are indexed from 0 to $n - 1$ where n is the number of elements in the array. Unlike other collections in *ACTOR*, array elements can *only* be referenced by their index position [Ref.4:p. 164]. Of special note in the Array class is the tuple method. This method is used for creating a new array at run time where the array elements can be

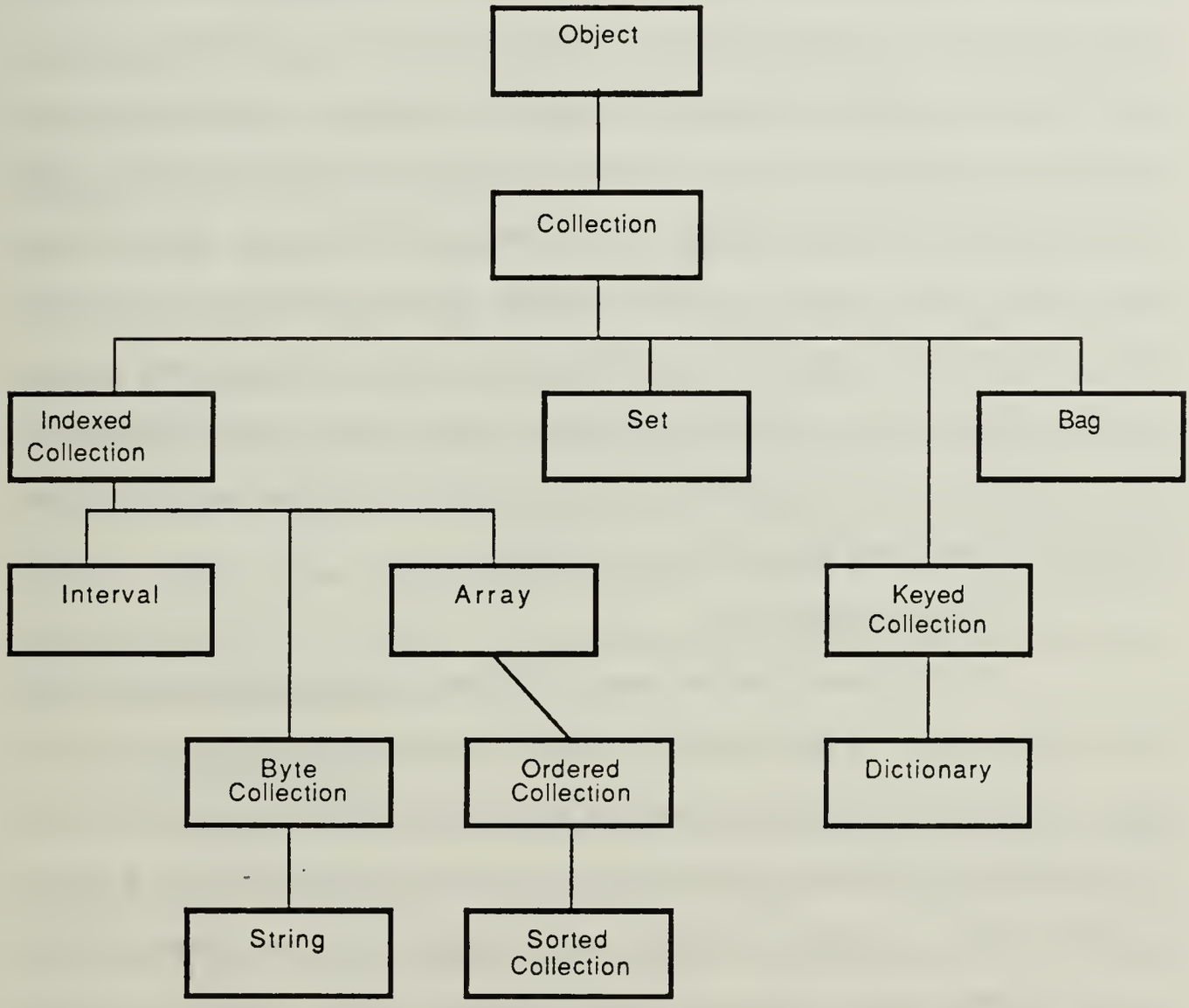

Figure 3. The Collection Subtree

either literals or variables [Ref.4:p. 167]. An example of this can be found in Appendix A in the section on the Customer class.

b. The Ordered Collection Class

The Ordered Collection class is a direct descendant of the Array class. It differs from the array in that it has a variable size. It can dynamically allocate additional space if it is tasked to hold more elements than it has room for. An ordered collection also allows more flexibility in accessing its elements. Access can occur by index, just as with arrays. In addition, elements can be added to or deleted from the beginning or the end of the ordered collection. Because of this behavior and the methods which the class provides, both stacks and queues are easily simulated. The push method adds the desired object to the stack or queue, the pop method removes the last object pushed and the removeFirst method removes the object which has resided in the ordered collection the longest [Ref.5:p. 1.6.5]. The Ordered Collection class is a good example of the flexibility of the collection classes within the *ACTOR* system.

c. The Dictionary Class

The Dictionary class, as shown in Figure 3, is a direct descendant of the Keyed Collection class. A dictionary is a collection of *association* objects which bundle together a key and a value associated with that key [Ref.4:p. 209]. Access to an element in a dictionary object is normally most convenient when performed via the key. Both the key and the value, as expected in *ACTOR*, can be objects of any class. Many different keys in a dictionary object can be associated with the same value. However, each key itself must be distinguishable from every other key in the dictionary. The implementation package described in Chapter III makes frequent use of dictionary objects and the *do* enumeration method previously described in this chapter.

3. The Window Class

The Window class is a formal, unifying class whose descendants perform all of the display functions built into the *ACTOR* environment. It contains many methods which serve as the communications link to Microsoft Windows. The effect of the methods in this class is to send requests to Microsoft Windows to create, update and manage all window objects which *ACTOR* desires to manipulate [Ref.4:p. 233]. Windows can be either of the parent or child type. A parent window is an independent object which maintains control over all its children windows. Typically, children windows are used to split the parent window into smaller, more manageable pieces such as menu areas or scroll boxes [Ref.4:p. 236]. There are many details involved in the way that *ACTOR* manages its windows which are not of concern for the purposes of the research involved in this thesis. The main descendant of the Window class used in this research is the Text Window class. The purpose of objects of this class is to display textual output to the user within a window. These windows operate in only one direction: text may be displayed but keyboard input is not accepted. The implementation of Chapter III makes extensive use of the Text Window class.

4. The Dialog Class

The Dialog class and its descendants constitute the methods used to initiate communication with the user and to receive their responses. The appearances of dialog objects are similar to popup windows which appear superimposed on top of existing windows on the user's screen. They are used to notify the user of unusual occurrences, errors, messages and to solicit user input. There are two broad categories of dialogs, *modal* and *modeless*. A modal dialog takes control of the application when it is created and normally requires some user action such as clicking on "OK" to regain control. A modeless dialog will not take over from the application but will allow the various other windows and menus to continue to function [Ref.4:p. 287]. Dialogs can be defined to

have many combinations of *buttons* such as "OK", "YES", "NO" or "Cancel". These are small areas on the screen designed to detect user activation of a mouse button when the pointer is located inside the button. There are two ways to create the dialogs to be used by an application [Ref.4:pp. 288-302]. The first involves defining *templates* which hold the patterns to be used for the dialogs, menus and icons. These definitions are held in an ASCII file called a resource script file. The second way to create dialogs is at run time through the invocation of objects of the Input Dialog class. This method is better suited to a prototyping application and is employed and discussed in the development of the implementation package of Chapter III.

E. OBJECT-ORIENTED DESIGN

An integral portion of a good object-oriented program is a good object-oriented design. Although it bears some similarities to both conventional design processes and standard systems analysis, object-oriented design has unique steps and considerations. Viewed from the top, it begins with the familiar task of determining system requirements. From that point, however, it progresses through specific steps which diverge from the standard design of computer programs.

Designing an application package in *ACTOR* generally involves six distinct steps [Ref.5:pp. 2.4.1-2.4.5]:

1) Establish system requirements

2) Model the real system

3) Determine the logical entities

4) Review the existing classes

5) Define new classes

6) Define public and private protocols

1. Establish System Requirements

At the outset of an object-oriented design, one must ask two basic questions: (1) How should the system appear to the user? and (2) How does the designer want the system to work? Several diagnostic questions can help to clarify the answers to these two basic questions. Of primary concern is the overall plan for the user interface. Should the application have one main parent window and a number of popup children windows as required, or are several parent windows more appropriate? Will communication with the user be best served with textual or graphical output or a combination of both? What degree of interaction is expected of the user? How can the system be designed so that its performance is compatible with the expectations of the typical user population? The answers to these questions serve to clarify the goals of the system and provide a basis for evaluation of the completed design.

2. Model the Real System

One of the most intuitive characteristics of object-oriented design is the close correspondence between entities in the physical world and entities in the program environment. *It is important at this point that the designer think in terms of entities, not in traditional terms of data structures and procedures* [Ref.5:p. 2.4.1]. For this reason, it is often found that people with no programming experience or preconceived biases are more successful at this initial modeling than those with significant procedural programming experience. If the problem being modeled involves obvious physical objects, the division of the application into these entities and ultimately into classes is straightforward. Sometimes, however, more thought is necessary to partition an abstract problem domain into distinct entities. For example, what would be an appropriate way to divide into entities the tasks which a pilot performs in the course of a flight? These tasks have no physical counterparts which can be modeled directly. From the author's

experience, an appropriate breakdown might be to consider tasks associated with aviating, those with navigating, and those with communicating. "Objects" in these three categories have natural relationships with one another and share many common aspects. Further subclasses of these three could be defined to encompass greater levels of detail. The goal is to partition the domain space into entities in a way which makes good sense to someone well versed in the field being modeled.

3. Determine the Logical Entities

This step in design is often quite complicated when building an application in a procedural language. *ACTOR*, on the other hand, offers a simple transition from the physical model to the logical model. The main difference between the two is that the designer must focus more on computer concerns when formulating the logical model. He should explore how the objects will be maintained and how they will relate to one another. Also of concern is the method of referencing the objects, that is, what will be the keys used for storage and retrieval of information from the objects? The goal of this step is the determination of the various classes which will be needed to make the system function as desired. [Ref.5:pp. 2.4.1-2.4.2]

4. Review the Existing Classes

Previously in this chapter, an abbreviated version of the *ACTOR* class tree was given. The complete class tree contains nearly 80 predefined classes of diverse description and hierarchical location [Ref.4:pp. 118-119]. It is essential that the designer familiarize himself with all of these classes for three reasons. First, the functions to be provided in the classes to be built may already exist within the standard classes. In this case, there is no need to redefine capabilities which already exist in the predefined language package. Second, a knowledge of the class tree is necessary to determine where to locate the classes which are to be defined. Proper inheritance is the main concern here. This subject is discussed in further detail in the next section. Finally, it is

necessary to know the features of each class so that if needed, they may be incorporated into a class to be written through either methods or instance variables.

5. Design New Classes

This step in the design constitutes the backbone of the implementation task. It involves transforming the logical entities formed in step (3) into well defined classes. For each class, the designer must determine its ancestor, its instance variables, and define its methods. There are two criteria by which classes can be constructed and evaluated for effectiveness. Most importantly, each class to be designed should have a single and clear function which should be largely apparent from its name. Secondly, all classes should be moderate in size. An unwieldy class size is an indication that a further logical division may need to be considered.

The basic decision which faces the designer at this point is where to place the new classes in the language hierarchy. The decision boils down to two alternatives. Either the class should be placed near the root of the class tree as a close descendant of the Object class or it should be placed well down in the tree hierarchy as the descendant of an appropriate class and as a distant descendant of Object. The guiding principle in making this decision is the wise use of inheritance. The designer needs to ask himself whether an object of the class which he is defining is fundamentally similar to an object of an existing class or whether it needs to borrow just a small portion of the behavior of that existing class. If it seems that the new class has much in common with the particular existing class in question, then it is probably a good idea to make the new class a descendant of the class in consideration. However, if the new class only needs to borrow small details, it is probably better to inherit that behavior through the use of an instance variable. That is, an instance variable in the new class can be caused to hold an object of the existing class, thereby gaining the functionality of the existing class without being a descendant of it.

Both strategies of hierarchical location of a new class have their strengths and weaknesses. If it is decided that the new class should have an ancestor which is specialized and distant from Object, the designer must make sure that the inherited methods and instance variables make logical sense. Inappropriate behavior inadvertently built into a class of objects may only be discovered later when it manifests itself in the form of nasty side effects. The main advantage in building a class from a specialized existing class is that most of the new class's functionality is inherited so that only a small amount of new code will have to be written. Generally, it is safer to build a class as a close descendant of the Object class because inappropriate inheritance is minimized [Ref.5:p. 2.4.4]. Also, if the concept of the new class is unique, it will make more logical sense to have the class start its own branch of the class tree. The main disadvantage in this arrangement is that little functionality is inherited so that the new class's methods will have to be written to perform much of the work of the class.

6. Define Public and Private Protocols

In the course of defining the classes of the implementation, it is necessary to determine what constitutes the interface with other classes and what information the class will hold as private, not to be shared. Although it is not a required declaration, the designer should know the difference between the methods within the class which can be called externally and those which are for the internal use of the class only. These are the *public protocol* and *private protocol* of the class, respectively. [Ref.5:pp. 2.4.2-2.4.3]

F. SEALING OFF AN APPLICATION

After an application package has been developed, debugged and optimized, *ACTOR* makes a provision for *sealing off* the code [Ref.4:pp. 392-393]. Once this has been done, the package can function autonomously without depending upon the entirety of the full *ACTOR* distribution image. With an interpreted language such as *ACTOR*, the

development of the stand-alone package is not as straightforward as is the case with a compiled language. The custom package needs to contain enough of the executable portion of the basic language so that it runs successfully. The formulation of the stand-alone application amounts to a pruning of the *ACTOR* tree. All tool classes such as browsers, inspectors and debuggers are deleted to form an efficient skeleton tree containing only the programmer's custom classes and the classes essential for package execution. This sealed off package, when run, has the same appearance as any other Microsoft Windows application [Ref.4:p. 388].

III. A HIGH-LEVEL APPLICATION

The first portion of the implementation phase of this thesis research is the topic of this chapter. The objective is to evaluate the value of the *ACTOR* language in the development of a high-level application package. Specifically, the investigation centers on the application of the object-oriented design steps described in the last chapter. A detailed description of their use and an evaluation of their effectiveness in developing an accurate programming model for the desired system are performed. The goals of the implementation are to determine whether *ACTOR* is a good language choice for the relatively rapid design, modeling and coding of a high-level application and to compare the package to a functionally similar Pascal version. The question of whether the language is suited to the programmer with little previous coding experience is also examined.

A. A VIDEO MANAGEMENT PROGRAM

It was desired to develop a non-technical application package in *ACTOR* with a scope broad enough to offer variety yet not so broad as to require a great amount of detail or repetition. In this way, the material to be modeled would be familiar to a wide variety of individuals, the design task would appeal to the intuition of a person without significant programming experience and a rich cross-section of the capabilities of an object-oriented language could be demonstrated. Another consideration in the choice of a system to model was that it must also be an application which lends itself to successful implementation in a procedural language such as Pascal.

With these considerations in mind, the decision was made to implement the heart of a management system for a generic video tape rental store. This particular application

involves a good mixture of data base management considerations, specialized objects in the case of the object-oriented implementation, specialized data structures in the case of the procedural implementation and some arithmetic computation and management for calendar dates. A video store management package, due to the degree of independence of the subtasks which it involves, also offers the opportunity to develop features in an incremental fashion. This allows the avoidance of the frequent writing of software harnesses to test low-level methods or procedures and the avoidance of the writing of stubs to test higher level structures and classes. Thus, the application may be suitable for rapid prototyping. An additional reason why a video store management package is a good vehicle to compare object-oriented and procedural implementations is the importance of the user interface which it involves. Such a management system naturally involves a highly interactive and responsive user interface. This is a strength of an *ACTOR* application and object-oriented applications in general.

B. THE PHYSICAL MODEL OF THE SYSTEM

1. Determining System Requirements

The first step in the object-oriented design process was given in Chapter II as the establishment of system requirements. The basic issue which needs to be addressed is the question of what exactly a video store management package encompasses. At this point in the design, this should be a "what" question and not a "how" question. To answer it, it is necessary to define the purpose of such a management system so that it may be broken into functional divisions. Appealing to the intuition and experience of the layman as a patron of a video tape rental store, the necessary functions to be performed are readily defined. The purpose of this management system is the efficient matching of customer tape rental requests with the store's video tape assets. This broad purpose may be divided into three general functional areas: the control of the video tape traffic and

inventory, the management of the store's customer database and the administration of fee collection.

Within these general functional areas, it is necessary to determine the specific tasks which a typical video tape management system might be expected to perform. An aid to this determination of tasks is to examine the use of the management system from both the point of view of the user (store employee) and the customer. The employee is primarily interested in controlling the inventory and traffic of video tapes and with the management of the customer database. The customer is most concerned with the efficient rental and return of tapes. The employee needs a facility to both add and delete a tape from the inventory of tapes in the case of new acquisitions and disposals. This should be as automated as possible, should provide a uniform collection of information on each tape held in the inventory and should not require any specialized knowledge on the part of the employee as to how this information is stored. A similar facility is needed by the employee to manage the store's database of customers. When a customer rents a tape, they are concerned with the cost of it, when it is due and efficient processing. They should be provided with all the necessary information to answer these concerns.

To support these demands, the management system needs to provide the store employee with an efficient system for renting and returning tapes. It should check such restrictions as the possibility that the customer has already checked out the maximum allowable number of tapes, whether they already have tapes checked out which are overdue and whether it is permissible for the customer to rent a tape which has the rating of the tape which they desire to check out. It should provide this information on an exception basis, not mentioning anything which does not affect the transaction with the customer. Fee calculations and due dates should be done automatically and updated status information on the customer displayed at the end of each rental and return event.

In addition to these functions, the store employee should be able to access information on a particular customer or display a list of overdue tapes at any time they choose.

The goals of the user interface are that it be intuitive in nature and in accordance with employee expectations, easy to learn and use, responsive to employee needs and, insofar as possible, to enforce the store policies on pricing, rental restrictions and special rules. The employee should be able to tell at a glance the exact options available to them. They should not be required to input any more detail than absolutely necessary concerning tapes, customers, dates or choice of operation. A minimum of input control manipulation and training should be involved. They should also be prevented from taking an action which would either violate store policy or damage store assets.

2. The Real World Entities

With these functional system requirements of the physical model determined, the next step is to identify the significant "players" or entities which are involved in the system. For this application, this is a straightforward task. There are obviously customers and video tapes involved. There is the store employee interfacing with the management system. Taking a collective view, it is necessary to consider the group of tapes held by the store, those checked out and the group of customers which the store has. Lastly, the frequent use of the word store in describing the physical entities leads to the conclusion that the store itself constitutes a form of an entity which must be modeled. There is a natural relationship between the store entity and the tape groups, the customer group and the store employee. The store entity seems to encompass the others, suggesting some sort of a hierarchical relationship. Figure 4 illustrates this proposed physical model. In the figure, the boxes represent the entities while their relative positions and connecting lines represent any hierarchical relationship which may exist. The dashed line connecting the store entity to the store employee entity signifies that this

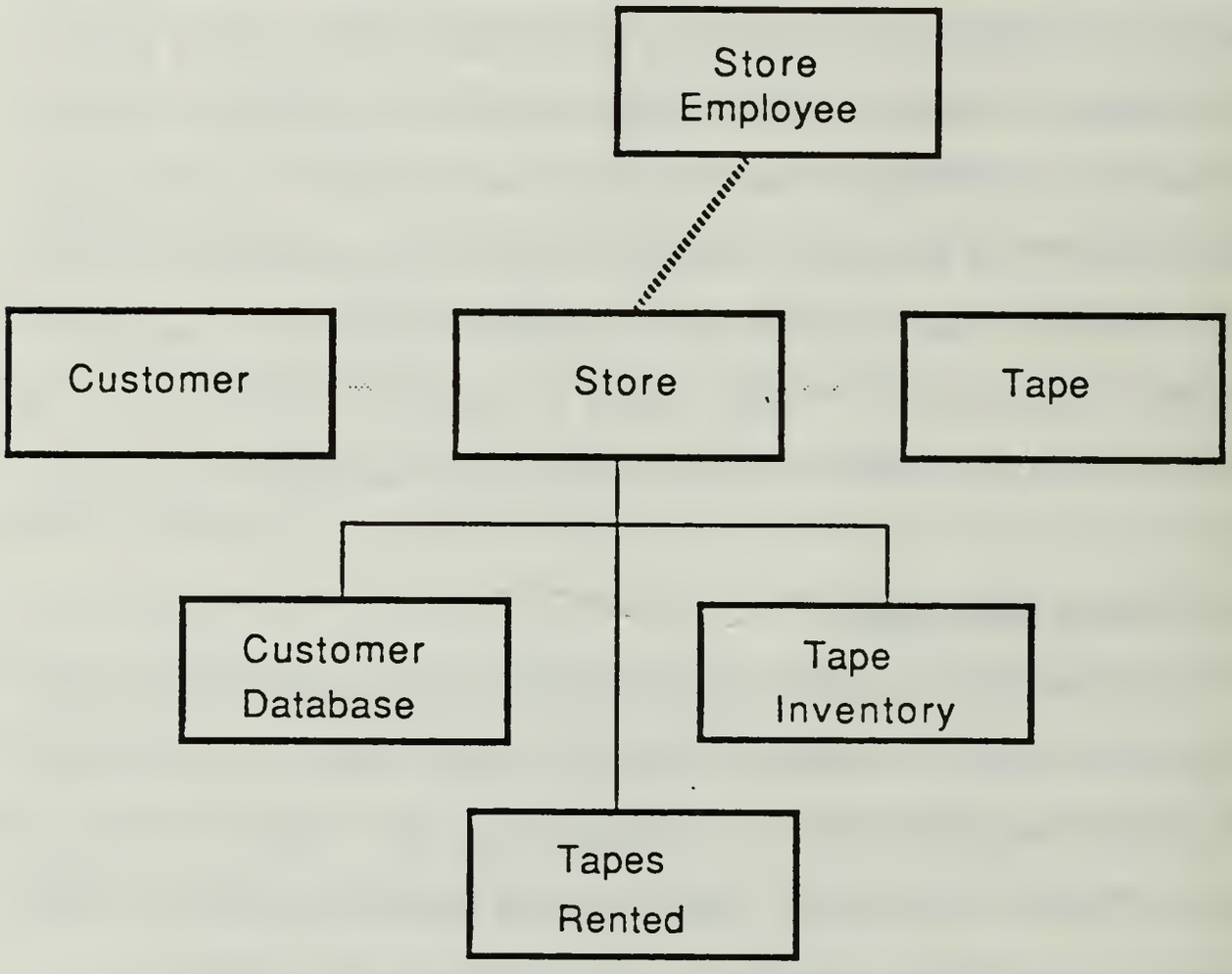

Figure 4. The Physical Model of the Video Management System

is not the same sort of relationship as between the store and the customer or tape databases. The store employee is a part of the store in one sense but they are also the system user. For the system user, the store entity acts as an interface to the various databases which are components of the store. These components are essentially transparent to the user/employee. The light lines between the store and customer and

store and tape boxes indicate that communication must occur between the entities but that no hierarchical relationship exists between them.

C. THE LOGICAL MODEL OF THE SYSTEM

The next step in the design of the object-oriented model is to translate the physical entities which have been identified into logical objects which are meaningful in the programming context. The goal of this step is the determination of the actual classes which are needed in the implementation to model the desired behavior encompassed in the physical model. The central entity in the physical model is the store. This is a good choice for the initial class to specify in the logical model. Both the ideas of a tape entity and a customer entity are nicely translated directly into objects. They each have a number of internal details (such as a customer's address or a tape's rating) concerned with both data and methods which are best managed in an autonomous manner. This is precisely the strength of an object. The entities depicted below the store object in Figure 4 can actually be thought of as components of the store which are not distinctly different from the tape and customer classes of objects. Instead, they are collections of either tape or customer objects. As such, they should not be specified as new classes but stored as instance variables within the store. From this it can be seen that there is a direct correlation between the parent/component relationship in the physical model and a class and its instance variables in the logical model. The only portion of the physical model which has not been addressed is the store employee. This entity is more accurately viewed as the user than as a part of the store. As such, it should not be modeled as an object but should be the determining factor in the design of the user interface portion of the implementation.

With these decisions made, a clearer picture of the necessary classes is developing. Classes for the store, tape and customer objects are required. The existing, predefined

classes of *ACTOR* need to be reviewed so that the new classes to be written can be most effectively located in the class tree. Reference to Figure 2 in Chapter II helps in this decision. Both the tape and customer objects need to have a number of specialized instance variables for holding pertinent data about the characteristics of the individual object. Examination of the class tree yields no class which possesses either instance variables or methods which would be appropriately inherited to import desired behavior to the tape or customer class. For this reason, it is best to define both of these new classes as immediate descendants of the Object class. This gives the two classes only the most basic of built in behavior and necessitates the custom implementation of most of their functionality. However, this is a better choice than to inherit utterly inappropriate characteristics from being located elsewhere in the class tree.

The store class amounts to the core of the application. It needs to manage the tape inventories and the customer database, control the traffic of both tape and customer objects, contain the functionality for renting and returning tapes and should act as the user interface for the package. In a window driven language like *ACTOR*, these jobs are well suited to an object in the Window class family. The descendants of the Window class all have specific functions which are too narrow in scope for the broad responsibilities of the store object. Because the Window class itself has rich functionality suitable for a user interface, the store class is defined as a descendant of Window. It is given the more appropriate and descriptive name of Store Window class. This accounts for all of the entities identified in the physical model. They have all been mapped into logical classes whose tree locations have been determined.

The last required class is not readily apparent at the intuitive level. In the determination of system requirements portion of the design, the calculation of tape due dates and the listing of late tapes were given as required functions. The Store Window

class needs access to logic which can provide manipulation of dates. This is an independent function and is best kept distinct from the Store Window class. The Calendar class is specified as a direct descendant of Object to fulfill these needs. As with the Tape and Customer classes, there are no appropriate classes other than Object which should act as the ancestor for Calendar. The Calendar class is an auxiliary tool to be used by the Store Window class and is completely isolated from the user. Its structure has less basis in the physical world than the other classes and its task is less central to the package. For these reasons, its existence was not directly suggested by the physical model although its functionality was specified indirectly in the system requirements. The completed logical model is shown in Figure 5. Boxes printed with shadows represent those classes written for this application.

1. The Store Window Class

A detailed description of the implementation of each of the custom classes begins with the Store Window Class. The complete *ACTOR* code for the Store Window class may be found in Section 1 of Appendix A. Only illustrative sections are reproduced in this discussion. The class header appears as follows:

```
inherit(Window, #StoreWindow,
#(customers tapes inStock rentedOut maxnum date), 2, nil)!!
```

ACTOR automatically constructs this header when the user selects a special option in the browser to make a descendant of a specified class. A template is presented which allows the user to provide the salient details of the class without being burdened with the exact format of the header statement. As described in the section in Chapter II on syntax, this statement specifies that the class descends from the Window class. The " # " symbol is the *ACTOR* designation for a literal. Here, it precedes the class name and the names of each of the instance variables contained in the class. The "customers" instance variable

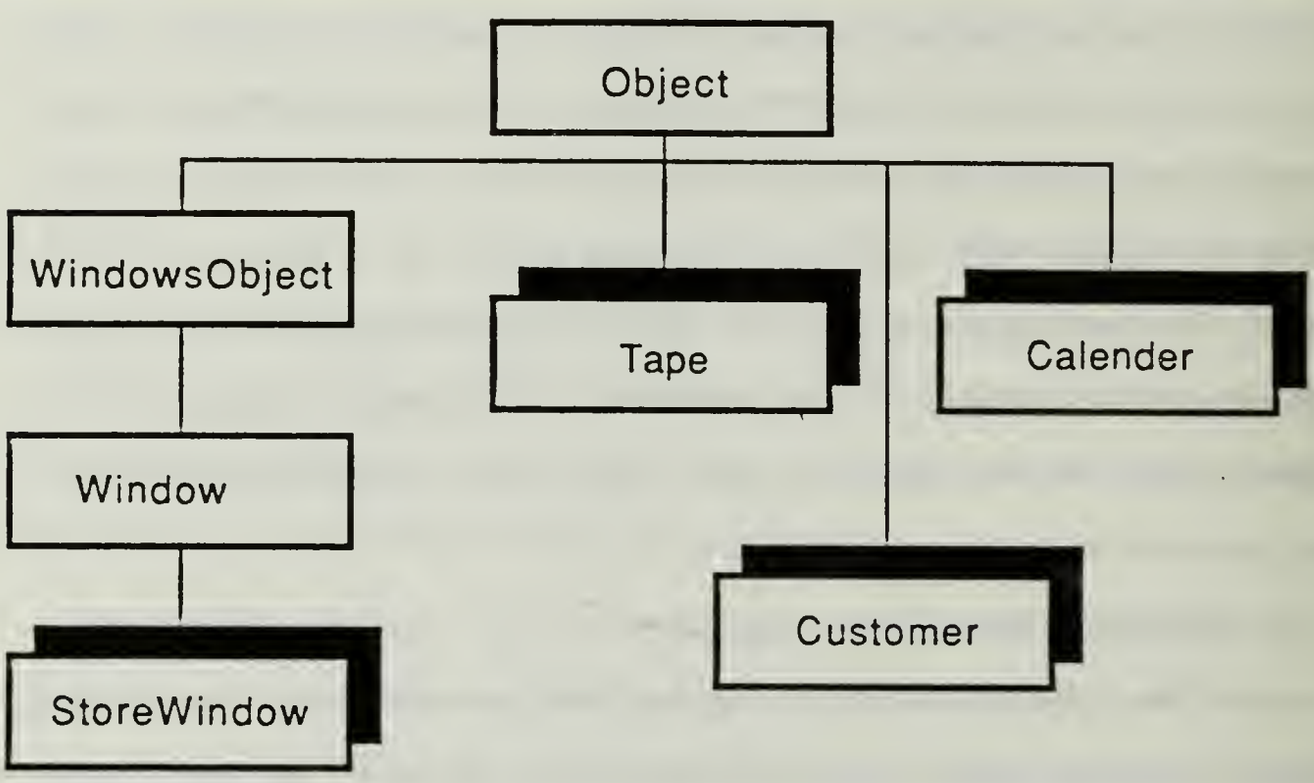

Figure 5. The Logical Model of the Video Management System

holds a dictionary consisting of all of the customer objects in the store database. Recall that a dictionary object consists of a collection of associations of keys and values [Ref.4:p. 209]. In this case, the keys to the dictionary are customer names while the values are customer objects. The "tapes" instance variable holds a dictionary consisting of all of the tape objects which the store owns. The keys to the dictionary are the tape numbers for each of the dictionary entries. The "inStock" instance variable holds another dictionary which is a collection of all tape objects not rented out. The keys here are the tape numbers. Similarly, the "rentedOut" instance variable is a dictionary of all tape

objects which are checked out. "Maxnum" simply holds the last tape number assigned when a new tape was added to the "tapes" dictionary. The instance variable "date" holds a string object which is passed to the Calendar object when date calculations need to be performed. The only date input required from the user is the current date when the system is initialized at the beginning of a business day.

The methods included in the Store Window class are addCust, deleteCust, addTape, deleteTape, rentTape, doCheckOut, returnTape, lateTapes, getCustName, displayCust, loadCust, saveCust, loadTapes, saveTapes, command, start and close, as can be seen by examining Section 1 of Appendix A. A discussion of the critical features of the most important of these follows. Figure 6 contains an abbreviated version of the code for the start method. This method demonstrates the use of a menu created dynamically instead of via a predefined format as stored in a resource file. The createMenu and changeMenu options are inherited by the Store Window class from the Window class. The changeMenu method is an example of a *pass-through* method which invokes the ChangeMenu function in Microsoft Windows [Ref.4:p. 241]. In the method, the "0" following the receiver "self" signifies that an item is to be added to the menu, not changed. The string which follows the "0" is the entry which will appear in the menu. The three digit number which comes next is just a constant associated with the new menu item and the "MF-APPEND" is the Microsoft Windows constant that specifies that an item is to be added to the end of the menu. The four messages which start with "load..." invoke the methods which read the given disk files into newly created dictionary objects held in the instance variables of the Store Window class. The message at the end of the method within the "do" construct invokes the "updateStanding" method which examines a customer object and writes a "bad" into the standing instance variable if the object has a tape rented which was due before today's date. It is important to note that the updateStanding method is within the Customer class since it deals with the customer

```
Def start(self | aString)
 {createMenu(self);
  aString := " Rent tape ";
  changeMenu(self,0,lP(aString), 100, MF_APPEND);
  aString := " Return tape ";
  changeMenu(self,0,lP(aString), 110, MF_APPEND);
       .
       .
       .
  freeHandle(aString);
  show(self,1);
  drawMenu(self);
  /* Initialize the tape numbering system */
  maxnum := 0;
  /* Initialize the instance variables tapes, inStock,
  and rentedOut.*/
  tapes := new(Dictionary,1);
  inStock := new(Dictionary,1);
  rentedOut := new(Dictionary,1);
  loadTapes(self,"tape.dat");
  loadTapes(self,"instock.dat");
  loadTapes(self,"rented.dat");
  loadCust(self);
  /* Initialize the Calendar object which holds today's
  date as entered by the user, methods for calculating
  future dates as well as differences between two dates. */
  date := new(Calendar);
  init(date);
  /* Initialize the standing of each customer based upon
  today's date and if they have late tapes. */
  do(customers,
  {using(cust)
   updateStanding(cust,date,rentedOut);
  });  }  !!
```

Figure 6. The Start Method

object. In accordance with the information hiding principle, the Store Window object doesn't know about the instance variables which the customer object uses or any details of the class since this information is not a part of the public protocol.

The command method, part of which is shown in Figure 7, is simply the means for recognizing the user's menu choice and invoking the appropriate method to carry it out. The command method is actually defined in the Window class but its behavior as so defined is inappropriate for a store window object. Thus it is necessary to redefine the method so that a store window object, when sent a "command" message will use its own

```
/* Accept the users choice from the menu and perform it. */
Def command(self, wP, lP)
 {select
   case lP <> 0      /* A menu item was not chosen */
    is ^0
   endCase

   /* lP = 0 for the remaining cases, implying that a
      menu choice was made. */

   case wP == 100
    is rentTape(self)
   endCase
      .
      .
      .
   case wP == 170
    is displayCust(self)
   endCase
 endSelect;
 ^0  }  !!
```

Figure 7. The Command Method

internal version of the method, not the version inherited from the Window class. The wP and lP arguments to the method are named after parameters passed from Microsoft Windows. The wP stands for *word parameter* and in this method signifies which menu selection the user clicked on. The lP stands for *long parameter* and here is used to indicate whether a valid menu choice was made at all.

The close method is reproduced in Figure 8. It simply invokes methods which write the current contents of the dictionaries held in the instance variables customers, tapes, inStock and rentedOut to disk files. The last line of the method demonstrates the use of early binding. The close method is already defined by the Window class. However, to accomplish the desired storing of customer and tape objects, it is necessary to redefine the close method within the Store Window class to perform this and then call the close method of the Window class. Notice that this is done by specifying the receiver (self) as a window object. If this early binding was not used, this would amount to a

```
/* Invoke methods to save the customer and tape
databases and then invoke the Window class close
method. */
Def close(self)
{
 saveCust(self);
 saveTapes(self,tapes,"tape.dat");
 saveTapes(self,inStock,"instock.dat");
 saveTapes(self,rentedOut,"rented.dat");
 ^close(self:Window);
}   !!
```

Figure 8. The Store Window Close Method

recursive call to this close method within the Store Window class. The overall effect of the close method is to perform the automatic storage of all customer and tape data when the user closes the store window object, presumably at the end of a business day.

One of the longest and most procedural-like methods of the Store Window class is the rentTape method, depicted in Figure 9. The design of this method is an if-then-else filter which checks for six exception conditions. If none of the conditions are found to be true, the detailed methods which perform the necessary bookkeeping of renting a tape are invoked. The "at" method used repeatedly throughout the method is defined by a dictionary object. It takes two arguments. The first is the name of the dictionary and the second is the name of the key to the dictionary. This method demonstrates the readability of *ACTOR* code due to the familiar syntax and self documenting variable names and strings.

A method which illustrates the use of polymorphism is the doCheckOut method. This is a short but powerful method illustrated in Figure 10. Notice that there are two invocations of doCheckOut within the method itself. These are not recursive calls. The first is a message sent to a customer object. It invokes the doCheckOut method within the customer object which inserts the tape number checked out. The second doCheckOut message is sent to a tape object. This causes the customer name, the date checked out and the date due to be inserted into the tape object. Again, the details of how the tape object manages this information are hidden from the store window object.

2. The Customer Class

The header definition statement for the Customer class is as follows:

```
inherit(Object, #Customer, #(name street city standing
tapesRented allowedToRent), 2, nil) !!
```

```
Def rentTape(self | cust,tape)
 {cust := getCustName(self);
 if not(at(customers,cust))
 then errorBox(cust,
      "This individual is not in the store database.");
 else tape := inputBox("TAPE NUMBER",
      "Enter the number of the tape to be rented:");
  if not(at(tapes,tape))
  then errorBox(tape,
      "That tape number is not in the store database.");
  else if (at(rentedOut,tape))
      then errorBox(tape,
           "That tape is already checked out!");
      else cust := at(customers,cust);
          tape := at(tapes,tape);
          if not(okRating(cust,getRating(tape)))
          then errorBox(getRating(tape),
  "This customer is not allowed to rent a tape with this rating!");
          else if tapeRentLimit(cust)
              then errorBox(cust.name,
  "This customer has reached the limit on number of tapes rented.");
              else if not(goodStanding(cust))
                  then errorBox(cust.name,
          "This customer is not in good standing!");
                  else doCheckOut(self,tape,cust);
                  displayCust(cust,tapes);
                  endif;
              endif;
          endif;
      endif;
  endif;
endif;   }
```

Figure 9. The RentTape Method

```
/* Adds the tape to be checked out to the rentedOut
  Dictionary and deletes it from the inStock Dictionary. */
Def doCheckOut(self,tape,cust)
{
  add(rentedOut,tape.number,tape);
  reclaim(inStock,tape.number);
  doCheckOut(cust,tape.number);
  doCheckOut(tape,cust.name,date);
}       !!
```

Figure 10. The DoCheckOut Method

As previously stated, the Customer class is a descendant of the Object class, the root of the *ACTOR* class tree. The first three instance variables are self explanatory and contain string objects. The "standing" instance variable simply contains a "good" or "bad" entry. A "bad" entry signifies that the customer object has tapes checked out which are overdue. The "tapesRented" variable contains an array object of four elements. These elements are strings representing tape numbers, not actual tape objects. The "allowedToRent" variable contains an ordered collection which is used to hold a collection of strings. These strings represent the permissible ratings which a customer is allowed to check out, e.g., "PG-13". Since the Customer class inherits far less behavior than the Store Window class, it contains many methods. The methods in the Customer class are updateStanding, doCheckIn, goodStanding, displayCust, readCust, writeCust, tapeRentLimit, okRating, doCheckOut, checkForDelete, getCustName and buildCust.

The buildCust method, given in Figure 11, effectively illustrates the structure of a customer object. The getCustName method is invoked early in the buildCust method. This solicits the name to be inserted into the customer record and ensures that

```
/* Builds a new Customer object, inserting information
  provided by the user in response to queries. */
Def buildCust(self | ratings)
{
 name := getCustName(self);
 street := inputBox("ADDRESS",
   "What is the street portion of their address?");
 city := inputBox("CITY AND STATE",
   "What is the city, state, and zip code?");
 standing := "good";
 tapesRented := new(Array,4);
 fill(tapesRented,"000");
 allowedToRent := new(OrderedCollection,1);
 ratings := tuple("G","PG","PG-13","R","X");
 do(ratings,
 {using(rat)
  if questionBox(rat,
  "Is this person permitted to rent tapes with the above rating?")
        == IDYES
  then add(allowedToRent,rat);
  endif;
 });
```

Figure 11. The BuildCust Method

the name is in a precise format when stored. This enables effective searches and manipulations of customer objects to be performed. The inputBox method is used to solicit and receive address information from the user. This method is defined in the string class so that the receiver of the message can be a string. This receiver becomes the caption of a specialized popup input window which displays the second argument, normally a question string, within the box. An example of a typical window is given in Figure 12. An input area is also displayed which enables the user to enter an appropriate answer to the displayed question. The string entered by the user then becomes the

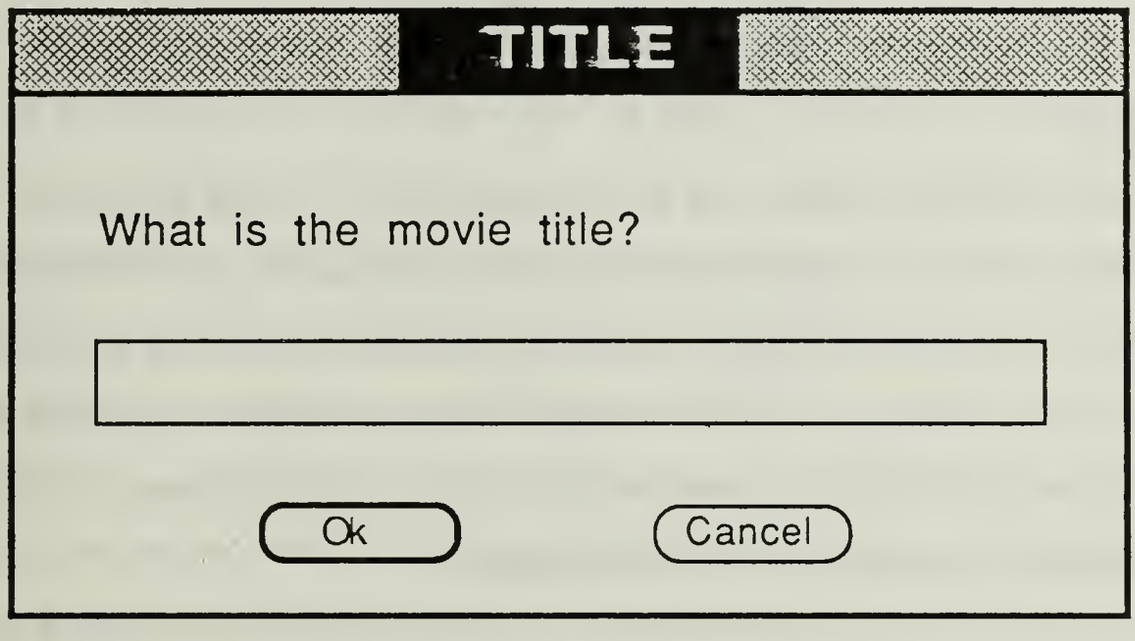

Figure 12. A Typical Input Box

returned value of the method. The code for the inputBox method can be found in Section 5 of Appendix A. The effect of the do construct in Figure 11 is to cycle through the tuple of possible tape ratings, asking the user if each one is to be permissible. The "==" sign in this part of the method is the *ACTOR* equivalence operator. *ACTOR* objects are actually pointers to the data required by the object. An "=" operator compares the two values at the memory locations which are referenced by their pointers. The equivalence operator just compares the pointers themselves. Hence, *equivalence is a much faster operation than equals* and should be used whenever possible [Ref.4:p. 126]. The identifier "IDYES" is simply an *ACTOR* constant signifying that a "yes" button has been clicked on.

Two of the methods in the Customer class illustrate the power which can be packed into a small amount of *ACTOR* code. These are the okRating and tapeRentLimit

methods. They both consist of one line of code. The okRating method, minus the definition line is:

```
^find(allowedToRent,rating);
```

This method is passed a rating string as an argument which it then looks for in the customer object. The " ^ " return operator is used in this method to send the rating back to the caller if the requested rating is found in the customer object. If the rating is not found, the method returns a value of nil. This allows the method to be treated as a boolean operator which helps to simplify the code in the calling environment. The body of the tapeRentLimit method consists of the following:

```
^not(find(tapesRented,"000"));
```

Here, the "boolean" use of the method can be the same. The difference is that a non-nil (true) value is returned if the string "000" is not found in the customer object. "000" is the sentinel signifying that no tape occupies a particular index in the array which holds the record of checked out tapes. If the array contains nothing but actual tape numbers, a nil or false value is returned by the method.

3. The Tape Class

The third class written for the video management package is the Tape class. Instances of it are tape objects which hold all the pertinent details concerned with an individual tape. The class header line, generated automatically by the browser, is as follows:

```
inherit(Object, #Tape, #(name number rating dateRented
dateDue rentedTo fee), 2, nil)!!
```

As with the Customer class, the Tape class is a direct descendant of the Object class, as was depicted in Figure 5. The "name" and "number" instance variables hold strings and are self explanatory. The "rating" variable is also used to hold a string, one of the five standard tape rating choices. The "dateRented" and "dateDue" variables also hold strings which represent exactly those dates. The format for these strings is that used by the Calendar class, mm/dd/yyyy. The "rentedTo" instance variable holds a string object which represents the name of the customer who has rented the tape. Notice that this is not an actual Customer object but the key to the object. The "fee" variable is used to hold a real object which is the daily rental cost of the tape.

The Tape class contains 12 relatively simple methods which serve to manage instances of the class. These methods are getTitle, addZeros, printTape, checkIfLate, figureCost, doCheckIn, rented, writeTape, readTape, getRating, doCheckOut and buildTape. Analogous to the buildCust method in the Customer class, the buildTape method is illustrative of the structure of a tape object. The code for the method, shown in Figure 13, is invoked by the addTape method in the StoreWindow class. The name, rating and fee of the tape object are solicited with the inputBox method and assigned to the appropriate instance variables. The "dateRented", "dateDue" and "rentedTo" variables are all initialized automatically to strings which signify that they contain a null entry. The "number" instance variable for the tape object is assigned the value of *maxnum* which is set to the next available integer for the tape inventory. This value, after conversion by the asStringRadix method, is stored as a string object.

```
/* Obtain information concerning a tape from the user
and build it into a new Tape object. */
Def buildTape(self,maxnum)
{
 name := inputBox("TITLE","What is the movie title?");
 /* Assign the next available number to the tape */
 number := asStringRadix(maxnum,10);
 rating := asUpperCase(inputBox("RATING",
                "What is the movie rated?"));
 dateRented := "999999";
 dateDue := "999999";
 rentedTo := "  ";
 fee := inputBox("RENTAL FEE",
          "What will the rental price be?");
}   !!
```

Figure 13. The BuildTape Method

Another example of the power of the *ACTOR* language is the checkIfLate method of the Tape class. The only statement in this method, beside the definition statement is as follows:

```
^(dateDifference(date,dateRented,date.today) >= 2);
```

The date object is an instantiation of the Calendar class and is passed as an argument to this method with the "today" instance variable treated as public protocol. The dateDifference method, defined in the Calendar class, calculates and returns the number of days difference between two date strings passed to it in the form mm/dd/yyyy. The way that the dateDifference method performs its function is hidden from the tape object. The checkIfLate method can be employed as a boolean function since it returns true if

there is at least a two day difference between the present date and the date the tape was checked out and returns nil (false) otherwise. This helps to simplify the code in the calling environment and makes it more readable.

A final method of interest in the Tape class is the getTitle method. This is another one line method consisting only of "^name". This may seem to be trivial or even a needless complication. Why can't the calling environment just make direct reference to "tape.name" when the title of the tape is needed? Although this can be done, it is generally not recommended. The "name" instance variable is a part of the private protocol of a tape object. No objects of other classes should use the knowledge that the title is stored in an instance variable called "name". Instead, the preferred way is to access the title through the getTitle method. There are numerous places throughout the video management package where reference is made to tape.number. This is because the "number" instance variable of a tape object acts as the key in the dictionaries used for the various tape inventories. It is therefore considered to be a part of the public protocol of a tape object.

4. The Calendar Class

The Calendar class, found in its entirety in Appendix A, is not a central component of the video management package but fills an essential utility role. Its purpose is to manage all of the irregularities of the modern 12 month calendar, accounting for different numbers of days in various months and leap years. The class is written in a general manner so that the time span over which it is defined can be simply changed by substituting occurrences of the beginning and ending years of the period. As currently implemented, the class functions for dates from January 1, 1988 to December 31, 2010. Within the class, dates are treated as integers with the beginning date corresponding to one and the last valid date mapped to one more than the difference in days between the beginning and last dates. As part of the external protocol of the class, dates are passed to

and from the Calendar class as strings in the format mm/dd/yyyy. They are mapped to the appropriate integer for manipulation within the class and converted back to appropriate string format in accordance with the external protocol. Since *ACTOR* normally runs on a personal computer, the range of dates covered by the Calendar class is limited to the value of the maximum integer representable by the hardware.

The Calendar class is a descendant of the Object class because no other classes exist in the *ACTOR* tree which are appropriate ancestors of Calendar. This inheritance can be seen from the header statement for the class:

```
inherit(Object, #Calendar, #(today tomorrow janFirst startMonth), 2, nil)!!
```

The "today" instance variable holds the string object representing the current date. This is input by the user when the initialization method of the Calendar class, shown in Figure 14, is invoked. The "tomorrow" instance variable also holds a string object. During the initialization method, the value of this string is determined by calculating today plus one and converting it to standard mm/dd/yyyy format. The "janFirst" instance variable contains an array which holds the integer representations for the first days of all of the years in the period covered by the calendar object. The "startMonth" variable contains an array of size 12 which holds integer representations of the first day of each month of a non-leap year, e.g., 1, 32, 60 and so forth.

The numToDate and dateToNum methods are used to convert the integer representation of a date to the standard string format and for the reverse conversion, respectively. The numToDate method starts by assuming the year is equal to the last year of the calendar period. It then compares the value of janFirst of that year to the integer passed to it as an argument. If janFirst is less, the correct year has been found,

```
Def init(self | aInpDlg)
{
 aInpDlg := new(InputDialog,"SET THE DATE",
  "Please type today's date in the format: mm/dd/yyyy",
  "");
 if runModal(aInpDlg,INPUT_BOX,ThePort) == IDOK
 then today := getText(aInpDlg);
 endif;
 initStartMo(self);
 initJanFirst(self);
 tomorrow := numToDate(self,(dateToNum(self,today) + 1));
} !!
```

Figure 14. The Calendar Initialization Method

otherwise the year is decremented and compared with the integer until the correct year is found. The month is determined in a similar manner, with logic included to account for leap years. The day is easily calculated by taking the difference between the integer argument and the first day of the calculated month. The dateToNum method is much simpler since the integer values for the first days of the given year and month are easily determined. Again accounting for leap years, the result is obtained by summing the year, month and day components. It should be noted that the Calendar class is very particular about the format of dates which are passed to it. For this reason, it is not hard to cause a malfunction by failing to use this format. Making the input aspect of the class more rigorous, although not difficult, would entail much code, would not exploit any of the strengths of *ACTOR* and is not in keeping with the notion of a prototype implementation.

D. AN ALTERNATIVE: THE PASCAL SOLUTION

An excellent way to appreciate the strong points of implementing this video management package in an object-oriented language is to make a direct comparison with a procedural version of the package. Appendix B contains a partial listing of a Pascal implementation of a video store management package functionally similar to the *ACTOR* version. The package was co-written by the author as a separate project. The listing contains the main declaration section, all procedure headers together with descriptions of how they perform, the begin and end delimiters of the procedures and the main body of the program. Between the procedure delimiters are found notations as to how many lines of code are contained in the procedure. There are two reasons why the program is not included in its entirety. First, the subject of this research is not procedural programming such as Pascal. Second, the program, being some 1900 lines in length, is too large to include as an auxiliary document.

The design of the program in Appendix B is accomplished along traditional lines for a procedural language. Essentially, the first two design steps used are the same as those used in developing the *ACTOR* application. Determining the functional requirements and modeling the real world entities are part of the problem domain and separate from the programming domain. However, there begin to be significant differences between the procedural and object-oriented approaches when the logical entities are developed. In developing the Pascal implementation, the close link between the real world entities and the logical entities which existed in the object-oriented design cannot be exploited. At this point in the Pascal design, a basic departure from the physical model and transition to an artificial programming regime takes place.

In developing a procedural application, a top down design is commonly recommended and applied. When programming in Pascal, one of the first steps in the top

down approach is to complete the declaration section for the program and to decide what major procedures are needed to accomplish the defined tasks. The requirements of the language force the programmer to define early in the implementation phase the exact characteristics of the program's constants, types and necessary global variables. Such is the case with the Pascal version of the video management package. The portion of the code which is present in Appendix B is representative of the skeleton which the programmer forms upon which to flesh out the details of the program's procedures. The designer's train of thought at this point is far separated from entities and objects. They are not concerned with objects which possess integral data and methods and accomplish work by passing messages. Instead, they are concerned about passing data in the form of shared data structures between using procedures. These data structures often have no close counterpart in the physical model yet must be firmly defined early in the programming phase. Often, there is a close mapping between the tasks which the physical model must perform and the family of procedures found in the Pascal program. In this application, it is easily seen that each procedure can be matched with a particular required function such as checking out a tape or adding a customer to the store database. Thus, the development of the Pascal package is a two pronged approach: developing the required data structures to provide shared information and developing the necessary procedures to provide the desired functionality.

1. Comparing the Two Programs

There are many contrasts which can be drawn between the object-oriented and procedural implementations of the video package. Perhaps the most obvious is the length of the two implementations. The length of the *ACTOR* package is just under 1000 lines of code while the Pascal package is over 1900 lines. There are some minor differences in the capabilities of the two packages. The Pascal package allows easier changes to be made to existing records of tapes or customers. However, the *ACTOR* package has a

much more sophisticated system for managing calendar references and calculations. It is more general in nature than the Pascal version and is easily modified to function for any desired time period. All in all, the functionality of the two is very comparable, leading to the conclusion that the characteristics of the languages are largely responsible for the great difference in program size.

Development time for the *ACTOR* version of the package was markedly less than that of the Pascal version. In fact, the ratio of development time for the *ACTOR* version to the development time for the Pascal version was approximately equal to the ratio of the size of the *ACTOR* program to the size of the Pascal program. Since the development environment of *ACTOR* is much more sophisticated than that of most implementations of Pascal, it might be expected that an *ACTOR* package could be developed much quicker than a Pascal package of the same size. In the case of the *ACTOR* implementation performed in this thesis, there was a unique factor present which caused this expectation to be wrong. The author's experience with *ACTOR* at the beginning of the implementation of that version was much more limited that his experience with Pascal at the inception of the Pascal package. Even though the assimilation of *ACTOR* syntax occurs quickly for someone with Pascal experience, the necessity of having to learn the details of the language during the implementation phase lengthens the development time. Another factor which prevents the development time of an *ACTOR* program from being extraordinarily short, when compared with a Pascal program of the same length, is the power of the language. The typical *ACTOR* statement has much more leverage in performing work than the typical Pascal statement. In the case of an *ACTOR* program and a Pascal program of the same length, the *ACTOR* program will most certainly have greater functionality and accomplish much more work.

Another contrast between the two programs is the user interface. Although some extensions to Pascal are available which offer windowing and graphics, their use tends to be more complicated and less natural than such facilities in *ACTOR*. The windowing environment used by *ACTOR* is a familiar, friendly and efficient way to communicate with the user. The main menu and auxiliary communication of the Pascal package are also effective and easy to use. However, these features in the Pascal package normally come at considerable cost in development time. Many "writeln" and "readln" statements and much format planning is required to accomplish what is quickly and simply accomplished in *ACTOR*. There is also considerable difference in the debugging facilities available in the two languages. Most implementations of Pascal force the programmer to halt execution to fix errors and then to recompile at least the offending portion of the code so that it can be rerun. *ACTOR*, because it is an interpreted language, has a debugger which allows errors to be corrected "on the fly" so that execution can then proceed from the point of stoppage. A last difference of interest is the use of loop control variables. These are common and necessary in Pascal. *ACTOR* manages to avoid the use and management of many of these variables through the use of enumeration methods such as the "do" method discussed in Chapter II.

2. Drawbacks to the Pascal Approach

The main problem with the Pascal program as compared with the *ACTOR* program stems from the fact that it incorporates a significantly larger public protocol between its components. This protocol is found either in the form of large or complicated global data structures such as the customer record type or in the passing of complex types between procedures. The requirement that information found in these data structures be available to many users within the program is not good incorporation of the information hiding principle. As a result, two negative characteristics manifest themselves. First, the number and seriousness of possible side effects greatly increase

with an increased size of the interface between program modules or procedures. Second, maintainability is adversely affected if details of any given program aspect are spread over many places in the program. A small change to one portion of the program may necessitate changes in other locations which are not at all obvious and may be poorly documented. [Ref. 7]

E. CONCLUSIONS

Experience in the design and implementation of the video management package in both *ACTOR* and Pascal brings out the clear contrast between the two languages. Many of the specifics of the *ACTOR* version can be generalized into observations applicable to all *ACTOR* applications. Conclusions which can be drawn about the advantages of the *ACTOR* implementation and the use of the *ACTOR* language for high-level applications include the following:

1) The focus on objects naturally encapsulates data and manipulations which can be performed on those data. This enforces the information hiding principle.

2) The use of inheritance by an object-oriented language increases the power and speed of development of application packages. Many common features are abstracted out by the inheritance scheme. There is much less duplication of code and structure than with a procedural language.

3) The use of polymorphism by an object-oriented language efficiently replaces much of the logic associated with the use of the Pascal "case" statement. An object is inherently capable of deciding how to handle a particular message. Objects of different classes can respond to the same message in any way that the designer desires.

4) *ACTOR* supports the concept of program modularity in two ways. First, division of objects into classes forces the programmer to think in terms of modules which have narrow interfaces with one another. Second, the methods which accomplish the work of the program are kept distinct from one another across object boundaries yet intimately connected with their associated objects.

5) The use of late binding in *ACTOR* lends much more flexibility to the language than the compile time binding used by Pascal. The resulting code can be more general in nature and more powerful.

6) The excellent user interface available with *ACTOR* allows an interactive prototype application to be developed much faster than with Pascal.

7) The significant amount of built in capability of *ACTOR* in the form of predefined classes allows effective and rapid prototyping through the augmentation of these classes.

IV. A LOW-LEVEL APPLICATION

The subject of this chapter, as the previous chapter, is an implementation package done in *ACTOR*. Chapter III explored the utility of *ACTOR* in the development of a high-level prototype. The focus of this implementation is a low-level package, a family of abstract data types.

A. INTRODUCTION TO ABSTRACT DATA TYPES

The notion of an abstract data type is an idea which has arisen out of the need to increase the modularity and maintainability of computer programs. An abstract data type is defined as a mathematical model together with a collection of operations defined on that model [Ref. 8]. The essence of this definition is that an abstract data type is an abstraction of information packaged with predefined ways to manipulate that information. Why are abstract data types viewed as so important? Shankar notes that:

> Integrity of a data structure can be better maintained if it is always manipulated by a predefined set of procedures. [Ref. 9]

In other words, the power of an abstract data type is that it allows a data structure to autonomously maintain its own state. Access to the data structure is strictly controlled by a standard set of operations available to the user of the abstract data type. The public protocol available to the user should contain no information on the implementation details of the abstract data type operations. The user should know how to reference the operations of the abstract data type without knowing how the operations perform their work.

It should be noted that standard Pascal has no inherent provision for abstract data types. It defines built-in types such as real and boolean which fit the definition of an

abstract data type, but the language does not allow the user to define their own such types. It is possible to design a package in Pascal which simulates an abstract data type. The data structures, internal operations on those data structures and the public protocol can be defined and adhered to just as if it were an actual abstract data type. However, there is nothing inherent in the standard Pascal language which *enforces* the abstract data type. In other words, if the user happens to know how the supposedly private operations are implemented, there is nothing to stop them from violating the abstract data type and accessing its data structures using other than the designated public operations. This shortcoming has been overcome by certain extensions to Pascal and by the language ADA [Ref. 9].

B. CORRELATION BETWEEN OBJECTS AND ABSTRACT DATA TYPES

This discussion of abstract data types makes it evident that there exist some close parallels between them and *ACTOR* objects. The key concept shared by both these entities is the idea of encapsulation. Both abstract data types and objects involve the encapsulation of data and operations through the process of abstracting common attributes of these data and operations. Abstract data types involve two types of abstraction: procedure and data abstraction [Ref. 9]. Procedure abstraction is the hiding of algorithms. It is well embodied by the Pascal procedure which can be invoked and used without knowing its details. Data abstraction is the hiding of data objects. It permits their use without knowing the details of their structure. An abstract data type results when a data type and its operations are implemented as a data abstraction supporting multiple instances [Ref. 9]. The last sentence sounds very much like the specification of an object-oriented class: the encapsulation of data together with methods which manipulate the data of objects of that class. Messages which are understood by a class of objects correspond to the names of operations in a abstract data type which form the

type's public protocol. Externally, abstract data types and objects can be viewed as very similar constructs which fulfill much the same purpose.

Internally, there are key differences between abstract data types and objects. The structure of an abstract data type is procedural in nature: work is performed by active procedures which act on passive data. Objects, on the other hand, involve the use of data which are themselves active in nature. Remember, in an object-oriented language, virtually *everything* is an object. This means that all data found in the instance variables and referenced within the methods of an object are objects themselves, incorporating their own data and operations in the form of methods. Within the abstract data type, data and procedures are treated as separate concepts. It is only at the level of the abstract data type package that they are treated as a unit. This separation of data and operations *never* occurs in an object. Entities at the most primitive level in an object oriented language are still objects. Another internal difference between abstract data types and objects is the contrast between procedure invocation and message passing. Instead of passing data to procedures, objects are asked to perform operations on themselves [Ref. 6].

C. A TREE PACKAGE

To study the effectiveness of implementing an abstract data type package in an object-oriented language, it was decided to provide the functionality of binary search trees and AVL, or height balanced, trees in a collection of *ACTOR* classes. For clarification, these terms are defined. It is assumed that the reader is familiar with the general concept of trees, nodes and edges. Stubbs and Webre define a binary tree recursively as being either empty or consisting of a node, called the root node, together with two binary trees. These two binary trees are disjoint from each other and from the root node and are known as the left and right subtrees of the root [Ref.10:p. 181]. Reingold and Hansen define a binary search tree as a binary tree in which every node has

the property that elements in its left subtree come before the node in natural order and those in its right subtree come after the node in natural order [Ref. 11]. This assumes that the nodes of the tree each contain a key element which can be compared with key elements of other nodes for ordering purposes. The height of a tree is the number of levels in the tree from the root to the most distant descendant. An AVL, or height balanced, tree is a binary search tree in which the difference in height of the two subtrees of any node is at most one [Ref.10:p. 225]. Figure 15 depicts an example of a tree which is not an AVL tree and the AVL tree which results after applying a transformation known as a double rotation to the first tree. This transformation, others similar to it and the meaning of the numbers within each node of the tree in Figure 15 are explained later in this chapter in the section which discusses the implementation of the AVL Tree class.

The task of modeling a tree package in *ACTOR* is significantly different from that of developing a high-level application such as the video management package. The very things which are to be modeled and implemented are already well defined. They are abstract in nature and have no physical counterparts. Trees, as abstract data types, have their logical entities defined in terms of nodes, the contents of those nodes and the connections between them. In the cases of the binary search tree and the AVL tree, the performance criteria of the application are also well defined. Thus, the object-oriented design task starts with the determination of the classes required and where they should be located in the *ACTOR* class hierarchy. Examination of the *ACTOR* class tree reveals no classes which exhibit tree-like behavior which would serve as appropriate ancestors for any of the classes in a tree package. Therefore, any classes developed will either be descendants of the Object class or descendants of one another.

The most basic building block of a tree is the node. This is an excellent starting point for an initial class for the *ACTOR* tree package. The node class is called the Binary

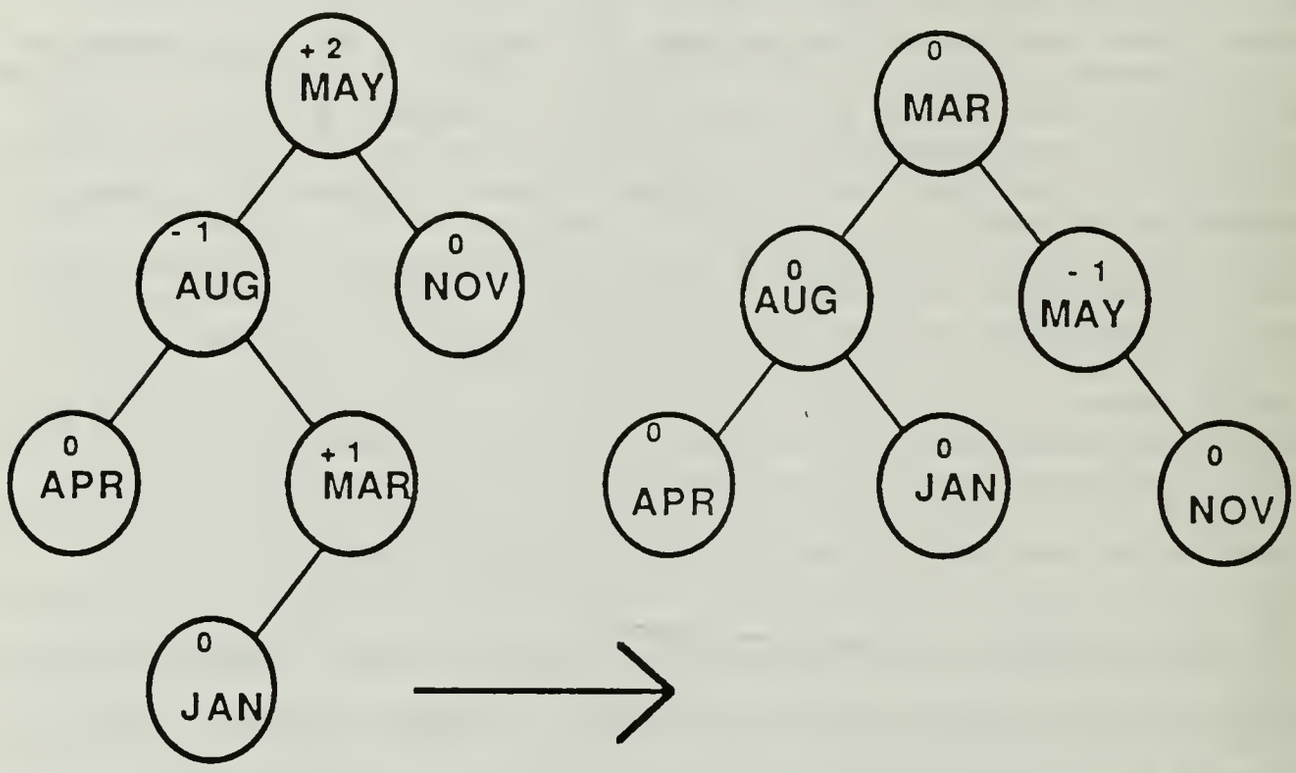

Figure 15. Forming an AVL Tree

Node class and is a direct descendant of the Object class. It contains basic behavior common to all nodes including the basics of traversing a tree which the node is a part of. Since the nodes of an AVL tree are just more specialized versions of the nodes in the Binary Node class, they are objects in the AVL Node class. This is a direct descendant of the Binary Node class. The logic to handle the connections between nodes could also be incorporated into the methods of the node classes. However, manipulations such as single and double rotations required in an AVL tree do not properly fit within the realm of classes concerned with nodes. This is because these manipulations are done on trees

as a whole, not on the individual nodes of a tree. For this reason, two additional classes are formed. The Binary Search Tree class is a direct descendant of the Object class. As a tree object, it is concerned with methods for building, deleting from and traversing trees. The class AVL Tree is specified as a direct descendant of the Binary Search Tree class. The resulting hierarchy of custom classes which make up the package is depicted in Figure 16. The details of each of these classes are covered in the following sections.

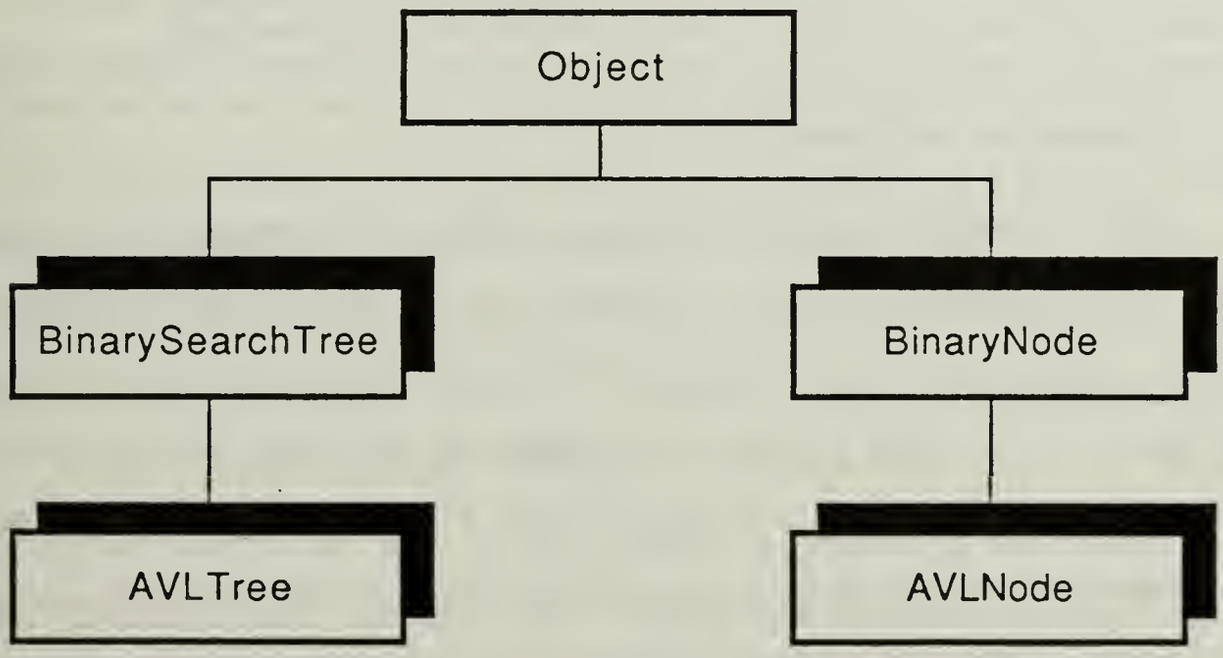

Figure 16. Class Hierarchy for the Tree Package

1. The Binary Node Class

This class is a direct descendant of the Object class and has three instance variables, "left", "right" and "data". As with trees defined in procedural languages, the first two of these instance variables contain pointers to the left and right subtrees, respectively. Actually, they should be thought of as containing the left and right subtree objects but remember that *ACTOR* uses pointers to its objects instead of copies of them for space efficiency purposes. The methods included in the Binary Node Class are compareNode, preOrder, postOrder, inOrder, deleteNode and subdeleteNode. The compareNode is simply a method used to determine which of the key elements of two nodes is greater. The two values are passed as arguments in the message to the method. If the two values are equal, the method returns a value of zero. If the first is greater than the second, a value of -1 is returned, and if the first is less than the second, a value of +1 is returned. The method uses the "=" and ">" operators which must be defined in the class of the key values being compared.

Tree traversals amount to the systematic visiting and processing of each node in a tree. For the purposes of this tree package, the processing of the node consists of printing its contents to the screen. The preOrder method is shown in Figure 17 as well as in Appendix C. This method is based upon a sample section of code included with the commercial implementation of *ACTOR* as distributed. It is a simple recursive algorithm which traverses the tree starting from the given node. For each node visited, the order of printing is the node itself, the subtree contained in the "left" instance variable, then the subtree contained in the "right" instance variable. The postOrder method prints the nodes of the whole tree by processing the left subtree, the right subtree, then the node itself for each node in the tree. The inOrder method results in the printing order left subtree, the node itself and then the right subtree. The postOrder and inOrder methods look identical to the preOrder method except for the placement of the print messages.

```
/* Traverse a BinaryTree pre-order. That is, visit
  the parent node first, then the left leaf, and lastly
  the right leaf. */
Def preOrder(self)
{ print(data);
  print(',');
  if left
  then preOrder(left);
  endif;
  if right
  then preOrder(right);
  endif;
} !!
```

Figure 17. The PreOrder Method

The deleteNode and subdeleteNode methods, contained in Appendix C, are based on the functionally similar procedures found in Stubbs and Webre [Ref. 10]. The deleteNode method appears quite procedural in nature. It is a recursive method which is used to search the tree until the node to be deleted is found or it is determined that the node does not reside in the tree structure. If the node is found and it has an empty "right" instance variable, the value of the node is replaced by the object held in the "left" instance variable of the node. Conversely, if the node has an empty "left" instance variable, the value in the "data" instance variable is replaced by the contents of the "right" instance variable. If neither "left" nor "right" hold nil objects, a message is sent to the node object held in "left" to invoke the subdeleteNode method. The subdeleteNode method is a private method in the sense that it should not be invoked directly from outside the Binary Node class but is reserved for the use of other methods within the class. Its job is to find and return the right most node of the left subtree of the node to be deleted. This node is effectively detached from the tree and the method deleteNode

replaces the node to be deleted with it. Within the subdeleteNode method, direct references to an object's instance variables by other objects are made with syntax such as *parent.right*. Although this is not generally recommended practice, in this instance it is a design trade off. The choice is between directly referencing the children nodes of another node in the tree or invoking a one line method to return the value of the desired variable with a message of the form *getRight(parent)*. The first alternative provides code which is cleaner and easier to read when frequent references must be made. In the case of passing messages between different nodes of the same tree as here, there is little danger of corrupting an object's management of its instance variables. Therefore, the decision is made to go against the normal recommendations of *ACTOR* coding to avoid awkward referencing syntax.

2. The AVL Node Class

The AVL Node class is defined to descend directly from the Binary Node class since it needs access to that class's methods through inheritance. The only reason for defining a class for AVL nodes separate from the Binary Node class is to incorporate the capability to store *balance* information in the AVL Node object. The "balance" instance variable is used to hold an integer object which represents the difference in heights between the left and right subtrees held in the node's "left" and "right" instance variables [Ref. 12]. If the left subtree has one more level than the right, the balance is +1. If the right subtree has one more level than the left, the balance value is -1. If both subtrees are of the same height, the balance is zero. Nodes which have subtrees differing in height by more than one have a balance equal to the difference in height, adjusted for the proper sign. Note that by the definition of an AVL tree, such a tree can have no nodes with balance factors less than -1 or greater than +1 [Ref. 12].

The AVL Node class contains only two methods, new and init. As discussed in Chapter II, a class normally does not define the new method but makes use of the

predefined version in the Behavior class. The reason that the new method is redefined in the AVL Node class is to accomplish automatic initialization whenever a new AVL Node object is instantiated. The new method appears as follows:

```
^init(new(self:Behavior));
```

In this statement, self is bound before run time to the Behavior class. The effect of the method is that, when a new message is sent to the AVL Node class, the standard new method of the Behavior class creates a new AVL node which is then initialized via invocation of the init method. The init method merely initializes each instance variable of the new node. The balance is set to zero since a new node never has children. This technique is detailed in the Actor Training Course Manual [Ref.5:p. 2.3.5] .

3. The Binary Search Tree Class

The Binary Search Tree class inherits directly from the Object class and has just one instance variable, "root". It is comprised of the six methods buildNode, addNode, deleteNode, preOrder, postOrder and inOrder. The last three are one line methods which send preOrder, postOrder and inOrder messages to the node held in the "root" instance variable. Locating the logic of the traversals in the Binary Node class instead of the Binary Search Tree class is a design decision made to equalize the size of the two classes. The buildNode method invokes the new method of the Binary Node class to create a new Binary node object, assigns its "data" instance variable to the value passed in and sets root to contain the newly created node if root is currently empty. The deleteNode method merely invokes the deleteNode method for the *node* held in root. This is another example of the use of polymorphism in *ACTOR*: let the receiver of the message determine the proper action to take. The addNode method is based upon a portion of an algorithm found in Horowitz and Sahni [Ref. 12]. It involves a

non-recursive search for the proper insertion point of the data to be added to the tree. When the point is located, the buildNode method is invoked and the newly created node is attached to the tree.

4. The AVL Tree Class

The AVL Tree class descends from the Binary Search Tree class in order that it may inherit the basic tree behavior of that class. Because the AVL tree is a specialized, more complicated version of the binary search tree, additional instance variables and methods are required. In addition to the "root" instance variable inherited from the Binary Search Tree class, the AVL Tree class defines four additional instance variables: "pivot", "pivotparent", "pivotchild" and "adjustbalance". The "pivot" variable is used to hold the node on the path of an insertion which is closest to the point of insertion and which has a balance of either +1 or -1. The "adjustbalance" variable holds an integer value of either +1, 0 or -1 and is used in the method which resets the balance factors of all nodes in the tree after an insertion [Ref.10:p. 227]. The "pivotparent" variable is self explanatory and the "pivotchild" variable is used to designate the child of the pivot node in the direction of the insertion. The methods of the AVL Tree class are new, init, buildNode, addAVLNode, findPivot, resetBalance, singleLRotate, singleRRotate, doubleLRotate and doubleRRotate.

The new and init methods of the AVL Tree class are analogous to those of the same name in the Binary Search Tree class. The init method performs three tasks. It inserts a new AVL node into the root variable, puts the same node into the pivot variable and sets the adjustbalance variable equal to the zero object. The buildNode method is used in adding a node to the AVL tree. It instantiates a new AVL node, inserts the value provided and sets the root to the new node if root currently holds a nil object.

The addAVLNode method, shown in Figure 18, is based upon the algorithm of Horowitz and Sahni [Ref. 12]. The method, by the nature of tree manipulation, involves a good bit of left and right child manipulation and adjusting of balance factors. The first message which the method sends is an addNode message to self. Here, self is an AVL tree. Since the AVL Tree class does not define this method, the immediate ancestor, Binary Search Tree class, is searched and the method is located and performed there. Notice that the buildNode method invoked by the Binary Search Tree addNode method is the buildNode method of the AVL Node class. In this way, the same basic add method can be used for both types of trees. The next message sent within the addAVLNode method is the findPivot message. This method locates the pivot node, as previously defined, and inserts this node into the "pivot" instance variable. It also puts the proper nodes into the "pivotparent" and "pivotchild" instance variables. All of this is done after the new node has been inserted. The resetBalance method is invoked next to update the balance factors of the nodes in the tree. At this point, there are two possible cases which require only the balance factor of the pivot node to be adjusted for the insertion to be complete. These are when the pivot has a balance factor of zero, in which case it takes on a balance factor equal to adjustbalance, and when the sum of the pivot node's balance factor and adjustbalance equals zero, in which case pivot's balance factor is set to zero. If neither of these cases holds, a rotation about the pivot node is called for. The transition which was shown in Figure 15 is a double rotation, performed by the doubleLRotate method. The case where a single rotation is applied to reform an AVL tree is illustrated in Figure 19 and given as the singleLRotate method of Appendix C. These methods are also based on well known algorithms as illustrated in Horowitz and Sahni [Ref. 12]. The methods singleRRotate and doubleRRotate are just mirror images of the previously mentioned methods, with lefts and rights reversed.

```
/* Insert an item into an AVL tree. */
Def addAVLNode(self, item | result);
{
 addNode(self,item);
 findPivot(self,item);
 if (pivotchild = nil)
 then ^self;
 else resetBalance(self, item);
 endif;
 if (pivot.balance = 0)
 then pivot.balance := adjustbalance;
  ^self
 endif;
 if (pivot.balance + adjustbalance = 0)
 then pivot.balance := 0;
  ^self
 endif;
 if (adjustbalance = 1)
 then if (pivotchild.balance = 1)
     then singleLRotate(self);
     else doubleLRotate(self);
     endif;
 else if (pivotchild.balance = -1)
     then singleRRotate(self);
     else doubleRRotate(self);
     endif;
 endif;
 if (pivotparent = pivot)
 then root := pivotchild
 else if pivot = pivotparent.left
     then pivotparent.left := pivotchild;
     else if (pivot = pivotparent.right)
         then pivotparent.right := pivotchild;
         endif;
     endif;
 endif;
}          !!
```

Figure 18. The AddAVLNode Method

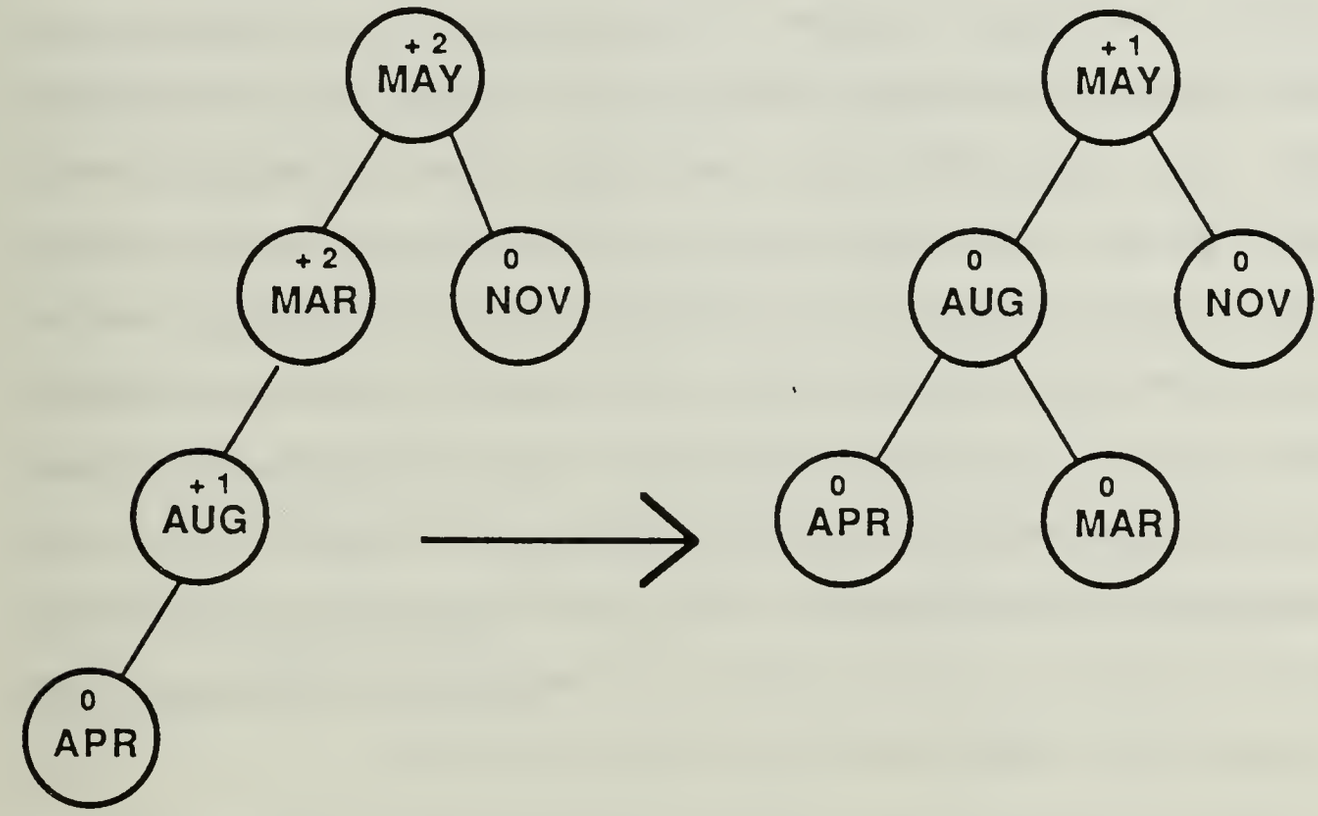

Figure 19. Application of a Single Rotation to form an AVL Tree

D. ANALYSIS OF RESULTS

The implementation of this tree package in *ACTOR* demonstrates a number of important concepts concerning the suitability of an object-oriented language for implementing abstract data types. First, by comparing the design of the package with the video management package, it can be seen that it is much harder to exploit the strengths of an object-oriented language when there are tight constraints on the problem from the outset. In the video management case, the designer is free to formulate both the physical

and logical model in any way which seems natural, meets the functional requirements of the system and is in accord with the object-oriented paradigm. This allows all of the benefits of a good object-oriented design to be reaped and the strengths of the *ACTOR* language to be exploited. In the case of the tree package, however, the designer is not just constrained by the functional requirements of the package. The internal details of binary search trees and AVL trees are fairly well specified before the object-oriented design task begins. The concepts of nodes and connections are hard to divorce from the idea of trees. In the same way, the ideas of balance factors, pivot nodes and rotations are hard to separate from the AVL tree. Thus, at the start of the design, many of the logical entities have already been solidified. If there are other general approaches and techniques to use in implementing these abstract data types, they are clouded by the existing schemes which were developed in the realm of procedural languages. The difference between the high-level and low-level design implementations is the difference between being told what to accomplish and being told how it is to be done.

There are a number of practical difficulties which arise when an abstract data type such as the tree package is implemented in *ACTOR*. Trees and their maintenance are based upon the management of pointer variables. The manipulation of these pointers involves logic which is characteristically procedural in nature. This style of programming can also be done in *ACTOR*, but it is forced and does not take advantage of the autonomous properties of objects. When dealing with frequent pointer references and manipulations, the designer is forced to choose between violation of the private protocol of objects or the construction of code which involves rather awkward referencing. A further complication arises in the implementation of trees when the design of *ACTOR* classes takes place. The division of the tree into node and tree classes is a choice which must be made if there is need to store such objects as roots, pivots and adjustbalances. However, this division is not intuitive and results in some overlap of function between

the classes. This is exemplified by the traversal methods of the Binary Node and Binary Search Tree classes.

Another difference in high-level and low-level modeling in *ACTOR* is related to the contrast between abstract data types and objects. Viewed externally as units, the two entities bear striking resemblance to one another. However, internally, the abstract data type is a procedural structure, composed of active procedures manipulating passive data. Notwithstanding the packaged nature of the abstract data type, it still must be separated into its component parts of data and algorithms when viewed internally, from the designers point of view. This division never occurs in the case of objects. The data and the methods which operate on the data always adhere to one another with a molecular-like bond. This constitutes a fundamental difference between abstract data types and objects which profoundly affects the manner in which design and implementation of them is approached and carried out.

V. CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSIONS

It is evident from the results of this research that *ACTOR* is a modern high level language with many strengths and advantages. It is a powerful language which should prove to have wide appeal among both individuals with significant programming experience and those with limited experience. The following specific conclusions can be drawn about its use as a result of the experience gained from this thesis.

1) *ACTOR* is a good language choice for those with procedural programming experience wishing to learn an object-oriented language. *ACTOR* syntax is very easily assimilated by C or Pascal programmers, being specifically designed with such a user in mind. This is in contrast to the syntax of Smalltalk which is unique and requires a moderate effort to master.

2) *ACTOR* is a good language choice for individuals with little or no programming experience. This is due to the intuitive appeal of the object-oriented paradigm. It is very natural to think in terms of objects when forming a physical model of a system and very natural to map these objects into logical entities within the programming domain.

3) *ACTOR* is well suited for top down design and use in prototyping. There is considerable functionality built into the predefined classes. This allows the programmer to begin their task at a high-level of development and use inheritance to incrementally put together a prototype. The user is assisted along the way by excellent interaction with and feedback from the system and a capable and flexible debugging facility.

4) Objects support the information hiding principle. The encapsulation of data and methods within an object promotes the use of information hiding by limiting the distribution of implementation details and maintaining narrow interfaces between different classes.

5) Modularity is a natural result of good object-oriented design. From the outset, the design of an *ACTOR* program hinges on dividing the problem into distinct entities which are capable of autonomous operation and management. Classes which have been implemented by different individuals or modified internally can readily communicate and interact so long as their public protocols remain intact.

6) Polymorphism in an object-oriented language models real world behavior. Humans have an inherent expectation that different things will respond in different ways upon

receipt of a general instruction. This phenomenon is effectively modeled by the use of polymorphism whereby different objects can be made to respond in different ways to the same message. This is one of the keys to the power of an object-oriented language.

7) The most efficient object-oriented implementation is the result of a problem statement which is not over-specified. A strength of the *ACTOR* language is the intuitive appeal of its design process. If excessive constraints or implementation restrictions are placed on the application at its initiation, creativity is stifled and the maximum benefits of the object-oriented approach are not realized. This is especially true if the constraints force procedural qualities to be incorporated into the design.

8) There are striking similarities between objects and abstract data types. Externally, the two fulfill the encapsulation of data and operations on those data and support information hiding. However, data and algorithms are distinctly separate within an abstract data type while data and methods remain integrated within an object.

9) Although possible, it is not recommended to implement in *ACTOR* an abstract data type with an inherent procedural character. Such an implementation is not efficient in terms of utility per line of code. Most importantly, it does not exploit the strengths of the object-oriented paradigm.

B. RECOMMENDATIONS

The use of *ACTOR* is recommended for modeling and prototyping applications to be performed by programmers of all experience levels and backgrounds. It is a language which appeals to the intuition and does not depend upon the user possessing a wide background in traditional computer science topics. The strength of its user interface and ability to develop applications which operate within a windowing environment make it highly suitable for the development of packages which require powerful yet friendly user interfaces.

C. ANSWERS TO INVESTIGATIVE QUESTIONS

The two investigative questions posed by this thesis, together with their answers, are as follows:

1) How natural is the connection between a group of interrelated physical objects and the logical objects which are developed to model them in the *ACTOR* environment? In other words, is an object-oriented language a good choice for a modeling or simulation task to be performed by someone without significant programming experience?

There is a very strong inherent connection between the physical and logical objects involved in an object-oriented design. This intuitive connection makes such a language an excellent choice for someone without significant programming experience.

2) Is an object-oriented language such as *ACTOR* a suitable tool to use in the development of abstract data types? Specifically, can its strengths be exploited in the development of a set of classes designed to model the behavior of both ordinary and height balanced binary search trees?

The *ACTOR* language may be useful for implementing some abstract data types. However, the basic structure and component interrelationships of trees are well defined and tend to be procedural in nature. In accordance with the seventh and ninth conclusions drawn above, this is not a situation where the strengths of the *ACTOR* language can be exploited. *ACTOR* is a powerful and innovative language but, like many things, can lose its appeal if applied in an area where not intended.

APPENDIX A - A VIDEO STORE MANAGEMENT PACKAGE

1. THE STORE WINDOW CLASS

```
/* Presents the main menu for the video store application.
Menu choices are the possible operations which the employee
may perform. */!!

inherit(Window, #StoreWindow, #(customers tapes inStock
rentedOut maxnum date), 2, nil)!!

now(StoreWindowClass)!!

now(StoreWindow)!!

/* Check in a Tape object by invoking methods which delete
if from the appropriate Customer object, reinitialize
the checked out information in the Tape object, add the
Tape object to the inStock instance variable, delete
the tape from the rentedOut instance variable, calculate
and display charges due, and update the standing of the
Customer object.  A customer is in bad standing if they
have overdue tapes. */
Def returnTape(self | tape,cust)
{
 tape := inputBox("TAPE NUMBER",
         "What is the number of the returned tape?");
 figureCost(at(rentedOut,tape),date);

 /* Send a check in message to the Customer object who
   had the tape. */
 cust := at(customers,rented(at(rentedOut,tape)));
 doCheckIn(cust,tape,date,rentedOut);
 doCheckIn(at(tapes,tape));
 add(inStock,tape,at(tapes,tape));
 reclaim(rentedOut,tape);
 displayCust(cust,tapes);
}    !!

/* Set up the window to display a list of Tape objects
which have been checked out for two or more days.  Then
invoke the appropriate methods of the tape class to list
these objects if there are any. */
Def lateTapes(self | aTW)
{
 aTW := defaultNew(TextWindow,"LATE TAPES");
 show(aTW,1);
 printString(aTW," #");
```

```
  printString(aTW,"      TITLE");
  printString(aTW,"                RENTED TO");
  printString(aTW,"    DAYS LATE");
  eol(aTW);
  printString(aTW,"  ");
  do( over(1,28),
    {using(x) printString(aTW,"--");
    });
  eol(aTW);
  do(rentedOut,
  {using(tape)
    if (checkIfLate(tape,date))
    then printTape(tape,aTW,date);
      eol(aTW);
    endif;
  });
}       !!

/* Delete a customer from the store database unless he
is not in good standing, has tapes still checked out or
cannot be located. */
Def deleteCust(self | toDelete,cust)
{
  toDelete := getCustName(self);
  if (cust := at(customers,toDelete))
  then select

  case checkForDelete(cust) == 2
    errorBox("UNABLE TO DELETE",
          "This individual is not in good standing!");
  endCase

  case checkForDelete(cust) == 1
    errorBox("UNABLE TO DELETE",
        "This individual still has tapes rented out!");
  endCase

  case checkForDelete(cust) == 0
    removeUsing(customers,toDelete,{});
  endCase
  endSelect;
  else errorBox(toDelete,
        "This customer is not in the store database!");
  endif;
}   !!

/* Adds the tape to be checked out to the rentedOut
  Dictionary and deletes it from the inStock Dictionary. */
Def doCheckOut(self,tape,cust)
{
  add(rentedOut,tape.number,tape);
```

```
 reclaim(inStock,tape.number);
 doCheckOut(cust,tape.number);
 doCheckOut(tape,cust.name,date);
}       !!

/* Check to see if it is possible and permissible to
check out a particular tape to a particular customer.
If it is, perform the checkout, doing the necessary
"bookkeeping". */
Def rentTape(self | cust,tape)
{
 cust := getCustName(self);
 if not(at(customers,cust))
 then errorBox(cust,
      "This individual is not in the store database.");
 else tape := inputBox("TAPE NUMBER",
      "Enter the number of the tape to be rented:");
  if not(at(tapes,tape))
  then errorBox(tape,
      "That tape number is not in the store database.");
  else if (at(rentedOut,tape))
      then errorBox(tape,
           "That tape is already checked out!");
      else  /* Reassign cust and tape to Customer and
               Tape objects instead of just keys to
               these objects. */
      cust := at(customers,cust);
      tape := at(tapes,tape);
      if not(okRating(cust,getRating(tape)))
      then errorBox(getRating(tape),
  "This customer is not allowed rent a tape with this rating!");
      else if tapeRentLimit(cust)
          then errorBox(cust.name,
  "This customer has reached the limit on number of tapes rented.");
          else if not(goodStanding(cust))
              then errorBox(cust.name,
          "This customer is not in good standing!");
              else doCheckOut(self,tape,cust);
                displayCust(cust,tapes);
              endif;
          endif;
      endif;
    endif;
  endif;
 endif;
}
          !!

/* Get the customer's last and first names from the user,
concatenate and capitalize them and return this string. */
Def getCustName(self | last)
```

```
{
 last := inputBox("LAST NAME",
         "What is the last name of the individual?");
 ^asUpperCase(last + ", " + inputBox("FIRST NAME",
         "What is the first name of the individual?"));
} !!

/* Create a new Customer object, invoke the method which
fills it with relevant data, then add it to the Dictionary
held in the instance variable "customers". The key to
this Dictionary is the customer's name. */
Def addCust(self | cust)
{
 cust := new(Customer);
 buildCust(cust);
 add(customers,cust.name,cust);
} !!

/* This method deletes the desired Tape object from
the "tapes" and "inStock" Dictionaries of the video
store. */
Def deleteTape(self | toDelete)
{
 toDelete := inputBox("DELETE A TAPE",
     "What is the number of the tape to be deleted?");
 if not (at(tapes,toDelete))
 then errorBox(toDelete,
    "This tape number is not in the store database!");
 /* Make sure it isn't currently rented out. */
 else if rented(at(tapes,toDelete))
     then errorBox("STILL CHECKED OUT TO:",
                 rented(at(tapes,toDelete)));
     else  /* Remove the Tape from the tapes and
              inStock Dictionaries. */
       reclaim(tapes,toDelete);
       reclaim(inStock,toDelete);

       /* Check to see if the highest number assigned
          to a tape needs to be reassigned. */
       if asInt(toDelete,10) = maxnum
       then maxnum := 0;
       keysDo(tapes,
         {using(aKey)
          maxnum := max(asInt(aKey,10),maxnum)
         });
       endif;
     endif;
 endif;
}          !!

/* Cause a new Tape object to be made, initialized
```

```
with the desired information, and added to the store's
database held in the Dictionary of the instance variable
"tapes".  The key to the tape is its number, assigned
as maxnum after maxnum is incremented by 1. */
Def addTape(self | tape)
{
 maxnum := maxnum + 1;
 tape := new(Tape);
 buildTape(tape,maxnum);
 add(tapes,tape.number,tape);
}   !!

/* Display requested pertinent data on a customer
 using a text window for output. */
Def displayCust(self | cust)
{
 cust := getCustName(self);
 /* if the name given by the user is in the store
 database, then display the associated attributes. */
 if (at(customers,cust))
 then displayCust(at(customers,cust),tapes);
 else errorBox(cust,
              "Name not found in store records.");
 endif;
}   !!

/* Load the file containing the store customer data
 base into a SortedCollection held in the instance
 variable customers. */
Def loadCust(self | cust, line, custfile)
{ custfile := new(TextFile);
 setName(custfile, "cust.dat");
 open(custfile, 0);
 checkError(custfile);
 customers := new(Dictionary,4);
 loop
 while ((line := readLine(custfile)) and (line <> ""))
 begin cust := new(Customer);
      readCust(cust,custfile,line);
      /* Add the new Customer object to the customers
        dictionary using the name as the key. */
      add(customers,cust.name,cust);
 endLoop;
 close(custfile);
 checkError(custfile);
}       !!

/* Store the given tape collection to disk in the given
  file.  Include all changes which have been made to the
  collection during the current store usage session. */
Def saveTapes(self,collection,actualFile | tapefile)
```

```
{
 tapefile := new(TextFile);
 setName(tapefile, actualFile);
 create(tapefile);
 checkError(tapefile);
 do(collection,
 {using(tape)
   writeTape(tape,tapefile);
 });
 close(tapefile);
 checkError(tapefile);
}       !!

/* Load the file whose name is passed as the string
"actualFile" into an object of class Dictionary and
store it in the appropriate instance variable. */
Def loadTapes(self,actualFile |
          tape,tapefile,line)
{
 tapefile := new(TextFile);
 setName(tapefile, actualFile);
 open(tapefile,0);
 checkError(tapefile);
 loop
 while ((line := readLine(tapefile)) and (line <> ""))
 begin tape := new(Tape);

   /* Read the data into the first tape object. */
   readTape(tape,tapefile,line);

   /* Update the maxnum variable if this tape's
      number is the largest seen so far. */
   maxnum := max(asInt(tape.number,10),maxnum);

   if (actualFile = "tape.dat")
   /* Add the newly created Tape object to the tapes
      Dictionary using the tape number as the key. */
   then add(tapes,tape.number,tape);
   else if (actualFile = "instock.dat")
        then add(inStock,tape.number,tape);
        else add(rentedOut,tape.number,tape);
        endif;
   endif;
   readLine(tapefile);
 endLoop;
 close(tapefile);
 checkError(tapefile);
}               !!

/* Invoke methods to save the customer and tape
databases and then invoke the Window class close
```

```
method. */
Def close(self)
{
 saveCust(self);
 saveTapes(self,tapes,"tape.dat");
 saveTapes(self,inStock,"instock.dat");
 saveTapes(self,rentedOut,"rented.dat");
 ^close(self:Window);
}   !!

/* Store the customer database to disk.  Include all
 changes which have been made to the database during
 the current store usage session. */
Def saveCust(self | custfile)
{
 custfile := new(TextFile);
 setName(custfile, "cust.dat");
 create(custfile);
 checkError(custfile);
 do(customers,
 {using(cust)
  writeCust(cust,custfile);
 });
 close(custfile);
 checkError(custfile);
} !!

/* Accept the users choice from the menu and perform it. */
Def command(self, wP, lP)
{
 select
  case lP <> 0      /* A menu item was not chosen */
   is ^0
  endCase

  /* lP = 0 for the remaining cases, implying that a
     menu choice was made. */

  case wP == 100
   is rentTape(self)
  endCase

  case wP == 110
   is returnTape(self)
  endCase

  case wP == 120
   is addTape(self)
  endCase

  case wP == 130
```

```
    is deleteTape(self)
  endCase

  case wP == 140
    is lateTapes(self)
  endCase

  case wP == 150
    is addCust(self)
  endCase

  case wP == 160
    is deleteCust(self)
  endCase

  case wP == 170
    is displayCust(self)
  endCase
 endSelect;
 ^0
}       !!

/* Set up the menu and display the window */
Def start(self | aString)
{
 createMenu(self);
 aString := " Rent tape ";
 changeMenu(self,0,lP(aString), 100, MF_APPEND);
 aString := " Return tape ";
 changeMenu(self,0,lP(aString), 110, MF_APPEND);
 aString := " Add tape ";
 changeMenu(self,0,lP(aString), 120, MF_APPEND);
 aString := " Delete tape ";
 changeMenu(self,0,lP(aString), 130, MF_APPEND);
 aString := " Late tapes ";
 changeMenu(self,0,lP(aString), 140, MF_APPEND);
 aString := " Add cust ";
 changeMenu(self,0,lP(aString), 150, MF_APPEND);
 aString := " Delete cust ";
 changeMenu(self,0,lP(aString), 160, MF_APPEND);
 aString := " Display cust ";
 changeMenu(self,0,lP(aString), 170, MF_APPEND);
 freeHandle(aString);
 show(self,1);
 drawMenu(self);

 /* Initialize the tape numbering system */
 maxnum := 0;

 /* Initialize the instance variables tapes, inStock,
 and rentedOut.*/
```

```
  tapes := new(Dictionary,1);
  inStock := new(Dictionary,1);
  rentedOut := new(Dictionary,1);

  loadTapes(self,"tape.dat");
  loadTapes(self,"instock.dat");
  loadTapes(self,"rented.dat");
  loadCust(self);

  /* Initialize the Calendar object which holds today's
  date as entered by the user, methods for calculating
  future dates as well as differences between two dates.
  */
  date := new(Calendar);
  init(date);

  /* Initialize the standing of each customer based upon
  today's date and if they have late tapes. */
  do(customers,
  {using(cust)
    updateStanding(cust,date,rentedOut);
  });
}                    !!
```

2. THE CUSTOMER CLASS

```
/* Objects of this class represent one of the store's
customers. Each pertinent characteristic of the customer is
stored in one of the instance variables of the class. */!!

inherit(Object, #Customer, #(name street city standing
tapesRented allowedToRent), 2, nil)!!

now(CustomerClass)!!

now(Customer)!!

/* Given the current date, checks to see if the Customer
has tapes in the tapesRented variable which are two or
more days late. If this is the case, "bad" is placed
in the standing instance variable, if not "good" is
placed there. */
Def updateStanding(self,date,rentedOut)
{
 do(tapesRented,
 {using(num)
  if num <> "000"
  then if checkIfLate(at(rentedOut,num),date)
       then standing := "bad";
         ^self;
       endif;
  endif;
 });
 standing := "good";
}  !!

/* Replace the number of the tape checked out by the
Customer object with the nil sentinel, "000". In
addition, call the method updateStanding which checks
to see if the Customer has any other tapes checked out
which are late. */
Def doCheckIn(self,tape,date,rentedOut)
{
 if (find(tapesRented,tape))
 then tapesRented[find(tapesRented,tape)] := "000";
 endif;
 updateStanding(self,date,rentedOut);
}  !!

/* Return true if the Customer object's "standing"
instance variable contains "good". */
Def goodStanding(self)
{
 ^(standing = "good");
} !!
```

```
/* Display requested pertinent data on a customer
  using a text window for output. */
Def displayCust(self,tapes | rentflag, aTW)
{
  aTW := defaultNew(TextWindow,name);
  show(aTW,1);
  printString(aTW,name);
  eol(aTW);
  printString(aTW,street);
  eol(aTW);
  printString(aTW,city);
  eol(aTW);
  printString(aTW,"Customer in ");
  printString(aTW,standing);
  printString(aTW," standing!");
  eol(aTW);
  printString(aTW,"Tapes rented: ");
  do (tapesRented,
  {using(num)
    if num <> "000"
    then rentflag := true;
      eol(aTW);
      printString(aTW,"        ");
      printString(aTW,num);
      printString(aTW,": ");
      printString(aTW,getTitle(at(tapes,num)));
    endif;
  });
  if (rentflag = nil)
  then printString(aTW," none.");
  endif;
  eol(aTW);
  printString(aTW,"Allowed to rent tapes rated ");
  do (allowedToRent,
  {using(rat) printString(aTW,rat);
    printString(aTW," ");
  });
}       !!

/* Read data from the given file into the given Customer
object, using the data in "line" as the initial entry. */
Def readCust(self,custfile,line)
{
  name := line;
  street := readLine(custfile);
  city := readLine(custfile);
  standing := readLine(custfile);
  tapesRented := new(Array,4);
  tapesRented[0] := readLine(custfile);
  tapesRented[1] := readLine(custfile);
```

```
  tapesRented[2] := readLine(custfile);
  tapesRented[3] := readLine(custfile);
  allowedToRent := new(OrderedCollection,1);
  line := readLine(custfile);
  loop
  while ((line <> "") and (line <> nil))
  begin add(allowedToRent,line);
   line := readLine(custfile);
  endLoop;
} !!

/* Write a Customer object to a given file using a
standard format. */
Def writeCust(self,custfile)
{
  write(custfile, name + CR_LF + street + CR_LF +
                city + CR_LF + standing + CR_LF);
  do(tapesRented,
  {using(num)
   write(custfile, num + CR_LF);
  });
  do(allowedToRent,
  {using(rat)
   write(custfile, rat + CR_LF);
  });
  write(custfile, CR_LF);
} !!

/* Check to see if the Customer object contains four
non-zero tape numbers in the tapesRented instance
variable.  "000" is the "nil tape" marker.  Return true
if so and nil otherwise. */
Def tapeRentLimit(self)
{
  ^not(find(tapesRented,"000"));
} !!

/* Check to see if the given rating is in the Ordered
Collection of allowable ratings for the Customer object.
Return a non-nil value if it is a permissible rating,
return nil otherwise. */
Def okRating(self,rating)
{
  ^find(allowedToRent,rating);
} !!

/* Insert the given tape number into the first array
element of "tapesRented" which is "000". */
Def doCheckOut(self,aTapeNum | ndx)
{
  ndx := 0;
```

```
 loop
 while tapesRented[ndx] <> "000"
  ndx := ndx + 1;
 endLoop;
 tapesRented[ndx] := aTapeNum;
} !!

/* Check to see if the Customer object is in good
standing and that it contains no checked out tapes. */
Def checkForDelete(self)
{
 if standing <> "good"
 then ^2
 else if not (do(tapesRented,
              {using(num)
                num = "000"
              }))
     then ^ 1;
     else ^0;
     endif;
 endif;
} !!

/* Get the customer's last and first names from the user,
concatenate and capitalize them and return this string. */
Def getCustName(self | last)
{
 last := inputBox("LAST NAME",
         "What is the last name of the individual?");
 ^asUpperCase(last + ", " + inputBox("FIRST NAME",
         "What is the first name of the individual?"));
} !!

/* Builds a new Customer object, inserting information
 provided by the user in response to queries. */
Def buildCust(self | ratings)
{
 name := getCustName(self);
 street := inputBox("ADDRESS",
  "What is the street portion of their address?");
 city := inputBox("CITY AND STATE",
  "What is the city, state, and zip code?");
 standing := "good";
 tapesRented := new(Array,4);
 fill(tapesRented,"000");
 allowedToRent := new(OrderedCollection,1);
 ratings := tuple("G","PG","PG-13","R","X");
 do(ratings,
 {using(rat)
  if questionBox(rat,
  "Is this person permitted to rent tapes with the above rating?")
```

```
        == IDYES
  then add(allowedToRent,rat);
  endif;
 });
}   !!
```

3. THE TAPE CLASS

```
/* Objects of this class represent video tapes.  Each of the
tapes characteristics is held in an instance variable.  The
possible manipulations of the tape object can be performed by
invoking one of the class methods. */!!

inherit(Object, #Tape, #(name number rating dateRented
dateDue rentedTo fee), 2, nil)!!

now(TapeClass)!!

now(Tape)!!

/* Return the name of the tape. */
Def getTitle(self)
{
 ^name;
} !!

/* Concatenate the appropriate number of zeros onto the
fee to be displayed when a tape is returned. */
Def addZeros(self,cost | difference)
{
 difference := size(asString(cost)) -
               indexOf(asString(cost),'.',0);
 if difference == 1
 then ^"00";
 endif;
 if difference == 2
 then ^"0";
 else ^"";
 endif;
}       !!

/* Print pertinent data on the Tape to the given window. */
Def printTape(self,aTW,date)
{
 printString(aTW,number);
 printString(aTW,"  ");
 printString(aTW,name);
 do(over(1,(28 - size(name))),
   {using(x)  printString(aTW," ");
   });
 printString(aTW,rentedTo);
 do(over(1,(22 - size(rentedTo))),
   {using(x)  printString(aTW," ");
   });
 printString(aTW,asStringRadix(dateDifference
               (date,dateDue,date.today),10));
}    !!
```

```
/* Given the date, check to see if the Tape object is
two or more days late. */
Def checkIfLate(self,date)
{
 ^(dateDifference(date,dateRented,date.today) >= 2);
}    !!

/* Calculate and display the amount due upon return of
the Tape object. */
Def figureCost(self,date | msg)
{
 if date.today = dateRented
 then msg := "Amount due: $" + asString(fee) +
                          addZeros(self,fee);
 else msg := "Amount due: $" + asString
  (dateDifference(date,dateRented,date.today) * fee)
  + addZeros
  (self,(dateDifference(date,dateRented,date.today)
                                * fee));
 endif;
 errorBox(name,msg);
}          !!

/* Reinitialize the dateRented, dateDue, and rentedTo
instance variables to nil sentinels. */
Def doCheckIn(self)
{
 dateRented := "999999";
 dateDue := "999999";
 rentedTo := "  ";
} !!

/* If the tape object is rented, return the string
holding the customer it is rented to.  If the tape is
not rented, return nil. */
Def rented(self)
{
 if rentedTo = "  "
 then ^nil;
 else ^rentedTo;
 endif;
} !!

/* Given a file name, write the Tape object to that file. */
Def writeTape(self,file)
{
 write(file, name + CR_LF + number + CR_LF + rating +
  CR_LF + dateRented + CR_LF + dateDue + CR_LF +
  rentedTo + CR_LF + asString(fee) + CR_LF + CR_LF);
} !!
```

```
/* Given a file name and an initial line of input, read
data from the file and store it into a tape object. */
Def readTape(self,tapefile,line)
{
  name := line;
  number := readLine(tapefile);
  rating := readLine(tapefile);
  dateRented := readLine(tapefile);
  dateDue := readLine(tapefile);
  rentedTo := readLine(tapefile);
  fee := asReal(readLine(tapefile));
} !!

/* Return the tape's rating to the caller. */
Def getRating(self)
{
 ^rating;
} !!

/* Insert information into the tape object concerning
who rented it, when they rented it, and when it is due. */
Def doCheckOut(self,aCustName,date)
{
 rentedTo := aCustName;
 dateRented := date.today;
 dateDue := date.tomorrow;
} !!

/* Obtain information concerning a tape from the user
and build it into a new Tape object. */
Def buildTape(self,maxnum)
{
 name := inputBox("TITLE","What is the movie title?");
 /* Assign the next available number to the tape */
 number := asStringRadix(maxnum,10);
 rating := asUpperCase(inputBox("RATING",
                "What is the movie rated?"));
 dateRented := "999999";
 dateDue := "999999";
 rentedTo := "  ";
 fee := inputBox("RENTAL FEE",
           "What will the rental price be?");
}   !!
```

4. THE CALENDAR CLASS

```
/* This class manages calendar dates for such applications as
the StoreWindow class. When an object of this class is
created and initialized, it solicits the current date from
the user. From this and through its methods, it calculates
the month, day, and year which represent tomorrow and stores
this as a string in the instance variable "tomorrow". The
class also contains methods which can return the difference
between two dates which are given in month, day, year format.
Input and output to objects of this class are always in the
string format mm/dd/yyyy whereas all intraclass manipulations
are performed on dates converted to successive integers.
The valid date range for the class if 01/01/1988 to
12/31/2010. However, the methods are written in a general
way so that all that is required to change the valid range
of years is to change the occurrences of the limiting years
themselves. */!!

inherit(Object, #Calendar, #(today tomorrow janFirst startMonth), 2, nil)!!

now(CalendarClass)!!

now(Calendar)!!

/* Convert the integer representation of a date used
internally by this class to a string in the format
mm/dd/yyyy. */
Def numToDate(self,aInt | year,month,day)
{
 /* Initialize at the last year and decrement until
 correct. */
 year := 2010;
 loop
 while (janFirst[year - 1988] > aInt)
  year := year - 1;
 endLoop;
 if (isLeapYear(self,year)) and
     ((aInt - janFirst[year - 1988]) = 59)
 then month := 2;
  day := 29;
 else if (isLeapYear(self,year)) and
     ((aInt - janFirst[year - 1988]) > 59)
    then aInt := aInt - 1;
    endif;
    month := 12;
    loop
    while startMonth[month] >
       (aInt + 1 - janFirst[year - 1988])
       month := month - 1;
    endLoop;
```

```
    day := aInt - (janFirst[year - 1988] +
      startMonth[month] - 2);
  endif;
  ^(asPaddedString(month,2) + "/" +
    asPaddedString(day,2) + "/" +
      asStringRadix(year,10));
}      !!

/* Given a date in mm/dd/yyyy format, convert it to its
integer representation with 01/01/1988 being day number
one. */
Def dateToNum(self,aStr | day,month,year)
{
 month := asInt(subString(aStr,0,2),10);
 day := asInt(subString(aStr,3,5),10);
 year := asInt(subString(aStr,6,10),10);

 /* If the date falls in a leap year and it is after
 Feb 29, adjust the day up by one. */
 if (isLeapYear(self,year)) and (month > 2)
 then day := day + 1;
 endif;
 ^(janFirst[year - 1988] + startMonth[month] + day - 2);
}   !!

/* Given two dates in string format, the first coming
chronologically before the second, return the difference
in days between the two. */
Def dateDifference(self,firstDate,laterDate)
{
 ^(dateToNum(self,laterDate) - dateToNum(self,firstDate));
}  !!

/* Determine if the given year is a leap year.  Works
for any positive integer. */
Def isLeapYear(self, year)
{
 ^((year mod 4 = 0) and (year mod 100 <> 0))
   or (year mod 400 = 0);
}  !!

/* Initialize the array which holds the integer day
numbers corresponding to the beginning of each year
from 1988 to 2010.  Ie., the entry for 1988 is
1 while the integer stored for 1989 (index 2) is 367. */
Def initJanFirst(self | thisyear)
{
 janFirst := new(Array,23);
 janFirst[0] := 1;
 thisyear := 1989;
 loop
```

```
 while thisyear < 2011
 if isLeapYear(self,thisyear - 1)
 then janFirst[thisyear - 1988] :=
   janFirst[thisyear - 1989] + 366;
 else janFirst[thisyear - 1988] :=
   janFirst[thisyear - 1989] + 365;
 endif;
 thisyear := thisyear + 1;
 endLoop;
}   !!

/* Initialize the array which holds the integer
representations of the first days of each of the twelve
months of a year which is not a leap year.  The array
indices run from 0 to 12 but the first day values are
stored in elements 1 through 12 so that the month
numbers match the array indices. */
Def initStartMo(self)
{
 startMonth := new(Array,13);
 startMonth[1] := 1;
 startMonth[2] := 32;
 startMonth[3] := 60;
 startMonth[4] := 91;
 startMonth[5] := 121;
 startMonth[6] := 152;
 startMonth[7] := 182;
 startMonth[8] := 213;
 startMonth[9] := 244;
 startMonth[10] := 274;
 startMonth[11] := 305;
 startMonth[12] := 335;
}  !!

/* Initialize the Calendar object by receiving the
current date from the user in the format mm/dd/yyyy,
initializing the instance varibles startMonth and
janFirst, and calculating the correct value to be
stored as a string in the instance variable "tomorrow". */
Def init(self | aInpDlg)
{
 aInpDlg := new(InputDialog,"SET THE DATE",
   "Please type today's date in the format: mm/dd/yyyy",
   "");
 if runModal(aInpDlg,INPUT_BOX,ThePort) == IDOK
 then today := getText(aInpDlg);
 endif;
 initStartMo(self);
 initJanFirst(self);
 tomorrow := numToDate(self,(dateToNum(self,today) + 1));
}  !!
```

5. ADDITIONS TO THE STRING CLASS

```
/* Display an input dialog box with self as the caption,
 and str as the prompt to the user.  Then return the
 user's input to the caller. */
Def inputBox(self,str | aInpDlg)
{
 aInpDlg := new(InputDialog,self,str,"");
 if runModal(aInpDlg,INPUT_BOX,ThePort) == IDOK
 then ^getText(aInpDlg);
 endif;
} !!

/* Show a question box with self as the caption, str as
message, and yes and no response boxes. */
Def questionBox(self,str | ans)
{
 ans := Call MessageBox(handle(ThePort),lP(str),lP(self),
  MB_YESNO bitOr MB_ICONQUESTION);
 freeHandle(str);
 freeHandle(self);
 ^ans;
} !!

/* Show an error box with self as the caption, str as message.  */
Def errorBox(self, str)
{ new(ErrorBox, ThePort, str, self,
            MB_OK bitOr MB_ICONHAND);
} !!

/* Compare the values of the item to be inserted in a
binary tree and the data in the node of interest. */
Def compareNode(self, item)
{
 if (self = item)
 then ^0
 else if (self > item)
  then ^-1
  else if (self < item)
   then ^1
   endif;
  endif;
 endif;
} !!
```

APPENDIX B - THE VIDEO MANAGEMENT PACKAGE IN PASCAL

```
program VIDEOS_R_US;

            {-----------------------------}
            {    TITLE : VIDEOS_R_US      }

{ It is designed as a daily management program for the video store, Videos-}
{ R-Us. It includes procedure SET_UP which allows the manager to get the  }
{ program running every business day morning and to set desired rental rate}
{ values. After these have been initialized, the program's main menu is   }
{ displayed. The menu allows the store clerks to perform 12 diferent      }
{ operations such as checking out a tape or adding a new customer to the  }
{ store data base. After the option is completed, the menu is again       }
{ displayed until the quit option is selected at the end of the day.      }
{-------------------------------------------------------------------------}


const
  CLOW = 1001;              { Lowest number assigned to a customer }
  CHIGH = 1299;             { Highest number assigned to a customer}
  TLOW = 101;               { Lowest number assigned to a tape     }
  THIGH = 199;              { Highest number assigned to a tape    }
  SHSTRING = 15;            { Common string length for short items }
  MEDSTRING = 30;           { Common string length for medium items}
  LSTRING = 50;             { Common string length for long items  }
  MAXOUT = 4;               { Greatest number of tapes that can be }
                            {   checked out by one customer        }

type
  CNUMBER = CLOW..CHIGH;          { Index for customer array       }
  RATINGS = ( G, PG, PG13, R, X );
  OK_RATINGS = SET OF RATINGS;    { Allowable ratings for a customer }
  NUM_CHKD_OUT = 0..MAXOUT;       { Number of tapes which a customer }
                                  {   has checked out              }
  TNUMBER = TLOW..THIGH;          { Index for tape array           }
  SHORT_STR = STRING[SHSTRING];
  TAPES_CHKD_OUT = SET OF TNUMBER;
  MED_STR = STRING[MEDSTRING];
  LONG_STR = STRING[LSTRING];

  CADDRESS = RECORD               { Customers address }
    STREET : MED_STR;
    CITY_ST : MED_STR;
    PHONE : SHORT_STR;
  end;

  CNAME = RECORD                  { Customers name    }
    LAST_NAME : SHORT_STR;
    FIRST_NAME : SHORT_STR;
```

```
  end;

  CUSTOMER = RECORD
    NAME : CNAME;
    ADDRESS : CADDRESS;
    OK_RATING : OK_RATINGS;
    NUM_CHK_OUT : NUM_CHKD_OUT;
    TAPES_CHK_OUT : TAPES_CHKD_OUT;
    DELETED : CHAR;              { Is the customer data still valid  }
  end;

  DATE = RECORD
    YEAR : INTEGER;
    MONTH : 1..12;
    DAY : 1..31;
  end;

  TAPE = RECORD
    TITLE : LONG_STR;
    RATING : RATINGS;
    CHECKED_OUT : CHAR;          { Is it checked out  }
    WHO_HAS_IT : CNUMBER;        { Customer number of person who has it}
    DATE_OUT : DATE;
    DELETED : CHAR;
  end;

  CUST_ARRAY = array [CNUMBER] of CUSTOMER;
  TAPE_ARRAY = array [TNUMBER] of TAPE;


var
  TAPE_LIST : TAPE_ARRAY;
  CUST_LIST : CUST_ARRAY;
  CUR_DATE : DATE;               { Current date }
  BARG_RATE,                     { Rental rate on bargain days }
  TAPE_RATE,                     { Normal tape rental rate     }
  X_TAPE_RATE,                   { X-rated tape rental rate    }
  TAX_RATE,                      { Current tax rate            }
  LATE_FACTOR : REAL;            { Multiplier for rate levied  }
                                 {  against late tapes         }
  BARG_DAY : BOOLEAN;            { Is this bargain day?        }
  CHOICE : INTEGER;              { Users choice from MENU      }
  {$I B:CHECKOUT.PAS}
                                 { Turbo Include file name which}
                                 { contains the procedure for   }
                                 { checking out a tape.  Space  }
                                 { limitations required this.   }


procedure SET_UP;
```

```
  { This procedure is designed to be used by the store manager. It    }
  { prompts him to enter the current date, and to confirm or redefine  }
  { rates for tape rental, tax, and late charges.                      }

var
  ANS : STRING[5];                { User input }
  I : INTEGER;          { Dummy variable used in string-to-real conversion }

begin           { procedure SET_UP }

  { 73 LINES OF PASCAL CODE }

end;            { procedure SET_UP }


procedure LOAD_CUST_ARRAY;

  { This procedure loads the array of customer data from the disk file     }
  { CUSTFILE. It initially tags all array elements as deleted and then     }
  { reads the tape numbers and loads the records of the tapes actually in  }
  { the stores inventory.                                                  }

var
  DATA_FILE : TEXT;
  I : INTEGER;                  { Loop control variable }
  SPACE : CHAR;                   { Dummy variable used in reading file }
  RATED : STRING[5];                { Used to read tape rating from file  }
  TAPES : STRING[3];                { Used to read tape numbers from file }
  TAPE : INTEGER;                 { Integer version of TAPES }
  CUSTNUM : CNUMBER;                    { Customer number in file }

begin

  { 50 LINES OF PASCAL CODE }

end;        { procedure LOAD_CUST_ARRAY }


procedure LOAD_TAPE_ARRAY;

  { Reads data on the store's tapes and loads it into the array        }
  { TAPE_LIST. The first item read is the tape number which is the     }
  { index of the array but is not a field of the record.               }

var
  DATA_FILE : TEXT;
  SPACE : CHAR;               { Dummy variable used in reading file }
  RATED : STRING[5];            { Used for input of tape rarings }
  I : INTEGER;                { Loop control variable }
  TAPENUM : TNUMBER;
```

```
begin

  { 40 LINES OF PASCAL CODE }

end;                    { procedure LOAD_TAPE_ARRAY }


procedure MENU (var CHOICE: INTEGER);

  { This procedure presents the user with twelve options which manipulate   }
  { the customer or tape arrays or both to accomplish the desired action.   }
  { After the action is completed, the MENU is again displayed, waiting for }
  { another request from the user. A quit option is also given which causes}
  { the updated arrays to be written to disk at the end of the business day.}

var
  ANS : CHAR;             { User response to confirm that quit was chosen }
  VALID_CHOICE : BOOLEAN; { Flag which prevents illegal menu CHOICE       }

begin

  { 41 LINES OF PASCAL CODE }

end;                  { procedure MENU }


procedure FIND_TAPE_NAME (var TAPE_NUM: INTEGER;
                          var FOUND, GIVE_UP : BOOLEAN );

  { This procedure is called by procedures FIND_TAPE, DEL_TAPE, and   }
  { CHANGE_TAPE. It is used to locate a tape by searching the name    }
  { fields of the TAPE_LIST array.                                    }

var
  ANS : CHAR;                 { User input }
  TAPE_TITLE : LONG_STR;
  I : INTEGER;                { Loop control variable }

begin

  { 35 LINES OF PASCAL CODE }

end;                        { procedure FIND_TAPE_NAME }


procedure FIND_TAPE;

  { This procedure calls FIND_TAPE_NAME TO search for the tape name that  }
  { the user desires. If FOUND, all of the pertinent information on the   }
  { tape is displayed.                                                    }
```

```
var
  TAPE_NUM : INTEGER;              { Number of tape, determined after }
                                   { tape record is located           }
  FOUND,                           { Flag for a located tape title    }
  GIVE_UP : BOOLEAN;               { Flag for and aborted search      }
  ANS : CHAR;                      { User response                    }

begin

  { 41 LINES OF PASCAL CODE }

end;       { procedure FIND_TAPE }


procedure CHANGE_TAPE;

  { Given either the tape number or the tape title that the user is looking }
  { for, this procedure will locate it if possible. It will then allow the }
  { user to change either the tape title or the rating. The checked-out    }
  { status of the tape connot be changed via this procedure.               }

var
  TAPE_NUM : INTEGER;              { Same }
  FOUND,                           { as in }
  GIVE_UP : BOOLEAN;               { FIND_TAPE }
  ANS : CHAR;
  ANS2 : INTEGER;                  { User responses }
  ANS3 : 1..5;
  QUIT,                            { Flag indicates changes complete }
  CONTINUE,                        { Indicates valid change option   }
  SATISFIED : BOOLEAN;             { Indicates user is satisfied     }
                                   { with changes he has made        }

begin

  { 103 LINES OF PASCAL CODE }

end;             { procedure CHANGE_TAPE }


procedure DEL_TAPE;

  { Sets the delete flag in the record of a tape given either the tape }
  { number or its title. Calls procedure FIND_TAPE_NAME to search on   }
  { the tape name.                                                     }

var
  TAPE_NUM : INTEGER;
  FOUND,
  GIVE_UP : BOOLEAN;
```

```
  ANS : CHAR;              { User input }

begin

  { 21 LINES OF PASCAL CODE }

end;


procedure ADD_TAPE;

  { Searches through the tape array and allows user to add new tape   }
  { information to the lowest number tape record which has been   }
  { deleted.   }

var
  I : TNUMBER;             { Incremented to search tape array   }
  AVAIL_NUM_FOUND : BOOLEAN;   { If never true, tape array is full  }
  RATED : STRING[5];          { I/O version of field RATING  }
  CORRECT : BOOLEAN;          { Provides user opportunity to  }
                    { check accuracy of input  }
  ANS1 : CHAR;             { User input  }
  ANS2 : INTEGER;            { User input  }
  J : INTEGER;             { Loop control variable  }

begin

  { 100 LINES OF PASCAL CODE }

end;      { procedure ADD_TAPE }


procedure CHK_OUT_TAPE;

  { This lengthy procedure requires the tape number as an input from the   }
  { user.  It checks that the tape number is valid, not deleted, and not   }
  { checked out.  It can either accept the customer's number who wants to  }
  { check out the tape or search for his number given his first and last   }
  { names.  It checks that the customer's number is valid and not deleted.  }
  { Once these checks have been made and passed, the nested procedure   }
  { CHECK_RECORDS is called.  This procedure determines if the customer   }
  { can check out a tape with this tape's rating, if he already has four   }
  { tapes checked out, and if this tape is checked out.  If these checks   }
  { are passed, the double nested procedure DO_CHECK_OUT is called to set   }
  { the proper values in the customer and tape arrays and to calculate the   }
  { amount that the clerk needs to collect.  The clerk is then given a     }
  { chance to repeat the whole procedure for another tape and a running    }
  { total of COST is kept.   }

var
  COST : REAL;               { Total amount to be collected }
```

```
  COST_OUT : REAL;          { Cost of each tape }
  I,                  { Loop control variable }
  TAPE_NUM,
  CUST_NUM : CNUMBER;
  ANS : CHAR;               { User input }
  LNAME,
  FNAME : SHORT_STR;          { User inputs for last and first names }
  VALID_CUST,               { Flag for valid CUST_NUM }
  VALID_TAPE,               { Flag for valid TAPE_NUM }
  TAPE_OUT,                 { Tape check out has been allowed }
  DONE : BOOLEAN;           { No more tapes to check out }

procedure CHECK_RECORDS (CUST_NUM, TAPE_NUM: INTEGER);

  { Procedure is nested within CHK_OUT_TAPE and performs as described above }

procedure DO_CHECK_OUT (CUST_NUM, TAPE_NUM : INTEGER);

  { This procedure is nested one level deeper, within procedure CHECK_RECORDS }
  { and performs as described above. }

begin           { procedure DO_CHECK_OUT }

 { 29 LINES OF PASCAL CODE }

end;    { procedure DO_CHECK_OUT }


begin    { procedure CHECK_RECORDS }

 { 39 LINES OF PASCAL CODE }

end;    { procedure CHECK_RECORDS }


begin       { Main procedure, CHK_OUT_TAPE }

 { 93 LINES OF PASCAL CODE }

end;          { procedure CHK_OUT_TAPE }


procedure ADD_CUST;

{ This procedure allows the user to add customer data to the next available  }
{ record in an array of customer records. i.e., the lowest available customer }
{  number }

var
  AVAL_NUM: CNUMBER;
```

```
procedure UP_CASE (var CHANGES :SHORT_STR);

var
  I:INTEGER;

begin { Start the upcase procedure }

  { 2 LINES OF PASCAL CODE }

end;  { Finish the upcase procedure }


procedure UP_CASEM (var CHANGES :MED_STR);

var
  I:INTEGER;

begin { Start the upcase procedure }

  { 2 LINES OF PASCAL CODE }

end;  { Finish the upcase procedure }


procedure VALID_RATING;

{ This procedure assigns ratings that can be checked out by the customer }

var
   S: SHORT_STR;
   ANS: CHAR;
   T: RATINGS;

begin { nested valid_rating }

  { 19 LINES OF PASCAL CODE }

end; { Conclude the nested procedure of valid_ratings }

begin { start the body of procedure add_cust }

  { 38 LINES OF PASCAL CODE }

end; { Conclude the main body of the ADD_CUST procedure }


Procedure DEL_CUST;

{ This procedure allows the user to delete a customer by setting the }
```

```
{ DELETED variable to  char Y if the record is deleted and available }
{ for reassignment to another customer }

var
   DEL_NUMBER: CNUMBER;

begin  { start the procedure  DEL_CUST }

  { 10 LINES OF PASCAL CODE }

end; { conclude procedure DEL_COST }


procedure INVENTORY;
{ This procedure counts the total number of tapes in the inventory }
{ and assigns the total inventory to a variable (TAPE_COUNT), it also  }
{ takes the total number of tapes checked out and assigns them to the  }
{ variable (TAPES_OUT) , it calculates the total number of tapes on   }
{ the shelf and assigns it to the variable (TAPES_ON_SHELF) }

var
   CUST_NUM: CNUMBER;
   TAPE_NUM: TNUMBER;
   TAPES_OUT,
   TAPE_COUNT,
   TAPES_ON_SHELF: INTEGER;
   TOTAL_OUT : INTEGER;
   ANS : CHAR;                    { User input }

begin  { start the main body of the inventory procedure }

  { 39 LINES OF PASCAL CODE }

end; { Conclude the inventory procedure }


procedure LATE_TAPES;

{ This procedure searches the tape array for tapes that have been checked }
{ out for more than one day and displays the names of customers who have }
{ had these tapes. In addition, it also calculates the total days late up }
{ to 28 days and assigns any tapes later than 28 Days  the max. value of 28 }
{ When a tape is 28 days late the customer will be required to pay for }
{ the replacement cost of the tape. }

const
   PAID = 1; { one days rental of the tape }

var
   SUBDAY,
   TAPE_NUM,
```

```
   TAPE_NUM,
   TDAYS_LATE,
   DAYS_LATE: INTEGER;
   ANS : CHAR;                { User input }

procedure FIGURE_DAYS;

{ This nested procedure assigns total days in the month to varible subday }

begin

  { 13 LINES OF PASCAL CODE }
end;

begin { Start the main body of the LATE_TAPES procedure  }

  { 116 LINES OF PASCAL CODE }

end; { Conclude the LATE_TAPE Procedure }


procedure CHANGE_CUST;

{ This procedure finds a given customer, displays his record, and allows }
{ user to make changes to any of the applicable fields }

var
   VALIDATE,
   VALID: CHAR;
   CUST_NUM_NAM: STRING[5];
   INDX: CNUMBER;
   OPTION,
   NEW_LNAME,
   NEW_FNAME: SHORT_STR;
   NEW_STREET,
   OPTION2,
   NEW_CITY_ST: MED_STR;
   I,                    { Dummy variable use in VAL function }
   TEMP_NUM,
   CHOICE: INTEGER;
   FOUND: BOOLEAN;

procedure CH_FNAME; { changes the customers first name }

begin

  { 6 LINES OF PASCAL CODE }

end;
```

```
procedure CH_LNAME; { changes the customers last name }
begin
  { 5 LINES OF PASCAL CODE }
end;

procedure CH_STREET; { this procedure changes the customers street address }
begin
  { 4 LINES OF PASCAL CODE }
end;

procedure  CH_CITY_ST;  { this procedure changes the city, state, and zip }
begin
  { 4 LINES OF PASCAL CODE }
end;

procedure CH_RATINGS; { this procedure changes the customer ratings }
var
   S: STRING[5];
   ANS: CHAR;
   T: RATINGS;
begin   { Start the CH_RATINGS procedure }
  { 24 LINES OF PASCAL CODE }
end; { conclude the procedure CH_RATINGS }

procedure CH_PHONE; { this procedure changes the customers phone number }
var
  TELEPHONE: SHORT_STR;
begin    { start the change phone procedure }
  { 7 LINES OF PASCAL CODE }
end;     { conclude the change phone procedure }
```

```
begin {start the main body of CHANGE_CUST }

  { 74 LINES OF PASCAL CODE }

end; { conclude the change customer procedure }


procedure FIND_CUST;

{ This procedure takes the customers last name and searches the customer }
{ list array and prints out the customer record so customer can be found }

var
  FNAME,
  LNAME: SHORT_STR;
  FNUMBER: CNUMBER;
  TAPE_NUM: TNUMBER;
  ADDRESS: CADDRESS;
  S:STRING[5];
  T:RATINGS;
  FOUND: BOOLEAN;
  I: INTEGER;                { Loop control variable }

begin { Start the procedure FIND_CUST }

  { 79 LINES OF PASCAL CODE }

end; { Conclude the procedure FIND_CUST }


procedure  CHK_IN_TAPE;

var
  TEMP_NUM : CNUMBER;
  TAPE_NUM : TNUMBER;
  QUIT,
  OPTION : STRING[2];

begin

  { 32 LINES OF PASCAL CODE }

end;              { procedure CHK_IN_TAPE }


procedure SAVE_CUSTFILE;

   { This procedure is called when the quit option is selected from      }
   { the main program menu, presumably at the end of the business day.   }
```

```
  { It takes all non-deleted customer records from the array CUST_LIST   }
  { and writes them back to the text file called CUSTFILE.TXT.  At the   }
  { start of the next business day, procedure LOAD_CUST_ARRAY  retrieves }
  { the file and reloads the array.                                      }

var
  DATA_FILE : TEXT;
  I,
  J : INTEGER;                { Loop control variables }
  NUMBLANK : INTEGER;         { Number of blanks added to LAST_NAME  }
                              { to write it in readable format       }

begin

  { 49 LINES OF PASCAL CODE }

end;                  { procedure SAVE_CUSTFILE }


procedure SAVE_TAPEFILE;

  { Saves the undeleted records in the array TAPE_LIST to a disk text    }
  { file named TAPEFILE.TXT.  If a tape is not checked out, dummy values }
  { are placed in the WHO_HAS_IT and DATE_OUT fields.                    }

var
  DATA_FILE : TEXT;
  I : INTEGER;                { Loop control variable }

begin

  { 40 LINES OF PASCAL CODE }

end;                  { procedure SAVE_TAPEFILE }



begin           { MAIN BODY OF PROGRAM }

  SET_UP;
  LOAD_CUST_ARRAY;
  LOAD_TAPE_ARRAY;
  MENU (CHOICE);

  while CHOICE <> 13 do
    begin
      case CHOICE of
        1: CHK_OUT_TAPE;
        2: CHK_IN_TAPE;
        3: ADD_TAPE;
        4: DEL_TAPE;
```

```
          5: FIND_TAPE;
          6: CHANGE_TAPE;
          7: LATE_TAPES;
          8: INVENTORY;
          9: ADD_CUST;
         10: DEL_CUST;
         11: FIND_CUST;
         12: CHANGE_CUST;
        end;

        MENU (CHOICE);

      end;          { while CHOICE <> 13 do }

    SAVE_CUSTFILE;
    SAVE_TAPEFILE;

end.              { program VIDEO_R_US }
```

APPENDIX C - A BINARY TREE PACKAGE

1. THE BINARY SEARCH TREE CLASS

```
/* Objects of this class are binary search trees.
Specifically, for each node of type BinaryNode in the
tree, all nodes in the left subtree of that node have
key values which are less than that of the node in
question.  Of course, all nodes in the right subtree of
the node in question have key values greater than that
node. */!!

inherit(Object, #BinarySearchTree, #(root), 2, nil)!!

now(BinarySearchTreeClass)!!

now(BinarySearchTree)!!

/* Delete a node from a BinarySearchTree by invoking
methods in the BinaryNode class to search the proper
node path in the tree. */
Def deleteNode(self,item)
{
 deleteNode(root,item);
} !!

/* Insert an item into a binary search tree.  The
variable "item" is the value to be inserted.  The
compareNode method returns:
        -1 if data > item
        +1 if data < item
         0 if data = item    */
Def addNode(self,item | result, search, searchparent)
{
 search := root;
 searchparent := root;
 loop
 while (search <> nil)
  result := compareNode(search.data,item);
  if result == 0
  then ^self;
  endif;
  searchparent := search;
  if result == -1
  then search := search.left
  else search := search.right
  endif;
 endLoop;
 search := buildNode(self,item);
```

```
 if result == -1
 then searchparent.left := search;
 else if result == 1
     then searchparent.right := search;
     endif;
 endif;
} !!

/* Build a new node to be a part of the BinarySearchTree.
Insert the given value in the "data" instance variable of
the BinaryNode class and check to see if this node should
be the root node of the tree. */
Def buildNode(self,item | node)
{
 node := new(BinaryNode);
 node.data := item;
 if root = nil
 then root := node;
 endif;
 ^node;
} !!

/* Invoke the preOrder traversal method of the BinaryNode
class by sending the appropriate message to the root node. */
Def preOrder(self)
{
 preOrder(root);
} !!

/* Invoke the postOrder traversal method of the BinaryNode
class by sending the appropriate message to the root node. */
Def postOrder(self)
{
 postOrder(root);
} !!

/* Invoke the inOrder traversal method of the BinaryNode
class by sending the appropriate message to the root node. */
Def inOrder(self)
{
 inOrder(root);
} !!
```

2. THE AVL TREE CLASS

```
/* Define a new class called "AVLTree". Members of this class are
nodes of an AVL, or height-balanced tree. Specifically, the left
and right subtrees of any node can differ in height (number of levels)
by no more than one. Objects of this class share the property of the
BinarySearchTree class that the data values of all nodes contained in
the left subtree are less than the parent node and all such values in
the right subtree are greater than the parent node. Special insertion
and deletion methods are required to maintain the height-balanced
property of the AVLTree class. The class has the following instance
variables:
  balance-- stores the difference in height between the left and
        right subtrees of the parent node. A value of 0 means
        subtrees of equal height. A value of 1 means the left
        subtree is higher than the right while a value of -1
        means the right is higher than the left.
  pivot--  the node on the path from the tree root to the node
        inserted or to be deleted with a balance of 1 or -1
        and which is closest to the node inserted or to be
        deleted.
  pivotparent-- the node whose child is the pivot node.
  pivotchild-- the node which is the child of the pivot node on the
        path to the node which was inserted or is to be deleted.
  adjustbalance-- amount by which a node's balance factor is adjusted
        by a rotation method.  */ !!

inherit(BinarySearchTree, #AVLTree, #(pivot pivotparent pivotchild
adjustbalance), 2, nil)!!

now(AVLTreeClass)!!

/* Redefine the new method to invoke automatic
initialization. */
Def new(self)
{
 ^init(new(self:Behavior));
} !!

now(AVLTree)!!

/* Perform a double rotation where the inserted node
 occurs in the right subtree of the pivot node's left
 child.  */
Def doubleLRotate(self | grandchild)
{
 grandchild := pivotchild.right;
 pivotchild.right := grandchild.left;
 pivot.left := grandchild.right;
 grandchild.left := pivotchild;
 grandchild.right := pivot;
```

```
  select
    case grandchild.balance = 1
    is pivot.balance := -1;
      pivotchild.balance := 0;
    endCase
    case grandchild.balance := -1
    is pivotchild.balance := 1;
      pivot.balance := 0;
    endCase
    default pivotchild.balance := 0;
    pivot.balance := 0;
  endSelect;
  grandchild.balance := 0;
  pivotchild := grandchild;
}     !!

/* Perform a double rotation where the inserted node
  occurs in the left subtree of the pivot node's right
  child.  */
Def doubleRRotate(self | grandchild)
{
  grandchild := pivotchild.left;
  pivotchild.left := grandchild.right;
  pivot.right := grandchild.left;
  grandchild.right := pivotchild;
  grandchild.left := pivot;
  select
    case grandchild.balance = -1
    is pivot.balance := 1;
      pivotchild.balance := 0;
    endCase
    case grandchild.balance := 1
    is pivotchild.balance := -1;
      pivot.balance := 0;
    endCase
    default pivotchild.balance := 0;
    pivot.balance := 0;
  endSelect;
  grandchild.balance := 0;
  pivotchild := grandchild;
}    !!

/* Perform a single rotation where the inserted node
  occurs in the left subtree of the pivot node's left
  child.  */
Def singleLRotate(self);
{
  pivot.left := pivotchild.right;
  pivotchild.right := pivot;
  pivot.balance := 0;
  pivotchild.balance := 0;
```

```
}   !!

/* Perform a single rotation where the inserted node
  occurs in the right subtree of the pivot node's right
  child.  */
Def singleRRotate(self);
{
  pivot.right := pivotchild.left;
  pivotchild.left := pivot;
  pivot.balance := 0;
  pivotchild.balance := 0;
}     !!

/* This method is used for resetting the balance
  values for the nodes in an AVL tree from the pivot
  node to the newly inserted node.  This applies before
  any rotations are performed. */
Def resetBalance(self,item | search,result)
{
  result := compareNode(pivot.data, item);
  if(result = -1)
  then search := pivot.left;
    pivotchild := search;
    adjustbalance := 1;
  else search := pivot.right;
    pivotchild := search;
    adjustbalance := -1;
  endif;
  loop
  while (search.data <> item)
    result := compareNode(search.data, item);
    if (result = -1)
    then search.balance := 1;
      search := search.left
    else search.balance := -1;
      search := search.right;
    endif;
  endLoop;
}     !!

/* Finds the node closest to the insertion point
  which has a balance factor of +/- 1, called the pivot
  node. */
Def findPivot(self,item | result,search,searchparent)
{
  search := root;
  searchparent := root;
  pivot := root;
  loop
  while (search.data <> item)
  begin result := compareNode(search.data, item);
```

```
  if (search.balance <> 0)
  then pivot := search;
    pivotparent := searchparent;
  endif;
  if (result = -1)
  then pivotchild := pivot.left;
    searchparent := search;
    search := search.left;
  else pivotchild := pivot.right;
    searchparent := search;
    search := search.right
  endif;
 endLoop;
}    !!

/* Insert an item into an AVL tree. */
Def addAVLNode(self, item | result);
{
 addNode(self,item);
 findPivot(self,item);
 if (pivotchild = nil)
 then ^self;
 else resetBalance(self, item);
 endif;
 if (pivot.balance = 0)
 then pivot.balance := adjustbalance;
  ^self
 endif;
 if (pivot.balance + adjustbalance = 0)
 then pivot.balance := 0;
  ^self
 endif;
 if (adjustbalance = 1)
 then if (pivotchild.balance = 1)
     then singleLRotate(self);
     else doubleLRotate(self);
     endif;
 else if (pivotchild.balance = -1)
     then singleRRotate(self);
     else doubleRRotate(self);
     endif;
 endif;
 if (pivotparent = pivot)
 then root := pivotchild
 else if pivot = pivotparent.left
     then pivotparent.left := pivotchild;
     else if (pivot = pivotparent.right)
         then pivotparent.right := pivotchild;
         endif;
     endif;
 endif;
```

```
}          !!

/*  Build a new node for an AVL tree by creating a new
AVLNode and setting its data value equal to "item". */
Def buildNode(self,item | node)
{
 node := new(AVLNode);
 node.data := item;
 if root.data = nil
 then root := node;
 endif;
 ^node;
}      !!

/* Initialize instance variables as necessary. */
Def init(self)
{
 root := new(AVLNode);
 pivot := root;
 adjustbalance := 0;
}   !!
```

3. THE BINARY NODE CLASS

```
/* Objects of this class form nodes in a binary search tree.
The instance variable "left" holds a pointer to the left
subtree, the variable "right" a pointer to the right subtree,
and the variable "data" the value stored in the node.  Note
that the type of "data" is not specified and can be anything,
so long as a meaningful comparison operation can be performed
on it. */

inherit(Object, #BinaryNode, #(left right data), 2, nil)!!

now(BinaryNodeClass)!!

now(BinaryNode)!!

/* This method is used by deleteNode to find the
  right- most member of the current node's left
  subtree. */
Def subdeleteNode(self | parent, remove)
{
  if (right = nil)
  then ^nil;
  endif;
  parent := self;
  remove := parent.right;
  loop
  while (remove.right <> nil) parent := remove;
    remove := remove.right;
  endLoop;
  parent.right := nil;
    /* Detach right-most node */
  ^remove.data;
  /* Put its value in the calling node */
} !!

/* Delete an item from a binary search tree */
Def deleteNode(self, item | result)
{
 result := compareNode(data, item);
 if (result = 1)
 then right := deleteNode(right, item);
 /* item > data, return the right subtree */
 ^self
 else if (result = -1)
   then left := deleteNode(left, item);
   /* item < data, return the left subtree */
   ^self
   else if (right = nil)
   /* No right child, so replace node with left subtree */
     then data := left;
```

```
     ^self
    else if (left = nil)
    /* No left child, replace node with right subtree */
     then data := right;
     ^self
     else data := subdeleteNode(left);
     ^self
    endif;
  endif;
 endif;
endif;
} !!

/* Traverse a BinaryTree pre-order. That is, visit
  the parent node first, then the left leaf, and lastly
  the right leaf. */
Def preOrder(self)
{ print(data);
 print(',');
 if left
 then preOrder(left);
 endif;
 if right
 then preOrder(right);
 endif;
} !!

/* Traverse a BinaryTree post order. That is, visit the
   left leaf, the right leaf, and then the parent node. */
Def postOrder(self)
{
 if left
 then postOrder(left);
 endif;
 if right
 then postOrder(right);
 endif;
 print(data);
 print(',');
} !!

/* Traverse a BinaryTree in order. That is, visit the left
   node, the parent node, and then the right node. */
Def inOrder(self)
{
 if left
 then inOrder(left)
 endif;
 if (data <> nil)
 then print(data);
   print(',');
```

```
  endif;
 if right
 then inOrder(right)
 endif;
}
 !!

/* Compare the values of the item to be inserted in
 the tree and the data in the node of interest. */
Def compareNode(self, item)
{
 if (self = item)
 then ^0
 else
  if (self > item)
  then ^-1
  else ^1
  endif;
 endif;
} !!
```

4. THE AVL NODE CLASS

```
/* Instances of this class form nodes of an AVL or height-
balanced tree. The "left" and "right" instance variables are
pointers to either nil or other AVL nodes. The "balance"
instance variable holds one of the three values -1, 0, or +1.
These values are interpreted as follows:
     -1: the subtree starting at "right" is one level higher
        than the subtree starting at "left"
      0: both the tree rooted at "left" and the tree rooted
        at "right" have the same height
     +1: the subtree rooted at "left" is one level higher than
        the subtree rooted at "right"
The "data" instance variable simply holds the data object which
is to be stored in the tree. Note that this can be of any
class, so long as there is a basis for comparing and ranking
data values represented in the objects. */!!

inherit(BinaryNode, #AVLNode, #(balance), 2, nil)!!

now(AVLNodeClass)!!

/* Invoke the init method to initialize the instance
variables of the AVL node. */
Def new(self)
{
  ^init(new(self:Behavior));
}   !!

now(AVLNode)!!

/* Set initial values for the class instance variables. */
Def init(self)
{
  left := nil;
  right := nil;
  data := nil;
  balance := 0;
} !!
```

LIST OF REFERENCES

1. MacLennan, B. J., *Principles of Programming Languages,* Second Edition, pp. 443-483, Holt, Rinehart and Winston, New York, New York, 1987.
2. Brooks, F. P., Jr., *The Mythical Man-Month,* Addison-Wesley, Reading, Massachusetts, 1975.
3. Standish, T. A., *Data Structure Techniques,* Addison-Wesley, Reading, Massachusetts, 1980.
4. Duff, C., and others, *Actor Language Manual,* Version 1.1, The Whitewater Group, Inc., Evanston, Illinois, 1987.
5. Duff, C., and others, *Actor Training Course Manual,* Version 1.1, The Whitewater Group, Evanston, Illinois, 1988.
6. Pascoe, G. A., "Elements of Object-Oriented Programming," in *Byte*, pp. 139 - 144, August 1986.
7. Parnas, D. L., "Information Distribution Aspects of Design Methodology," in *Information Processing*, pp. 339 - 342, North-Holland Publishing Company, 1972.
8. Aho, A. V., Hopcroft, J. E., and Ullman, J. D., *Data Structures and Algorithms,* pp. 11 - 15, Addison-Wesley, Reading, Massachusetts, 1983.
9. Shankar, K. S., "Data Design: Types, Structures, and Abstractions," in *Handbook of Software Engineering*, ed. Vick, C. R. and Ramamoorthy, C. V., pp. 247 - 266, Van Nostrand Reinhold Company, New York, New York, 1984.
10. Stubbs, D. F. and Webre, N. W., *Data Structures with Abstract Data Types and Pascal,* Brooks/Cole Publishing Company, Monterey, California, 1985.
11. Reingold, E. M. and Hansen, W. J., *Data Structures in Pascal,* p. 321, Little, Brown and Company, Boston, Massachusetts, 1986.
12. Horowitz, E. and Sahni, S., *Fundamentals of Data Structures,* pp. 442 - 456, Computer Science Press, Rockville, Maryland, 1982.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Chief of Naval Operations Director, Information Systems (OP-945) Navy Department Washington, D.C. 20350-2000	1
4.	Commandant of the Marine Corps Code TE 06 Headquarters, U.S. Marine Corps Washington, D.C. 20360-0001	1
5.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943-5000	2
6.	Curriculum Officer, Code 37 Computer Technology Naval Postgraduate School Monterey, California 93943-5000	1
7.	Professor C. Thomas Wu, Code 52Hq Computer Science Department Naval Postgraduate School Monterey, California 93943-5000	5
8.	Major Dana E. Madison Academy of Health Science, U.S. Army Attn: Health Care Administration Division Fort Sam Houston, Texas 78234-6100	1

9.	Captain Michael O. Rowell c/o Mr. Ivan Rowell 3874 Sandhill Road Lansing, Michigan 48911	3

Thesis
R81862 Rowell
c.1 The suitability of an object-oriented language for prototyping and abstract data types.