MIT/LCS/TM-162

A MANAGER FOR NAMED, PERMANENT OBJECTS

Alan Michael Marcum

April 1980

A MANAGER FOR NAMED, PERMANENT OBJECTS

by

Alan Michael Marcum

June, 1979

© 1979 by Alan M. Marcum

Massachusetts Institute of Technology

Laboratory for Computer Science

Cambridge Massachusetts 02139

A Manager for Named, Permanent Objects

by

Alan Michael Marcum

Submitted to the Department of
Electrical Engineering and Computer Science
on May 17, 1979 in partial fulfillment
of the requirements for the Degrees of
Bachelor of Science and Master of Science.

ABSTRACT

Storing data in a computing system for a long time has been of interest ever since it was possible to do so. Classically, one stores bit- or byte-strings, or perhaps arrays of "records." Yet, current programming philosophy stresses data abstraction techniques and concepts.

This report describes an object-oriented filing system which stores abstract objects, and allows the user to view the system as though one were storing abstract objects, rather than storing some external representation of the abstractions. Names may be attached to the (permanent) objects, and objects may be contained in (and may contain) other objects. Furthermore, an object may be contained in more than one object, thereby allowing the naming structure to be a network.

CR Categories: 3.73, 4.33, 4.34, 4.9.

Key Words: filing system, data abstraction, permanent storage.

Thesis supervisor: David D. Clark,
Research Associate in Electrical Engineering and Computer Science

Dedication

I dedicate this thesis report to my parents, Stan and Helen. They have provided support and encouragement always. Sometimes, it has been difficult for them -- I would drive home from school at the end of the term, spend the night there, and then leave the next morning to go skiing, to see friends, to go out to HP to begin a work assignment. Often, I would spend more time driving home than I would spend at home.

Thanks, folks, for your love, your support, your understanding, your friendship. To express my appreciation, in whatever small way this might be, I dedicate this work to you.

Acknowledgements

The research on which this thesis is based was performed under the Electrical Engineering and Computer Science Department's Co-operative Education Program ("VI-A"), at Hewlett-Packard Laboratories, Computer Research Laboratory, in Palo Alto, California. To express my appreciation for the opportunity of participating in VI-A, my first thanks go to the Department's Co-operative Education Program, and especially to its director, John Tucker, and his secretary, Lydia Wereminski.

My thesis supervisor, Dave Clark, provided me with constant guidance, advice, and support. His efforts to bridge the continental distance between us while I worked at HP were extraordinary. His efforts to help me clarify my thoughts and the exposition of those thoughts were remarkable. Dave read preliminary versions more quickly than I could reasonably expect. My deepest thanks go to him.

The members of the Computer Systems Research Group at MIT's Laboratory for Computer Science have helped me crystallize some of the ideas presented in the following chapters. Many of them helped me, despite my very brief association with the group. Some of the people, deserve special thanks. Allen Luniewski has been mentioned in several of the other CSR theses I have read recently; despite a very busy schedule, and a thesis of his own to write, Allen has taken time to talk with me and help me. Karen Sollins likewise took time out from writing her thesis to discuss some of my ideas. Wayne Gramlich helped me find several references on Hydra. Gene Ciccarelli

Acknowledgements

and I talked a great deal late several nights. In general, people were just there, ready to talk, or to listen.

Thanks are due also to Roy Levin, of Xerox's Palo Alto Research Center (Xerox PARC). His exceptionally prompt reply to a request for information about Hydra is greatly appreciated.

Jerry Morrison, again of Xerox (but with their System Development Division -- Xerox SDD), helped transfer drafts of this report to HP Labs. My thanks to him for his assistance.

During the previous five years I have been associated with the MIT Varsity Rifle Team, first as a team member, then as team captain, and this year as assistant coach. In addition to the Rifle Team, there was also the entire MIT shooting community, in which I include the Varsity Pistol Team and the Pistol and Rifle Club, in addition to the Varsity Rifle Team. A finer, more fun-loving bunch of people exists nowhere. They gave me their friendship, companionship, and competition, and provided a refuge when I had to "get away from it all." Thank you for everything.

Finally, my thanks go to my co-workers at HP Labs. Many theses I have read claim that a list of such people is too long to include; I feel they all deserve notice. Jim Duley, Bob Fraley, Bruce Hamilton, Ron Johnston, Nancy Kendzierski, Jeff Levinsky, Martin Liu, Dave Means, Darrell Miller,

Acknowledgements

Bruce Nordman, Jim Stinger, Howard Steadman, Paul Stoft, and Ken Van Bree waded through drafts of the thesis proposal and the thesis report, giving me their comments and ideas, often with not nearly enough time to do what was asked, but it was always done nonetheless. Besides doing their jobs, these people, in addition to John and Lydia, help keep VI-A going. We had many discussions, ranging from friendly chats to heated debates, between just two of us, or with the entire group. Bob Fraley, Bruce Nordman, and Dave Means deserve special thanks: Bob for his special help in sorting out my ideas; Bruce for his assistance in transferring copies of drafts of this report to the people at the Lab; Dave for his consultation when I most needed it. Thank you, people, for your support -- both personal and technical.

I hereby grant to MIT and to the U.S. Government a non-exclusive, royalty-free, irrevocable, license to use, reproduce and distribute copies of my work, entitled A MANAGER FOR NAMED, PERMANENT OBJECTS.

The research on which this report is based was supported in part by the Computer Research Laboratory of the Electronics Research Center of Hewlett-Packard Company through the Electrical Engineering and Computer Science Department's Co-operative Education Program. It was also supported in part by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under Contract Number N00014-75-C-0661.

Disclaimer

The English language has no explicitly neuter personal pronoun. Many people consider this an unfortunate omission. However, traditional proper usage dictates that the personal pronoun "he" and its derivatives be used when a neuter personal pronoun is required. I shall follow tradition, and use "he," the Women's Liberation Movement notwithstanding. I do not mean to offend with my use of "he", merely to express myself cleanly and easily.

Table of Contents

ABSTRACT 2

DEDICATION 3

ACKNOWLEDGMENTS. 4

DISCLAIMER 7

TABLE OF CONTENTS. 8

TABLE OF FIGURES 10

I. INTRODUCTION. 11
- A. The Problem. 12
- B. The Environment. 14
- C. The Programming Language: Syspal. 16
- D. The "Things" 20
- E. Related Work 22
- F. Plan for the Remainder of this Presentation. 23

II. ANTECEDENTS 24
- A. Honeywell's Multics. 25
- B. Hewlett-Packard's MPE/3000 31
- C. Miscellaneous. 35
  - 1. Unix. 36
  - 2. Hydra 37
  - 3. Version Maintenance 38
- D. Summary. 39

III. DEFINITION OF A "CATOAN-OBJECT" 41
- A. Issues: Containment, and Trust. 42
  - 1. Containment and Catoan. 43
  - 2. Trust and Catoan. 45
- B. The Basic Object 47
  - 1. The Operations of the Basic Object. 52
  - 2. Comments on the "_SET" Operations 57
  - 3. Naming and the DIRECTORY. 58
  - 4. Storing Data: The CONTENTS 60
  - 5. Protection and Security 62
- C. A Refined Object 65
  - 1. Protection and Security 66
  - 2. Cross-Referencing 70

Table of Contents

D. Versioned Objects. 73
1. Version Naming. 74
2. Storing and Implementing Versions 76
3. More on Version Naming. 84
E. Summary. 86

IV. AN EXAMPLE: A SYSPAL PROGRAM OBJECT 87
A. Motivation 88
B. Definition 90
C. Use. 94
D. Summary. 96

V. IMPLICATIONS OF MULTIPLE NAMING ENVIRONMENTS 97
A. Disjoint Naming Spaces 98
B. A Standard Interface for Filing Systems. 100
C. Garbage Collection .103
D. Summary. 107

VI. SUMMARY, AND EVALUATION OF THE PROPOSED SOLUTION 108
A. Summary. 109
B. Completeness 111
C. Trade-offs 115
D. Remaining Work 118

APPENDIX A 120

REFERENCES 133

Table of Figures

Figure 1: Sample Representations of Multics Objects 29
Figure 1a: Directory. 29
Figure 1b: Segment. 30
Figure 1c: Link 31

Figure 2: Sample Representation of an MPE/3000 File 34

Figure 3: The Basic Catoan-Object 49

Figure 4: Catoan-Object with CONTENTS of Type "text". 52

Figure 5: An Access Control List Scheme for Catoan. 69

Figure 6: Additions to the Basic Object for Cross-Referencing . . . 72

Figure 7: Version Naming Hierarchy. 75

Figure 8: Additional Information and Operations for Version Maintenance 78

Figure 9: Definition of VERSION_GENERATING_PROCEDUREs 80

Figure 10: A Syspal-Program Object 91

Figure 11: Standard, Minimal Interface for a Filing System 101

Figure 12: A Module Implementing a Stack 127

CHAPTER ONE

INTRODUCTION

In this chapter, I describe the problem to which this report is addressed. The environment which was assumed during my research is described (including the types of computing systems at which the results presented here are aimed), as are the assumptions about that environment. The programming language used in the examples and descriptions in this report is also briefly described. A short description of the entities in the computing system which are addressed here is presented. I then discuss related work, and present a plan for the remainder of the presentation.

I.A. The Problem.

How does one store and reference things in a computing system? Especially, how does one store and reference things whose existence is longer than that of the process which created them? What is the structure of these "things" which are stored? How can they be manipulated? What are the common characteristics of most of the "things" in a computing system? Is there anything that can be done to those "things" which does not fit the model of "common characteristics"?

One of the important trends in current computer science research is data abstractions: programming using abstract data objects, whose representation is not only of no concern to the user, but is forcibly hidden from him.

When using a computing system, one usually wants to retain some data for long periods of time. This requires some form of permanent storage on the computing system, and a mechanism for accessing the data stored in the permanent storage. Unfortunately, many abstraction languages ignore the issue of permanence, retaining objects only for the life of the process which created them. Yet, users want permanent storage of their objects.

Chapter One — Introduction

Section A — The Problem

Once an object exists for longer than the life of its creating process, it is desirable to attach a human-usable, hopefully mnemonic, name to it. Such a desire requires a managing program for the names, and objects: to translate names to internal object references, to provide a uniform semantic interpretation for the names, and to manage the stored objects.

Classically, in order to permanently store an (abstract) object, and to attach a name to it, the object had to be transformed from its internal representation to some external representation (like a stream of bits). This external representation was then passed to a "file system," which stored the stream of bits representing the abstract object in a "file." Usually, the conversions from internal representation to external representation was very visible to the user. Such a transformation is undesirable, as it negates some of the benefits of data abstraction techniques.

In this thesis report, I shall address these, and other, issues. I shall describe the "things" stored in a computing system, and how one might manipulate, define, and characterize them. I shall compare and contrast this work with that of other schemes for referencing and manipulating "things." I shall examine how the definition of the "things" affects their naming and other properties.

I.B. The Environment.

Described here is a scheme aimed at a range of environments. It will work equally well on single user computing systems and on multiple user, shared systems. Often, on single user systems, some of the problems of concurrent accessing and of protection become moot points, and so the focus of this report will be on shared systems.

A virtual machine is similar to both the single and multiple user systems. Within one process or collection of processes, it appears to be single user. However, many virtual machines running on the same real machine often share logical, as well as physical, resources. For example, multiple virtual machines may share the same file system for permanent storage, thereby sharing not only the physical storage devices but also the logical naming space. The scheme presented in the following chapters will also fill the needs of a virtual machine environment.

Loosely connecting autonomous systems together to form a network of computers presents some problems which I shall not address. For example, there are the problems of naming resources on remote systems, locating resources on remote systems, and network-wide sharing and protection. It is hoped, however, that the general network case is a simple extension of the work described here for a single, multiple user system.

The specific environment assumed in this work is a single, multi-user computing system, with a large address space (for example, at least a trillion bits). Storage entities are accessed by presenting a unique identifier for the entity (such as an address, a segment number, or a capability) and an address within the entity to the memory management system, which is responsible for the allocation of and access to the memory resource. Within each entity in the system, references to other entities may exist, and they may exist anywhere within the entity (rather than in some particular location within the entity).

The memory resource is presumed to be virtual, though it could be entirely real memory, provided there is a sufficiently large non-volatile component. Permanent storage of an entity is achieved by not deleting the entity; future accessing must be done with the unique identifier used to create the entity. Memory _appears_ to be single level; all entities exist in the same collection of memory. In particular, the notion of separate permanent and temporary memories is foreign to my presumed environment.

In my assumed system environment, security is a major concern. An objective is to minimize the number of trusted components in the system. By "trust" I mean to give access to one's data, when that access is not for reason of explicit use. In most existing systems, the filing system is

trusted -- it can delete, modify, make inaccessible, or leak the data in any file in the system. In the proposals following, the filing system (object manager) need not be trusted to not modify or leak data. (It will still be able to delete data, and to make them inaccessible.) The only component of the system that will have to be trusted with one's data is the memory management system, which deals with data on a bit (or collection of bits) level, and can place data in any address space in the system. (If a single-level, non-volatile storage system is used, the memory manager need not have the "power" it would in a multiple-level, volatile (virtual) storage system.)

An additional aspect of my presumed environment is that the operating system provided is a kernel, to which some user-environment features have been added. The user-environment features need not be used if one desires to write a replacement (or simply do without the feature). The filing system provided with the kernel is part of the optional section of the system; therefore, multiple filing systems could exist.

I.C. The Programming Language: Syspal.

The examples presented in the following chapters use the "Syspal" programming language. Syspal [10] is a Pascal-based systems programming language being developed at Hewlett-Packard's Computer Research Laboratory.

Syspal is an object-oriented language, similar to MIT's CLU [21, 22] or Carnegie-Mellon's Alphard [34, 35, 37]. One defines an object by defining the operations one can perform on the object; the actual realization of the abstract object is not visible to its users. Following is a short summary of some of the features of Syspal which are used in this report; a summary of the relevant features of Syspal is in Appendix A.

Syspal provides only a very few types, and allows the programmer to extend those types. Specifically, Syspal includes no "string" for direct use. Throughout this report, strings will have the representation

```
string(size: 0 TO 100) = TYPE RECORD
    length: 0 TO size;
    chars:  ARRAY(1 TO size) OF CHAR;
    END;  !string
```

with all the usual string operations defined.

The definition of STRING points out several features of Syspal. Defined types can take one or more parameters which further specify the type. The string definition shown above takes "size" as its parameter, specifying the length of the string. The statement

```
life_history: string(50);
```

declares a variable as a string of length fifty.

There are two kinds of comments; a "here to end of line" comment (denoted by "!"), and a "here to end of comment" comment (which uses "(*" to open the comment and "*)" to close it).

Syspal allows pointers to be declared. Pointers are typed; that is, a pointer refers to an object of some particular type, rather than a pointer to anything (PL/1 pointers are of the latter flavor). As an example, the following could be the representation of a list. Like strings, lists are parameter-based: the type of the list's elements is supplied by the "abstraction" user.

```
list(element_type: TYPE) = TYPE RECORD
    first: @element_type;
    rest:  @list(element_type);
    END;  !list
```

The field "first" is a pointer to an object of type "element_type"; "rest" is a pointer to a list of type "element_type."

As a further example, shown in Figure 12 in Appendix A is a definition of a STACK abstraction which takes, as its parameters, the type of the objects on the stack, and the number of elements the stack will be able to contain. The definition takes the form of a "module," the Syspal equivalent of the CLU "cluster." The operations on stacks, a

representation of a stack, and various "interfaces" for, or "means of referencing," stacks are shown.

Within a module, the keyword SELF is bound to the object on which the operation was called. SELF is not included in the header of the function, but is supplied as the first argument to the operation when it is called. The name of the module need not be provided in the CALL statement; it is recognized from the type of the first argument. For example, with the declarations

```
envr: stack(algol_stack_frame);
x: algol_stack_frame;
algol_stack_frame = TYPE . . . ;
```

CLU would require a CALL similar to

```
CALL stack$push(envr, x);
```

whereas in Syspal, the same statement would be

```
CALL push(envr, x);
```

or, optionally,

```
CALL stack.push(envr, x);
```

if the fully-qualified operation name was desired. Within the module implementing stacks, "SELF" would refer to "envr" in the above example.

I.D. The "Things."

At the beginning of this chapter, I referred to the "things," the entities, stored in a computing system. What are those "things"? What are their properties, what is their structure, what operations can be performed on them?

The "things" to which I refer are the abstract data objects which are stored in the computing system's long-term ("permanent") storage. Such objects may be viewed as files, segments, programs, hierarchical or relational databases -- whatever one might want to retain for long periods of time. The various kinds of objects are defined by the operations which can be performed on them, in addition to those which can be performed on ALL objects. Most existing permanent-storage systems do not take this view, but, rather, view storage as a collection or stream of bits or bytes, or possibly as an array of "records." Indeed, some of the reports on current research on storage systems take a byte-stream view of storage, when such a view is not necessary (see, for example, [20]).

The view of objects as abstractions is similar to that which CLU, ALPHARD, Smalltalk [12], and Syspal take of data. An object is an abstract data type, out of which other abstract data types are made. An example of this is building a first-in, first-out queue from a linked list. The

programmer implementing the queue is not concerned with the implementation of the list abstraction, merely with the definition of the operations of the list (FIRST, REST, APPEND). If the input-output specifications of the operations on lists remain the same, changing the implementation of lists does not matter. Perhaps the person maintaining lists may decide that lists larger than some critical size should be stored using a different format; the user of lists does not care about internal representation.

Syspal provides the abstractions ARRAY, RECORD, INTEGER, CHAR and BOOL for direct use. And yet, one is not concerned with the implementation of such things; one merely wants to use them, often, as here, to build other, more complicated abstractions.

(In addition to the languages mentioned above taking a view of objects similar to mine, Hydra [36] has a similar view of objects which are to be stored for long periods of time. Again, objects are abstract (and explicitly extensible). There are other similarities between the Hydra view of objects and mine; these will be mentioned later, as appropriate.)

More details on abstract data types can be found in the previously cited references on CLU and Alphard.

I.E. Related Work.

The work which has most influenced my thinking about object management has been the research on data abstractions. Much of this work has its origins in SIMULA [6]. Parnas describes abstraction techniques [23]; CLU, Alphard, and Syspal all embody these concepts. It was the desire to store objects, rather than files, and to view storage as a collection of abstract data, rather than as bit or byte strings, which motivated this research.

The file systems of Honeywell's Multics [15], Bell Labs' Unix [26, 29, 32], and Hewlett-Packard's MPE/3000 [13] helped me determine the characteristics of the objects stored in a computing system. The naming structure is derived directly from Multics. Hydra's file system [36] views objects in a manner similar to that presented here.

Much of my thoughts on protection also were influenced by Multics. The capability-based schemes described by Wulf (Hydra, [36]), Lampson and Sturgis (Cal, [19]), and Saltzer [28] provided an interesting alternative to the Multics Access Control List (also described in [28], and in [15]).

Various mechanisms have been developed for version maintenance. Most of them simply store the object as a linear sequence of complete versions (for example, TENEX [7], ITS [9], and OS/VS1 [16, 17]). The Source Code

Control System (SCCS) [5, 11, 27], part of Unix's Programmer's WorkBench [8, 18, 29], implements a novel way of maintaining versions as a set of updates. SCCS also allows a (limited) hierarchy of versions. The scheme I propose is an immediate extension of that embodied in SCCS.

I.F. Plan for the Remainder of this Presentation.

In the following chapters, I describe "Catoan" (pronounced ku-tōn′ (1)), an object-oriented filing system for large, multi-user computing systems. Chapter Two describes previous work which influenced my thinking, especially about those attributes which are common to all permanently stored objects in a computing system such as the one I assume. In Chapter Three, my view of a "basic" object is developed, followed by a discussion of a "refined" object and a "versioned" object. In Chapter Four, I present an example of how one might use Catoan to store a Syspal program. Chapter Five examines the problems which arise when other filing systems, and, therefore, other naming schemes and spaces, are allowed to co-exist with Catoan. The final chapter, Chapter Six, contains an evaluation of Catoan, and describes areas where further research is needed.

----

(1) Notation from _Webster's New World Dictionary of the American Language_.

CHAPTER TWO

ANTECEDENTS

In this chapter, I shall discuss previous work which had a large influence on my research and thinking. The systems discussed here were studied as examples of ways to manage particular kinds of objects.

The typical kind of object in each of these systems is the "classical file," often appearing under different names (such as "segment"). A "classical file" is presented to the user as a string or stream of bits or characters. It does not have any structure, save in the way in which it is interpreted by the user. Usually, files are stored as blocks of contiguous bits, along with some system overhead information.

Sample representations of the files in Multics and MPE/3000 will be described using Syspal notation.

II.A. Honeywell's Multics.

The Multics file system is described abstractly by Saltzer [28], and concretely in the Multics Programmer's Manual [15]. Here, those features which most influenced this work are described.

There are two major kinds of objects in the Multics file system: "directories" and "segments." Directories contain mappings of character-string names to object references (unique identifiers); the objects can be either segments or other directories. Segments contain the data stored in the system. In addition to directories and segments, there are also "links" and "multi-segment files"; these will be discussed only briefly.

The objects in the Multics file system are arranged in a hierarchical fashion, starting from a directory called the "ROOT." Directories can be either nodes or leaves (generally, they are nodes; only an empty directory can be a leaf); segments must be leaves. Any object in the hierarchy can be named directly, by specifying the names of all the containing directories in order, starting from the ROOT. For example, the payroll for the month of June might be specified, using "^" as a name separator,

"ROOT^Accounting^payrolls^June" (assuming that the payroll function is part of the accounting department).

In addition to specifying a fully-qualified name (like that in the previous paragraph), local names are allowed, with the system automatically supplying the higher levels of qualification. This requires a slight change in the form of fully-qualified, or global, names: if the search for an object is to start at the ROOT, the first component of the name is not supplied, thereby beginning the name with the separator character. Therefore, the above example would become "^Accounting^payrolls^June"; a user executing in the "Accounting" (beneath the ROOT) directory could reference the same segment with "payrolls^June," and someone in the "Accounting^payrolls" directory (again, beneath the ROOT) could use simply "June."

Each object in the file system has some system information associated with it. Some of this information is part of all the types of objects; some of it is object-type particular. An example representation of a Multics directory, segment, and link appear in Figure 1. The most interesting parts of this information concern protection and sharing: the "access_control_list" and "ring_brackets." The access_control_list specifies the types of access granted to each user in the system. Directory access types are search (look in the directory), modify (change

entries in the directory), and append (add entries to the directory); segment access types are read (get the contents of the segment), execute (interpret the segment as a program), and write (change the contents of the segment). The ring_brackets specify the position in the system's protection rings (an extension of the supervisor-user mode concept; see [28]) in which the object can be accessed.

The Multics file system implements a strict hierarchy; therefore, each object in the system has exactly one parent (1), though directories can have multiple children. To allow an object to appear to exist in more than one directory, Multics provides "links". A link is a mapping of a local (one component) name to a global (fully qualified) name. Returning to the above payroll example, assume that top-level management wanted to access the payroll files, and desired to do so directly, rather than through the entire ^Accounting^payrolls^June name. A link might be created in the CorpMgt directory called "JunePay," which would be mapped into the name "^Accounting^payrolls^June."

An important point about Multics links is that they map local names to global names, not local names to object references. Such links are called

---

(1) This is true for all objects in the system except the ROOT, which has no parent.

"soft" links (1); their resolution is a two-step process: resolving the local name to a global name, and then resolving the global name to a unique internal identifier (segment number). This position need not be taken; Unix, for example, links local names directly to object references (see Section III.C.1).

A multi-segment file allows more than one segment's worth of data in one object (segments have a limited size). A multi-segment file appears to be very similar to a normal segment, though it is implemented as a directory, with the segments comprising the multi-segment file as children of the directory.

---

(1) A "hard" link maps a local name directly to an object reference.

```
multics_directory = TYPE RECORD
(* Defined types (such as ACCESS_ID) are shown in Figure 1c. *)
    access_class: string(32);       !Eg. Classified, Top Secret.
    access_control_list: ARRAY(*) OF RECORD
         id: access_id;             !Principal identifier
         modes: RECORD
              (s, m, a): BOOL;      !Search, Modify, Append
              END;  !modes
         END;  !access_control_list
    author: access_id;
    current_length: INTEGER;        !Number of pages.
    (date_time_dumped,
     date_time_entry_modified,
     date_time_modified,
     date_time_salvaged,
     date_time_used): multics_date_time;
    initial_access_control_lists: RECORD
         segment: LIKE multics_segment.acl;
         directory: LIKE multics_directory.acl;
         END;  !initial_access_control_list
    multisegment_file_indicator: INTEGER;    !Segments in multi-segment
                                    !file; 0 if not msf.
    names: ARRAY(*) OF string(32); !Names of this directory.
    quota: INTEGER;                 !Pages allowed under directory.
    records_used: INTEGER;          !Secondary storage.
    ring_brackets: RECORD           !Rings of protection.
         (m_a, s): rings;
         END;  !ring_brackets
    safety_switch: BOOL;            !Query user upon DELETE?
    security_out_of_service_switch: BOOL;    !Access class discrepancy
                                             !has been detected.
    type: ARRAY(3) OF BOOL (*segment, directory, link*) :=
              (FALSE, TRUE, FALSE);
    unique_id: INTEGER;
    name_map: ARRAY(*) OF RECORD    !Segments under this directory.
         name: string(32);
         object: UNION(@multics_directory,
                       @multics_segment,
                       @multics_link);
         END;  !name_map
    END;  !multics_directory
```

Figure 1a: Sample Representation of a Multics Directory.

```
multics_segment = TYPE RECORD
    access_class: string(32);       !Eg. Classified, Top Secret.
    access_control_list: ARRAY(*) OF RECORD
         id: access_id;             !Principal identifier
         modes: RECORD
              (r, e, w): BOOL;      !Read, Execute, Write.
              END;  !modes
         END;  !access_control_list
    author: access_id;
    bit_count: INTEGER;
    bit_count_author: access_id;    !Principal who last set BIT_COUNT.
    copy_switch: BOOL;              !Copy on write?
    current_length: INTEGER;        !Number of pages.
    (date_time_dumped,
     date_time_entry_modified,
     date_time_modified,
     date_time_used): multics_date_time;
    maximum_length: 0 To 262144;    !256K words.
    names: ARRAY(*) OF string(32); !Names of this segment.
    records_used: INTEGER;          !Secondary storage.
    ring_brackets: RECORD           !Rings of protection.
         (w, r, e): rings;
         END;  !ring_brackets
    safety_switch: BOOL;            !Query user upon DELETE?
    type: ARRAY(3) OF BOOL (*segment, directory, link*) :=
              (TRUE, FALSE, FALSE);
    unique_id: INTEGER;
    contents: ARRAY(262144) OF data_word;   !256K words.
    END;  !multics_segment
```

Figure 1b: Sample Representation of a Multics Segment.

```
multics_link = TYPE RECORD
    author: access_id;
    (date_time_dumped,
     date_time_entry_modified,
     date_time_used): multics_date;
    names: ARRAY(*) OF string(32); !Names of this link.
    type: ARRAY(3) OF BOOL (*segment, directory, link*) :=
              (FALSE, FALSE, TRUE);
    unique_id: INTEGER;
    linked_to_path: string(168);
    END;  !multics_link


access_id = TYPE RECORD
    person: string(15);
    project: string(15);
    instance: string(1);
    END (*access_id*);

rings = TYPE DISTINCT 0 TO 7;              !Rings of protection.

multics_date_time = TYPE 0 TO 2**64-1;     !Microseconds since
                                           !January 1, 1901 00:00 GMT.

data_word = TYPE 0 TO 2**36-1;
```

Figure 1c: Sample Representation of a Multics Link.

II.B. Hewlett-Packard's MPE/3000.

I examined MPE/3000 file system as an example of a "limited-hierarchical" file system. Users cannot create their own directories. Rather, the naming hierarchy is a fixed three-level system: file_name (segment name), group_name, account_name. The file_name is the "lowest" level name, the account_name, the "highest." If a higher level is

specified, all lower levels must also be specified. There is a very strict rule for interpreting names: a one level name is extended with the current group and account; a two level name is extended with the current account. A process executes under exactly one account and one group within that account for its entire lifetime; the notion of changing the "working directory" of the process does not exist.

Segments can be created only in the process's current group within the current account. Segments exist in exactly one place in the hierarchy, and have exactly one name; neither soft nor hard links exist. To reference a segment by another name, it must be renamed (if staying in the same group and account) or copied (in which case it becomes an entirely new entity).

Security is specified in two ways: with an aggregate-level access control list (called the "security matrix"), and with a password (lockword). The latter, if required, must be supplied whenever the segment is "opened" (made ready for use) or deleted. The security matrix is checked at times similar to those when the password is checked, and specifies the types of access various groups of users are granted.

The access types which can be granted are: read, append (write at the end of the segment), write (anywhere in the segment), lock (access the segment exclusively), and execute. The groups are: any (anyone in the

system), account user (anyone in the same account), account librarian (an account member deemed responsible for all the segments in an account), group user, group librarian, and creator. In addition, the "account manager" (a user who is responsible for administration of the account) has access to all the segments in that account, and the "system manager" (a user who is responsible for administration of all the accounts in the system) has access to all segments in the system.

Figure 2 shows a sample representation of a file in the MPE/3000 file system. This representation is rather abstract, and incomplete in detail. More detail can be found in [13].

```
HP3000_MPE_file = TYPE RECORD
    label: RECORD
         name: fname;                !File name.      --|
         group: fname;               !Group name.       |-- Full file name.
         account: fname;             !Account name.   --|
         creator: fname;
         lockword: UNION(null, fname);  !Must be supplied at OPEN
                                        !if non-NULL.
         security_matrix: ARRAY(5) OF RECORD !Who can access file.
           (* Subscripts:   1 - read      2 - append
              3 - write     4 - lock      5 - execute. *)
              (any,
               account_user,
               account_librarian,
               group_user,
               group_librarian,
               creator): BOOL;
              END (*security*);
         secure: BOOL;               !Is SECURITY_MATRIX enforced?
         date_created: julian_date;
         date_accessed: julian_date;
         date_modified: julian_date;
         file_type: word;            !Type (eg. program, APL workspace).
         access_flags: RECORD        !How file is being accessed.
              store: BOOL;           !File being backed-up to tape.
              restore: BOOL;         !File being recovered from tape.
              load: BOOL;            !Memory-resident program file.
              exclusive: BOOL;       !Opened for exclusive use.
              END;  !accesses
         how_open: RECORD
              write: BOOL;
              read: BOOL;
              END;  !how_open
         user_labels_written: halfword;
         user_labels_max: halfword;
         max_records: dbl_word;
         private_volume_info: bit_string(32);
         logical_record_size: word;
         block_size: word;
         last_block_size: word;
         records_in_file: dbl_word;
         END;  !label
    data: ARRAY(1 TO 2**47) OF CHAR;
    END;  !mpe_file
```

```
fname = TYPE alpha_string(8);
alpha_string(size: 0 TO 100) = TYPE RECORD
    length: 0 TO 100;
    char1: letters;
    charn: ARRAY(2 TO size) OF UNION(letters, "0" TO "9");
    END;  !alpha_string
letters = TYPE UNION("a" TO "z", "A" TO "Z");
halfword = TYPE 0 TO 255;
word = TYPE 0 TO 32767;
dbl_word = TYPE 0 TO 2147482711;
julian_date = TYPE RECORD
    year: 0 TO 99;
    day: 0 TO 366;
    END;  !julian_date
```

Figure 2: Sample Representation of an MPE/3000 File.

II.C. Miscellaneous.

In addition to the file systems of Multics and MPE/3000, various other file systems influenced my thinking on Catoan. Unix influenced my ideas on links and the structure of the naming environment (that is, whether to use a hierarchy or a network). Hydra's form of objects proved interesting. TENEX's file system supplies a form of version maintenance, as do those of ITS, OS/VS1, and many others. This section presents the various systems which were investigated and which made some (at least minor) contributions to this work.

II.C.1. Unix.

The Unix file system is similar to the Multics file system. Like Multics, Unix provides a hierarchical file system, with an access control list protection scheme. However, the hierarchy is not strict, and the access control list is more coarse than that of the Multics system.

Like Multics, Unix has, conceptually, two types of objects: directories and segments (files). However, unlike Multics, Unix segments can have multiple parents. Also, links in Unix are "hard" links (those in Multics are called "soft"). The local name is translated directly to a unique identifier (segment number -- "i-node" in Unix terminology), without the intervening global name. This is a more efficient form of link (it skips the additional name resolution step (1) when following the link), but that is relatively unimportant. Soft links provide greater indirection facilities than do hard links (because they can be bound to another link). Hard links, though, provide a known interpretation of a link, and make it easier for the owner of a segment to determine all the people using it. Implementing a complete cross-reference with soft links, for example, would require that the link be completely traced when it was created; in a hard link system, the link is directly resolved.

----

(1) Or steps: a soft link can bind a local name to another soft link.

Although Unix segments can have multiple parents (can be contained in multiple directories), directories cannot. This precludes building a general network in the Unix file system.

The protection scheme in Unix allows the object owner to specify access for certain groups of users, rather than on a user-by-user basis. The scheme is tied to the accounting system, with access being granted to the owner, to members of the owner's project (account), and to all users in the system. See [26, 29, 32] for more details.

II.C.2. Hydra.

The Hydra file system [1, 13] stores Hydra-objects, which are pseudo-abstract, and are each of a particular type or type extension. Each Hydra-object (call one "CRL") has two parts: the data part, and the "c-list." The actual data in CRL is stored in the data part. The c-list contains references to Hydra-objects which are contained in CRL. Every object in Hydra has both parts.

Because each Hydra-object has both a data and a c-list part, there is need for only one kind of object, which can function as both a "segment" and a "directory." However, one other important reason for including both

parts in all objects is that references to other objects cannot exist in the data part, but only in the c-list.

II.C.3. Version Maintenance: TOPS-20, ITS, OS/VS1, and SCCS.

Version maintenance has been a topic of interest for some time. TOPS-20 [7], ITS (1) [9], and OS/VS1 [16, 17] all provide similar forms of version maintenance. All three systems store each version in its entirety (as opposed to storing updates relative to some base version). Versions in TOPS-20 and ITS are linear, time-ordered sequences, referenced by numbers which increase from older to newer (more recent) versions. The default version (the version obtained if none is explicitly specified) is always the most recent version. The symbol ">" in ITS, and the (special) version number 0 in TOPS-20, reference the latest version on read and create a new version on write. The symbol "<" in ITS and the (special) version number "-2" in TOPS-20 access the oldest version.

OS/VS1's version naming scheme differs from that of TOPS-20 and ITS. It is a two-level system, allowing both a "generation" and a "version" specification. The specification becomes a suffix of "GnnnnVmm" to the regular file name, where "nnnn" is the "generation number" and "mm" is the

---

(1) ITS is an operating system developed at MIT for the PDP-10 family of computers.

"version number." This provides a limited tree-structure for version naming: generation within the file, and version within the generation. The suffix "(0)" references the latest generation; "(+1)" creates a new generation; "(-1)" references the previous generation, and "(-n)" references the nth previous generation. The automatic version maintenance system does not use the version field; it can be accessed directly by the user, however.

The Programmer's WorkBench under Unix provides a facility called the Source Code Control System [5, 11, 27] for version maintenance. SCCS allows versions to be arranged in a hierarchy, with the names representing a derivation sequence. Versions are stored as sets of updates to the previous version. I shall discuss SCCS further in Section III.D, "A Versioned Object."

II.D. Summary.

In this chapter, I have discussed various existing systems which significantly influenced the research presented in the following chapters of this report. The file systems of Honeywell's Multics and of Hewlett-Packard's MPE/3000 were described, with an examination of their abstract file structures. The Unix file system is very similar to that of Multics, except that a segment can be contained in more than one directory.

The structure of the Hydra file system was also discussed, especially the structure of the objects stored. Finally, existing version maintenance systems were described, including TOPS-20, ITS, OS/VS1, and the Source Code Control System.

CHAPTER THREE

DEFINITION OF A "CATOAN-OBJECT"

In this chapter, I describe the objects managed by Catoan. First, the "basic" object, its characteristics, its operations, and its representation will be defined. Then, a "refined" object, whose operations are less primitive than those of the basic object, will be presented. Lastly, objects which have explicit versions (such as programs) will be described.

III.A. Issues: Containment and Trust.

As will be shown later in this chapter, there are three ways to put data in a Catoan-object; all of them are different, all have different semantics and characteristics. But, why three ways?

In Chapter One, I wrote that "multiple filing systems could exist." Furthermore, "the filing system (object manager) need not be trusted to not modify or leak data." Both of these issues involve trust: need one trust the filing system, and, if not, what can be done about it?

What does it mean to "contain" something? What does it mean to "trust" something? In this introductory section, I shall explore these ideas as they relate to Catoan. Some of the issues I shall raise may not be clear until later in the chapter; I think this is better than delaying their discussion, however.

First, though, a little groundwork must be laid. The unit of storage in Catoan is the "Catoan-object"; let a typical Catoan-object be called "CRL." In data abstraction terminology, Catoan implements the abstraction "Catoan-object." Catoan-objects can "contain" other Catoan-objects, and other kinds of abstractions, too. Each Catoan-object has a DIRECTORY and a

CONTENTS; the things one normally puts in each of these is different, and things are put in them for different reasons, as will be explained.

III.A.1. Containment and Catoan.

What does it mean for a Catoan-object to "contain" another Catoan-object. What does it mean for a Catoan-object to "contain" any kind of (abstract) object?

Each Catoan-object (such as CRL) has a "CONTENTS," which specifies the abstract object which is the data of CRL. This is one form of "containment": containment in the CONTENTS. The primary reason for creating a Catoan-object is to provide a means for permanently storing, referencing, and naming the data of the CONTENTS.

The data which the object contains -- its CONTENTS -- should be readily accessible. It should be easy to read, easy to set, and easy to change the CONTENTS. The CONTENTS could be used to hold the text of a letter which was stored in a computing system which implemented Catoan.

In addition to a CONTENTS, Catoan-objects have a DIRECTORY. The DIRECTORY has two parts: a "named" part, and an "unnamed" part. In the named DIRECTORY of CRL, one would store references (hard links) to those

Catoan-objects considered to be sub-objects of CRL. This is usually a logical grouping, and can be thought of as placing a segment in a certain directory in a Multics or Unix file system. The sub-objects of CRL are Catoan-objects in their own rights; changing their relationship with CRL (that is, the exact sub-object to which a particular name refers) usually is not done.

In the unnamed part of CRL's DIRECTORY are references (hard links, but without local names attached to the reference) to Catoan-objects which are physical sub-objects of CRL. Those Catoan-objects referenced in the unnamed part of the DIRECTORY are part of the implementation of the particular Catoan-object, and are not usually of interest to the object's users. As with objects referenced in the named part of the DIRECTORY, the relationship between CRL and the sub-objects in the unnamed part of the DIRECTORY usually is not changed.

The objects in the DIRECTORY of a Catoan-object are considered less accessible than the CONTENTS. Once a reference to an object is added to the DIRECTORY, it cannot be replaced, but must be deleted and then added. This reflects the accessibility semantics of such an inclusion. If these semantics are not appropriate for a particular application, the CONTENTS could be used to implement a directory which is interpreted by some program. Because the CONTENTS of a Catoan-object can be an arbitrary

abstract object, the DIRECTORY portion of a Catoan-object can be ignored, and the CONTENTS used to implement a filing system which is more natural for the particular application.

All three of the forms of including data in an object might be used to represent a system composed of a collection of programs (1). The highest-level module in the system is a program, with the source stored in the CONTENTS, representing the view of the source as the abstract program. In the unnamed portion of the DIRECTORY would be the implementation of the program object, including such things as documentation and object-code. Named references to the programs comprising the system would be in the named portion of the DIRECTORY. Chapter Four, "An Example: A Syspal Program Object," describes the aspects of this example relating to programs in more detail.

III.A.2. Trust and Catoan.

What does it mean to "trust" a non-sentient entity? What does it mean to "trust" a filing system? What does it mean to "trust" Catoan?

"Trust" in general is very difficult to define, especially when applied to non-sentient entities. However, "trusting" a filing system is easier to

---

(1) I shall return to this example throughout the chapter.

define. In this report, to trust a filing system is to give the filing system access to data when it doesn't explicitly require such access to perform its duties. My perception of a filing system's duties does not include access to the CONTENTS (as defined in the previous section). Rather, a filing system is a manager for named, permanent objects -- not the CONTENTS of those objects.

A "trusted" module is a module which a) the user believes is secure, and will not access things except on explicit instructions from the user, and b) does not allow other users to access it, except as is appropriate for that user. Part (a) is primarily a belief on the part of the user; part (b) has some implications on the kind of information which the trusted module can supply to environments outside the module.

Specifically, a trusted module, in order to prevent other entities from accessing its protected data, cannot give out any references to any portion of the protected data's internal representation. Rather, it must give out an indirect reference, which the trusted module, and only the trusted module, can translate into the actual representation of the protected data.

Catoan, however, gives out a pointer to portions of the representation of a Catoan-object. The CONTENTS_READ operation (see Section III.B.1.b) returns a pointer to the CONTENTS of the Catoan-object. This allows

entities besides Catoan to access part of the representation of the Catoan-object.

Because a module which gives out portions of its data's representation does not have total control over the representation, it does not have total control over what can be done to the representation, and so is unable to ensure certain kinds of internal consistency. In the case of Catoan, for example, the information accessed by the "principals" and the "dates" sub-classes of operations may not be accurate. Furthermore, Catoan has no way of verifying the identities of those accessing the data in the CONTENTS of the Catoan-object, because they may be accessing the data without using Catoan.

This report examines some of the implications of not trusting the filing system. The filing system will have access to *its* objects (Catoan-objects), but not to the data in the CONTENTS of the Catoan-object. This is done partly out of lack of trust, and partly to allow more than one filing system to exist in the host computing system more easily.

III.B. *The Basic Object*.

An OBJECT ("Catoan-object") is the basic unit of data in Catoan. Catoan-objects conceptually have three parts: SYSTEM OVERHEAD INFORMATION,

a DIRECTORY, and a CONTENTS. The first is information the system keeps about each object, such as when it was created. The DIRECTORY and CONTENTS were described in Section III.A.1 above.

Figure 3 shows the operations and representation of a Catoan-object. Many points in the figure and the immediately ensuing discussion may be unclear. Subsequent sub-sections in this section will clarify the problems.

Most of the operations on an object are related to the "SYSTEM OVERHEAD INFORMATION" in the object. There are only eight operations dealing with the DIRECTORY, and only two with the CONTENTS. Yet, these two parts of an object are the most interesting. The SYSTEM OVERHEAD INFORMATION is very structured, and has a very limited scope; we know the form it will take long before the object is actually defined. The DIRECTORY, on the other hand, may change drastically during the existence of the object -- it may start off empty, have some objects added to it, have some objects deleted from it, and will have an unpredictable size. Similarly, the structure and size of the CONTENTS is unpredictable, and the structure might never be known to Catoan.

```
MODULE catoan_object(contents_type);

new:PROCEDURE
    RETURNS(o:@catoan_object)
    (* Make a new catoan_object. *);
delete:PROCEDURE
    (* Delete a catoan_object. *);

contents_set:PROCEDURE(c:@contents_type)
    (* Stow the contents of the object. *);
contents_read:PROCEDURE
    RETURNS(c:@contents_type)
    EXCEPTION(contents_doesnt_exist)
    (* Retrieve this object's contents. *);

directory_unnamed_add:PROCEDURE(o:@catoan_object, n:INTEGER)
    EXCEPTION(directory_full, directory_slot_occupied)
    (* This object now includes unnamed object number N. *);
directory_unnamed_delete:PROCEDURE(n:INTEGER)
    EXCEPTION(directory_doesnt_exist
              directory_doesnt_contain_object)
    (* Remove Nth entry from unnamed portion of DIRECTORY. *)
directory_unnamed_lookup:PROCEDURE(n:INTEGER)
    RETURNS(o:@catoan_object)
    EXCEPTION(directory_doesnt_exist
              directory_doesnt_contain_object)
    (* Return Nth entry from unnamed portion of DIRECTORY. *)
directory_named_add:PROCEDURE(n:object_name, o:@catoan_object)
    EXCEPTION(directory_full, directory_slot_occupied)
    (* This object now includes another named object. *);
directory_named_delete:PROCEDURE(n:object_name)
    EXCEPTION(directory_doesnt_exist,
              directory_doesnt_contain_object)
    (* This object no longer includes a certain object. *);
directory_named_contains:PROCEDURE(n:object_name)
    RETURNS(b:boolean)
    EXCEPTION(directory_doesnt_exist)
    (* Does this object contain object 'n'? *);
directory_named_lookup:PROCEDURE(n:object_name)
    RETURNS(o:@catoan_object)
    EXCEPTION(directory_doesnt_exist,
              directory_doesnt_contain_object)
    (* Translates a contained object-name into an object reference. *)
directory_named_read:PROCEDURE
    RETURNS(n:ARRAY(*) of object_name)
```

```
    EXCEPTION(directory_doesnt_exist)
    (* Which objects does this one contain? *);

owner_read:PROCEDURE
    RETURNS(p:principal_id)
    (* Who owns this module? [obtained from mem mgr] *);
creator_set:PROCEDURE(p:principal_id)
    (* Indicate that principal 'p' is object's creator. *);
creator_read:PROCEDURE
    RETURNS(p:principal_id)
    (* Who created this object? *);
last_modifier_set:PROCEDURE(p:principal_id)
    (* State who last modified this object. *);
last_modifier_read:PROCEDURE
    RETURNS(p:principal_id)
    (* Who last modified this object? *);

date_created_set:PROCEDURE(d:date)
    EXCEPTION(date_invalid)
    (* Indicate when the object was made. *);
date_created_read:PROCEDURE
    RETURNS(d:date)
    (* When was this object created? *);
date_last_modified_set:PROCEDURE(d:date)
    EXCEPTION(date_invalid)
    (* Indicate when this object was last modified. *);
date_last_modified_read:PROCEDURE
    RETURNS(d:date)
    (* When was this object last modified? *);
date_last_accessed_set:PROCEDURE(d:date)
    EXCEPTION(date_invalid)
    (* Indicate when this object was last accessed. *);
date_last_accessed_read:PROCEDURE
    RETURNS(d:date)
    (* When was this object last accessed? *);

size_read:PROCEDURE
    RETURNS(s:integer)
    (* How big is this object?
      [overhead+mem_mgr.size(CONTENTS)+mem_mgr.size(DIRECTORY)] *);
```

```
object_name = string(20);
principal_id = string(20);

date = RECORD
    year: 1975 TO 3975;  !assumption: system will last <2000 years
    month: 1 TO 12;
    day: 1 TO 31,
    hour: 0 TO 23,
    minute: 0 TO 59,
    second: 0 TO 59.9999 PRECISION 4
    END;  !date

SELF = RECORD      !Representation of an object.
    contents = @contents_type;
    (date_created,
     date_last_modified,
     date_last_accessed) = date;
    (creator,
     last_modifier,
     owner) = principal_id;
    directory = RECORD
         named: ARRAY(*) OF RECORD
              n: object_name;
              o: @catoan_object;
              END;  !named
         unnamed: ARRAY(*) OF @catoan_object;
         END;  !directory
    END;  !SELF

END MODULE;  !catoan_object
```

Figure 3: The Basic Catoan-Object.

Assume that the CONTENTS of a Catoan-object holding a Syspal program is to be of type "text." Figure 4 shows how CRL would be declared, and how one would store and retrieve its CONTENTS.

```
text = TYPE . . . ;
edit_buffer: text;
crl: catoan_object;
        .
        .
        .
crl := NEW catoan_object(text);
        .
        .
        .
edit_buffer := contents_read(crl);
        .
        .
        .
contents_set(crl, edit_buffer);
```

Figure 4: Catoan-Object with CONTENTS of Type "text."

III.B.1. The Operations of the Basic Object.

The operations on an object can be classified according to the information they reference. The classes of operations are overhead: instance, principals, dates, miscellaneous; contents; and directory. Each class will be considered below.

III.B.1.a. Overhead-Class Operations: Instance.

The "instance" operations are NEW and DELETE. These operations are invoked whenever a Catoan-object is created or deleted. NEW sets up the initial contents of the overhead information, and initializes the DIRECTORY and CONTENTS to be empty (NULL). DELETE passes a message to each of the

Catoan-objects referenced in the DIRECTORY and to the object referenced in the CONTENTS indicating that they are no longer referenced by CRL, and deletes CRL.

III.B.1.b. CONTENTS-Class Operations.

The "contents" operations are CONTENTS_SET and CONTENTS_READ. They deposit data into, and extract data from, a Catoan-object's CONTENTS. The argument to _SET (1) and the return value from _READ are pointers to the type of the CONTENTS, as specified by the module parameter ("contents_type" in Figure 3) when the Catoan-object was instantiated by NEW. (For example, if CRL is defined as in Figure 4, _SET takes and _READ returns something of type @TEXT.)

The effects of _SET and _READ are to translate between Catoan-object references and Syspal-object references. Notice that both operations work with pointers, and not directly with the data. The _READ operation is analogous to the OPEN operation in a classical file system; the _SET operation is analogous to CLOSE.

---

(1) This is a shorthand notation for "CONTENTS_SET." When there will be no confusion as to the meaning and context, the prefix (portion of the name before the "_") will be omitted. A similar convention will be used for eliding suffixes.

The _SET operation must ensure that the Catoan-object and its components are safely stored in non-volatile storage. Hopefully, part of the interface of the memory manager is an operation like MAKE_NON_VOLATILE, which, if all memory is non-volatile, and there is no buffering in volatile memory (by the memory manager), may be a null operation. Similarly, _READ might "stage" some part of the contents by calling the memory manager's PRIME_BUFFER operation.

III.B.1.c. DIRECTORY-Class Operations.

The DIRECTORY of CRL specifies those Catoan-objects which are sub-objects of CRL. There are two parts to the DIRECTORY of a Catoan-object: the named part, and the unnamed part. The two parts represent different logical relationships between CRL and its sub-objects. The DIRECTORY is described in Section iii.A.1.

The unnamed portion of the DIRECTORY represents those Catoan-objects which are internal sub-objects of CRL. Generally, these are part of the implementation of the abstraction which uses CRL, and are of no concern to CRL's users. An example of using the unnamed portion of the DIRECTORY is shown in Chapter Four, "An Example: A Syspal Program Object," where it is used for (among other things) the object-code of a program.

The named portion of the DIRECTORY represents Catoan-objects which the user feels are logically parts of CRL. He might, for example, build CRL from several component objects, thereby forming one Catoan-object from several sub-Catoan-objects.

The operations on the unnamed portion of the DIRECTORY are DIRECTORY_UNNAMED_ADD, _DELETE, and _LOOKUP. _ADD inserts a sub-object in the nth DIRECTORY slot; _DELETE removes a specified unnamed entry from the DIRECTORY. _LOOKUP returns the object referenced by the Nth entry in the unnamed portion of the DIRECTORY.

The operations on the named portion of the DIRECTORY are DIRECTORY_NAMED_ADD, _DELETE, _CONTAINS, _LOOKUP, and _READ. _ADD associates a name and an object reference in CRL's DIRECTORY; _DELETE removes such an association. _CONTAINS is a predicate which indicates whether the supplied name is in the DIRECTORY; _LOOKUP translates a name to an object reference. _READ returns a matrix containing all the names in the DIRECTORY, and is supplied so that a DIRECTORY can be searched.

III.B.1.d. Overhead-Class Operations: Principals.

The "principal" operations obtain and manipulate the principal-identifiers stored in the overhead portion of a Catoan-object. The identity of the Catoan-object's owner (the principal paying for the

storage), creator, and last modifier are accessed through the operations OWNER_READ, CREATOR_SET and _READ, and LAST_MODIFIER_SET and _READ (1). The creator and last-modifier can be changed; the owner is obtained from the memory management system.

III.B.1.e. Overhead-Class Operations: Dates.

The "date" operations provide access to the time and date when various operations last occurred for the Catoan-object. Available are times and dates for the Catoan-object's creation, last modification, and last access. These operations are DATE_CREATED_SET and _READ, DATE_LAST_MODIFIED_SET and _READ, and DATE_LAST_ACCESSED_SET and _READ. The dates automatically maintained by Catoan are for creating, modifying, and accessing the Catoan-object, not the CONTENTS of the Catoan-object. This is related to the trust issue discussed in Section III.A.2.

III.B.1.f. Overhead-Class Operations: Miscellaneous.

The "miscellaneous" operations provide information about the physical size of the Catoan-object. SIZE_READ obtains the sizes of the CONTENTS, DIRECTORY, and overhead from the memory manager, and returns their sum.

---

(1) The _SET operations are generally not explicitly used, and exist primarily for completeness.

III.B.2. Comments on the "_SET" Operations.

The inclusion of some of the _SET operations (1) may be puzzling. For example, why is there a DATE_MODIFIED_SET operation? Won't Catoan take care of such things?

Recall that Catoan is part of the optional extensions to the kernel operating system. Furthermore, Catoan is not necessarily trusted, and it is possible to access portions of Catoan-objects (specifically, the data in the CONTENTS) without using Catoan. A user who directly accesses the data in the CONTENTS (for example) might want to update the SYSTEM OVERHEAD INFORMATION in a containing Catoan-object so that it accurately reflects what has happened.

It is possible that a failure of the host computing system's hardware, the operating system kernel, or Catoan may introduce errors into Catoan-objects. These errors may require human intervention. Even in a trusted filing system like that on Multics, the ability for people to access some of the "overhead" fields is considered necessary. In a non-trusted filing system, such abilities are mandatory so that "expected errors" (2) can be corrected.

----

(1) Specifically, the CREATOR_, LAST_MODIFIER_, DATE_CREATED_, DATE_LAST_MODIFIED_, and DATE_LAST_ACCESSED_ _SET operations.

(2) One of the reasons a system might not be trusted by its users is that

III.B.3. Naming and the DIRECTORY.

Each Catoan-object contains a DIRECTORY part. This DIRECTORY specifies all those objects which are sub-objects of, for example, CRL; the contained objects need not have names associated with them, in which case they are referenced numerically. See Section III.A.1 for a discussion and example of DIRECTORY use.

If one wanted to implement a Multics-like directory, the CONTENTS of the object would be NULL; for a Multics-like segment, the DIRECTORY would be empty. But, one can have a non-empty DIRECTORY and a non-empty CONTENTS at the same time, thereby allowing objects to "contain" other objects.

Multics has the concept of a "soft link," between a local name and a global name. No such concept exists in Catoan. Rather, because an object can be in the DIRECTORies of many objects, the same object can be referenced directly by many local names. This is often referred to as a "hard link," and is similar to the Unix link.

One of the implications of the unrestricted DIRECTORY inclusion is that, rather than implementing a naming hierarchy, Catoan realizes a naming

----

the users expect the system to make mistakes (that they can, perhaps, correct).

network. Just as object A can contain more than one object, so can more than one object contain object A. Furthermore, loops can be created in the network, by A containing B which contains A.

An advantage of this arbitrary network structure is that it can more readily reflect the structure of some objects. Recursive objects and objects which include other objects exist in the world; it would be nice if one could model them in a computing system. Such object inclusion also aids in modularity. For example, if one were implementing a network-model database, one could define the network parent-child relationships using the DIRECTORY of each object to contain the children.

Allowing a general network in the naming structure presents a problem only when the entire naming network must be walked. If it is deemed important to be able to walk the network, VISITED flags must be included in each Catoan-object, which must be reset upon completion of the network traversal. If such flags ARE included, it may be necessary to reset them all upon system restart, to guard against a failure during a walking of the network, and subsequent traversals encountering a non-existent loop because a VISITED flag stayed set from a previously aborted walk. Various problems besides system failure exist when the network must be walked; for example, what if a walk aborts for a reason other than system failure? I shall not discuss such problems here, but, rather, refer the interested reader to the

literature (garbage collection algorithms often solve this problem; see, for example, [3, 4, 30, 33]).

As long as the network does not have to be walked, loops and self-containment do not present a problem. The only other traversal of the naming network is for resolving a name, which is directed by the name to be resolved. If a name hits a loop, intentionally or unintentionally, the results may be unexpected, but the system will not incur any great problem (like an infinite loop), because the name must, by its physical properties, have a finite length. If there are soft links, however, name resolution may enter an infinite loop if a cycle of links is encountered (1).

III.B.4. Storing Data: The CONTENTS.

The purpose of Catoan, and of any filing system, is to allow the users of a computing system to retain data for long periods of time. For this purpose, Catoan objects have a component called the "CONTENTS." It is in the CONTENTS that the actual data are stored.

Most filing systems are "record" oriented: one retrieves ("reads") and deposits ("writes") bit- or byte-strings, or some collection of bits or

----

(1) Multics has this problem; its solution is to abort link resolution after encountering some number of consecutive links.

bytes ("records"). The structure of the data is very visible to the file system, and to the user of the file system. Furthermore, the user MUST know the structure of the data -- not necessarily how it is stored physically, but usually at least how it is stored logically ("logical records").

The CONTENTS of a Catoan object is of arbitrary type; Catoan has no explicit knowledge of the structure of the data in the CONTENTS. Therefore, the CONTENTS must be handled in its entirety through a pointer, rather than piecemeal (as in many other filing systems). Because Catoan works with abstract data, users of Catoan can view the contents abstractly, and can deposit and retrieve arbitrary data structures. There is no explicit notion of records in Catoan.

Because a pointer to the data in the CONTENTS is returned, rather than a copy of the data, sharing of the data in Catoan-objects is provided. If one wanted to implement an airline reservation system with many agents accessing a shared database, the database could be stored as a Catoan-object, retrieved from Catoan, and then manipulated by the operations defined on the database. If a text editor were implemented where it was desired to operate on a copy of the original data, a new object containing a copy of the data in the CONTENTS would be created,

operations performed on the copy, and then, perhaps, the copy stored in place of the old CONTENTS.

III.B.5. Protection and Security.

An interesting consequence of the way Catoan stores data is that Catoan need not be trusted with the data. True, it could maliciously delete an object, but it cannot leak parts of the contents of the object to other users. All Catoan could leak would be the entire object. If one wanted to store one's data securely, so that no one else could read it, one could store it as the CONTENTS of a Catoan object, and simply not give anyone an interface to the module that implements the data in the CONTENTS. All Catoan can do is to leak the entire CONTENTS of the object; if the interface is not also possessed, the CONTENTS does no one any good.

It might be undesirable to let even the CONTENTS of the object reach "unfriendly" hands. For example, it might be necessary for someone to have an interface to the module which implements the object stored in the CONTENTS, and yet he should be restricted from using the CONTENTS of a particular Catoan-object. Such protection can be provided through various schemes, ranging from passwords, to access control lists, to capabilities. Passwords can be included easily in Catoan, by adding PASSWORD_SET and

PASSWORD_VERIFY operations to the module, for example. This, however, require trusting Catoan to properly implement password protection.

Similarly, Catoan could implement access control lists, and could verify the right of some principal to perform certain operations on a given object. This, again, requires trusting Catoan to properly enforce the protection.

If one does not want to have to trust the object manager with his data, what can be done? Capabilities [28] offer a solution. If someone does not have the capability of something, that thing cannot be accessed, because it cannot be named. This level of protection must be enforced by the system's memory manager.

In a capability-based system, implementing a directory-walking mechanism for name resolution, where all resolution begins from a "root" as in a Multics-like file system, allows all users access to all objects. To resolve a name, start at the root (to which all have access); find the name in the directory, and use the capability there found to proceed to the next node, where the process is repeated. Since names are "translated" directly to capabilities, and since capabilities are the mechanism on which protection is based, naming and protection become equated. Since naming is universal in a Multics-like system, there is no protection.

What is needed is a restriction on the initial entry into the naming network. Providing a single node (object) from which all other nodes (objects) can be reached is the problem in the Multics-like name resolution in a capability system. Each user must be able to name, to find in some accessible directory, only those objects which should be accessible to him. This requires a per-user directory of objects initially accessible upon entry to the system, and then careful control of which capabilities are given to which users, and to which additional objects, besides the one directly referenced, access is granted (that is, which objects are contained in the DIRECTORY of the directly referenced object). This restriction is a general property of capability-based protection systems. A more detailed description of the issues underlying this discussion is in [28].

Part of Catoan's job is to produce internal names (like capabilities) from external names (like character strings). This is the job of the directory manipulation operations of an object. The DIRECTORY_LOOKUP operation "translates" a character string into an object, thereby generating an internal name, or capability. The solution here comes from a refinement to the basic capability mechanism, and requires introducing "locked" capabilities.

A locked capability has, in addition to the reference to an object, a "lock" associated with it. A locked capability is implemented by a trusted module, such as described in Section III.A.2. In order to access the capability protected by the lock, and the data protected by the locked capability, an accessor _must_ go through the proper type manager, which can verify the accessor's identify and rights in whatever way its implementor pleases. In other words, in order to use the locked capability, a "key" fitting the lock must be presented.

Capability locks and keys, like capabilities themselves, must be unforgeable (locked capabilities must also be unforgeable). Thus, if one wants to place an object in a somewhat publicly available directory (as may be required, because all directories may be "somewhat publicly available"), and yet retain control over who can access the object, a locked capability, rather than an ordinary capability, is placed in the directory. The key for the locked capability is then distributed in a secure manner to those who are allowed access to the object.

III.C. A Refined Object.

The basic object, described above, is rather spartan. Often, a more "civilized" object is desired which supplies features convenient for human use. For example, one might want to provide a "classical file" object,

supporting record-at-a-time access. Perhaps more security, automatic object cross-referencing, locking (or some other form of "sequencing"), or version control might be desired. This section describes a more refined, "civilized" object than that described above.

III.C.1. Protection and Security.

An important refinement to the basic Catoan-object is the addition of further protection features. Given the directory lookup mechanism, a capability-based protection system may provide little security, as previously noted (Section III.B.5). A solution to this problem is to provide an access control list scheme as a feature of a refined object. The access control list would be a matching of principal identifiers with a specification describing the types of access allowed the principal. This is the scheme Multics uses, and is described in [28].

An alternative to the complete access control list is a Unix [26] or MPE/3000 [13] protection scheme, which allows all members of particular groups the same access. For example, all members of a particular project, or of a particular sub-project, might be given access to the object. In Multics, this would be represented as "*.Syspal," where "Syspal" was the name of the project.

Further protection refinements can also be implemented. A security-clearance concept (confidential, secret, top secret) is a possibility, where each process would have an unforgeable indication of its current clearance; passwords could be provided, requiring that the correct password be supplied when the object is accessed; arbitrary protection schemes, requiring access only between certain times, or on certain days, or after a program sufficiently verifies the identity of the user, might be desired. By making the various "protected objects" each a different type, with different object managers, and allowing access only through the correct manager, access to the objects can be restricted as desired.

A point of note is: what is being protected by the access control list of the refined Catoan-object? Catoan is not necessarily trusted; furthermore, it is possible to access the data in the CONTENTS of a Catoan-object without the intervention of Catoan. Therefore, the access control list cannot protect the data in the CONTENTS in the general case. Rather, the access control list protects the Catoan-object, since that is the only thing for which access requires using Catoan.

How, then, might the data in the CONTENTS of a Catoan-object be protected? Locked capabilities, described above, offer one solution. Another solution is to control the distribution of the data's addressability. In a capability-based protection system, this implies not

distributing the capability for the data to other users, but instead requiring them to use Catoan to access the data. This requires the user to trust Catoan to enforce the access control list.

Figure 5 shows the operations and representation of an access control list scheme. The access control list is implemented as an array, matching principal identifiers with access rights. The access rights are specified by bits indicating DIRECTORY read, DIRECTORY search, DIRECTORY modify, DIRECTORY append, CONTENTS read, CONTENTS set, access control list read, access control list modify, and access control list append. Each operation protected by the access control list must verify that the principal requesting the operation is authorized to perform the operation on the object; if not, UNAUTHORIZED_ACCESS is signaled.

The ACL_ADD_PRINCIPAL operation gives a new principal access to a Catoan-object. The arguments are the identifier of the principal and the access specification. If the specified principal is already in the access control list, an exception is signaled. _ADD_ACCESS adds a the specified access rights to a principal in the Catoan-object's access control list.

_DELETE_PRINCIPAL rescinds a principal's right to access the Catoan-object. Similarly, _DELETE_ACCESS removes a particular access right of a principal in the Catoan-object's access control list.

```
acl_add_principal:PROCEDURE(new_acl:acl, prin:principal_id)
    EXCEPTION(unauthorized_access, acl_principal_already_in_acl)
    (* Inserts new principal in the access control list. *);
acl_delete_principal:PROCEDURE(prin:principal_id)
    EXCEPTION(unauthorized_access, acl_principal_not_in_acl)
    (* Removes a principal from the access control list. *);
acl_add_access:PROCEDURE(add_acl:acl, prin:principal_id)
    RETURNS(old_acl:acl)
    EXCEPTION(unauthorized_access, acl_principal_not_in_acl)
    (* Ensures PRIN has specified permission. *);
acl_delete_access:PROCEDURE(del_acl:acl, prin:principal_id)
    RETURNS(old_acl:acl)
    EXCEPTION(unauthorized_access, acl_principal_not_in_acl)
    (* Ensures PRIN does not have specified permission. *);
acl_read:PROCEDURE
    RETURNS(acl:access_control_list_rep)
    EXCEPTION(unauthorized_access)
    (* Formats the access control list for external perusal. *);
acl_set:PROCEDURE(new_acl:access_control_list_rep)
    RETURNS(old_acl:access_control_list_rep)
    EXCEPTION(unauthorized_access)
    (* Allows bulk setting of the access control list. *);

access_control_list_rep = ARRAY(*) OF RECORD
    prin:principal_id;
    the_acl:acl;
    END;  !access_control_list_rep
acl = RECORD
    dir_acl: ARRAY(4) OF BOOL (*Read, Search, Modify, Append.*);
    cont_acl:  ARRAY(2) OF BOOL (*Read, Set.*);
    acl_acl: ARRAY(3) OF BOOL (*Read, Modify, Append.*);
    END;  !acl
principal_id = string(20);
```

Figure 5: An Access Control List Scheme for Catoan.

_READ returns the entire access control list so that it can be examined externally. This operation might be used to obtain an access control list for use in setting some other Catoan-object's access control list, using

the _SET operation. _SET's argument is an entire access control list, like the value returned by _READ.

The representation of an access control list consists of a sequence of two-component RECORDs. Each RECORD consists of a PRINCIPAL_ID and an ACL. The ACL is a three-component RECORD: the DIR_ACL, the CONT_ACL, and the ACL_ACL. Each component is an ARRAY OF BOOL, with the several bits corresponding to the various modes of permission which can be granted. Each permission type is independent of all the others.

III.C.2. Cross-Referencing.

One often wants to determine which Catoan-objects reference CRL, and which Catoan-objects CRL references. This requires two collections of data: those objects referenced by CRL, and those objects which reference CRL. The first set is the DIRECTORY of CRL, and so is readily available. The second set, however, is not so readily available -- it must be explicitly collected.

How might such a cross-reference be implemented? Suppose each object had a structure and operations like those of Figure 6 as part of its definition. Then, upon adding a reference to an object's DIRECTORY, a call to the contained object's XREF_ADD_REF operation would be included in the

implementation of DIRECTORY_NAMED_ADD and DIRECTORY_UNNAMED_ADD. (Similar definitions and calls are required for XREF_DELETE_REF.)

There is a problem with the above method for storing cross-reference information: who pays for the storage? A straight-forward implementation of a versioned-object would have the object's owner paying for the storage of cross-reference information. This penalizes owners of very popular objects, for the object's owner may have little control over the number of referencing objects.

One solution is to ignore the problem; that is, to let the object's owner pay for the object's cross-reference information. Another possibility is for the accounting system to keep track of the number of cross-references to each object, and to deduct the charges for the cross-reference information from the object owner's bill. This would effectively make cross-reference information part of the system's overhead, and so all users would pay a share of the cross-reference storage costs.

```
xref_add_ref:PROCEDURE(name:object_name, obj:@object)
    EXCEPTION(xref_full);
    (* NAME is the name of the referencing object.
      OBJ is a reference to the referencing object. *)

    array_ref_out_of_bounds: EXCEPTION;

    EXCEPTION
        ON array_ref_out_of_bounds DO
          RETURN(xref_full);
        BEGIN
            SELF.xref.refd_bys[SELF.xref.next].obj := obj;
            END;
        END;
        SELF.xref.refd_bys[SELF.xref.next].name := name;

        SELF.xref.next :=# +1;
        END PROCEDURE;  !xref

xref: RECORD
    next: INTEGER;
    refd_bys: ARRAY(*) OF RECORD
        name: object_name;
        obj: @object;
        END;  !refd_bys
    END;  !xref
```

Figure 6: Additions to the Basic Object for Cross-Referencing.

Two problems exist with the system overhead solution. The first is that it is inequitable: if a system has two users, with the first referencing five objects not owned by him and the second referencing one such object, both users would probably pay for three references, thereby overcharging the second user. The second problem is: what prevents someone from informing the system of far more references than actually exist to his objects and (illegally) lowering his storage bills?

Assume that the following fragment is part of the XREF_ADD_REF operation:

CALL acctg_storage_add_ref(*arguments*)

The ACCTG_STORAGE_ADD_REF operation tells the accounting manager that a cross-reference entry has been added to a particular object, and that the object's owner should not be charged for the storage occupied by the entry. This operation must be carefully protected; the only entities which are allowed to call ACCTG_STORAGE_ADD_REF must be trusted by the accounting manager not to call it excessively (that is, more times than are appropriate for the number of references), because otherwise someone could obtain free storage.

III.D. A Versioned Object.

Another refinement to the basic object is the "versioned" object. Rather than directly modifying an object when changing it, a new instance of the object is created, which is somehow related to a previous instance. Therefore, rather than an object appearing mutable, it is a "history" of immutable versions. This provides access to instances of the object besides the most recent one, and facilitates, for example, concurrent support and development of software.

III.D.1. Version Naming.

For each object, a hierarchy of versions exists, which is reflected in each version's name. The hierarchical relationship is that of "logical derivation": if Version B is the child of Version A, then B was "logically derived from" A. For example, B might be a refinement of A, correcting an implementation error if the object were a program. Alternatively, B might become a sibling-version to A, which could imply that A and B were similar sorts of refinements (improvements, modifications) of their mutual parent. Whether a version is a child or a sibling is the decision of the version's author.

A version name consists of a sequence of qualifiers to the object name. These qualifiers are suffixes to the object name or to a "qualified" object name (an object name with a version name suffix). Each qualifier is a number, specifying the version number from the appropriate level in the version hierarchy which is desired. The name "CRL.3.62" is a qualified object name, whose object name is "CRL," and whose version name is "3.62."

Figure 7 shows an sample version hierarchy. The versioned Catoan-object is named "CRL." CRL has three "top-level" versions; that is, three versions which are, in some sense, major modifications of CRL. In system installation terms, this level in the tree might correspond to a

"release," with lower levels being called "level" and "fix." To obtain CRL release two, one would use the name "CRL.2"; to obtain CRL release three, level one, "CRL.3.1" would be used.

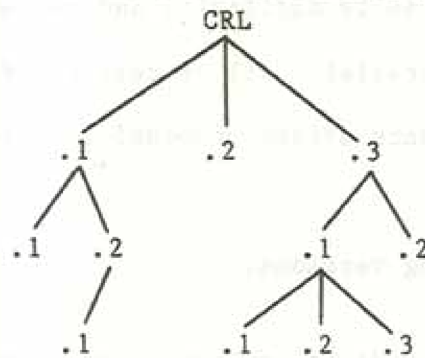

Figure 7: Version Naming Hierarchy.

Examining the CRL.3 subtree, there are two children of CRL.3: CRL.3.2 has no children; CRL.3.1 has three children. In system installation terms, one might reference CRL release three level one fix two as "CRL.3.1.2."

There are no restrictions on the semantics attached to the various levels in the hierarchy. For example, rather than "system installation," version management could be used in a class on software engineering. Suppose an exercise in modifying existing programs is to be given. The students might be broken into groups, with each group developing a

solution. The initial program is CRL; each group is to create its solution as CRL.n. While working on the assignment, various trial solutions might be attempted, with modifications being made in an attempt to produce a better solution. Perhaps one group has one small part of the problem remaining which is especially difficult, and so two of the group members attempt a solution in parallel. All of this could be handled very easily with the version maintenance system proposed in this chapter.

III.D.2. Storing and Implementing Versions.

Storing versions is a problem distinct from naming, though they are often coupled, especially if versions are stored as incremental changes to other versions, as in the Source Code Control System (SCCS) [5, 11, 27] available with the Programmer's WorkBench under Unix [8, 18]. SCCS stores a set of versions as a collection of updates run against the parent version. A version is created from some particular existing version, is named relative to that version, and is generated from that version. (The version generation process is recursive if necessary.)

By de-coupling version naming from version generation, additional flexibility is obtained, without sacrificing the potential benefits of coupling naming and generation (coupling can be done by the user if desired). Furthermore, the proposed mechanisms allow version generation to

be done in any manner desired, allowing the user to specify space-time trade-offs, derivation relationships, policies for creating new versions (as opposed to including the changes in an existing version for efficiency), et ceterae.

The additional information contained in a Catoan versioned object to provide version maintenance and the operations on such objects are shown in Figure 8.

A versioned Catoan-object consists of four types of information: information describing how to generate the version (VERSION_GEN_INFO), the logical children of the node in the version naming hierarchy (CHILDREN), the logical parent of the node (PARENT), and whether some other version is physically derived from this version.

```
version_new:PROCEDURE(v_name: version_name,
                      v_base: @versioned_catoan_object,
                      update_info: updates_specification,
                      v_gener: version_generating_program)
    RETURNS(new_version: @versioned_catoan_object)
    EXCEPTION(version_exists)
    (* Creates a new version;  SELF = parent. *);
version_delete:PROCEDURE(v_name:version_name)
    EXCEPTION(version_nonexistent)
    (* Remove a version from the history;  SELF = parent. *);

version_get:PROCEDURE(v_name:version_name)
    RETURNS(v_obj:@versioned_catoan_object)
    EXCEPT(version_nonexistent)
    (* Translate a name into a versioned object;  SELF = parent. *)
version_read:PROCEDURE
    RETURNS(o:@catoan_object)
    (* Translates a version into an object;  SELF = the version. *)

version_replace:PROCEDURE(v_name: version_name,
                          v_base: @versioned_catoan_object,
                          update_info:updates_specification,
                          v_gener:version_generating_program)
    EXCEPTION(version_nonexistent, version_not_replaceable)
    (* Replace a (leaf) version with a new one. *);

additional_versioning_information(updates_specification) = TYPE RECORD
    version_gen_info: RECORD
         base_version: @versioned_catoan_object;
         updates: updates_specification;
         version_gen_pgm: @version_generating_procedure;
         END;  !version_gen_info
    children: ARRAY(*) OF RECORD
         name: version_name;
         version: @versioned_catoan_object;
         END;  !children
    parent: RECORD
         name: version_name;
         version: @versioned_catoan_object;
         END;  !parent
    used_as_base: BOOL;
    END;  !versioned_catoan_object
```

```
version_name(size: 0 TO 100) = TYPE RECORD
    length: 0 TO size;
    chars: ARRAY(1 TO size) OF
        UNION("0" TO "9", ".");
    END;  !version_name;
```

Figure 8: Additional Information and Operations for Version Maintenance.

The VERSION_GEN_INFO contains three pieces of information. The BASE_VERSION denotes the version from which the current version is physically derived. To generate the current version, as is done by VERSION_READ, start with the BASE_VERSION and apply the UPDATES. The UPDATES specify the transformation under which the base version must go to obtain the current version. The UPDATES are applied by the VERSION_GENERATING_PROCEDURE, in which the semantics of the UPDATES are embodied. The minimal definition of VERSION_GENERATING_PROCEDUREs is shown in Figure 9.

The definition of the UPDATES_SPECIFICATION (the parameter to the VERSIONED_CATOAN_OBJECT type) is up to the user, as is that of the VERSION_GENERATING_PROCEDURE. The only requirements of either of these is that the VERSION_GENERATING_PROCEDURE meets the proper interface, and the VERSION_GENERATING_PROCEDURE and UPDATES_SPECIFICATION are compatible.

```
version_generating_procedure(updates_specification, contents_type) =
    TYPE
    PROCEDURE(base: version_name, updates: updates_specification)
    RETURNS(contents_type)
    EXCEPTION(versioned_object_nonexistent_base,
              versioned_object_inconsistent_updates)
    (* UPDATES_SPECIFICATION is a type definition describing the
     form of the updates.
       CONTENTS_TYPE describes the form of the CONTENTS of the
     version.
       BASE is the version from which this version is physically
     derived.
       UPDATES is the updates to be run against the base. *);
```

Figure 9: Definition of VERSION_GENERATING_PROCEDUREs.

VERSION_REPLACE allows certain versions to be mutable, rather than immutable, so that changes to certain versions need not create a new version (though one could be made, if desired). Any version with a child becomes immutable, and any version which is the BASE_VERSION of some other version also becomes immutable. However, if a version is a leaf in the naming structure, and no other versions depend upon it, it can be changed. This is an efficiency refinement, and allows small changes to be readily incorporated.

The VERSION_DELETE operation is not totally straightforward; it cannot merely remove the version. Some other version may be using the to-be-deleted version as its BASE_VERSION. If deleting a version will remove a BASE_VERSION, either the version cannot be deleted, or the

information in it must be included in those versions which depend on the version to be deleted. This may require a cross-referencing mechanism, similar to that presented in Section III.C.2.

The CHILDREN field specifies those versions which are immediate children of the current version. The CHILDREN fields of all the versions of a versioned object specify the logical relationships among the various versions, as described above. Because the CHILDREN information and the VERSION_GEN_INFO information are separate, the logical derivation of a version need not be related to the physical derivation of the version.

The CHILDREN field attaches names to the logically derived children of the current version. The name of the child, together with the names of all the eventual parents of the child, specify the position of the child in the version hierarchy. See Section III.D.1 for a discussion of version naming.

The PARENT information indicates the version which is the logical parent of this version. It allows tracing back up the version hierarchy when necessary.

To demonstrate how a VERSION_GENERATING_PROCEDURE and an UPDATES_SPECIFICATION might be defined, consider an example: maintaining versions of a program. The version history is that of Figure 7.

Logically, the UPDATES_SPECIFICATION could be a collection of "commands," specifying operations like "delete" or "insert" on a particular line of the document. (This is similar to the record-oriented update programs which exist in some batch-oriented computing systems for including updates in the source for a program.)

The VERSION_GENERATING_PROCEDURE would take the BASE_VERSION and "run the UPDATES against" the base. The result of this process is the text of the version represented by the BASE_VERSION and the UPDATES. Each pair of <UPDATES, BASE_VERSION> could represent a different logical version of the document (1), depending on how the VERSION_GENERATING_PROCEDURE interpreted the UPDATES relative to the BASE_VERSION.

How does one create the initial version of such a program? First, a VERSIONED_CATOAN_OBJECT, CRL, is created. The VERSION_GEN_PGM is specified to be the "Syspal_version_editor," which would apply the change directives properly. The BASE_VERSION is specified as NULL, indicating that there is no version on which this one is based. Then, the UPDATES which will create the initial version of CRL from "nothing" are supplied. CRL's initial version is now complete.

---

(1) In general, only a small subset of the <UPDATES, BASE_VERSION>s actually represent meaningful versions.

As an example, suppose that CRL.3.1.3 is to be created under CRL.3.1 (that is, CRL.3.1 is to be CRL.3.1.3's parent). For whatever reason, CRL.3.1.3 will be derived from CRL.3.2. What follows is a description of generating CRL.3.1.3 at the lowest level.

Call the version to be created NEW_VER, and let ADAM denote the most ancient ancestor in the version tree (in this case, CRL). First, the version on which NEW_VER is based must be obtained. The statement

```
base := version_get(adam, base_name);
```

finds the version denoted by BASE_NAME (which would have the value ".3.2") and assigns it to BASE. The program of CRL.3.2 would be obtained by

```
original_pgm := version_read(base);
```

This program would be provided as input to an editor, the output of which would be the new version of the program's source, which would be assigned to NEW_PGM. The incremental differences between ORIGINAL_PGM and NEW_PGM could be determined by

```
differences := Syspal_differences(original_pgm, new_pgm);
```

and everything is almost ready to complete the process. The parent of NEW_VER must be obtained:

```
parent := version_get(adam, parent_name);
```

assuming ".3.1" is the value of PARENT_NAME. Now, NEW_VER can be included in the version hierarchy of CRL, using the statement

```
new_ver := version_new(parent, new_name, base,
                       differences, Syspal_version_editor);
```

where NEW_NAME has ".2" as its value. This completes the creation of CRL.3.1.3.

To obtain the program as of a particular version, the version's name is supplied to VERSION_GET, which finds the version in the version naming hierarchy. VERSION_READ is then invoked, which passes the version's base and updates to the version generator (VERSION_GEN_PGM), which returns the version.

At some point, after the version history becomes very large, generating a given version may take a very long time. What could then be done is to create a version which is complete (similar to the initial version). Thereafter, future new versions could be generated off this new "complete" version, rather than having to incrementally generate all the previous versions before generating the desired one.

III.D.3. More on Version Naming.

In addition to the regular version names, one might want to have "sliding" names for versions. For example, when developing a program, one often has a backup, a current, and a test version of the program. Upon determining that the test version is ready for installation, one would want

to change the meanings of the names "backup," "current," and "test" to reflect the new state. This can be accomplished, and the general problem of "sliding" names can be solved, by introducing "variables" to reference versions.

A simple method of specifying variables for version references is to include an optional user-defined procedure for variable assignment which would be called whenever a new version is created. This procedure, or another one, could also be called directly by the user when he wanted to update the variable assignments. The variables' names and the objects they referenced could be stored in the named DIRECTORY in the highest-level Catoan-object.

It may be desirable to allow a general network of version names, rather than just a hierarchy. Catoan supports a general network for naming objects; version naming may require similar capabilities. At this point, the value of a version network has not been proven. Despite always referring to a hierarchy of naming versions, though, Catoan will support a network of versions using the definition presented in Figure 8 above. Any restrictions to a hierarchy would have to be done in the VERSION_NEW operation.

The operations presented here are very low level. Presumably, a higher-level interface to version maintenance would be presented to the user by, for example, the editor.

III.E. Summary.

Definitions of the "Basic Catoan-Object," a "refined" object, and a "versioned" object have been presented in this chapter. The operations of the objects, and sample representations, have been described. Issues of naming, protection, and (in some cases) efficiency were mentioned.

CHAPTER FOUR

AN EXAMPLE: A SYSPAL PROGRAM OBJECT

In this chapter, I shall demonstrate how Catoan might be used. The demonstration will be based on an example: a "Syspal program object." A Syspal program object is a convenient way to store a program written in Syspal using Catoan as the object storage mechanism.

In this object, one would store a Syspal program though the same general structure, if not the exact structure, could be used for storing programs written in most languages. The Syspal program object is an extension of the versioned Catoan object described in Section III.D, and the cross-referenced Catoan object described in Section III.C.2. In addition to the operations pertaining to Syspal programs, the operations of the versioned Catoan object and those for cross-referencing are part of the definition of the Syspal program object.

IV.A. Motivation.

Classically, a program is stored as a collection of files, each one containing some portion of the program. For example, one might have a source file, a documentation file, an object-code file, an interface file, a load-able (executable-code) file, and so on. These are usually differentiated by a suffix indicating the kind of file: ALG68 for an ALGOL68 source file, PL1 for a PL/1 source file, DOC for a documentation file, OBJ for an object-code file, et ceterae. Each file is individually visible to the user.

A typical scenario in a system like this is as follows. A user wants to write a program to help him balance his checkbook. Assume he wants to use the Syspal programming language. He types something like

edit CheckBook Syspal new

meaning that a new file, of "type" Syspal, named "CheckBook," is to be edited. Upon finishing his first attempts at writing the program, he might type

run CheckBook

with a resultant error message like

NO SUCH FILE: CheckBook.LOAD

which is reported because he had never compiled the program. Upon discovering his error, a likely follow-up might be

compile CheckBook

for which another error message might be generated, because there is no COMPILE command. Finally, after much aggravation, the user might realize that he should type

Syspal CheckBook

which would compile his program.

Thinking that he can now run his program (assuming it compiled properly), the example user might type

run CheckBook

for which an error message like the one he received the last time he tried RUN would be elicited. Eventually, he might realize that

link CheckBook

is needed, after which

run CheckBook

would work -- assuming that SYSPAL, LINK, and RUN did not require the user to supply the proper suffixes for CheckBook.

How many times does the user actually care about the object-code file, or the load-able file? How many times does the user actually care about compiling, or about linking (except to check for compile-time errors)? Why can't RUN simply produce a properly executable form of the program? Abstractly, the user is writing a Syspal program, not a machine-language

program; what does he care about the representation of his program? (Indeed, even if he were writing a machine-language program, the representation of the program may of no concern to him.)

The example presented in this chapter addresses these problems. The Syspal program object defined in the next section consists of several internal parts, which correspond to the classical object-code, load-able, documentation, source, et ceterae files. Normally, these are of no concern of the user, and so need not be dealt with explicitly (though the ability to do so exists).

IV.B. Definition.

Like any abstract object, a Syspal program object is defined by the operations one performs on it. The primary operations one performs on such objects are NEW, DELETE, EDIT, RUN, EDIT_DOCUMENTATION, and DEBUG. Secondary operations, which exist more for efficiency than for completeness, include COMPILE, and RESOLVE_REFERENCES. In addition to those operations specific to Syspal programs, the operations of the versioned Catoan-object and the cross-referenced Catoan-object are part of the definition of the Syspal program object. These extra operations are available directly to the user because of the %VISIBLY_EXTEND statement. Figure 10 shows the interface for and representation of the Syspal-program object.

```
new:PROCEDURE
    RETURNS(p:@Syspal_program)
    (* Instantiates a new Syspal program. *);
delete:PROCEDURE
    (* Destroys a Syspal program and its subsidiary objects. *);

edit:PROCEDURE
    (* Allows modification to a Syspal program. *);
run:PROCEDURE
    (* Executes the Syspal program. *);
edit_documentation:PROCEDURE
    EXCEPT(syspal_program_no_documentation)
    (* Modifies the documentation of a Syspal program. *);

compile:PROCEDURE
    EXCEPT(syspal_program_compilation_failed)
    (* Compiles the Syspal program. *);
resolve_references:PROCEDURE
    EXCEPT(syspal_program_unresolveable_reference)
    (* Resolves external references (calls the system LINKER). *);

debug:PROCEDURE
    (* Invokes the DEBUGGING subsystem. *);

%VISIBLY_EXTEND versioned_catoan_object,
                cross_referenced_catoan_object;

SELF: RECORD
    program: versioned_catoan_object;
    xref: cross_reference_information;
    (* Use of the VERSIONED_CATOAN_OBJECT:
      CONTENTS                 = source code.
      unnamed DIRECTORY slot 1 = object code.
      unnamed DIRECTORY slot 2 = documentation.
      unnamed DIRECTORY slot 3 = interface.
      unnamed DIRECTORY slot 4 = object code with external
                                 references resolved.
      named   DIRECTORY slots  = sub-programs. *)
    END;  !SELF
```

Figure 10: A Syspal-Program Object.

The NEW operation is invoked when a Syspal-program object is created. It takes no arguments, and returns as a result the new object. Usually, this operation is automatically invoked by the EDIT operation on a new program. NEW initializes the various fields in the representation of the program before returning.

DELETE destroys a Syspal-program, and all of its underlying sub-objects and versions.

The EDIT operation is invoked when changes are to be made to the program. As mentioned above, EDIT will invoke NEW if a new program is being edited. The only argument of the operation is the implicitly supplied program object; it returns nothing.

RUN attempts to execute some representation of the program. For Syspal programs, this may require compiling first. RUN verifies that valid, current executable code exists for the source; if it does not, RUN will implicitly invoke the COMPILE operation. If the supporting system requires pre-execution binding (linking), RUN will also invoke the RESOLVE_REFERENCES operation. Once current executable code is obtained, RUN will transfer execution-control to the program.

EDIT_DOCUMENTATION provides access to the DOCUMENTATION portion of the Syspal program.

DEBUG calls a debugging facility, allowing the programmer to control the execution of the program, to examine the state of its execution, et ceterae.

The secondary operations, COMPILE and RESOLVE_REFERENCES produce object- and bound-code, respectively. As mentioned, they exist primarily for efficiency. They would probably be used by a programmer to be sure that an error would not occur if someone else should cause the operations to be implicitly invoked.

In addition to the explicitly defined operations listed above, the operations of version management and cross-referencing, as well as those of the basic Catoan-object, are available for use with Syspal program objects. The %VISIBLY_EXTEND pseudo-statement causes the named interfaces to be included in this one. (Appendix A describes this in a little more detail). Syspal programmers can treat Syspal program objects as ordinary Catoan-objects, including them in other Catoan-objects, including other Catoan-objects in them, explicitly creating new versions, accessing the cross-reference information, et ceterae.

For example, assume that a user named "Ribak" was writing a system composed of several Syspal programs. One of the programs (called "DRIVER") is the top-level program, which controls dispatching the other parts of the subsystem. One way to reflect this structure in the external structure of the programs is to have the other parts of the subsystem be sub-objects of DRIVER, included in the DIRECTORY of the Catoan-object used to store the DRIVER Syspal program object. Then, Ribak could easily see the system's structure by NAMED_READing the DIRECTORY of the Catoan-object.

IV.C. Use.

To use the Syspal program object, a user would invoke the EDIT operation. EDIT would obtain the source code of the program, or initialize it to empty if the program was new. The user would make whatever changes had to be made, replace the old edition of the program with the updated one (or, perhaps, create a new version instead), and terminate the editing session.

If the editor was able to check some or all of the syntax and semantics of the program, a COMPILE merely to verify that no compilation errors existed would be unnecessary. If the editor was unable to perform such checks, the user might want explicitly to COMPILE the program if he was not going to run it immediately, and someone else might try to RUN it before he

had a chance to do so. Otherwise, he could invoke the RUN operation, which would automatically invoke COMPILE and, if necessary, RESOLVE_REFERENCES.

If an error is discovered while RUNning the program, the DEBUG operation could be invoked, allowing the programmer to examine the program and its environment. If changes were made to the program while debugging, EDIT could be called directly by DEBUG, thereby automatically incorporating changes which were made while DEBUGging into the permanent copy of the program.

Assume that the programmer finishes DEBUGging the program, and then neglects to COMPILE the program. One of the users of the program then tries to RUN the program. At this point, the COMPILE operation is implicitly invoked, and the program is transformed into some form which can be executed by the host system. The user had no knowledge of this transformation; it is an implementation detail.

The Syspal program object is an extension of the Catoan-object. This allows the programmer to use the properties of Catoan-objects when thinking about managing his programs. For example, if someone has written a utility program which produces a copy of a Catoan-object, that same program could probably be used with Syspal program objects with very little, if any, modification. If other computing systems or other naming environments (see

Chapter Five) could reference his Catoan-objects, then they could, likewise, reference his Syspal program objects. This allows the issues of object management to be left to the object manager, regardless of the use to which the objects are being put, regardless of the extensions which are made of the basic Catoan-object.

IV.D. Summary.

Many people do little with computers but write programs on and for them. Generally, the abstractions available for their use for actually writing the programs are very primitive. The Syspal program object presented above is a high-level abstraction for writing and storing programs which is based on the Catoan-object and its extensions.

CHAPTER FIVE

IMPLICATIONS OF MULTIPLE NAMING ENVIRONMENTS

As mentioned in Chapter One, I do not assume that Catoan is the only manager for named, permanent objects that exists in the system. Therefore, Catoan's is not the only naming environment in the computing system. If there exist other naming schemes, and another naming environment is created which is disjoint from the one I propose, what are the implications? Are the name spaces forever disjoint? Is there a way to refer to objects in one namespace while within another? Is there a way to transfer objects from one namespace to another, either from within either of the two namespace in question, or from a third one?

V.A. Disjoint Naming Spaces.

Given the existence of more than one object manager, it is very probable that the objects of one system cannot be handled by the others. In classical file systems, internal storage formats may differ, the system overhead information stored may differ, the structure of the files may differ -- in fact, the "type" (in the programming language sense of the word) of the files may be incompatible, so that the different kinds of files are implemented by different modules.

In Catoan, the naming mechanism is part of the object structure, and is handled by the object management mechanism. Separating names from objects is not part of Catoan's underlying philosophy. Therefore, regardless of the structures of other object managers, regardless of the naming mechanisms of other object managers, if an object is not a Catoan-object, it cannot be named within Catoan.

If Catoan-objects can be named and accessed directly by some other object manager, the naming structures are not disjoint. In this case, data transfer is no problem, and is, indeed, a moot point: the objects of both systems are accessible from one of the systems.

Let us assume that, not only can Catoan not access non-Catoan-objects, but other systems likewise cannot access Catoan-objects, either. In this case, the naming structures are truly disjoint, and data in the objects of one system cannot be transferred directly into objects of the other. What is needed for such data transfer is some procedure which can bridge the two naming structures.

To be able to write a "bridging" procedure, it must be possible to access both object managers from the same procedure. This requires that the interface for both systems be available to the procedure. The procedure must be able to name and to access (in a protection sense) the interfaces; if naming can be done at this level directly, with internal names (segment numbers, capabilities), then providing the procedure with the internal unique identifiers of the two object managers produces the necessary availability.

If naming cannot be done with internal names, then a mechanism is needed to allow translation of external names (character strings) to internal ones. This requires, essentially, another name manager for "special" interfaces which are needed between, among, and above the normal naming structures.

Once the interfaces (and modules) for both object managers are available to the bridging procedure, transferring data between the two naming environments involves obtaining the necessary information from one environment (using the operations of its objects), and supplying that information to the other environment (using the operations on its objects). The author of the procedure must, therefore, know the interfaces for both systems. Such bridges might be provided as part of a system-wide library of "utility" routines.

V.B. A Standard Interface for Filing Systems.

An alternative to forcing someone who is trying to transfer data between object managers into learning the idiosyncrasies of both systems is to have all object managers meet the same interface (if standard data transfer is to be possible). This interface would specify the minimal set of operations required of an object manager, and would also allow data to be transferred freely among object management systems and their naming environments.

Because of the wide variety of storage techniques, protection schemes, and information collected, access to the "overhead" information will not be included in the "standard, minimal interface" which will be defined. Because there are many ways to interpret names, many ways to organize a

naming structure, many ways to attach semantics to a naming structure (be it a hierarchy, a network, or even a list), passing components of names to the object manager may not make sense. Because there are many ways to structure data, a limited means for accessing an object manager's data will be provided. Figure 11 shows the standard, minimal interface for object managers.

```
lookup: PROCEDURE(name:string(*), root:@TYPE(SELF)
     RETURNS(obj:@TYPE(SELF))
     EXCEPTION(name_not_found(index: INTEGER),
               name_invalid(index: INTEGER))
     (* Translates a character-string name into an object
        reference, relative to "root."  "index" is the
        position in "name" up to which the name could be
        found or parsed. *);

contents_read: PROCEDURE
     RETURNS(cont:@contents_type)
     (* Extracts the CONTENTS from the object. *);

contents_set: PROCEDURE(cont:@contents_type)
     EXCEPTION(contents_type_inappropriate)
     (* Places "cont" in the CONTENTS of the object. *);
```

Figure 11: Standard, Minimal Interface for a Filing System.

Names are handled in their entirety only, and are relative to some point which is supplied by the caller. This "root" pointer may be NULL, in which case the object manager determines the root. If the root is not NULL, the name is parsed relative to the supplied root. For example, if one wanted to have a Multics file system parse the name

"^udd^CSR^Marcum^thesis," the root would not have to be specified, because Multics has one global root. If "Marcum^thesis" were to be located relative to "^udd^CSR", then "^udd^CSR" could be supplied as the root.

If the above example were to be executed in Catoan, and the object "thesis," a sub-object of the object "Marcum," were to be found, a pointer to CSR would be supplied as the root, and "Marcum^thesis" would be supplied as the name.

Just as names are handled in their entirety, the data contained in an object are accessible only in their entirety. One is allowed to _SET and _READ the CONTENTS of some object as a whole. Returned by _READ is a pointer to the CONTENTS, which may be of arbitrary type, just as the CONTENTS of a Catoan object may be of arbitrary type. _SET's argument is a pointer to a datum of arbitrary type to be used as the CONTENTS of the object.

Some object managers may have to place restrictions on the types of the objects which are the CONTENTS being stored. It is the responsibility of the object manager to verify that the type of the CONTENTS is sensible for that particular style of object manager. The exception CONTENTS_TYPE_INAPPROPRIATE is provided to allow a standard mechanism for signalling such a problem.

Catoan does not meet, as described so far, the standard, minimal interface. The operations on the CONTENTS (_SET and _READ) are compatible, but an additional DIRECTORY operation is needed to take a full name and a root point, and return a pointer to the named object. This is a simple addition, with which Catoan meets the standard, minimal interface of Figure 11.

V.C. Garbage Collection.

Reclaiming storage used by objects which are inaccessible may be necessary. If such "garbage collection" is needed, how does the existence of multiple naming environments affect garbage-collection?

Garbage collection is a reclamation of the physical storage used by logical entities (objects) which become inaccessible. Garbage collection techniques have been a topic of investigation for a long time; they still are. I shall not discuss the actual techniques here; the interested reader is referred to [3, 4, 30, 33]. Rather, what follows is a discussion of the effects of multiple name spaces on garbage collection.

Usually, garbage collection is performed by the object manager. If this view of garbage collection is taken, all works well while there is

only one object manager. Indeed, all may work well within each of the individual object managers. Each object manager has enough information to garbage collect its own objects. What happens, however, if there exist inter-namespace references? What happens if an arbitrary object can refer to another arbitrary object, as can happen in Catoan?

A possible solution is to extend the standard, minimal interface for object managers (see Figure 11) to include operations for communicating garbage collection information. Suppose two object managers, Catoan and Namit, exist in one computing system. Let "C1," "C2," et ceterae be Catoan-objects; let "N1," "N2," et ceterae be Namit-objects. There can be references in C1 to C2, for example, and there might be references permitted between two Namit-objects. Objects in Catoan can certainly reference objects in Namit; whether objects in Namit can reference Catoan's objects is immaterial.

Perhaps C1 references C2, and C2 references N6. Catoan reaches a stage when garbage collection is required, and so it scans its objects for inter-object references. It records the C1-C2 reference. Upon discovering the C2-N6 reference, it must transmit the information that N6 is referenced to N6's manager, Namit. How might this be done?

Let us assume that Catoan can determine that N6 belongs to Namit (I shall return to this issue shortly). Catoan must (conceptually) send a message to Namit indicating that N6 is referenced from some other naming environment. Perhaps Catoan would even specify that N6 was referenced from the Catoan naming environment, by object C2. How would Catoan name N6 to Namit? If all inter-namespace references are symbolic, Catoan could use the same name that C2 used. (This also solves the problem of determining the object manager of N6, mentioned above.) If, however, references are direct (rather than symbolic), as they could be in Catoan, it would be necessary to pass to Namit the direct reference (which might be a segment number). This presents no problem if garbage collection can be done without object names, as is usually the case.

Direct references pose another problem: how does Catoan determine that Namit is the manager of N6? Perhaps some extra information is stored with the reference in C2 to N6 enabling Catoan (or any other object manager) to determine that the reference is to an object of some other object manager. (Indeed, some such information is needed to allow an object manager to determine at least that an object reference is to one of its objects or to an object of some other object manager.) Another possible solution is to maintain a directory of references to objects of other object managers.

Regardless of the exact methods for solving the various problems of inter-namespace references, garbage collection will require much inter-object manager communication to convey the inter-namespace references. Furthermore, additional complexity is introduced into the standard, minimal interface for object managers of Figure 11, into the information stored for references, into the mechanics of garbage collection. [4] contains a discussion of garbage collection in multiple address spaces with inter-address space references. When the address spaces are logical rather than physical, when they are name spaces rather than address spaces, when they are managed by more than one entity, garbage collection is even more difficult than as described in [4].

Another solution, which I prefer, is to make garbage collection the function of the memory management system. This is especially appealing in an addressing system in which all references must be made through tagged "pointers." Such references can be recognized easily by the memory manager (because they are tagged). Generally, as long as the memory manager can determine that a reference to an area of storage exists somewhere, the precise form of addressing is immaterial -- it can be through segment numbers, disc addresses, capabilities, et ceterae.

If the memory management system can determine that an area of memory is referenced, regardless of where the reference is located within the memory

system, it can do the garbage collection. The memory management system is below the object managers. Furthermore, because the memory management system is part of the operating system kernel, all object managers use the same (the only) memory manager. Therefore, because a single entity has access to all the object references, and can determine when something is and is not an object reference, the problem of garbage collection in multiple naming environments is solved.

V.D. Summary.

Chapter Five has presented the issues surrounding the existence of multiple naming environments in a computing system. The effects of multiple naming environments on system-wide naming, on transferring data among name spaces, and on garbage collection (storage management) were discussed. A "standard, minimal" filing system interface was described.

CHAPTER SIX

SUMMARY, AND EVALUATION OF THE PROPOSED SOLUTION.

In the following, I look at my proposals, commenting on what they are and "where I am," on their completeness, and on the trade-offs that have been or could be made. I examine them with regard to previous work and what "might be done." Lastly, I present my recommendations for further research in the area of managing named, permanent objects in computing systems which range in size from a single-user personal computer to a distributed network composed of many autonomous hosts (which range in size from personal computers, to multiple-user computing utilities, to networks themselves).

VI.A. Summary.

This report has presented the results of an investigation into storing things in modern computing systems. The investigation has produced a design of a system called "Catoan," which is a manager for named, permanent objects. Colloquially, such a manger could be considered an object-oriented filing system.

A description of existing ways of viewing permanent storage was presented in Chapter Two, describing Honeywell's Multics and Hewlett-Packard's MPE/3000 in depth. Bell Telephone Laboratories' Unix was briefly described, as was Carnegie-Mellon University's Hydra. The file systems in each of these influenced my thinking about permanently storing objects in a computing system. A few methods for maintaining versions of objects were also described in Chapter Two.

In Chapter Three, I described Catoan. The "Basic Catoan-object" was defined and described, and a representation of the information in the Catoan-object was presented. Refinements of the basic object were shown, including an access control list protection scheme, cross-referencing, and version maintenance. A general scheme for storing versions was described, which allows the user to make the space-time trade-offs which most other version maintenance schemes make for the him.

An example of using Catoan was described in Chapter Four. A Syspal-program object was built using the cross-referenced and versioned Catoan-objects.

Chapter Five related the problems which occur when multiple naming environments exist in the same computing system. It is assumed that Catoan might not be the only object manager in the computing system, and that users might desire to transfer information among object managers and their naming environments. The effects of multiple naming environments on garbage collection were also stated.

More globally, more abstractly, in this report I have described a view of storing data in a computing system which departs from the classical view. I have made this departure because the classical views of data storage are not amenable to many of the current philosophies on programming, software engineering, and data abstraction. Catoan allows one to think of data storage in the abstract; it allows one to think of storing abstract data objects, rather than storing "piles of bits."

Catoan is merely a type manager, for a Catoan-object. However, it is a rather odd type manager: it gives out references to portions of the representation of its data -- namely, a pointer to the CONTENTS. It is

this aspect of Catoan which makes it untrusted: part of the representation of a Catoan-object is not secure.

VI.B. Completeness.

Catoan has also been a vehicle for exploration. Very rarely is the permanent data storage mechanism of a computing system not trusted. Very rarely do multiple filing systems exist within the same computing system. Yet, these are two important issues in the design of Catoan.

When one is exploring and experimenting, there is a good chance that the results will not be perfect. So it is with Catoan. The decision that Catoan need not be trusted, and will not be trusted, limits its use. Because of the lack of trust, Catoan cannot enforce extended controls on access to the data of a Catoan-object.

If one were to trust Catoan, and make Catoan the only object manager, then other filing systems and naming environments could still exist. However, rather than building directly on the memory management facilities, the other filing systems would build on Catoan. Although this does solve the trust issue, it introduces inefficiency by imposing another layer of mechanism between the user and permanent storage. It may limit flexibility if, in fact, a particular application is ill suited to Catoan (a possibility if for no other reason than Catoan is not implemented).

Hopefully, Catoan could be implemented efficiently, so that the additional layer would not cost very much.

The naming scheme of Catoan allows a network of Catoan-objects to be built. This introduces additional complexity by making it more difficult to traverse the naming environment. When writing a program to traverse a tree, it is known that there will be no loops encountered during the traversal. But, when traversing a network, it is possible to encounter a loop; therefore, loop detection is needed. However, the additional flexibility gained by allowing multiple parents and, therefore, a naming network often outweigh the cost of additional traversing complexity. Furthermore, because a network is a superset of a hierarchy, a naming hierarchy can be used, foregoing the generality (and cost) of a network.

Catoan has no concept, analogous to the soft link, of associating an external name with another external name. Catoan recognizes only hard links, and multiple parents of an object. There are semantics of soft links which cannot be modeled using hard links. For example, allowing a user to use the same (local) name for some object, regardless of the modifications made to the object, is much easier using soft links. If it is possible at all with hard links (and this depends on the type of internal name to which a hard link translates an external name), substitution is usually much more visible to the unconcerned user than with

soft links. Nonetheless, because changing the CONTENTS of a Catoan-object does not affect the containing objects, the "soft substitution" provided by soft links is easier to approach with Catoan hard links than with, for example, Unix hard links.

The Catoan philosophy would dictate that, because of uniformity, each object should contain a section for soft links, if they were to be included in Catoan. An alternative is to introduce a new type of Catoan-object, a "soft_link." This points out another feature of Catoan: there is only one type of Catoan-object. This forces the overhead of both portions on all the users of Catoan, even if eighty-seven percent of their objects do not use the CONTENTS.

One of the most important questions to be answered about Catoan is: "Can one do everything with Catoan that one can do with 'conventional' file systems?" I claim that, except for issues of trust, one can, and that, in fact, one can do some things in Catoan that cannot be done in many existing file systems. As to trust, the overhead operations are most greatly impacted by not trusting Catoan -- the SYSTEM OVERHEAD INFORMATION is not necessarily correct.

The data-oriented operations in Catoan are the "CONTENTS" operations, described in Section III.B.1.b. The operations are very simple, and from

their simplicity comes much generality. Also, because of the lack of constraints on the structure of the CONTENTS, anything which can be described in Syspal can be stored directly in a Catoan-object. (It can be argued that Syspal's data description facilities are universal; such arguments are outside the coverage of this report.)

Because Catoan allows an arbitrary network of objects in its naming structure, relationships which cannot be expressed in some other systems (for example, hierarchical naming environments) can be easily expressed in Catoan. Objects can be composed of sub-objects, which may themselves be composed of further sub-objects, any of which (at any level) may be part of other objects.

In the basic Catoan-object, there is no provision for enforcing protection (except at the CONTENTS's type level, which is somewhat clumsy). Protection is, however, introduced as a refinement. This refinement is merely a suggestion, and is presented as such to re-enforce its optionality. For similar reasons, cross-references and version maintenance schemes are extensions and refinements, and are not critical to the basic theory.

No mechanisms for concurrency control have been presented in this report. This is because there are very many schemes, ranging from

"classical" locks, to monitors [14], to semaphores, to event counts [25], to some very recent, perhaps esoteric schemes aimed primarily at distributed systems [24]. If one desired to implement concurrency control atop the basic Catoan-object, or any of its refinements, this could be done, and should not impact the abstractions which exist.

When designing a computing system, recovery from semi-catastrophic failures and from human errors is often considered. The concept of off-line backup of on-line storage is crucial to a system which portends to be a safe repository for its users' data [31]. However, backup is not discussed in this report. To make Catoan complete, some form of off-line backup must be included, at some level. This was not done here because of the implications that lack of trust has on the ability to access data so as to transfer it to off-line backup. If Catoan is, in fact, not trusted, the task of backup must be relegated to the memory manager, which is trusted, or to a higher level abstraction which is in a better position to implement backup when it is needed.

VI.C. Trade-Offs.

An implicit trade-off has been responsibility for memory management. Most filing systems perform their own buffering between primary and secondary memory; Catoan relies on the underlying memory management system

for this. While this certainly simplifies Catoan, and helps support the multiple-level, abstract system concept [23, 36, 2], there may be a sacrifice in control over buffer management, resulting in a decrease in system performance.

In a memory system which is "automatically" managed, the performance degradation will generally be local, visible only to the user of Catoan whose application would benefit from detailed control over the buffer management. However, such local control will often result in degraded global performance, because the memory (buffer) manager, which has more global information than the filing system, is being circumvented.

An instance of the "classical space-time trade-off" can be found in version maintenance. One has the option of very fast access to any version (at the expense of storing each version in its entirety), or of very little storage (at the expense of building the requested version from a "base" by applying "updates"). This trade-off has been left to the user of Catoan's version maintenance system, by allowing him to specify a "base," a set of "updates," and a program to apply the updates to the base. See Section III.D for further details.

The view of stored objects presented by Catoan is very unlike that presented by most existing object managers (filing systems). Usually,

stored objects are viewed as a one-dimensional array of records (byte strings). This view allows the object to be access in pieces, rather than requiring that it be accessed in its entirety (as far as the object manager is concerned). This decision allows objects to be viewed abstractly, and to have an internal structure which is unknown to Catoan. If a more classical view is desired (because, for example, most of the object are very large, and one generally wants to access only a small portion of them, anyway), a record-at-a-time view could be built atop Catoan, using Catoan to actually store the object. Because Catoan's CONTENTS_READ operation returns a pointer to the contents, rather than the entire contents itself, such a system would not require modification to Catoan, nor would it generate excessive memory referencing from reading in the entire contents.

What happens if some portion of memory is volatile? How must Catoan be changed so that a user can be assured that his data is in stable storage? Catoan must provide the user with a MAKE_NON_VOLATILE operation which performs a "synchronous write" so that, upon termination of the operation, the user is assured that the object has been transferred to non-volatile storage. This requires a similar operation exist for the memory manager, since the view it presents to Catoan is that of non-volatile storage.

A very important trade-off is that of trust. Because Catoan need not be trusted, the information in the DATEs and PRINCIPALs fields may be

inaccurate. Lack of trust implies a certain difficulty in enforcing security and in implementing backup, and implies certain uncontrolled accessibility to Catoan-objects (in particular, to the CONTENTS). But, Catoan is optional. If Catoan provides protection mechanisms, if Catoan is secure, then it must be trusted, and it probably becomes mandatory.

VI.D. Remaining Work.

Much has been done on and with Catoan. Much is left to do: more theory needs developing, practical experience needs to be gained with the concepts embodied in Catoan. This section describes some of the work which remains to be done relating to Catoan and the ideas presented in this report.

As mentioned in Chapter One, Catoan might be used on a machine which is part of a multi-node network. In such an environment, one often wants to name resources which exist at remote nodes. Furthermore, one often wants to locate a resource thought to exist somewhere in the network, but at an unknown node. Despite the need for investigation into this area, this report on Catoan does not address network-wide filing systems or naming environments. One possible view of a network-wide filing system built using Catoan is to consider the remote nodes as representing other members of a collection of multiple naming environments. It might then be possible

to apply the concepts presented in Chapter Five to the problems of network-wide filing systems.

Issues of protection, security, and sharing are relevant to the goals of Catoan. These have been discussed briefly throughout this report; additional work is needed to present a unified view of protection and sharing to the users of Catoan that is both convenient and powerful.

As discussed in Section III.C.2, when implementing cross-references there is a problem of who pays for the storage occupied by the cross-reference information. This is part of a more global problem of how to determine the amount of storage in one principal's space which is occupied by the data of another principal (including "The System"). I know of no previous work done in this area.

Designing a system which is robust in the face of host-system failures is still a large open research question. Because Catoan manages permanent data objects, it should provide stability in the face of failure.

Lastly, how might one implement Catoan? How difficult would it be? Is the environment Catoan presents to its users really the right one? Is Catoan complete, sufficient, and easy to use? Only an attempted implementation can answer these questions.

APPENDIX A

SUMMARY OF THE SYSPAL PROGRAMMING LANGUAGE.

This appendix summarizes the salient features of Syspal (1) [10] as they relate to this presentation. The reader is warned that this is not a definitive explanation of the language, nor is it complete. The reader is warned further that this represents Syspal as I knew it in May, 1979, while the language was still undergoing active development. The language as it actually is defined at the time this paper is read, or even published, may differ substantially from the summary presented here.

Syspal is a data-abstraction language, based on Pascal, and geared toward systems programming. Much of the syntax and semantics are derived from Pascal, and from CLU. One of the design goals of Syspal is to support modular programming conveniently.

---

(1) Syspal is an experimental programming language under development at Hewlett-Packard Laboratories, Electronics Research Center, Computer Research Laboratory, in Palo Alto, California.

Data Types.

Syspal provides the programmer with a few standard, "built in" data types. Various forms of enumeration types, which specify the range of values of a type, are available. Using enumerations, the usual INTEGER, REAL, BOOL, and CHAR types can be defined. For example, INTEGER might be defined

INTEGER = TYPE -1000000 TO 1000000

if INTEGERs between positive and negative one million were desired. The REAL type might be

REAL = TYPE PRECISION 6 EXPONENT 32

stating that six digits of precision and an exponent between positive and negative thirty-two was available. BOOL, representing truth and falsehood, could be defined

BOOL = TYPE UNORDERED(TRUE, FALSE)

where UNORDERED specifies that the relations based on order (less, greater, et ceterae) are not defined on BOOLs (though equal and not equal still are). The CHAR type represents the ASCII character set, and is an ORDERED collection of the values according to the ASCII collating sequence.

In addition to the scalar types, aggregates are provided by Syspal. Two kinds of aggregates exist: RECORDs and ARRAYs. ARRAYs are homogeneous collections of elements which can be referenced using numeric subscripts.

A definition like

```
x: ARRAY(1 TO 6) OF INTEGER
```

defines "x" to be a six element ARRAY of INTEGERs. The declaration

```
y: ARRAY(*) OF CIRCULAR(0, 1, 2)
```

specifies "y" as an array with unknown size of modulo-three integers.

RECORDs allow non-homogeneous data to be included in the same aggregate. The elements of RECORDs are accessed by their field names. For example, suppose the following definition were part of a Syspal program:

```
employee: RECORD
    name: string(30);
    addr: RECORD
        street: string(35);
        city_state: string(35);
        zip_code: 0 TO 99999;
        END;  !address
    salary: 10000 TO 500000;
    monthly_productivity: ARRAY(1 TO 12) OF 0 TO 10;
    END;  !employee
```

This defines the variable "employee" to contain four fields: "name" (a character-string of length thirty; see Section I.C for a definition of strings); "addr" (which itself is a RECORD, consisting of two thirty-five character strings and a non-negative integer less than 100,000); "salary" (an integer between 10,000 and 500,000), and "monthly_productivity" (which is another aggregate: an ARRAY containing twelve elements, each of which is an integer between zero and ten).

In addition to being able to define variables, the Syspal programmer is allowed to define new types. This is done in the same way that INTEGER, REAL, et ceterae were defined above. For example,

```
address = TYPE RECORD
    street: string(35);
    city_state: string(35);
    zip_code: 0 TO 99999;
    END;  !address
```

defines a type called "address," with the same structure as the "addr" field in the "employee" structure above (also called "employee.addr"). A programmer-defined type (call it "PDTP") can be an extension of some other type (the "base type," call it BTP)., meaning that PDTP is built on BTP and "extends" it. Unless specifically prohibited, an extension of a type will match the base type for the purpose of compile-time type checking.

Defined types can have user-specified parameters, as shown in the definition of the "string" type found in Section I.C. Parameters are very useful when defining modules, such as a stack consisting of INTEGERs, or of REALs; see below for a discussion of modules.

One can also define a variable or type as the UNION of two or more types. This specifies that any of the base types might be the type of the defined variable.

Syspal provides pointers. Pointers are typed, and can refer to only one kind of object (as opposed to PL/1 pointers, which can reference anything). A pointer to an INTEGER is declared

pint: @INTEGER;

and a pointer to an address would be

paddr: @address;

If the value of "pint" were assigned to "paddr," an error would be raised.

Control Structures.

Most of the "usual" flow control constructs exist in Syspal. Conditionals (IF-THEN-ELSE and CASE), iteration (WHILE, REPEAT, FOR, and LOOP [infinite repetition]), exception handling (EXCEPTION), and procedure calling (CALL), among others, are provided. In addition, iteration can be controlled by a "sequencer" (1), which is a co-routine to provide the next value for iteration.

---

(1) This is very similar to the CLU "iterator" [22].

Procedure and Function Definition and Calling.

Procedure declarations have the form:

```
name: PROCEDURE(parm1:type1p, parm2:type2p, ...)
    RETURNS(var1:type1v, var2:type2v, ...)
    EXCEPTION(cond1(exvars1), cond2(exvars2), ...);
```

This defines a PROCEDURE called name. The parameters are parmN (N being 1, 2, et ceterae), of types typeNp. The procedure returns values of types typeNv through the internal names varN. Exceptional conditions condN can be raised in this procedure; they will return with parameters exvarsN, respectively. The parameters, RETURNS clause, EXCEPTION clause, and vars portion of the exceptional conditions ("condN") are optional.

As mentioned in Section I.C, Syspal recognizes the type of the implicit operand to module operations, and, furthermore, assigns this implicit operand to the keyword "SELF." Type checking is performed for calling sequences, as well as for other variable references.

In addition to a normal procedure termination, an abnormal termination can occur. There is only one way for a procedure or function to terminate normally: assign a value to the RETURNS variable defined in the function header (if any exist), and exit through the end of the procedure or function. An abnormal termination is indicated by the RETURN statement.

Abnormal termination can, in addition to returning the name of the exceptional condition, return values which can be used by the calling procedure to diagnose the error.

Modularity, Data Abstractions, and Interfaces.

Syspal is a data-abstraction language, similar to CLU [22], for example. The Syspal analogue to the CLU cluster is a "module." When one defines an abstract data type, one does so by defining the module which will manage the abstraction. Variables of the abstract type are then declared to be of the module's type.

The abstraction is defined by the "interface" of the module. The interface defines those things (operations, constants, type declarations, et ceterae) which are to be visible to users of the abstraction; all other information about the module is invisible to all but the module itself. A module can have many interfaces; for example, the creator of an object might be able to modify the object, but he might not want others to be able to modify it, only to read it. Figure 12 shows the definition for a module implementing a STACK abstraction. The module definition, including the operations and representation, and three interfaces are presented.

```
MODULE stack(element_type: TYPE, stack_lim: INTEGER):
       stack, strict_stack, loose_stack, pseudo_stack;
   new: PROCEDURE
       RETURNS(stk: @stack)
       (* Creates a new STACK, of "type" ELEMENT_TYPE, with
         STACK_LIM elements (maximum). *);
       ALLOCATE SELF;
       SELF.tos := 0;
       stk := EXT(SELF);
       END PROCEDURE;  !new

   push: PROCEDURE(val: element_type)
       EXCEPTION(stack_overflow)
       (* Puts VAL onto the top of the stack. *);
       IF SELF.tos=stack_lim THEN
          RETURN(stack_overflow);
          ELSE SELF.tos :=# +1;
               SELF.elements(SELF.tos) := val;
               END;
       END PROCEDURE;  !push

   pop: PROCEDURE
       RETURNS(top: element_type)
       EXCEPTION(stack_underflow)
       (* Return and discard the top of the stack. *);
       IF SELF.tos=0 THEN
          RETURN(stack_underflow);
          ELSE top := SELF.elements(SELF.tos);
               SELF.tos :=# -1;
               END;
       END PROCEDURE;  !pop

   is_empty: PROCEDURE
       RETURNS(ans: BOOL)
       (* Returns TRUE if the stack has no elements. *);
       ans := SELF.tos=0;
       END PROCEDURE;  !is_empty
```

```
make_empty: PROCEDURE
    (* Forces the stack to have no elements. *);
    i : INTEGER;
    operation_not_defined_on_type: EXCEPTION;

    i := 1;
    EXCEPTION
        ON operation_not_defined_on_type DO
            i := stack_lim+1;
        BEGIN
            WHILE i<=stack_lim DO
                SELF.elements(i) := NULL(element_type);
                i :=# +1;
                END;
            END;
    SELF.tos := 0;
    END PROCEDURE;  !make_empty

extract: PROCEDURE(index: INTEGER)
    RETURNS(elem: element_type)
    EXCEPTION(stack_nonexistent_element(size: 1 TO stack_lim))
    (* Returns the INDEXth-from-top element (top = 1). *);
    IF index>SELF.tos THEN
        RETURN(stack_nonexistent_element(SELF.tos));
        ELSE elem := SELF.elements(SELF.tos-(index-1));
    END PROCEDURE;  !extract

insert: PROCEDURE(val: element_type, index: INTEGER)
    EXCEPTION(stack_nonexistent_element(size: 1 TO stack_lim))
    (* Sets the INDEXth-from-top element to VAL (top = 1). *);
    IF index>SELF.tos THEN
        RETURN(stack_nonexistent_element(SELF.tos);
        ELSE SELF.elements(SELF.tos-(index-1)) := val;
    END PROCEDURE;  !insert


SELF: RECORD
    tos: 0 TO stack_lim;
    elements: ARRAY(1 TO stack_lim) OF element_type;
    END;  !SELF

END MODULE;  !stack
```

```
!Interface definitions.

(stack(element_type: TYPE, stack_lim: INTEGER),
 strict_stack(element_type: TYPE, stack_lim: INTEGER)): INTERFACE;
    new: PROCEDURE
        RETURNS(stk: @stack);
    push: PROCEDURE(val: element_type)
        EXCEPTION(stack_overflow);
    pop: PROCEDURE
        RETURNS(top: element_type)
        EXCEPTION(stack_underflow);
    is_empty: PROCEDURE
        RETURNS(ans: BOOL);
    stack_overflow, stack_underflow: EXCEPTION;
    END INTERFACE;  !stack, strict_stack


loose_stack(element_type: TYPE, stack_lim: INTEGER): INTERFACE;
%VISIBLY_EXTENDS strict_stack(element_type, stack_lim);
    make_empty: PROCEDURE;
    extract: PROCEDURE(index: INTEGER)
        RETURNS(elem: element_type)
        EXCEPTION(stack_nonexistent_element(size: 1 TO stack_lim));
    stack_nonexistent_element(size: 1 TO stack_lim): EXCEPTION;
    END INTERFACE;  !loose_stack


pseudo_stack(element_type: TYPE, stack_lim: INTEGER): INTERFACE;
%VISIBLY_EXTENDS loose_stack;
    insert: PROCEDURE(val: element_type, index: INTEGER)
        EXCEPTION(stack_nonexistent_element(size: 1 TO stack_lim));
    END INTERFACE;  !pseudo_stack
```

Figure 12: A Module Implementing a Stack.

The STACK module has two parameters: defining the type of the STACK's elements ("element_type") and its maximum size ("stack_lim"). These parameters are passed to STACK when a new STACK is created. They are supplied by the programmer when the particular STACK variable is declared. For example,

```
inventory: stack;
inventory := NEW stack(inven_control_record, 150);
```

declares "inventory" to be a STACK, and instantiates it as a stack of "inven_control_records," with at most one hundred fifty inven_control_records. The list of names after the last colon in the MODULE statement is a list of the interfaces which this module meets.

The NEW operation, invoked by the NEW statement, initializes the fields in the representation of the STACK, and returns the external (abstract) representation of a stack ("EXT(SELF)").

PUSH and POP present no particular surprises. They do illustrate, however, the exception-handling mechanisms of Syspal. The only way to terminate the execution of a procedure normally is to exit through the last statement of the procedure body, having previously assigned to the appropriate variables whatever values are to be returned. If an

exceptional return is to be performed, the RETURN statement is used, naming the exception, and specifying the parameters which might be returned with the exception (see EXTRACT and INSERT).

The IS_EMPTY operation is a predicate to allow the user to see if the stack has any elements. MAKE_EMPTY alters the stack to ensure that, if IS_EMPTY were called immediately after make_empty, IS_EMPTY would return TRUE.

EXTRACT and INSERT allow direct access to the elements of the stack. If an undefined element is accessed, the exception STACK_NONEXISTENT_ELEMENT is signalled, and the current size of the stack is returned with the exception name.

The interfaces allow various forms of access to the STACK abstraction (module). If a strict stack discipline is desired (access to only the top of the stack), the "stack" or "strict_stack" interface would be used. If a slightly looser stack discipline is desired, allowing writing only through PUSH but reading anywhere in the stack, "loose_stack" would be used. If no controls over the use of the stack, but the convenience of a stack, were desired, the "pseudo_stack" interface would be appropriate.

Note that the "loose_stack" and "pseudo_stack" interfaces are built on other interfaces. The "%VISIBLY_EXTENDS" statement specifies that the named interface should be considered as part of this interface, and that this interface extends it. It further specifies that all information in the extended interface should be explicitly visible to the user. (In contrast, %EXTENDS would allow the extending interface access to the operations of the extended interface, but would not allow the user access to the information in the extended interface unless it was explicitly given.)

References

[1] Almes, G. and G. Robertson. "An Extensible File System for HYDRA," Carnegie-Mellon University, Department of Computer Science, CMU-CS-78-102, February 1978.

[2] Anderson, T., P.A. Lee, and S.K. Shrivastava. "A Model of Recoverability in Multi-Level Systems," IEEE Transactions on Software Engineering SE-4 (November 1979), pp. 486-494.

[3] Baker, Henry G., Jr. "Actor Systems for Real-Time Computation." M.I.T. Laboratory for Computer Science Technical Report TR-197, 1978.

[4] Bishop, P.B. "Computer Systems with a Very Large Address Space and Garbage Collection." M.I.T. Laboratory for Computer Science Technical Report TR-178, 1977.

[5] Bonanni, L.E., and A.L. Glasser. SCCS/PWB User's Manual. Bell Telephone Laboratories, 1977.

[6] Dahl, O.-J., and K. Nygaard. "SIMULA -- an ALGOL-Based Simulation Lanugage," Communications of the ACM 9 (September 1966), pp. 671-678.

[7] DEC. DECSystem-20 User's Guide. Digital Equipment Corporation, AD-4179B, 1978.

[8] Dolotta, T.A., R.C. Haight, and E.M. Piskorik, editors. PWB/Unix User's Manual -- Edition 1.0. Bell Telephone Laboratories, 1977.

[9] Eastlake, D., et al. ITS 1.5 Reference Manual. M.I.T. Artificial Intelligence Laboratory Memo AIM-161A, July 1969.

[10] Fraley, Robert A. "Syspal: A Pascal-Based Language for Operating System Implementation," Proceedings of Compcon, Spring 1978. IEEE, 1978, pp. 32ff.

[11] Glasser, Alan L. "The Evolution of a Source Code Control System," preprint of a paper submitted to the IEEE Transactions on Software Engineering. Bell Telephone Laboratories, 1978.

[12] Goldberg, A., and A. Kay, editors. SMALLTALK-72 Instruction Manual. Xerox Palo Alto Research Center, SSL-76-6, 1976.

[13] HP-GSD. MPE Commands Reference Manual, Second Edition. Hewlett-Packard Company, General Systems Division, 1978.

References

[14] Hoare, C.A.R. "Monitors: an Operating System Structuring Concept," Communications of the ACM 17 (October 1974), pp. 549-557.

[15] HISI. Multics Programmers' Manual Reference Guide. Honeywell Information Systems, Incorporated, 1975.

[16] IBM. OS/VS1 JCL Services. International Business Machines GC24-5100-4, 1976.

[17] ------. OS/VS1 Utilities. International Business Machines GC26-3901-0, 1977.

[18] Ivie, E.L. "The Programmer's Workbench -- A Machine for Software Development," Communications of the ACM 20 (October 1977), pp. 746-753.

[19] Lampson, B.W., and H.E. Sturgis. "Reflections on an Operating System Design," Communications of the ACM 19 (May 1976), pp. 251-265.

[20] ------. "Crash Recovery in a Distributed Data Storage System," to be published in Communications of the ACM.

[21] Liskov, B.H., et al. "Abstraction Mechanisms in CLU," Communications of the ACM 20, (August 1977), pp. 564-576.

[22] Liskov, B.H., et al. "The CLU Reference Manual," Computation Structures Group Memo Number 161. M.I.T. Laboratory for Computer Science, July, 1978.

[23] Parnas, D.L. "On the Criteria to be Used in Decomposing Systems into Modules," Communications of the ACM 15 (December 1972), pp. 1053-1058.

[24] Reed, D.P. "Naming and Synchronization in a Decentralized Computer System," M.I.T. Laboratory for Computer Science Technical Report TR-205, 1978.

[25] ------, and R.K. Kanodia. "Synchronization with Eventcounts and Sequencers," Communications of the ACM 22 (February 1979), pp. 115-123.

[26] Ritchie, D.M., and K. Thompson. "The Unix Time-Sharing System," Communications of the ACM 17 (July 1974), pp. 365ff.

References

[27] Rochkind, M.J. "The Source Code Control System," IEEE Transactions on Software Engineering SE-1 (December 1975), pp. 364-370.

[28] Saltzer, J.H. "Topics in the Engineering of Information Systems." M.I.T. Department of Electrical Engineering and Computer Science, 1977.

[29] Schindler, G.E., Jr., editor. "Unix Time-Sharing System," The Bell System Technical Journal 57 (July-August 1978), part 2.

[30] Steele, G.L., Jr. "Multiprocessing Compactifying Garbage Collection," Communications of the ACM 18 (September 1975), pp. 495-508.

[31] Stern, J. "Backup and Recovery of On-Line Information in a Computer Utility." M.I.T. Project MAC Technical Report TR-116, January, 1974

[32] Thompson, K., and D.M. Ritchie. Unix Programmer's Manual. Bell Telephone Laboratories, 1975.

[33] Wadler, P.L. "Analysis of an Algorithm for Real Time Garbage Collection," Communications of the ACM 19 (September 1976), pp. 491-500.

[34] Wulf, W.A. "ALPHARD: Toward a Language to Support Structured Programs." Carnegie-Mellon University, Department of Computer Science, April 1974.

[35] ------, editor. "An Informal Definition of ALPHARD." Carnegie-Mellon University, Department of Computer Science, CMU-CS-78-105, February 1978.

[36] ------, R. Levin, and C. Pierson. "Overview of the Hydra Operating System Development," Proceedings of the Fifth Symposium on Operating System Principles, November 1975.

[37] Wulf, W.A., R.L. London, and M. Shaw. "Abstraction and Verification in ALPHARD: Introduction to Language and Methodology." Carnegie-Mellon University, Department of Computer Science, June 1976.