Acknowledgements

I would like to thank my supervisor Professor Pagurek for all of the help and encouragement he offered during the writing of this thesis.

Most of all, I would like to acknowledge the contribution of my wife Cathie. Without her steady love and encouragement, through what was a difficult period of my life, none of this would have been possible.

An Object-Oriented Database for a Computer Aided Software Engineering Environment

by

S. Michael Milinkovich, B. Comm.

A thesis submitted to the

Faculty of Graduate Studies and Research

in partial fulfillment of the requirements

for the degree of

Master of Science

Faculty of Engineering

Department of Systems and Computer Engineering

Carleton University

October, 1988

ISBN 0-315-51197-4

The undersigned hereby recommends to the Faculty of Graduate Studies and Research acceptance of the thesis "An Object-Oriented Database for a Computer Aided Software Engineering Environment" submitted by S. Michael Milinkovich, B. Comm., in partial fulfillment of the requirements for the degree of Master of Science

[illegible]

Professor B. Pagurek, Ph. D.

[illegible]

Chairman,
Department of Systems and Computer Engineering

Department of Systems and Computer Engineering

Faculty of Engineering

Carleton University

November, 1988

Abstract

Object-oriented approaches to the design and implementation of computer systems has been an active area of research during the 1980's. This thesis explores the application of this methodology to the design and implementation of a persistent object storage system to support a Computer-Aided Software Engineering (CASE) environment.

The system uses a commercially available database management system (ZIM) as the starting point for a persistent object manager for the programming language Objective-C. The system itself is implemented in Objective-C and some assembler.

Acknowledgements

I would like to thank my supervisor Professor Pagurek for all of the help and encouragement he offered during the writing of this thesis.

Most of all, I would like to acknowledge the contribution of my wife Cathie. Without her steady love and encouragement, through what was a difficult period of my life, none of this would have been possible.

Trademarks

Apollo is a trademark of Apollo, Inc.

DEC and **VAX** are trademarks of Digital Equipment Corporation.

GemStone and **OPAL** are trademarks of Servio Logic Corp.

Objective-C is a trademark of Stepstone Corp.

SUN and **SunView** are trademarks of SUN Microsystems, Inc.

UNIX is a trademark of AT&T

VM/CMS and **IBM** are trademarks of International Business Machines.

ZIM is a trademark of Zanthe Information Inc.

Table of Contents

List of Figures

1. Introduction

1.1. Motivation for the Research

As computer-based systems have become larger and more complex, much of the research in computer science and software engineering has gone into tools and methodologies to improve the manner in which they are specified, designed and implemented The goals of researchers in many seemingly different facets of software technology are often remarkably similar. A few examples of issues being addressed are: the complete and accurate specification of requirements and their communication to the implementers; the design of an effective solution to the problem at hand; the efficient use of both computer and human resources for the implementation, operation, and maintenance of the system; and the management of change during its entire life.

It has been argued persuasively (Brooks, 1987), that creating software systems to solve large and interesting problems is inherently complex. There is no undiscovered magic that will make the process suddenly faster or easier; any improvements will be in increments of percentages, rather than orders of magnitude. However, as computers become more pervasive in society and as they are called on to perform more challenging functions, software issues such as reliability, maintainability, and malleability are becoming more critical Two areas of research which address these concerns are software methodologies and tools

Software methodologies provide a consistent approach to the specification, design, and implementation of systems. They are the principles by which software engineers build their systems. One software methodology which has been the focus of much interest in the last decade is the 'object-oriented' approach. This paradigm has been used to design and build systems, to create new programming languages, and as a data modeling technique.

Software tools - commonly referred to as Computer-Aided Software Engineering (CASE) tools - attempt to automate as much of the software process as possible. Software is a labour intensive endeavour, and software engineers are expensive. As a result, there is on-going research into automating virtually every facet of the software process. Some examples include:

- project planning and control,
- change and configuration management,
- specification and design capture (using one or more methodologies),
- source code control and version release,
- performance analysis and testing,
- documentation.

While individual CASE tools are certainly useful, an even more important goal is the development of CASE *environments*, where multiple tools can work together in a seamless fashion. Perhaps the greatest motivation for CASE environments comes from the need to handle change. For example, keeping documentation up to date with changes in the source code is done best when there exists a tool which creates and maintains the dependencies between them. CASE environments are a promising technology for dealing with large, complex and multi-person software projects.

One of the major requirements for the creation of a CASE environment is that the tools supported be able to share data between them. As a number of researchers have noted (Wile and Allard, 1986), (Belkhatir and Estublier, 1986) (Gallo, Minot and Thomas, 1986) (Atwood, 1985), the best way to achieve this is through the use of an integrated database system.

1.2. The Application: The ARTTisan Toolset

The goal of this research is to provide a stable, persistent object store for a Computer-Aided Software Engineering (CASE) environment being developed at the Department of Systems and Computer Engineering under the auspices of the Advanced Real Time Toolset (ARTT) project. The environment is based on a network of SUN workstations running Berkley UNIX. While the ARTT project includes a number of different thrusts, this work primarily addresses the needs of the integrated toolset known as ARTTisan.

The following sections briefly describe the ARTTisan toolset and the design notations that it supports.

The Toolset

ARTTisan attempts to meet the design needs of a programmer developing real-time software. It is primarily targeted to addressing the needs of '*programming-in-the-small*', as it presently has none of the communication and co-ordination facilities (such as project planning and tracking, etc.) required to support teams of programmers working together on large software development efforts ('*programming-in-the-large*').

ARTTisan presents the developer with a hierarchy of notations, each of which plays a different role in the design process. Each layer in the hierarchy of notations represents a further refinement in the design. The initial layer is highly abstract, with each of the additional layers becoming more detailed. The design notations are tightly coupled, in that symbols in one of the notations may be related to symbols in one or more of the other layers. For example, a task in a process model may be further specified by a structure graph, and a procedure within the structure model may be further described by a state machine model. Each of these different notations has a separate *tool* associated with it, and the *toolset* is comprised of these

A major goal of the toolset is to have the different design tools *integrated*: Changes in one layer of the design should be automatically reflected in the other layers which are related. One way to accomplish this goal is to share a common, persistent data store. Previous generations of the ARTT toolset had used a filter-based approach to inter-tool communication. Data representing a design done in one of the tools would be massaged into the format expected by the next tool in the design process. This waterfall approach made it difficult to co-ordinate revisions between the different tools: filters, unfortunately, rarely work in reverse. Using a common, shared data store allows the toolset to deal directly with changes made at each level. Enhancing the data store to allow concurrent, multi-user access would also be a first step towards an environment which supports a more team-based approach to software development.

The user interface for the toolset is based on the graphical, direct manipulation paradigm. Icons representing the symbols of the design notations are manipulated by the designer using a pointing device (mouse) and keyboard. In its present form, the toolset is written entirely in Objective-C, using SunView to handle the graphical interface. Although the toolset itself is implemented using an object-oriented language, it should be noted that the design notations supported by the toolset do not support the object-oriented methodology.

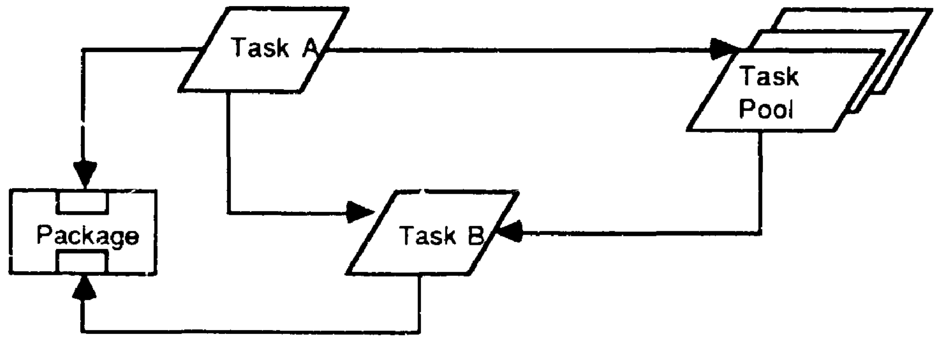
Tool integration is provided by highly inter-related, complex objects which implement the behaviour and maintain the relationships between the various toolset components. Figure 1.1 illustrates some of the symbols used in the various design notations, while Figure 1.2 shows some of the classes which implement the toolset. The basic approach is to treat each design notation as a directed graph, possibly with cycles. This allows a common approach to the implementation of each of the different notations. The nodes and arcs of the graphs have a specific meaning and representation within their tool, and (possibly) with the other tools in the toolset as well. Each node and arc is modelled using three complex objects: its display object, which defines how it is viewed graphically and manipulated by the user; its logical object, which implements its behaviour and maintains its relationships with the other objects in the toolset; and its tool component object which binds the display and logical components together.

Viewed in this way, each tool may be implemented as a collection of nodes and arcs of a certain type, along with methods which enforce the rules and constraints of the design notation.

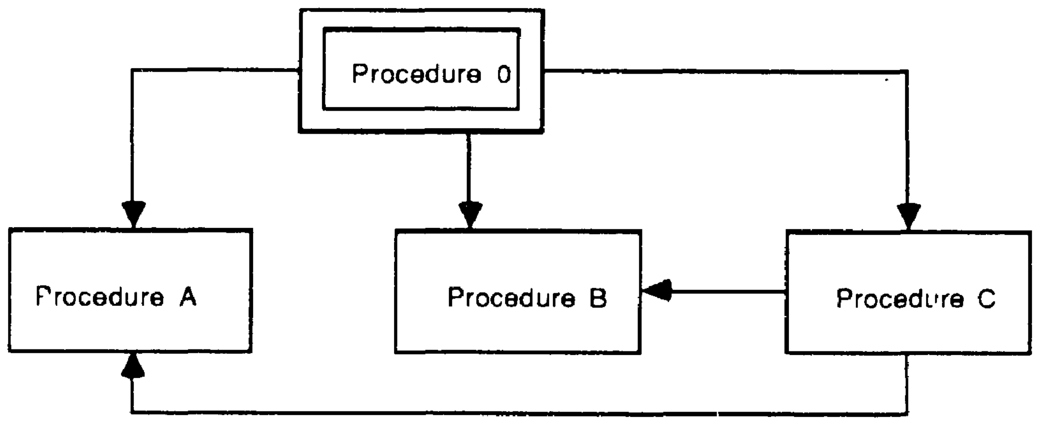
As can be seen in Figure 1.2, many of the objects are complex and highly structured, including a number which are comprised of collections of other objects. However, some of the objects in the toolset are also of the large, unstructured variety - such as graphics or text.

Figure 1.1: Design Notations

Layer 1: Task Diagram



Layer 2: Yourdon Procedure Diagram



Layer 3: State Machine Diagram

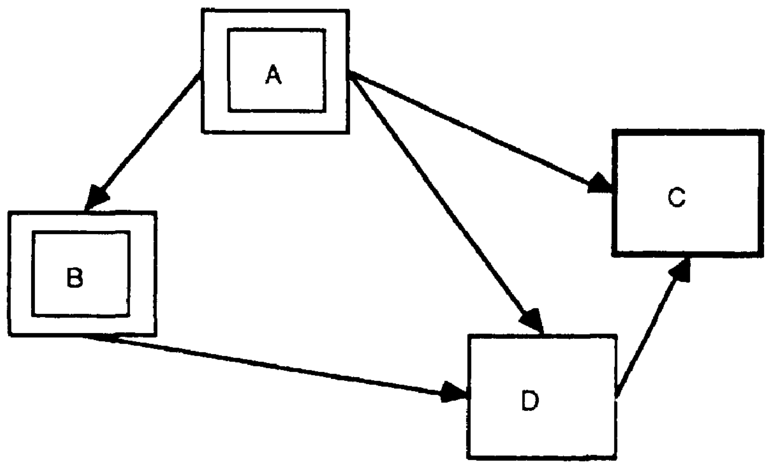


Figure 1.2

Notational Conventions:
- Each line represents an Objective-C class definition: the class name is first, followed by a list of instance variables, enclosed in ()'s
- The inheritance hierarchy is shown by the indentations
- Instance variables enclosed in {}'s represent collections

```
System (name, processModel)
Tool ({nodes}, {arcs}, startingNode)
  ProcessModel ({dataStores})
  StructureModel ({lcp}, {dataStores})
  StateMachineModel
ToolComponent (displayObject, logicalObject)
   Node
     ProcessNode
     StructureNode
     SMNode
     Arc
        ProcessArc
          ............
LogicalToolObject
      Node ({toArcs}, {fromArcs}, label)
         ProcessNode ({entries}, {lcps})
            Task (structureModel)
             TaskPool
            Package (structureModel)
            TaskGroup (processModel)
        DataStore
        LCP (task)
        StructureNode
            SMProc (stateMachineModel)
            Proc (fleshCode)
        SMNode
            States
            FSMEntry
            FSMReturn (returnValue)
            TaskEntry
           TaskEnd
            SMFCall (SMProcedure)
      Arc ({dependents}, toNode, fromNode, label)
           ProcessArc (toDataFlow, fromDataFlow)
              MessageArc ({ProcMsgArcs})
           StructureArc
            ProcedureCall
            ProcMsgArc
          SMArc
             StateTransition
```

1.3. Overview of the Research

This research has the following major goals:

- Provide a persistent object store for the ARTTisan toolset. This goal has a largely pragmatic motivation, since a persistent object store is required if the toolset is to be integrated.
- Evaluate the object-oriented approach to software engineering - both as a design methodology and as an implementation technique.
- Since the toolset is implemented using the Objective-C language, the object server must be consistent with the language and tightly integrated with it. Furthermore, since the toolset itself is being developed in parallel with the object server, the persistent object facilit; should be provided in a manner which is transparent to the toolset code.
- For the initial implementation, the object server is to be a single-user tool. However, the implementation must allow for future extensions to provide multi-user support.

These goals were met by the design and implementation of the persistent object server for Objective-C described in this thesis. The object server has been successfully integrated with a subset of the ARTTisan toolset. The major strength of this work are:

- It meets its primary goal of providing a persistent object store for the ARTTisan toolset. Objects may be stored and retrieved from the object base.
- The object server is transparent to the toolset code. For an example, persistent objects were provided to an existing subset of the toolset with only one line of code requiring modification. The toolset benefits from what is referred to as *locational transparency* - applications written using the object server do not have to know or care if an object is located in memory of disk at any point in time.

- The functionality provided by the object server has been provided with only minor changes to the Objective-C run-time kernel. No changes were required to the Objective-C compiler.

1.4. Outline of the Thesis

This thesis is organized as follows: Chapter Two introduces the reader to the basics of the object-oriented approach; and Chapter Three discusses the research into object-oriented databases. These two chapters may be omitted by a reader already familiar with these research areas. Chapter Four turns to the issues involved in implementing object-oriented database systems; Chapter Five discusses the design goals and issues for this work, and serves as an overview of the implementation; Chapter Six discusses the interface between the persistent object store and the Objective-C language; Chapter Seven presents the detailed design and implementation of a persistent object server for the Objective-C language; Chapter Eight presents the conclusions of the research and identifies areas for future research.

2. Object-Oriented Concepts

2.1. The Object-Oriented Approach

The basic notion of *objects* as they apply to computing is that they are abstractions of some entity, tangible or intangible, that is of interest to the designer. As such, they are a model of some thing that the designer wishes to simulate. One proposed definition of objects considers them to be "anything nameable" (Power and Weiss, 1988; p. 46). A more manageable definition of objects can be found in (Liskov, 1988).

Liskov's view of objects may be summarized by the equation:

Objects = Abstract Data Types + Inheritance

In this view, objects are extensions of data abstractions created by the software designer. They combine the properties of both procedures and data, since they perform computation and save local state (Stefik and Bobrow, 1984). The key idea is that:

> "Object-oriented programming is primarily a data abstraction technique, and much of its power derives from this. However, it elaborates this technique with the notion of "inheritance"....[used carefully] inheritance provides a useful addition to data abstraction." (Liskov, 1988; p.18)

While objects can be understood in terms of data abstractions and inheritance, that alone is not enough to fully understand the entire object-oriented approach to software development. The key additional concept is the notion of computation via message-passing. This computational metaphor is used in two of the most popular object-oriented programming languages, Smalltalk (Goldberg and Robson, 1983) and Objective-C (Cox, 1986)

The following sections describe data abstraction and inheritance, as they relate to the object-oriented approach. In addition, the message-passing computational metaphor is presented structure code found in programs written in more traditional languages. The testing of data types is now under the control of the procedure calling mechanism, rather than the programmer.

2.2. Object-Oriented Programming Languages

Smalltalk

Object-oriented programming finds its historical roots in the language Simula. However, it was with the Smalltalk-80 (Goldberg and Robson, 1983) system from Xerox's Palo Alto Research Center (PARC), that first brought the term 'object-oriented' to the attention of most software engineering researchers and practitioners. It was an important development in the computing field, and has had an enormous impact on the industry. The research that went into the Smalltalk system pioneered many concepts which are now considered standard: pointing devices, such as the now ubiquitous mouse, graphical user interfaces, and over-lapping windows all find their roots in the Smalltalk effort. It remains as perhaps the most consistently object-oriented programming language. As such, all of the characteristics of the object-oriented approach described in the previous section apply to the Smalltalk system.

Smalltalk is more than just a programming language. It is an entire environment built on the principles of the object-oriented methodology. There are four aspects of the Smalltalk environment (Rentsch, 1982): the programming language kernel, which comprises the Smalltalk language compiler and byte-code interpreter; a programming paradigm, which is the message-passing metaphor; a programming environment, which includes such tools as a debugger, editor and code browser, as well as a large class library; and the model-view-controller user interface model, which includes a number of classes for creating graphical end user facilities. It is important to note that these are not discrete units, they are inter-mingled and overlapping.

representation of the object within a set of operations. Users must use these operations to manipulate the object, rather than modifying the object's representation directly. This allows the separation of the object's behaviour from its implementation. Users of the object are restricted to knowing *what* operations an object will suffer, without knowing *how* they are performed. As a result, no object is dependent on the implementation details of other objects. This dramatically increases the robustness of the resulting system, since objects can be modified or re-implemented with minimal effect on other parts of the system. Encapsulation ensures that the code to perform a particular operation occurs only once - in the implementation of the type. This dramatically reduces code bulk, and simplifies change.

Software development using objects requires a fundamental shift in viewpoint. Objects are viewed entirely from the outside; as a result, when using an object, the developer is concerned with what it is, rather than how it is. Rentsch (Rentsch, 1982) notes that in Smalltalk, what is even more important is that the concept of getting 'inside' an object to examine or modify its state does not even exist in the language. Thus, objects are truly encapsulated in a purely object-oriented programming language.

Locality

One important side-effect of data abstraction and encapsulation is the property of *locality*. "Locality allows a program to be implemented, understood, or modified one module at a time." (Liskov, 1988; p. 20). Once an abstraction has been specified, its implementer and the implementers of other abstractions which use it, or are used by it, require minimal interaction. Everyone knows what to expect from the abstraction, which is the specified behaviour. This supports 'programming-in-the-large'. That is, it aids in the development of large, complex systems by teams of programmers. Developers can work on specific data types, without being concerned with the implementation strategies of others, since each can be understood in terms of its specification.

Locality also allows the system to be reasoned about (and hence, modified) one module at a time. The clear delineation of objects provided by locality allows the developer to concentrate on what must be done, rather than on what must be left alone.

Two key features which have become identified with the object-oriented approach result directly from the property of locality (Liskov, 1988). The first of these is fast prototyping, whereby a demonstratable system can be created quickly. This is due to the fact that objects can be initially implemented using simple representations and straightforward algorithms. As the system matures, the object implementations can become more sophisticated and robust.

A second, and closely related feature resulting from locality, is the support for program evolution. Requirements inevitably change over the life of a system; however, it is possible to structure the objects such that the effect of changes can be localized to their *class*.

Classes and Instances

In order for a programming language and/or environment to support data abstraction, there must exist facilities to add and use new data types. In Smalltalk and Objective-C, the primary tool for doing this is the *class*. Classes provide the designer with a tool to add new data types, to define their structure, and the operations that may be performed on them. A class is a description of one or more like objects, which are referred to as *instances* of the class. Structure is defined by specifying the instance and class variables of instances; behaviour is defined by specifying the methods that they can perform.

A class identifies the common properties of its instances. The operations that objects will perform are defined for the class, but only have an effect upon instances. The process of creating a new instance of a class is referred to as *instantiation*.

All objects are unique instances of some class; which gives rise to the notion that all objects have the property of *identity*. Every object has a (possibly system-defined) name which remains constant regardless of any changes in the object's state.

<u>Linguistic Support</u>

One of the greatest advantages in using object-oriented languages is that the data abstraction methodology is directly supported. There exists linguistic support for the definition of classes, for the binding of operations to data types, and for information hiding.

In Smalltalk, all data items are objects, including such primitive data abstractions as integers and characters. The result of making such primitives objects is a highly consistent programming environment: all items in the system are objects. Higher-level data abstractions (ie. user-defined objects) have exactly the same rights and privileges as system-provided ones. Furthermore, since all of the features of the language and its environment are implemented using objects, the usual dichotomy of system or language functions versus user-defined functions does not apply

<u>Composite Objects</u>

Once objects are the sole tool of abstraction, it is natural to view objects as being aggregates of other objects. Thus a composite object must deal with its subcomponents using the operations which they, in turn, have provided. Eventually, these layers boil down to the system-implemented objects. This hierarchy of abstract data types is referred to as the aggregation/decomposition hierarchy. In the Artificial Intelligence area, this is also referred to as the part-of relationship.

Two important side-effects of viewing complex objects as aggregations of other objects are:

- An object may be a sub-component of several objects simultaneously This structure-sharing is a powerful tool for modelling real-world entities, since different objects can share the concepts that they have in common. It should be noted that structure-sharing is greatly facilitated by the identity property of objects, since the references to objects are based on the identity of the object itself, rather than the value of some attribute.
- The subcomponents of an object may be other objects of arbitrary complexity. This is very different from the traditional approach where the attributes of a user-defined data type must be one of a small set of system-supplied data types (ie. integer, character). It is a powerful abstraction technique, one which allows the designer to easily and naturally layer both the design and the implementation of the objects in the system.

Objects which form collections of other objects are a special case of composite object, since their existence forms a relationship between their member objects. Objects which are *members-of the same collection are said to share an association* relationship.

Inheritance

Every class is a specialization of some other class, referred to as its *superclass* Each class, then, is said to *inherit* the structure and behaviour of its superclass This forms what is generally referred to as the class hierarchy. At the root of this hierarchy is the class Object, which defines the properties common to all objects in the system. New classes are defined as specializations of existing classes. This abstraction technique allows new concepts to be added quickly to the system as refinements to existing ones. It allows programmers to work in an incremental fashion. Since a class inherits the behaviour of its superclasses, new code must be written only for those aspects that make it unique, thus decreasing the amount of redundant information.

Different object-oriented languages support either single or multiple inheritance. In the former, a new class inherits its properties from a single super-class. In the latter, a new class may be defined by mixing the structure and behaviour of several classes. Thus, under single inheritance, classes are organized into a tree structure, while in multiple inheritance, classes are organized into a directed graph (often called a *class lattice*).

Inheritance, as described here, is one implementation of the data abstraction technique known as specialization/generalization. Generalization is the abstraction by which the common properties of several different classes are represented as a generic class. The constituent classes are said to be specializations of the generic class and inherit its properties. This abstraction establishes an is-a relationship between the objects. For example, Managers and Secretaries may be considered as specializations of Employees and would inherit such attributes as Name, Address and Salary from the higher level abstraction.

Message Passing

Under the object-oriented approach, computation is performed by the sending of messages between objects. Processing activity takes place within the object itself, but only after it has been initiated by the receipt of a message. A message is a request for an object to perform some operation, the meaning of which is dependent upon the class of the object that receives it. Different classes will perform different actions given the same message. What is done upon the receipt of a message is based on the method (procedure) which the class implements as a response to that message. Resolution of what action will be performed, based on a given message, is typically performed at run-time.

For most, programming in an object-oriented language will initially seem quite different. All state is captured within objects, and the only way to accomplish computation is to use the messages to which the object will respond. Programs are created by passing messages to seemingly intelligent objects which perform some computation; they, in turn send their message requests to other objects to extract data or to have some additional computation performed. While proven effective, this paradigm is quite different from the more traditional procedural approach.

There exists a subtle difference between the semantics of a procedure call and a message request (Rentsch, 1982). Under the traditional imperative approach, there exists an assumption that the calling procedure is somehow 'in control' of the called procedure. Conversely, a message is more accurately viewed as a request by the sender. The receiver object is totally responsible for the interpretation of the message and for 'doing the right thing'; hopefully, meeting the wishes of the sending object. Control is relinquished to the receiving object both conceptually and actually.

Bindings, Protocols and Polymorphism

All objects are instances of some class, and in this sense, all object-oriented languages are strongly typed. However, in both Smalltalk and Objective-C, the objects themselves are typed rather than the slots which hold the objects. For example, in most languages, when a variable is declared, its type is immediately disclosed to the compiler. In Smalltalk, when a variable is declared, it is known only to be an object. As a result, the type of an object is unknown until run-time. Other object-oriented languages, such as C++, follow the more conventional approach and support the compile-time binding of objects to types.

The methods which implement operations specified by messages are bound to the object classes, rather than to some global name space. As a result, different object classes may have functions (methods) with the same name. Since the type of an object is typically bound at run-time (as in Smalltalk and Objective-C), which method a message is calling can only be established then. This is generally referred to as the *late-binding* of messages.

Message sends are quite different from conventional function or procedure calls. In the latter, the name of the function is bound to a location in memory which is named in the program's symbol table, and the location is specified at compile-time. A function call is made up of the name of the function, and the parameters being passed to it. When the function is called, the parameters are placed on the stack, and control is branched to the specified memory location. Since the symbol table is typically global, function names must be unique. Methods, on the other hand, do not have unique names. It requires a <class, message selector> pair to uniquely identify the memory location where the method implementation resides. The *message selector* is a character string which names the method. A message send is made up of the object which is receiving the message (the *receiver*), the message selector which identifies the operation desired, and the parameters being passed to the method. Finding the address which implements the desired operation requires a table lookup.

One interesting side effect of late-binding is that it simplifies the creation of an incremental compiler for a language that implements it. Since all procedure addresses are resolved at run-time, a method may be re-compiled without impacting all of its callers (Duff, 1986). This furthers the goals of encapsulation and locality. On the other hand, late binding allows some errors in the code to remain undetected until run-time.

Late binding has some obvious performance penalties. Since the address of a function must be looked up at run-time, every procedure call implies some sort of searching. This is more expensive than simply branching to a pre-determined address, as is the case with traditional languages. This performance penalty is compounded by the fact that the message-passing paradigm results in a programming style which calls for a great many message sends. Lately, there has been a great deal of interest in performing more compile-time bindings. For example, in C++ (Stroustup), objects are strongly typed and as a result, messages may be resolved at compile time.

As a result of these binding mechanisms (ie. the static binding of methods to object classes and the run-time or dynamic binding of message sends to these methods), different object classes can share a *message protocol.* A protocol is a standardized set of messages used to implement a desired functionality. Two classes which implement the same set of messages are said to follow the same protocol.

> "There is additional leverage for building systems when the protocols are *standardized.* This leverage comes from *polymorphism.* In general the term polymorphism means "having or assuming different forms," but in the context of object-oriented programming, it refers to the capability for different classes of objects to respond to exactly the same protocols. Protocols enable a program to treat uniformly objects that arise from different classes. Protocols extend the notion of *modularity* (reusable and modifiable pieces as enabled by data-abstracted subroutines) to *polymorhpism* (interchangeable pieces as enabled by message sending)." (Stefik and Bobrow, 1984; p. 41)

Polymorphism then, allows different classes to perform similar operations, despite otherwise different representations. This is a powerful technique for the programmer to treat different classes in a uniform manner. Furthermore, it negates the need for much of the control structure code found in programs written in more traditional languages. The testing of data types is now under the control of the procedure calling mechanism, rather than the programmer.

2.2. Object-Oriented Programming Languages

Smalltalk

Object-oriented programming finds its historical roots in the language Simula. However, it was with the Smalltalk-80 (Goldberg and Robson, 1983) system from Xerox's Palo Alto Research Center (PARC), that first brought the term 'object-oriented' to the attention of most software engineering researchers and practitioners. It was an important development in the computing field, and has had an enormous impact on the industry. The research that went into the Smalltalk system pioneered many concepts which are now considered standard: pointing devices, such as the now ubiquitous mouse, graphical user interfaces, and over-lapping windows all find their roots in the Smalltalk effort. It remains as perhaps the most consistently object-oriented programming language. As such, all of the characteristics of the object-oriented approach described in the previous section apply to the Smalltalk system.

Smalltalk is more than just a programming language. It is an entire environment built on the principles of the object-oriented methodology. There are four aspects of the Smalltalk environment (Rentsch, 1982): the programming language kernel, which comprises the Smalltalk language compiler and byte-code interpreter; a programming paradigm, which is the message-passing metaphor; a programming environment, which includes such tools as a debugger, editor and code browser, as well as a large class library; and the model-view-controller user interface model, which includes a number of classes for creating graphical end user facilities. It is important to note that these are not discrete units, they are inter-mingled and overlapping.

Smalltalk is a 'pure' object-oriented language. As such, it is typified by the following characteristics (Rentsch, 1982):

- **uniformity of abstraction:** All items in the Smalltalk system are objects, running the gamut from those usually considered to be primitives - such as integers and characters - to such esoteric data types as ProcessScheduler, SortedCollection and BitBlt. Even more interesting, the language itself is implemented using objects. Therefore, such things as contexts, classes and messages are also objects which may be manipulated. This uniformity makes it easier to implement tools such as debuggers to support the language and browsers to support additions and modifications to the class hierarchy. Since everything is an object, there is no distinction between system supplied data types and those created by the programmer. There are no 'second-class citizens'. As a result, it is straightforward to implement new classes which are specializations or aggregations of the classes supplied with the Smalltalk image. This results in much of the time-savings typical of developing systems with Smalltalk.
- **uniformity of computational metaphor:** Smalltalk uses the message-passing metaphor. All processing results from the sending of messages between objects. This is applied consistently, to the point where arithmetic operators between integers are considered messages.
- **uniformity of reference:** In Smalltalk, the only way to reference an object is from the outside, using the messages to which it will respond to. There is no way to 'peek' inside an object to view or update its underlying data structures. At the same time, all variables use associative access; variable names refer to pointers, rather than to objects directly.

The consistent application of this approach means thai the 'dangling pointer' class of software bugs no longer becomes an issue.

The following is a list of interesting points and implementation details concerning the Smalltalk language. The reader should note that many of the details presented here will be referred to later in the research.

- The language is implemented using a byte-code interpreter. Programs are compiled to an intermediate language (as opposed to the machine language native to the hardware) comprised of single byte instruction codes. These instructions are then executed by a 'virtual machine'. This approach results in a more efficient interpreter.
- Garbage collection is used to free the programmer from the burdens of memory management. This is considered a key feature in terms of the ease of applications development and systems reliability, although it can obviously have an impact on execution performance. It should be noted, however, that newer Smalltalk implementations use a scavenging garbage collection technique (described in Duff, 1986) which significantly reduces the performance penalty associated with garbage collection.
- As is implied by the use of garbage collection, Smalltalk manages its own memory. One key detail to note is that pointers to objects are not simple pointers. All references to the same object are via a single 'object-oriented pointer' (OOP). As a result, it is possible to replace an object with an entirely new one and have all external references to it automatically updated. This is performed using a *become* message.

- Objects can send messages to themselves by referring to one of two special variables known as *self* and *super.* Messages to self call the method implemented by the class of the object, or its superclasses, if no such method is defined.

  Messages to super cause the message look up routine to start with the object's superclass. This allows behaviour to be specialized with a minimum of new code. For example, one common usage of super is when a subclass must perform some special processing to implement a message previously defined by one of its superclasses. ,ften, the best way to handle this is to implement the specialized behaviour in a new method, which then calls the superclass's routine to perform the more general process. This is done by sending a message to super.

- All of the source code above the virtual machine layer is available to be viewed and modified. This makes the system very malleable. A programmer can not only create the classes required for his own application, he can also modify all of the system classes provided with Smalltalk, if required.

- While the availability of source code is one of the benefits of Smalltalk, it also poses a problem. It is difficult to partition the Smalltalk environment in order to deliver a finished application. Imagine, if you will, the dangers involved in delivering an end-user application - say a banking system - where the tellers can view and modify not only the code that makes up their application, but the code for the compiler, graphics, disk IO, etc.. Furthermore, it would be more space efficient if the classes that were not used by an application could be excluded from the image delivered to the users.

- Object persistence is handled by saving the heap to disk or other storage device. This makes it difficult to allow data to be shared between multiple users of an application.

Objective-C

Objective-C (Cox) is a more recent language than Smalltalk. It is a hybrid language, in that it combines the efficiency and portability of the well-known C language with much of the expressive power of Smalltalk. Objective-C message sends can be mingled with ordinary C statements in the code; as well, ordinary C data structures can be mingled with objects. The language is implemented as a pre-processor to the C compiler, along with a run-time kernel written in assembler and Objective-C. It comes with an extensive class library (although not as large as Smalltalk's) which may be used to develop applications.

Objective-C supports the key concepts of the object-oriented methodology, such as classes, message-passing, polymorphism and inheritance. However, since it is based on C, it lacks much of the consistency that is the hallmark of Smalltalk. For example, the C-implemented data types such as pointers, integers, etc., are not objects, as they would be in Smalltalk. *Objects are added to C via defining one new data type, referred to as an id.* Id's are *first class* types in Objective-C, in that any operation that can be applied to a character pointer in C, can be applied to an id (with, perhaps, some appropriate casting). Objects themselves are essentially C structures.

The following is a list of interesting points and implementation details concerning the Objective-C language.

- The language is compiled, rather than interpreted. However, in order to handle the late binding inherent in message passing, there is a run-time kernel. Messages are compiled to an ordinary C function call to the assembler function *msg()*, with the object, its class, the message selector, and the parameters to the message passed as parameters. *Msg()* looks up the function which implements the correct method and branches to it, leaving the method parameters on the stack. For added efficiency, the most recently used methods are cached.

- There is no garbage collection, so memory management is left in the hands of the developer. The memory required for objects is allocated from the heap using the an Objective-C protocol which ultimately access the regular C memory routines such as *alloc()* and *free()*. As a result, dangling pointer bugs can occur when objects are deallocated.
- References to objects are ordinary C pointers, not OOP's, as in Smalltalk. This means that multiple references to objects cannot be updated in a straightforward manner. This is an additional source of dangling pointer bugs.
- Objects may be saved to disk. Complex objects will result in the saving of all objects reachable from the root.

3. Object-Oriented Database Management Systems

3.1. Introduction to Database Management Systems

Database management systems (DBMS) are a well known topic in computer science and engineering. Their importance can be seen by the fact that they exist for virtually all types of computers, large and small, and by the amount of research interest dedicated to them. Databases are distinct from most programming environments in that they deal with *persistent* data. Persistence means that the data is stored in some non-volatile storage media (typically disk), in contrast with other environments where all data structures are in memory. Once data is persistent, it is possible to separate it from individual applications. This gives rise to an important side-effect of persistence: it allows the *sharing* of data between applications and users. Furthermore, as will be discussed further below, this sharing can be sequential or concurrent.

Not only are DBMS's concerned with the storage of shared, persistent data, they are designed to efficiently access large quantities of data. While any operating system will deal with persistent data through its file system, few do so effectively once there is a large amount of data. The difference is that databases typically provide some mechanism for the direct access and updating of data stored in files.

In addition to the efficient access of large amounts of persistent data, databases have a number of common features. These include the support of data models and data abstraction, language support, access control and resiliency.

Data Models and Abstractions

Database management systems will provide support for at least one *data model*, which provides a mathematical abstraction with which to view the data. For example, the relational data model provides a set-theoretic view of data, with a well understood set of mathematical operations to manipulate that data; the network model is based on directed graphs.

Date (Date, 1983) defines data models to consist of the following three components:

- *a collection of object types;*
- a collection of operators;
- a collection of general integrity rules.

The object types provide the basic units of the data model; databases built from a particular data model will consist of objects of strictly those types. The operators provide a mechanism by which those objects may be accessed and manipulated. The integrity rules are a set of general constraints to which databases using the data model must conform; for example, relational tables must be normalized.

In addition to a data model, modern database systems will generally support some notion of *abstraction levels*. That is, a number of layers of abstraction between the bytes being written to the disk and the information being manipulated by the users. The most common model of abstraction levels has three layers: physical, conceptual and view. The *physical database* level is concerned with the files and indices that actually make up the storage structures of the database The *conceptual database* level defines the structures of the physical level such that they are meaningful in some way to the users of the system. In order to do this, DBMS's supply a data definition language (DDL), which allows the definition of the data in terms of some data model, and some degree of control over the implementation provided by the physical level. It is also common for the conceptual schema to be first defined in terms of some sort of higher level abstraction technique (such as a semantic data model, described below), and then translated to the data model supported by the target DBMS. The *view* (or subscheme) level provides a mechanism to partition the conceptual database into views which are meaningful to a particular user group. For example, the personnel department and the telephone operators would want very different views of an employees database. Views allow the database designer to create objects which are more complex than those defined in the conceptual schema.

Levels of abstraction give rise to another commonly referred to concept in databases, *data independence*. There are two types of data independence: physical and logical. Physical independence implies that the physical schema may be modified with minimum impact on the conceptual or view levels. This results from the fact that applications which access the database do so via the DBMS; this level of indirection allows the underlying file structures of the database to be modified without resulting in program changes. Logical independence implies, in a similar fashion, that changes may be made to the conceptual schema with minimal impact on the subschema level.

Language Support

High-level language support for data definition, manipulation and access are typically provided by database systems. In different systems, each of these three facilities may be provided by a separate language, by one language or by some combination thereof. Furthermore, some database systems are intended to be used with a host language such as COBOL, PL/1 or C.

(Ullman, 1988) claims that one of the key developments in modern database technology is the integration of the host and data manipulation languages. In other words, negating the need for some host language to write the application in. For example, even such recent database access languages such as SQL require a host language like COBOL to provide the flow of control logic needed to create a complete system. SQL statements embedded in the program interact with the database, while the host language provides the control structures, interacts with the user and performs many of the data manipulations in program memory. The key problem with using separate languages is that *impedance mismatch* may result (Zaniolo, et al). This occurs when the host language and database language are based on different notions of computation (ie. an imperative language mixed with a declarative query language) or have different typing systems.

Security and Integrity

Access control, security and data integrity facilities are also common in database systems. The first two are concerned with controlling access to the data to ensure that it is not corrupted by unintended or malicious use. Note that the view level of data abstraction is one commonly used technique to provide access control; users are restricted to manipulate only those views for which they are authorized. Data integrity facilities ensure that the values being stored are reasonable. For example, a field to store a person's age would not be expected to fall outside the range of 0 to 150.

Concurrency and Transactions

Concurrent access to the data by multiple users, in either a time-sharing or truly distributed environment, is one of the most important functions of database management systems. In fact, the efficient sharing of data between multiple users is one of the main motivations for using DBMS's to store and manipulate data, as opposed to ordinary file systems Facilities to support multi-user access are typically implemented in terms of locking protocols, schedulers and transaction managers. However they are implemented, concurrency schemes are intended to ensure that concurrent transactions performed by the users are *serialized*. That is, that they appear to have been performed in some sequential fashion, despite having been interleaved wherever possible. Serialization of transactions is critical because it ensures that each user of the database deals with a consistent view of the data. Without serialization, it is trivial to construct examples of multi-user database transactions which have not been serialized whose actions interfere with each other in such a way as to leave the database in an inconsistent state.

Transactions are user operations on the database which are to be applied *atomically;* either all of the operations within the body of a transaction will be committed to the database, or none will. Concurrency schemes common in today's DBMS's typically assume each transaction to be quite short - a second or two, at most, and more normally some fraction of a second. As a result, it is generally assumed that a transaction can be suspended momentarily (while waiting for a lock to be freed, for example) in order to ensure that serialization has been met.

Database locking is done to ensure that only one user may have access to a database entity at any given moment. One of the most commonly used locking protocols is known as *two-phase* locking. Date defines this protocol in terms of the following theorem (Date, 1983; p. 102):

"If all transactions obey the following rules:

one: before operating on any object the transaction first acquires a lock on that object; and
two: after releasing a lock the transaction never acquires any more locks;

then all interleaved executions of those transactions are serializable."

All of the locks required by a transactions are done before any items are unlocked; once the first item is unlocked, no more locks may be acquired by the transaction.

Since transactions may be made to wait until other transactions release portions of the database, the DBMS is also responsible for the detection and resolution of transaction *deadlock.* Deadlock occurs when two or more transactions are suspended while each is waiting for the other to release data. For example, transaction A holds a lock on record X and is waiting for transaction B to release its lock on record Y. Simultaneously, transaction B is waiting for A to release its lock on record X. This situation could remain unresolved forever. It is the database equivalent of the infinite loop.

In summary, database transactions have the following characteristics:

Duration: Database transactions are of short duration. Typically, they are designed so that they last for a few seconds at most, to minimize the possibility that one user is forced to wait for data being accessed by another. For example, interaction with the user is usually not contained within the bounds of a transaction. This is to ensure that the data is not locked while the user is viewing it on the screen. For example, a program which interacts with both the user and the database could have the following form:

```
user interaction 1
start transaction
        database update 1
        if error then
                abort transaction
                :
                :
        database update n
        if error then
                abort transaction
commit transaction
user interaction 2
        :
        :
```

Consistency: Database transactions are the primary method of maintaining database consistency. Implicit in this statement is the assumption that any transaction that commits will leave the database in a consistent state.

Volume: Database transactions touch fairly small amounts of data (or at least only a few tables). Large reports, data loads and bulk changes are typically done off-line to minimize their impact on user response-time.

Locking: If a record that the transaction wants to access is locked by another user, the transaction is typically put into a wait state until that lock is released; multiple transactions waiting on a record are queued. Note that this implementation strategy is based on the assumption that transactions are short in duration (otherwise users would be waiting for an unacceptable length of time).

Concurrency: In a typical database application, there are many users and hence, may transactions executing concurrently. There is a high probability that the many concurrent transactions may result in deadlocks occurring. Therefore, deadlocks should be detected and resolved automatically by the DBMS - usually by causing one of the offending transactions to abort.

Much of the function of a database management system therefore, is to provide some support for the correct and efficient access to the stored data in a multi-user environment.

Resiliency

Resiliency in the face of some unexpected event, such as abnormal program termination or media failure is also one of the tasks of a DBMS. Databases are expected to ensure that their integrity is maintained at all times, and when that is impossible, to provide backup and recovery mechanisms to return to a consistent state. It should be apparent that transactions play an important role in the ensuring the integrity of the database, since they are guaranteed to either commit or fail atomically.

3.2. The Motivation for Object-Oriented Databases

Before describing what is meant by 'object-oriented database management systems', let us first address the issue of why they are interesting. In other words, what is unsatisfactory about the present database technology that is driving researchers to look in new directions.

Why Relations Are Not Enough

The present state-of-the-art in database technology are those systems which support the relational data model. Relational DBMS's have been well-accepted in the traditional application domains of databases, such as business and accounting. There are a number of commercially available relational database systems. Although these systems may differ in implementation details, etc., they all share the same view of data. Recall that a data model consists of a set of object types, a set of operations on those types and a set of general constraints. Given this definition, the relational model may be quickly summarized as (Ullman, 1988), (Date, 1983):

- All objects are *tuples*, or members of a *relation*. Relations are a concept from set theory and can be defined as a "subset of the Cartesian product of a list of domains.....a domain is simply a set of values, not unlike a data type." (Ullman, 1988; p. 43) Relations may be viewed intuitively as tables, where the rows are the tuples (instances) and the columns are the attributes (values drawn from the domains of the relation) of those tuples.
- There is a fixed number of operations on relations, based on either the relational calculus or the relational algebra, which are logically equivalent. For example, relational algebra includes the operations of union, difference, selection, projection, and Cartesian product. The result of any of these operations on a relation is another relation, which then may be used as an operand to another operation. These operators have the property of orthogonality; that is, none may be expressed as a formula consisting of the

others. This, in turn, leads to the conciseness of languages based on the relational model.

- Relations, which form the conceptual schema, are constrained to be *normalized.* Normalization provides an algorithmic way to construct relations from the objects to be modelled by the database, such that each entity may be uniquely identified by a *primary key*; further, normalization avoids redundancy of data, which can lead to update anomalies. The relational model is *value-oriented* in that it is the attributes of a tuple which uniquely identify it, rather than some identity property.

Relational DBMS's also typically provide some mechanism to construct views. Views provide a mechanism to model complex objects which are not necessarily fully normalized.

> "A view is defined in a relational model as a query over the base relations, and perhaps also over other views. Current implementations do not materialize views, but transform user operations on views into operations over the base data." (Wieuerhold, 1986; p. 37)

Care must be taken when manipulating database views. Although they are themselves relations, they are not necessarily normalized. As a result, operations performed on them may result in update anomalies.

The relational model has a number of important strengths, as noted by (Ullman, 1988), (Tsichritzis and Lochovsky, 1982), (Date, 1983) and others.

- It provides a simple and intuitive tabular view of data. This is particularly useful in applications, such as accounting, where tne data naturally fits this mold.
- As noted above, the results of operations on relations are, in turn, relations.

- It supports concise, yet powerful, declarative languages for operating on data. This is partly due to the previous point, since operations can be nested easily, and also the orthogonality of the relational operators.
- It supports the notions of data abstraction and data independence.

If relational DBMS's are performing satisfactorily in many application areas, why is there a need for new database technologies? First and foremost, databases are being called on to support new and very different applications in such areas as Computer Aided Design, Computer Aided Manufacturing (CAD/CAM), Office Automation, Artificial Intelligence and Computer Aided Software Engineering (CASE). These applications are characterized as having:

- large amounts of data of diverse types, including both large, unstructured objects such as graphics and text, and complex, highly structured objects such as CAD designs;
- complex end-user interface requirements; and
- a high rate of change in the application.

Many of these applications are being modelled and understood while the software that implements them is being developed. Contrast this with the more traditional application domains of computers, where the systems being developed have often been understood and performed manually for many years. As a result, the features of object-oriented programming such as rapid prototyping, localization of change and inheritance make it better suited to these domains.

A number of researchers (Rumbaugh, 1987), (Zaniolo, et al), have commented on the general unsuitability of relational database technology to deal with these new application areas. Although the relational model is an elegant, mathematically-based data abstraction, it provides only one level of structure (the relation), and is poor at representing both highly structured, complex objects and unstructured objects. This is largely the result of the following factors.

- Complex objects must be represented by tuples spread over a number of relations, rather than by some direct representation. These tuples must be joined using relational operators to form the entire object. As a result, much of the representation of complex objects resides in the application code, rather than in the schema; further, it is difficult to manipulate complex objects as single entities. While the view mechanism, supported by many relational DBMS's provides a way to represent these objects in a more natural manner, the updating of views may result in update anomalies.
- The model is restricted to a small set of primitive data types - typically integers, characters and dates - and there is no way to extend this set.
- The lack of the notion of object identity often forces the creation of arbitrary key values in order to invent uniqueness. The inability of relational databases to handle data structures which are recursive to some arbitrary depth stems from the lack of object identity. Identity is also required for the structure sharing required for many knowledge-intensive applications. Structure sharing supports *referential transparency;* that is, any change in an object is automatically made available to all of the objects that refer to it. This is not the case with relational databases, where a change in a tuple's key is not reflected in the entities that share it (Zaniolo, et al). Rumbaugh (Rumbaugh, 1987) attributes many of the problems of normalization (ie. concern about update anomalies, referential integrity and redundant data) to the value-orientation of the model, and the refusal to admit the property of identity.

- The relational model's reliance on the record construct and a tabular view of data makes it unsuitable for representing unstructured objects.

Ullman (Ullman, 1988) argues that the value-orientation of the relational approach is, in fact, one of its strengths. He bases this argument on the unsupported claim that it is impossible to define a declarative language for an identity-based data model. However, this argument is ultimately counter-intuitive; identity is one of the key concepts in modelling the real world. For example, picture two brand new basketballs. They are the same in every way - colour, shape, manufacturer, etc.. But as even a child can tell you, they are *different* basketballs. In other words, they have the property of identity.

It should be noted that there have been a number of extensions to the relational model which have been proposed by researchers which address many of the issues raised above. (For example, the RM/T model, as described in (Date, 1983)).

Database Issues Addressed by the Object-Oriented Approach

From the previous section, it can be seen that the object-oriented approach promises a better approach to modelling complex real-world entities than do conventional methodologies. The following describe a number of other issues in database research and how they could be addressed by object-oriented database management systems.

Conceptual Modelling

There has been a great deal of interest by database researchers in ways to improve the methodology by which the conceptual schemas are designed. The goal is to capture more of the semantics of the problem domain in the schema. Traditional data modelling techniques capture only the *static* (structural) semantics of the application. This limits their usefulness in specification and design since only part of the problem can be adequately described, as the *behaviour* of the system is not addressed. This research area is broadly referred to as *conceptual modelling.*

The modelling techniques which have come out of this area are known as *semantic data models*. To a large degree, this area may be viewed as an attempt to combine advances in the areas of databases, programming languages and artificial intelligence. It must be stressed that conceptual modelling is not an implementation technique. Rather, it is a toolbox of concepts, notations and abstraction techniques which improve the specification and design of database schemas and transactions (ie. both structure and behaviour). Generally, it is assumed that the actual implementation of the database will be done using a conventional relational DBMS.

One of the ideas advanced by conceptual modelling is that much of the semantics of an application can be described using a number of *abstraction hierarchies*. The classification-instantiation hierarchy is common to virtually all database systems and languages. For example, this abstraction exists in relational systems since a relation defines the contents and meanings of the tuples actually stored in the database. This abstraction allows the grouping of like objects into classes; and an instance-of relationship is said to exist between the objects and their class (ie. {123, 'Fred'} could be an instance-of the relation Employees). The specialization-generalization hierarchy supports the notion that the common properties of several different classes may be represented by a generic class. The constituent classes are said to be specializations of the generic class and inherit its properties. This abstraction establishes an is-a relationship between the objects (ie. Managers and Secretaries could be specializations of Employees). The aggregation-decomposition hierarchy forms a part-of relationship between between objects which have been aggregated into some higher level object (ie. Name and Address could form part-of Employees). The association abstraction groups objects into collections. Objects are related by association if they may be viewed as being part of a higher level collection object. This establishes a member-of relationship between the collection object and its members.

Conceptual modelling also brought the notion of encapsulation to database design. In other words, the behaviour of the objects being manipulated by the database should also be somehow captured by the schema; and further, that it is easier to maintain the integrity of those objects if all access to them was via operations (transactions in database parlance) that they defined and implemented.

Conceptual modelling, therefore, introduced the object-oriented design methodology to the database community. Of course, once these design techniques became generally accepted, the next step was to provide object-oriented DBMS's for their implementation.

Views

Views were described briefly above, however, since they provide a mechanism which has been proposed as a method for the representatio... of complex objects by relational databases (Wiederhold, 1986), they warrant further discussion.

Views provide an abstraction whereby information on the database is tailored to meet the needs of a single class of users. There exists two types of views: partitions and aggregations.

Partitions: This class of views restrict and (perhaps) modify the database schema to meet the needs of a certain class of users. Partitions may restrict which tables and/or which fields a user may see in the database. There may also exist facilities to allow the renaming and reordering of fields within tables. Partition views are often used to control access to the database.

Aggregations: This type of view is often describea as a 'stored join'. The goal of aggregation views is to create a complex object from the relations in the database that more closely follows the users' conceptual view of the data. For example, in a CAD database, a printed circuit board may be represented in a number of different normalized relations; however, the user will want to view and manipulate the complex object as a whole. Note that although the relations that make up a view of this type will be normalized, there is no restriction that the view itself be so.

Aggregation views are generally considered a complex entity with poorly understood semantics in the database literature. However, views are really just complex objects. As such, their implementation would be a natural part of a OODBMS.

One of the reasons why views are considered complex is that the semantics of updates to them in terms of their underlying implementation is not always clear. The maintenance of database integrity in the face of view updates is also considered an issue. Furtado and Casanova (Furtado and Casanova, 1985) note that there are two basic approaches to handling the view update problem. The first is to treat the view as an abstract data type, including a definition of the allowable operations. In other words, treat the view as a complex object. The second is to define generalized procedures which take a view, the desired update and the current database state as operands and attempt to construct a satisfactory update to the underlying database. They note that treating views as ADT's has the following advantages:

- Certain updates are no longer ambiguous when they are not treated as straightforward applications of single tuple insertion, deletion or replacement.
- Ambiguity is avoided since the developer can make arbitrary decisions in the implementing the view operations.

- Specific view manipulation routines will be more user-friendly than the use of some general-purpose view update language.
- Constraints are automatically enforced if view operations are restricted to those implemented for the view abstract data type.

In terms of the object-oriented approach to views, the above holds, with the following extensions.

- Since the components of a complex object (view) are themselves objects, the view is not responsible for ensuring their integrity. As objects, they are also capable of intelligent behaviour and are ultimately responsible for maintaining their own constraints. This makes it more straightforward to implement view abstractions using an object-oriented approach.
- Inheritance is an integral part of the object-oriented paradigm. Therefore, it would be straightforward to define specializations of views. These would meet the need for partition views described above, especially if multiple inheritance is allows by the system

Objects, therefore, are the most suitable implementation vehicle for modelling complex entities.

Constraints

More advanced database systems supply facilities for the maintenance of *constraints* defined over the database. Constraints put bounds on what behaviour is acceptable for a certain class of entities or attributes of entities; they also put restrictions on the set of allowable states that the database may be in.

Three types of constraints exists (Brodie, 1984):

Inherent: These are constraints which are the result of the data model which the application is based on. For example, in under the relational model, there exists the constraint that relations must be normalized.

Inherent constraints are not necessarily a good thing. Each of the traditional data models (hierarchial, network and relational) , however, come with their own set of such constraints. While they enforce a certain rigor in the design of databases, they also ultimately restrict what it is capable of modelling.

Explicit: These are constraints which have been specified by the application designer or implementer. The earlier example, where age is restricted to be between 0 and 150 is such a constraint. How explicit constraints are most efficiently implemented is an active research area. One presently popular approach is to express the constraints as predicates in some first-order logic language.

Explicit constraints may be further divided into static and dynamic constraints. Static constraints include such things as data typing and range checking. Dynamic constraints, on the other hand, are concerned with specifying what state transitions are acceptable, given the present state of the database. They are most commonly expressed in terms of pre and post-conditions for database transactions.

Implicit: Implicit constraints are those which are the result of interaction between inherent and explicit constraints.

One of the goals in building object-oriented database systems is to minimize the inherent constraints, while at the same time allowing explicit constraints (both static and dynamic) to be concisely and correctly expressed. Static constraints could be expressed as predicates over the object classes in the database. The real advantage to the object-oriented approach, however, lies in the specification of dynamic constraints.

In traditional database implementations, dynamic constraints are typically contained in validation rules in the application programs that access the database. These programs have unrestricted access to all of the entities available to them in the schema. Each of them is responsible for ensuring that they do not perform some corrupt action on the database. Since objects support encapsulation, the only way to access the entities in the database is via the operations that they make available. Therefore, each object class is responsi le for the integrity of its instances. The creation of complex interactions becomes a more straightforward exercise when the programmer can rely on each of the database objects to maintain their own integrity.

Closely related to constraints is the notion of *active databases*. The concept implies that the database schema should contain triggers or demons which automatically execute under certain conditions. For example, an attribute of an entity can have a certain demon associated with it which executes whenever an instance in updated. Active schemas provide a facility to maintain constraints and to also perform certain 'housekeeping' tasks for the database.

Underlying the concept of active databases is one key assumption: the data in the database is passive. The concept does not apply per se to object-oriented databases since this assumption does not hold. In an object base, the only way to modify an object is to use one of the methods defined for its class. Integrity checks and/or housekeeping jobs may be included in these programs. As a result, OODBMS's are active by definition.

Meta-Data Management

Database management systems provide some mechanism for the management of *meta-data*. Meta-data refers to 'data about the data' being stored and mai [illegible] ed in the system In conventional systems, the data definition language (DDL) and the data manipulation language (DML) are distinct. More recent tools treat these in a consistent manner, with the advantage that the meta-data may be accessed, and in some cases manipulated, at run-time.

The traditional approach to meta-data management is to first design the schema and then populate the database. The schema is viewed as largely fixed, and applications cannot typically access or modify the information held there. Its purpose is to define the structure of new instances and to keep track of existing ones. This approach is ineffective in areas such as CAD, which have a high rate of change. In these environments, the DBMS must supply the users the ability to add, modify and remove classes in the database schema. The more conventional view of schema design and then database implementation is giving way to a cyclical approach to database design, definition and use.

In addition, as (Zaniolo, et al) have pointed out, the main source of knowledge for a DBMS lies in its meta-data. It should be available to be treated in a manner similar to regular data. As they put it:

"In the meta-data lies the knowledge" (Zaniolo, et al; p. 59)

An object-oriented database management system should supply the users with mechanisms to deal with data and meta-data in a consistent manner. The most straightforward way to do this is to represent the schema itself as objects. Encapsulation will ensure that users cannot access the underlying implementation of the DBMS system objects; furthermore, the operations defined for the classes implementing the schema objects will ensure that any constraints will be satisfied, thus maintaining the integrity of the system. But since it is composed of objects, users will be able to manipulate the schema, just like any other class of objects. This is analogous to the class description mechanism in Smalltalk.

Knowledge Bases

No discussion about the current research in object-oriented databases would be complete without addressing the area of Knowledge Bases (KB). They represent another, not entirely separate, stream of database research.

KB's address many of the same issues dealt with by object bases by taking a *knowledge-based* approach. Instead of viewing the semantics of an application as being described by the combination of structure and behaviour of the objects involved, KB's model the real world using *facts* and *rules*, along with some sort of deductive reasoning mechanism. The facts are stored as tuples in a relational database and the rules are expressed in a declarative language, typically based on some first-order predicate logic. The reasoning mechanism provides a way to deduce truths about the world being modelled by the KB, without the necessity that they be expressly stored as a tuple, as in conventional databases.

From a programming language perspective, knowledge bases versus object bases may be viewed as Prolog versus Smalltalk. And in a manner similar to programmers arguing for their most emotionally favourite language, you can find those who argue for either object or knowledge bases as being superior[1]. In fact, they represent two different and complementary methodologies for modelling the real world.

3.3. Object-Orier [illegible] Databases

What then, constitutes an object-oriented database management system (OODBMS)? It should be apparent from the previous discussions on the object-oriented approach and on databases, that many of the data abstraction issues in databases have been resolved by the

[1] For an example of such an argument, see Ullman, 1988, pgs 28-29

object-oriented approach. For example, class and inheritance mechanisms support the abstraction hierarchies of classification, specialization and aggregation; and the binding of methods to classes supports the notion of encapsulation. The problem then, is to apply the concepts inherent in the object-oriented methodology to the design and implementation of databases.

This section first describes the two viewpoints taken by various researchers interested in OODBMS's. It then turns to the key features which earn a database system the 'object-oriented' label, including: support for object identity and complex objects, encapsulation, extensible type systems and inheritance. We will define an *object-oriented database system* to be any DBMS which supports, as a minimum, those inter-related features. However, keep in mind that object bases first and foremost provide the functionality basic to any database, such as: concurrent access to large amounts of persistent data, language support, backup and recovery mechanisms and access control.

Two Perspectives In Object-Oriented Databases

Object-oriented databases are a relatively new research area, and much of the interest in it is coming from two, quite disparate, groups. The first group is the database community. From their viewpoint, the object-oriented approach provides powerful new ways to abstract data and to implement databases which support complex applications. Much of their emphasis is on efficient access methods, indexing schemes and concurrency schemes. The second group is made up of researchers in the programming languages area. From their viewpoint, OODBMS's provide a mechanism to efficiently share and access persistent data. Their interests lie in such areas as the sharing of objects in different memory spaces, remote message sends and distributed garbage collection. The perceived issues and proposed solutions advanced by these two groups are quite different (for an excellent discussion on the differences between these two camps, see (Bloom and Zdonik, 1987)). The first constructs a data model to support the object-oriented paradigm and then defines data definition and manipulation languages in order to implement a complete OODBMS. The second typically takes a programming language (such as Smalltalk) and devises ways to extend it to handle persistent data. These different viewpoints is not surprising, given that the database community traditionally concentrated on the structure of systems, while the programming language community concentrated on process.

There are a number of possible ways to connect object-oriented programming languages (OOPL's) with object-oriented databases. These include (Peter Lyngbaek, in Power and Weiss, 1988):

- embed database languages (such as embedded SQL) into an object-oriented language; this approach works well for sharing and querying information, but is poor in terms of flexibility and transparency;
- export certain database constructs, but use the OOPL to write the methods which access the database; this is more flexible, but makes it more difficult to optimize queries;
- make persistence completely transparent, which is obviously good for transparency, but makes it more difficult to share and query data.

Object Identity and Complex Objects

As discussed in the previous chapter, objects have the property of identity. That is, an object can be distinguished from all others, regardless of its attributes Recall from Chapter Two that this implies that objects retain their identity regardless of any changes in their state (Ullman, 1988) considers this to be the definitive feature of an object-oriented database system, to the point where he has given the DBTG network and the IMS hierarchial models the object-oriented label, simply because they support object identity.

Object identity is an important concept for a number of reasons, including the following:

- Object identity supports the definition and representation of *complex* objects which have, as attributes, other objects of arbitrary complexity. In addition, *composite* objects, such as collections, may be supported. The members of these composite objects may be of arbitrary types, as opposed to most other data models where collections must be homogeneous.
- It allows the representation of objects for which little or nothing is known. This is an important feature in applications such as CASE, where it is important to capture data that is incomplete.
- Any or all of the attributes of an object may be changed without affecting the fact that it remains the same object.
- Related to complex objects, identity allows a high degree of structure sharing. Objects which share a piece of knowledge do so directly. This feature reduces the update anomalies as per the relational model.
- Finally, object identity is intuitive. Recall the basketball example discussed in the previous section.

Encapsulation

For a DBMS to be considered object-oriented, it must provide support for the notion of encapsulation. Recall that encapsulation is an implementation hiding technique. Instances of a data are accessed and manipulated via a set of operations supplied by their class, rather than have their representation dealt with directly. As a result, the behaviour of an object is packaged with its structure. This is done by binding operations to types (classes), which define the behaviour of their instances.

Encapsulation provides a mechanism to model behaviour, since the operations allowed by a class are defined to the database system as well. All of the programs which wish to manipulate instances of a certain type must use the operations that they provide. Applications become more concise, since the code to perform a certain operation occurs only once; they become more reliable, since there is a higher degree of code sharing and reuse; they cope with change better, since the impact of a modification can be localized to (typically) a few classes; and the integrity of the data increases, since constraints on the objects can be reflected in their operations. As a result, encapsulation aids in the development of large, complex systems.

Extensible Type Systems

Extensible type systems refer to the ability to define new data types for use by the database management system. A data type includes both a representation and a set of allowable operations. In a conventional DBMS, developers are restricted to using a fixed set of pre-defined types, such as characters, integers, etc. and a small set of operations on those data types. The goal of an extensible type system, is to allow the creation of new types which are indistinguishable from the system-supplied ones; and further, to be able to use these user-defined types in the creation of even more new types. This allows the nesting of structure to arbitrary levels.

Relational DBMS's allow the specification of new relations in the schema. In an object-oriented database, the class construct is used. Classes are similar to schemas in that they define the representation of a database entity. However, they extend this with the ability to use other, possibly user-defined, classes in their definition and by packaging operations with structure

Classes provide the physical data independence of relational DBMS's without limiting the expressiveness of the data model.

(Fishman, et al, 1987) lists the following capabilities as necessary to support an extensible typing system.

- There must be a mechanism for declaring new data types. Specifically, filters are required to provide some translation between the new type's internal representation and the representation viewed and manipulated by the user.
- There must be a way to define operations on new types. This would typically presume the existence of some sort of language support.
- There must exist a way to implement new database access methods for the newly created types.

Inheritance

In addition to providing a mechanism for adding user-defined classes to the system, an object-oriented DBMS should support inheritance - either single or multiple. Recall that objects were defined as being abstract data types, plus inheritance.

4. Implementing Object-Oriented Databases

4.1. Implementation Issues for Object-Oriented Databases

The following sections describe issues concerning the implementation of object-oriented databases. Much of the information contained in this section was derived from (Stein and Maier, 1988), and from the transcribed discussions contained in (Power and Weiss, 1988).

Object-oriented database systems are still primarily a research field, with many unsolved questions, as the following quote illustrates.

"...the field of object-oriented databases is taking off almost exponentially as a strong market-driven activity. Over 25 efforts are currently underway to implement OODB systems. There are many difficult research problems that need to be solved: object-oriented data models, management of composite objects, OODB programming languages, distributed transaction management on abstract data types for co-operative design environments, change management for evolving objects, object sharing in multi-lingual and heterogeneous distributed environments, query optimization techniques for abstract data types, development of an appropriate performance matrix for OODB's, and performance and reliability issues....OODB technology will take five to ten years to transition (sic) from its current status of 'proof of concepts' to the status of 'full commercial systems'." (Satish M. Thatte, writing in Power and Weiss, 1988; p. 87)

Complex Objects - How Big is Big?

OODBMS's support large, complex objects whose inter-relationships are part of the data, unlike relational databases, where such mappings are stored in the application code in the form of joins, projections, etc.. As a result, applications utilizing OODBMS's often have a *navigational* feel to them. The user is provided with an environment where he can look at the contents of an object and decide where to go next. At the limit, he may be able to access the entire database from a single root. This poses an obvious question: what are the bounds of an object?

This problem has implications for concurrency control and for data integrity. For example

- How do you place bounds on the closure of an object? The user does not consider the entire database as his domain of interest at any one point in time. How does the system identify what he does consider to be the set of objects he is interested in.
- When you save an object, do you save every sub-object which is reachable from it, or some sub-set?
- What kind of locking strategy can be used when, in the bounds of a single session, the user can navigate the entire database?
- Where should related objects be placed in the database? What sort of clustering strategies best meet the needs of the application?

Concurrent Access

Concurrent access to object-oriented databases raises two issues which must be addressed in their implementation:

- the nature of the applications which OODBMS's would typically be targeted to require prolonged access and manipulation of data - unlike conventional database systems where transaction durations are typically measured in fractions of seconds;
- complex objects may often take the form of directed graphs; concurrent access to such data structures is a more difficult problem than that faced by other DBMS's.

Prolonged Transactions

(Fishman, et al, 1987) identifies the characteristics of OODBMS applications which require prolonged access to the database. These include:

- Applications where one unit of work includes many conventional transactions against a number of different databases, possibly distributed across a network.

- Artificial intelligence applications where highly interactive queries may represent several concurrent and inter-related transactions, many of which are read-only.
- Design applications in CAD/CAM and CASE which require long transactions, with durations often measured in days and where the intermediate results of a transaction may be of interest to other users of the system. Further, since these transactions may represent several days work, allowing the DBMS to automatically abort them due to deadlock, or some other anomaly, is not viable. These applications also require the simultaneous existence of a number of different versions of the same objects.

In order the satisfy the requirements of the design transaction scenario, Fishman, et al proposed that the basis for concurrency control be provided by a *version manager.* Under this approach, all transactions are allowed to commit. If a conflict is identified by the system, an alternative version of the objects in conflict are created. Users check out one or more object versions for extended periods, and as a result, the locks required for them are maintained in persistent storage. This is in contrast with most database systems, where the locks are maintained in the program memory of the DBMS.

One approach to the implementation of a multi-user design environment is to provide concurrency control, in the conventional sense, only for the length of time required to check out a consistent view of the object the user wishes to manipulate. The transaction manager of a traditional DBMS would be used while persistent locks are put on the objects being reserved for the user. Those objects would remain locked until they are returned (checked in) by the user However, instead of over-writing the old data at commit time, the DBMS will create a new *version* Since a design transaction may last for days, there must also be mechanisms to allow the user to save intermediate stages of his work. These partial saves may be made to the database itself, or to some data store local to the user. If they are made to the database, they may be made available to the other users of the system on a read-only basis.

Complex Objects (Directed Graphs)

While some research is available on how to perform locking on hierarchial data structures, it seems that concurrent access to structures which are essentially directed graphs remain an issue. Under the hierarchial approach, a number of locking protocols are available (Ullman, 1988). One approach is to set a lock on the parent object, and whenever a lock is required on a child object, the lock tree is traversed in the correct order. However, it must be noted that whether or not a lock on the parent object implies locks on all of its children is application dependent. In some applications, the entire database may be reachable from a single root.

In situations where objects are directed graphs, a strictly hierarchial locking strategy will not work, since there is more than one path to an object. As a result:

> "You have to know all of the objects within a closure of an object to determine whether the closures overlap. That is difficult for two reasons, one is that the closures are often very large, and that means keeping track of a large amount of information. When you try to determine whether two closures overlap so that you can allocate a lock, you have to look at an awful lot of objects, which may not even be on the same server. If it is a distributed system, determining the closure overlap is going to take you all over the place. It is not clear that the method will work in practice." (David Wells, Texas Instruments, as quoted in Power and Weiss, 1988; p. 84)

Concurrency control in object-oriented databases will largely be the responsibility of the application developers. The best approach seems to be to supply the developers with a number of locking and versioning primitives, which are then used to create application-specific strategies. These locks and versions are maintained in persistent storage. While there is still a need for conventional concurrency control mechanisms, they are used primarily to ensure that the user is getting a consistent view of an object while these persistent locks are being placed.

The One-Level Store Abstraction

The concept of a 'one-level store' comes primarily from those programming language researchers who are interested in the problems of providing persistent objects. Briefly, the idea is that users and developers of systems which use a persistent object store should remain unaware of whether an object is in memory or on disk at any point in time. The idea is to make the storage of objects in a database transparent to the application programs which manipulate them. This is in contrast with most current database systems which differentiate between persistent and dynamic objects in terms of how they are defined and what operations may be performed on them.

The motivation for this approach is to avoid the situation common to many database programming environments now, where the programmer must deal either with a number of function calls to some database access routines or with an embedded data manipulation language which does not mesh well with the host programming language (ie. impedance mismatch).

The major issue surrounding the concept of one-level stores is whether it is a useful abstraction. Those from a programming language background view it as a natural way to implement persistent objects for a language. From a database perspective, it poses a number of problems. For example, if you have a database manipulation language which has been highly optimized for querying and manipulating persistent data, how well will it handle dynamic data? How useful is a one-level store in a multi-user environment, where the programmer will have to be concerned with object locking and transaction management regardless?

Classes and Their Sets

One key issue regarding OODBMS implementation is the meaning of classes In programming languages, classes define the common characteristics of their instances; as such, they are strictly *intensional.* On the other hand, in most database systems, classes are *extensional,* in that a collection of all of the instances of a class are maintained by the system.

Those who are basically interested in adding persistent objects to an existing OOPL, such as Smalltalk, often find the idea of a relationship between a class and its instances a foreign concept, as there is no comparable notion in programming languages. Maintaining such a relationship causes problems for garbage collection. If an instance is referred to by its class, it will always have at least one reference to it, and as a result, will never be reclaimed. On the other hand, people whose background is in databases find the notion of extensional classes intuitive. Without class-based collections of objects, database queries based on classes would be impossible.

The key question is really whether a collection of all instances of a class is meaningful, or should applications maintain explicit collections for those instances they are interested in? Many applications use instances of the same class, but store them in separate collections (Maier, et al, 1986). This is in contrast to the conventional database approach, which would store all instances of the same class in a relation. The different collections manipulated by the application would either be defined as a view or buried in the application code as a database query.

Object Deletion

Should objects be deleted explicitly by the application programs, as in traditional databases, or should they be automatically reclaimed (garbage collected) by the DBMS, similar to the Smalltalk approach? One factor which may make garbage collection a better alternative is the sheer complexity involved for the user of identifying those portions of shared, complex objects which should be deleted from the system.

Many researchers hope to avoid this problem all together, by not allowing the deletion of objects whatsoever. While this may result in some wastage of storage, the cost of mass storage devices is steadily decreasing.

Indexing

Should indexing be supported on the basis of internal state of an object, as is the case in conventional database systems, or should indices be based on the results of some method execution, as encapsulation would seem to require? Recall that objects, by definition, have their *representation encapsulated within a set of operations specified by their class.* An object's state is hidden from view. If this approach is followed when indexing objects in the object-base, a number of issues result. For example, if an index is based on a procedure call, how can the system be sure that the procedure will always return the same value for the same object state? This implies that the system must know which structural changes for an object will change the result of a method, so that the appropriate indices may be updated. If classes are to share indices with their subclasses, what happens when a subclass over-rides the definition of a method which forms the basis for an index?

Basing indices on the internal state of an object violates encapsulation; however, it can be supported without the expense of procedure calls. One possible solution would be to view indices on objects as being part of their definition, and therefore allowed to access their internal structure. If indices are allowed on the structure of the object directly, the next question becomes, how deep within a complex object can you specify an index? Only on the named instance variables specified in that class, or on instance variables of instance variables?

Class Modifications

How to best handle changes to class definitions is an important research topic in OODBMS's (Penney and Stein, 1987), (Zdonik, 1986), (Banerjee, et al, 1987), (Kim, et al, 1987). It may be viewed as being related to the concept of data independence, since it deals largely with the same set of issues: how can the schema be changed with minimum impact on the applications using the database and how should persistent objects be brought into line with their modified class definition? The second issue is trivial in an environment where objects are not persistent, since the programs involved may be simply re-compiled.

The schema of an OODBMS is, in some ways, more complex than a conventional database This is largely the result of inheritance and the binding of methods to classes. The following is a table of possible class modifications allowed in the Orion OODBMS, as shown in (Kim, et al. 1987; p. 120). Under their notation, nodes and edges refer to classes and specializations in a class lattice which supports multiple inheritance.

- Changes to the contents of a node (a class)
  - Changes to an instance variable
    - Add a new instance variable to a class
    - Drop an existing variable from a class
    - Change the name of an instance variable of a class
    - Change the domain of an instance variable of a class
    - Change the inheritance (parent) of an instance variable
      (inherit another instance variable with the same name)
  - Changes to a method
    - Add a new method to a class
    - Drop an existing method from a class
    - Change the name of a method of a class
    - Change the code of a method in a class
    - Change the inheritance (parent) of a method
      (inherit another method with the same name)

Changes to an edge
- Make class S a superclass of class C
- Remove a class S from the superclass list of a class C
- Change the order of superclasses of a class C

Changes to a node
- Add a new class
- Drop an existing class
- Change the name of a class

Two basic approaches have emerged to handle class modifications The first (Zdonik, 1986), calls for the placement of filters between the old object representation and its current form. The persistent objects are not modified until they are actually used by methods expecting the new structure. The second approach is to convert all of the persistent objects to their new representation at the time of the class modification. These two alternatives correspond roughly to "pay me now or pay me later" (Penney and Stein, 1987, p 111)

4.2. A Data Model Which Supports Objects

One data model which has been proposed for the support of object-oriented database is the Decomposed Storage Model (DSM) (Copeland and Khoshafian, 1985) The model is decomposed, in that, each attribute of an object class is represented as a separate binary relation. As a result, the values of each attribute are stored in a separate file The model supports a number of concepts which are important for the storage of persistent objects, including

- the concept of object identity, through the use of surrogate keys,
- directed graphs of objects;
- heterogeneous records; and
- multi-valued attributes.

The basic ideas behind the DSM can be described best by comparing the model to a typical relational storage model, where all the attributes of an object are stored together and object identity is not supported. This approach will be taken in the following sections, which shall describe the key features and characteristics of the Decomposed Storage Model. Note that the examples used in this discussion were based on those in (Copeland and Khoshafian, 1985).

In order to get a feel for the model, however, first examine figures 4.1. a) and 4.1. b), which show how a relation would be defined and stored under a relational database, and one which supports the DSM

Figure 4.1 a): A Normalized Relation

R

a1	a2	a3
v11	v21	v31
v12	v22	v32
v13	v23	v33

The a's represent attributes, the v's represent values

Figure 4.1 b): A Fully Decomposed Relation

r

sur
s1
s2
s3

a1

sur	val
s1	v11
s2	v12
s3	v13

a2

sur	val
s1	v21
s2	v22
s3	v23

a3

sur	val
s1	v31
s2	v32
s3	v33

Object Identity and Directed Graphs

Object identity is supported through the use of surrogates, which are system-supplied keys which are independent of the attribute values of the objects they represent. The DSM requires that these surrogates are stored in a separate file. As a result, it is possible to represent the existence of entities for which no information at all is known, without the need for storing explicit nulls to represent unknown information.

Object identity, implemented through the use of surrogates, allows the representation of directed graphs of database entities, which is crucial for the support of the complex objects. This is done by allowing surrogates of the child entities as attributes. Under the DSM, this will result in a binary relation made up of the parent surrogate and the child surrogate. An example of this is shown in Figure 4.1 c). Note that object surrogates have been stored as the values of a relation

Figure 4.1 c): Representing Graphs Using the DSM

r 1

sur
s1
s3

a11

sur	val
s1	s2
s3	s2

a11

sur	val
s1	s3
s3	s1

r 2

sur
s2

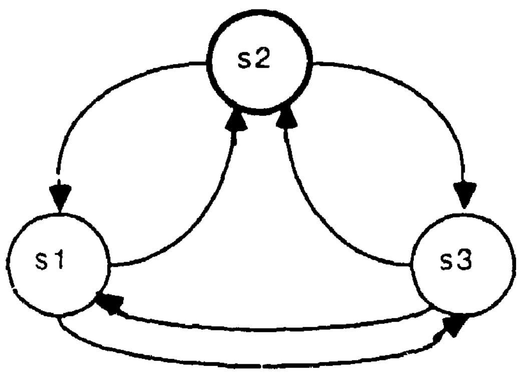
a21

sur	val
s2	s1

a22

sur	val
s2	s3

The DSM relations shown above correspond to the directed, cyclical graph shown below. S2 is highlighted since it is a different type than s1 or s3.



Heterogeneous Records

Heterogeneous records, refers to the ability of different tuples within a relation to have different attributes. For example, a relation which describes a set of employees might contain both engineers and salesmen. Both would include attributes a1 and a2 (say, name and birth date), but salesmen would have attribute a3 (company car) and engineers would have a4 (project). Under the relational approach, this would result in a storage structure shown in figure 4.2 a); note that an extra attribute, denoting the type of employee being represented must be added to the relation. Under a more sophisticated approach, where the relation has been partially decomposed is shown in 4.2 b) (Note that this representation assumes that a1 - the employees name - is a unique key).

Figure 4.2 a): Heterogeneous Relations

R

type	a1	a2	a3	a4
t1	v11	v21	v31	n/a
t2	v12	v22	n/a	v42
t2	v13	v23	n/a	v43

Figure 4.2 b): A Normalized Representation of a Heterogeneous Relation

R1

a1	a2
a11	a21
a12	a22
a13	a23

t1

a1	a3
a11	a31

t2

a1	a4
a12	a42
a13	a43

The DSM approach is shown in figure 4.2 c). While the partially decomposed relational structure is acceptable in terms of storage efficiency, it would involve spreading the representation of an employee over several relations. Furthermore, this decomposition is not performed automatically, as it would be using the DSM. Instead, its specification would have to be included in the design of the conceptual schema.

Figure 4.2 c): A DSM Representation of a Heterogeneous Relation

r

sur
s1
s2
s3

a1

sur	val
s1	v11
s2	v12
s3	v13

a2

sur	val
s1	v21
s2	v22
s3	v33

a3

sur	val
s1	v31

a4

sur	val
s2	v42
s3	v43

Multi-Valued Attributes

Multi-values attributes (also referred to as non-first normal forms), refers to the ability of a single attribute of a relation to have more than one value. For example, in an employees relation, you may wish to record the employees' children. Obviously, this attribute could involve several values for each tuple

Support of multi-valued attributes in a relational data model results in either reduced data independence or in greater complexity. If the storage model directly supports multi-valued attributes, a more complicated storage structure is required. The other alternative would be to further normalize the relations. Each attribute which could contain multiple values is decomposed into another relation. As a result, however, data independence is reduced, since a change from a single to multi-valued attribute (or vice versa) will result in a change in both the conceptual schema and the underlying file structures.

Figure 4.3 a): Multi-Valued Attributes

R	a1	a2	a3
	v11	v21	v31
	v12	v22	v32
	v13	v23, v24	v33

Figure 4 3 a) shows an example of a relation which contains multiple values for attribute a2. Under the DSM, this would result in a2 being represented, quite naturally, as an additional tuple in the relation for that attribute. This is illustrated in Figure 4.3 b). Thus the DSM approach has the following advantages:

- no additional complexity is introduced by the presence of multi-valued attributes, and
- data independence is not affected, since switches from multi to single-valued result in no changes to the storage structures.

Figure 4.3 b): A DSM Representation for Multi-Valued Attributes

a2	sur	val
	s1	v21
	s2	v22
	s3	v23
	s3	v24

4.3. GemStone: An Object-Oriented Database

GemStone, and its programming language OPAL, is one of the few commercially available object-oriented databases. It is also one of the best documented in the literature (Maier, et al, 1986), (Penney and Stein, 1987), (Purdy, Schuchardt and Maier, 1987) and (Maier and Stein, 1986). Other examples of object-oriented databases[1] include: POSTGRES (Stonebraker and Rowe, 1986), (Stonebraker, Anton and Hanson, 1987); ENCORE (Hornick and Zdonik, 1987), (Skarra and Zdonik, 1986), (Smith and Zdonik, 1987); ORION (Kim, et al, 1987), (Banjeree, et al, 1987); Iris (Fishman, et al, 1987); Emeraude/PCTE (Emeraude, 1987); VBASE (Andrews and Harris, 1987); and VOOD (Barbedette and Richard, 1986).

The basic approach taken in the design and implementation of GemStone was to extend the Smalltalk language with a number of database amenities. These include: support for queries over collections, persistent storage structures, class definition facilities and a multi-user environment for data sharing. Unlike most other efforts which extend programming languages with persistence, however, the GemStone group comes primarily from the database community.

[1] It should be noted, however, that although these systems share the label 'object-oriented database systems', they are often radically different in terms of their design and implementation.

As a result, it is a database first, with a Smalltalk-like language and programming environment available for applications development. The choice of Smalltalk as its basis was, in fact, largely market-driven since it is the best known of the object-oriented languages.

The basic GemStone architecture can be seen in Figure 4.4. The key things to note are:

- *Stone* provides the basics of a centralized, persistent object server This includes such things as storage management, concurrency control, transactions, recovery, support for collections and active session workspaces. Object identity is supported in Stone through the use of surrogates - referred to as 'object-oriented pointers' (OOPs) - similar to the Decomposed Storage Model described above. However, an object's instance variables are stored together, unlike a decomposed representation. Stone maintains an object table, in the form of a B-tree indexed on OOPs, which maps OOPs to physical locations

  Stone provides four basic storage structures: *self-identifying* (ie. SmallInteger, Character), *byte* (ie. String, Float), *pointer* (ie. Employee, SMEArc), and *non-sequencable collection (NSC)* (ie. Set). NSC's may be queried. Stone provides only the most basic operators for object access and manipulation.

- *Gem* corresponds roughly to the virtual machine layer in a Smalltalk implementation It adds the data abstractions required to add the 'object-orientedness' to the GemStone model. It also provides such facilities as the OPAL byte-code interpreter and access and session control. Gem also includes the system-supplied hierarchy of classes.

  Gem provides a facility to constrain named instance variables to be of a certain class. For example, an Employee's name variable could be constrained to be a String

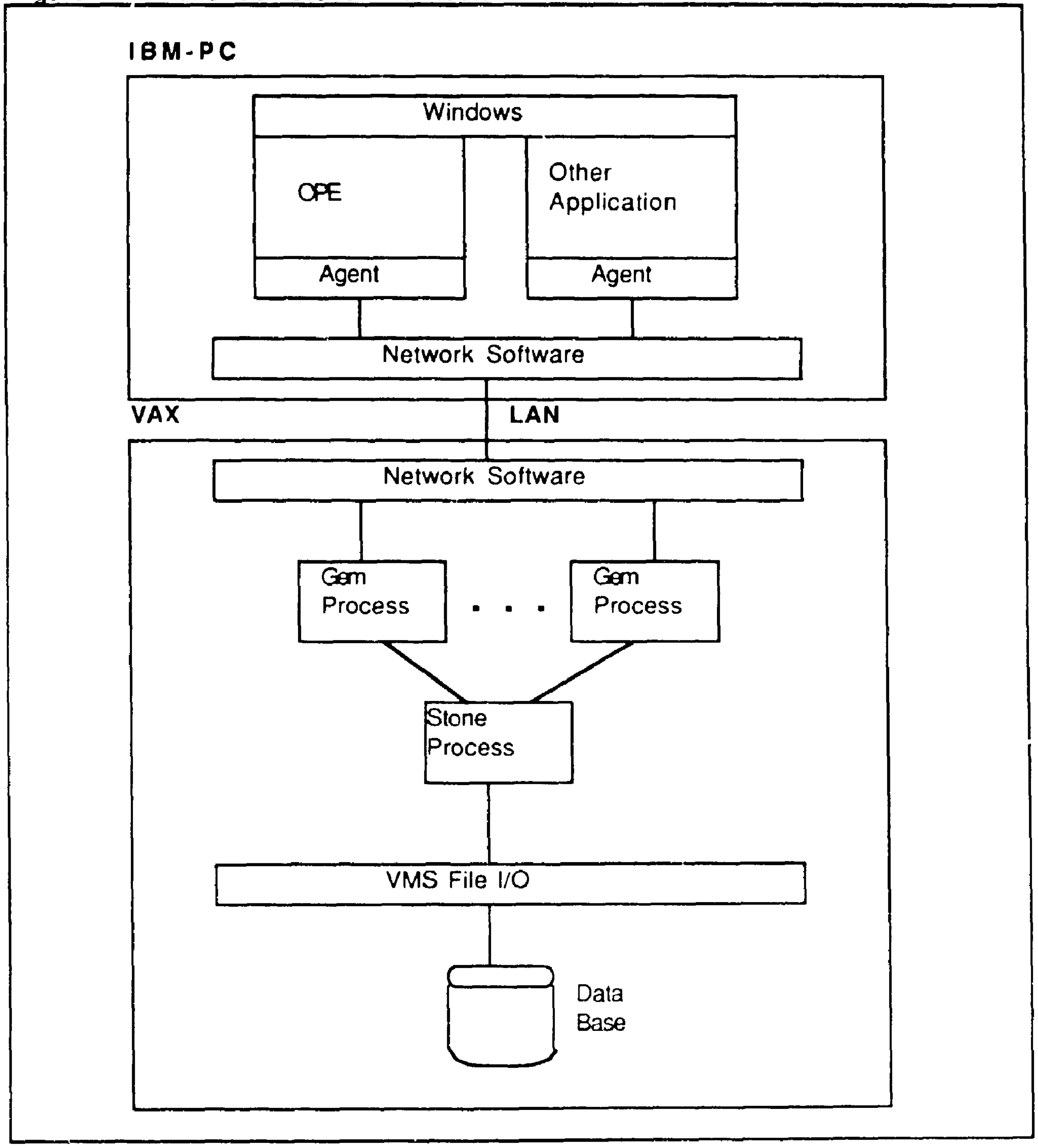
- *Agents* are a set of routines to facilitate communication between GemStone and applications written in other languages, such as C and Pascal Programs written in those languages may make calls to an Agent process for session and concurrency control, passing messages to GemStone objects, executing OPAL statements and compiling OPAL methods.

Information is passed between the Agent and Gem in the form of bytes and object pointers.

- *OPAL*, as mentioned earlier, is basically the Smalltalk language extended to support persistence. OPAL supports class definitions (data definition), data access and update (data manipulation), and control of the GemStone server. However, the user interface classes which are so much a part of the Smalltalk environment, have been removed from the OPAL image. Applications are instead expected to manage the human interaction via modules written in C or Pascal, with database access provided by an Agent process. The OPAL Programming Environment (OPE) provides a window-based interface for the creation and modification of GemStone classes.

Gem, Stone and Agent are separate processes. While there may be multiple Gem and Agent processes running (typically one each per user session), there is only one Stone process running per GemStone system.

Figure 4.4: The GemStone Architecture



Session and Access Control

All objects in a GemStone database belong to a particular *segment*. Segments are owned by users, and each user owns at least one segment. Access permissions to a segment may be granted by its owner to other users of the system, who are identified by userid and password control by GemStone. Note that having a pointer to an object is not the same as having permission to access it; furthermore, having permission to access an object does not imply access to all of its sub-objects.

Each GemStone user has a list of *name spaces* specified in his UserProfile. These are dictionaries which are used to provide individual users the illusion of a global name space. Whenever a name is encountered by the OPAL compiler which is neither an instance or a class variable, the user's name spaces are searched to find the object being referenced. Name spaces may be included in the UserProfile of a number of users, and are therefore a mechanism for sharing information.

Concurrency Control

Concurrent access to the database is provided by Stone, which maintains a *workspace* for each active session. This workspace contains a *shadow* copy of the object table, which was based on the most recently committed object table, referred to as the *shared table*. Whenever an object is modified by a user, a new copy of it is placed on a disk page which is inaccessible to other sessions The user's shadow copy of the object table is then modified to point to the new object location The shared table is not actually copied at the beginning of a session, instead, the top node of the B-tree which represents the table is copied, and nodes are added to the tree as objects are accessed during the session.

Unlike the locking two-phase locking protocol discussed in the previous ch ,pter, the concurrency control scheme implemented by Stone is optimistic. Conflicts are checked at commit time, rather than prevented through locking. Stone keeps track of the objects which a transaction has read and written, and conflicts with other transactions which have committed since it began are identified. If any conflicts are identified, the changes ·ı the shadow table are discarded, after the disk pages used to write new objects are reclaimed by the storage manager. This approach has the advantage that read-only transactions can never conflict, since they never write to the database at commit time. However, care must be taken to ensure that write transactions are not too big, since long periods between commits may result in lost work.

Collections and Indexing

Indexing in GemStone is provided for collections, rather than classes Those applications which require associative access to the instances of a class must implement that class to maintain its own instan· ə collection No such collection is provided automatically by the system

Indices therefore exist on explicitly maintained collections. They are created and removed by sending a message to an instance of Bag or Set, which specifies the path that the index is for. For example, if empSet if a set of Employee objects, an index may be added to the set by sending a message specifying that it is to be built on empName last (ie. the last name of the employee instances in the set).

Indices may be based on either the value of a path, or its identity. Note that since a value-based index (known as an equality index) is based on the internal state of an object, it violates the encapsulation of those objects for which it is defined. For an equality index to be defined, the types of the instance variables mentioned in the path must be constrained to be of a particular type.

Quer; .g

Associative access is performed in GemStone by querying collections which are maintained explicitly by the application. The query language itself is in the form of a mess:·r protocol implemented by class Collection and its subclasses, such as Set and Bag. For example:

```
aBag  select:  aBlock
```

The block may be thought of as a Boolean selection expression which is evaluated against each element of the collection. If the expressi. n evaluates to true, the element is included in the collection returned by the query Whether or not an index is to be used can be controlled by the programmer, by using braces instead of brackets to enclose the block expression. For example:

```
aBag select:
  { :anEmp | anEmp.empName.lastName = 'Sanders'}
```

would cause a index to be used, if one existed. while

```
aBag select:
  [ :anEmp | anEmp empName.lastName = 'Sanders']
```

would not

Garbage Collection

Garbage collection is done at two points in GemStone. The first is during user sessions, where objects which have been created and then later de-referenced are treated as garbage and their memory reclaimed. Once an object has been committed to the database, however, it may also later become garbage. In order to handle these, GemStone collects persistent garbage off-line, using a mark-sweep algorithm Note that this requires that the database be made unavailable during the time when the garbage collection process is being run.

5. Design Issues

Before turning to the design issues addressed in this work, a number of underlying definitions must be made clear to the reader These definitions are intended to describe the differences between an object-oriented database management system, an object server, and an object-oriented database programming language

Object-Oriented Database Management Systems: These systems support persistent objects using a complete environment. They typically have their own programming language (which may be embedded in a different host language), memory management, and data model. If the persistent objects are used by a host programming language, the applications developer must deal explicitly with the location of objects The objects must be explicitly read into memory by the program before being acted upon, or commands issued to the DBMS to perform actions upon the persistent objects in the database. An example of an OODBMS is GemStone, as described in the previous chapter.

Object Servers: These systems provide persistent object support for an existing programming language. The abstraction of a 'one-level store' may be supported In other words, the applications programmer no longer must be concerned with the location of the objects he is dealing with However, there remains a delineation between the programming language, and the persistent object server For example the object server requires a separate address space from the executing application This work represents the design and implementation of an object server for the Objective-C programming language.

Database Program ing Languages: These systems seek to provide programming languages wh c persistence is designed right into the language itself. There is no separation between the persistent object manager and the language. Such languages are describe l in (Cockshot, et al, 1983) and (Atkinson and Buneman, 1987). A number of thoughts on creating such a language are described in the concluding chapter, under Future Research.

5.1. Design Goals and Constraints

The following sections describe the short and long-term design goals for the object server The immediate goals are those which must be met by the initial implementation, while the long-term goals are those which must be allowed for in the design, so that future enhancements may be made with minimum effort.

Immediate Goals

The first goal of any object server must be the reliable, persistent storage of objects. That is the primary raison d'etre for any form of database system. The following describes the objectives which this work must satisfy in its implementation.

- Object-Oriented Data Model. Since the object server is intended to support an object-oriented programming language, its data model must provide the features inherent in the object-oriented methodology. These features include support for . object identity, inheritance, complex objects, encapsulation, and an extensible typing system. Recall that support for extensible types requires support for: the declaration of new types, the definition of operations for those types, and the ability to define new database access routines for those types.

Persistent objects are to have their allowable operations defined in the Objective-C language. Furthermore, class definitions used by the object server must also be shared with the Objective-C compiler. Therefore, some facility for maintaining the required relationships between the object server schema and the Objective-C application code must be provided.

• Schema Modifications: Access to the system's meta-data must be provided, and the user must be able to manipulate it in order to define and modify the class definitions which make up the database schema. Class modifications must be reflected in some reasonable manner in the data manipulated by the Objective-C methods.

• Object Granularity: The objects being manipulated by the toolset fall into two broad categories: complex, highly structured objects which are made up of numerous objects of relatively small granularity; and large, unstructured objects (such as text). The object server must provide access to objects of both types, with reasonable access and update performance.

The granularity of persistence must also be addressed in the implementation. For example, are entire complex object graphs, individual objects or individual variables within objects to be the unit which may be specified as persistent.

• Consistency with Objective-C: The goal of this work is to provide a persistent object server for the Objective-C language. Impedance mismatch must be kept to a minimum. The typing system must be as close as possible to that used by the language For example, introducing a stronger notion of typing in the object server's data model would not be beneficial. Similarly, splitting the storage of instances into their various types, as specified by their inheritance hierarchy, would not be useful, since Objective-C stores the entire object as one unit.[1] The server must also support the basic data types manipulated by the language. These include the C primitive data types, and the foundation object classes provided with Objective-C.

Along the same lines, the object server implementation should require as few modifications to the Objective-C language and run-time support facilities as possible. For example, modifying the compiler to recognize persistent objects as some special case is not an option.

- Integration with Objective-C: The object server, as a development tool, must be as simple to use as possible. First and foremost, this implies that the object server should be well integrated with the Objective-C language. Accessing persistent objects should not require the use of some special protocol; persistence must be provided to objects without requiring that they be a subclass of some special class; the location of persistent objects (ie. in memory or on disk) should be transparent to the program[2]; and the use of persistent objects should be syntactically identical to normal object usage.

Although the object server should be as straightforward to use as possible, it does not necessarily have to be invisible. Some reasonably small amount of interaction between the client programs and the object server is allowable.

[1] For example, say you had two classes - Person and Employee Employee inherits from Person. Instances of Employee have all of their instance variables contained within the same object. An alternative approach would be to have the Person portion of an Employee stored in one memory ' cation, and the Employee portion of an Employee stored in another. In a database sense, this would imply two entities (records) would be required to store one instance of Employee.

[2] This is generally referred to as the 'one-level store' abstraction, whereby the movement of data between disk and memory is hidden from the programs accessing the data.

Long-Term Goals

While not part of the initial implementation, multi-user access to the object base is of great interest - for both practical and academic reasons. Concurrent access to persistent objects would be an important step towards a complete CASE environment, with support for all of the team-oriented aspects of the software development process. It is also a difficult, and therefore interesting, problem. Multi-user access to databases which support CASE environments remains, to a large degree, a research issue. This is largely due to the fact that such environments require two complex features: support for prolonged access to data, while maintaining a consistent database, and for the maintenance of multiple versions of the design objects.

Proposed extensions to the object server implementation for multi-user access are contained in the Further Research section at the conclusion of this work.

5.2. The Basic Architecture

This section is intended to give the reader an introduction to the object server design. *Subsequent sections will describe its components in greater detail.*

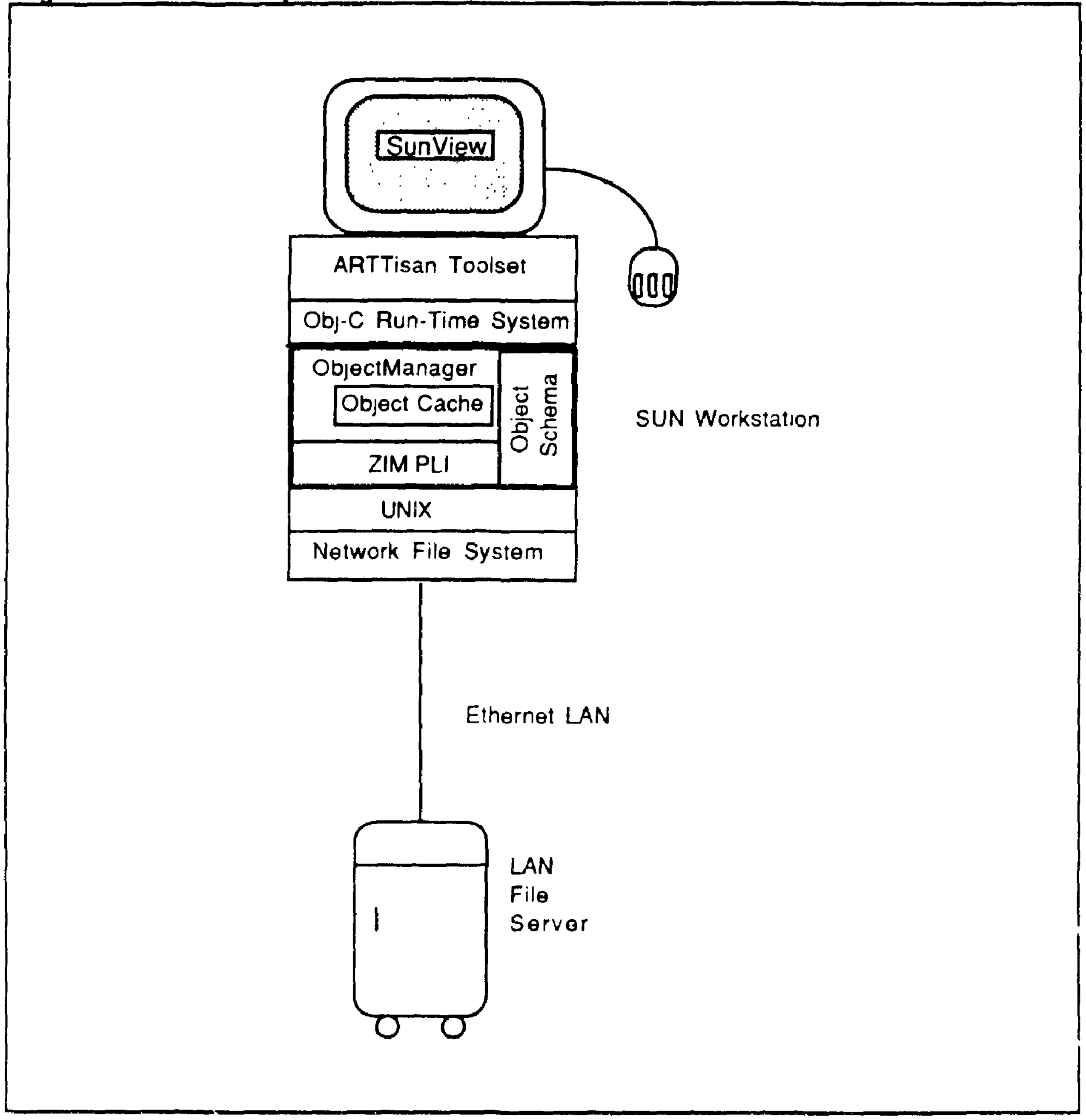
The basic structure of the object server, and how it relates to the toolset, can be seen in Figure 5.1. Some key things to note are:

- Objective-C applications utilize the object server by linking the class ObjectManager with the application. The ObjectManager has no instance methods - its functionality is implemented as a class protocol. When the application begins, the ObjectManager is initialized by sending it the *open* message, which returns the *database root* object. The ObjectManager session may be terminated by sending it the *close* message

- Persistent objects are read from, added to, and updated in the object base by making calls to the ZIM Programming Language Interface (PLI) routines described in the following chapter.
- All data manipulation of the persistent objects is performed within the address space of the Objective-C application. Updates cannot be forwarded to the object server to be performed in some asynchronous manner; all calls to the ZIM PLI routines are blocking. Objects which are required by the application are read, manipulated by Objective-C methods, and then returned to the database.
- The applications which use the object server do not access the persistent objects directly. Instead, database objects are represented in the application by instances of class Proxy. Messages which are sent to proxies have their receiver object transparently changed from the proxy to the underlying database object which it represents. This swap is performed by a modification to the Objective-C message sending kernel. No changes were required to the Objective-C compiler.
- When a message is sent to a Proxy instance, if the persistent object it represents has not already been read, it is loaded from the object base and cached by the ObjectManager. This provides the illusion of a 'one-level store'.
- The ObjectManager utilizes metadata (the "Object Schema") to describe the representation of objects - in both their object base and Objective-C formats.
- Objects are saved to the object base by explicitly telling the ObjectManager to commit. New objects, and changes to existing objects of a persistent class are saved to the database only if they are committed. This results in an implicit read/explicit write data access model.

- Persistent objects are stored in a ZIM relational database, using the ZIM Programming Language Interface. An object-oriented data model has been implemented using the PLI. All instances of a persistent class are stored in the same ZIM database file.

Figure 5.1: The Object Server Architecture



5.3. The Data Model

The data model implemented by the object server supports the concepts basic to the object-oriented methodology. These include: support for object identity, complex objects and multi-valued attributes. Unlike the Decomposed Data Model presented in Chapter Four, the *selected data model stores all of the attributes of an object on the same record* All instances of the same class are stored in the same ZIM database file. Note that the data representation of each object includes all of the instance variables defined for instances of its class - including all of the variables defined in superclasses. Figure 5.2 illustrates this point. It shows the definition of two classes (in Objective-C syntax), and how an instance of each class would be stored as records in their corresponding database file. Notice that instances of class Employee contain the variables defined in the superclass Person in the same structure.

Since the goal of this research effort was to provide a persistent object store for the Objective-C language, every attempt was made to keep the database representation of persistent objects close to the internal Objective-C format. In fact, the data model used by the object server is essentially a ZIM-based, persistent implementation of the Objective-C data model

The following sections describe the key features of the data model, and provide some insight into their ZIM PLI implementation.

Object Identity

Identity is one of the key properties of objects. It is supported in the data model by assigning every object a unique surrogate, called the Object Identifier (OID) The OID of each object has two parts: the first is the integer Unique Identifier (UID) of the object, the second is the class number of the object, which is actually the ZIM database file number which contains all instances of that class. The OID is actually represented as a C double precision number; the integer portion is used as the UID, and the decimal portion is used as the class number. This implies that every object in the database has a nine-byte header: eight bytes to represent the double, and one byte to hold the 'null' byte used by ZIM itself.

Since OID's contain both a unique identifier and the ZIM file number where the object resides, they can be used to locate objects in the database without the need for an object table The OID contains all of the information needed to find the physical location of the object. Figure 5 2 shows how this would be performed. Suppose you had read the object representing Fred le'Janitor, and you then wished to access the object representing his spouse. Her OID is stored as an instance variable of Fred. Using the ZIM PLI, you can find her object by searching the database file #101, as indicated by the class identifier contained in her OID.

Figure 5.2: Persistent Object Representation

```
= Person : Object {  // ZIM file 101
      char       name[30];
}
= Employee : Person {  // ZIM file 102
      int        empNum;
      id         spouse;
}
```

Employee Database file #102

OID	name	empNum	spouse
1111.0102	Fred le'Janitor	999	2222.0101

Person Database file #101

OID	name
2222.0101	Molly le'Janitor

A persistent object (called, say O_1) is therefore stored in the database with a unique 8-byte key - its OID - which allows its retrieval. Other persistent objects (for example, O_2) that reference O_1 do so by storing its OID as the value of an instance variable of type id. Note that when O_2 is read into memory, the 8-byte OID referencing O_1 must be converted into some 4-byte value, which is the size of an id[1]. In fact, the OID is used to create an instance of class Proxy, which will be discussed in detail in Chapter Seven.

Object Typing

Persistent objects follow the same typing conventions as ordinary Objective-C objects That is, their types are resolved at run-time, rather than at compile-time All objects are of the type *id* as far as the compiler is concerned. At run-time, the objects' isa pointers are used to identify which class the object is an instance of. No attempt is made to constrain the type of a certain instance variable to be of a certain object class. This applies only to instance variables which are of type *id*. For all other variable types, the regular C language typing rules apply.

Early in the design of the object server, introducing a stronger typing mechanism was considered. However, it seemed to buy relatively view benefits and presented the possibility of introducing impedance mismatch between persistent objects and Objective-C.

Data Types

The object store provides a mechanism to maintain persistent objects. It does not support C structures or unions as units of persistence The types of the instance variables within objects is limited to the following primitive types:

short, long, int, unsigned, double, float;
char (as a character array, not a C string), boolean,
id (objects).

[1] Recall that an id is nothing other than a C pointer, which on the SUNs is four bytes.

To get the functionality of a C null-terminated string, use instances of the class String.

Since ZIM supports only a subset of these data types, the object server must convert the types when reading and writing objects from the database. The types, and their ZIM storage type are:

C Type	Stored as
short	short
long	long
int	long
unsigned	long
double	double
float	float
char	char
id	double (an OID)

The object server presently supports the following Objective-C classes:

```
Object
  Cltn
    OrdCltn
      Stack
    Set
  IdArray
  String
  Point
  Rectangle
```

Thus the object server supports most of the "Foundation classes" which come with the Objective-C compiler[1]. As will be described later, many of these classes must be treated as special cases by the ObjectManager. This is due to the non-object-oriented techniques used to implement these classes. Specifically, the principle of encapsulating objects was abandoned in favour of speed in their implementation.

[1] The following Objective-C Foundation classes are not presently supported: Bag, BalNode, SortCltn, Assoc, Dictionary, BytArray, IntArray, and Sequence. Their absence is primarily the result of the fact tha at present, they are not utilized by the toolset code.

Complex Objects

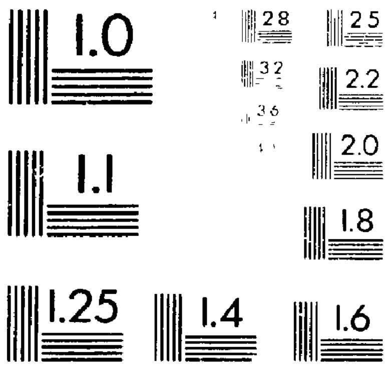
Complex objects are essentially directed graphs of inter-related objects. The connections between related objects in the database may be represented using the Object Identifiers (OIDs) of the various objects. A persistent object (called, say O_1) which refers to second persistent object (O_2) does so by storing O_2's OID as the value of an instance variable of type id. In some ways then, the data model is similar to a network model, where the links between objects are maintained as unique database identifiers.

Consider the previous example concerning Fred le'Janitor. An extended example, using a more purely object-oriented representation, could use instances of the class Name, in which a Person's first and last names are String objects (see Figure 5.3). In this example, we have introduced two new classes - Name and String. Name is a complex object which has two instance variables, namely firstName and lastName, which are both instances of class String. Note also that the Person class has been extended, such that the instance variable name is now an object (an instance of class Name), and that the instance variable spouse has been added to it. In the example using Fred le'Janitor and his wife Molly, the database must now store eight distinct objects which form a directed graph, with a cycle[1]

The concept of complex objects may also be used to explain how the object server is to be utilized by the toolset. Essentially, the contents of the entire object base may be viewed as one large, complex object. Every object in the database is reachable from a single *database root* object. When a session with the ObjectManager is started, the root object is supplied to the

[1] Molly is referred to by Fred as a spouse, and vice versa.

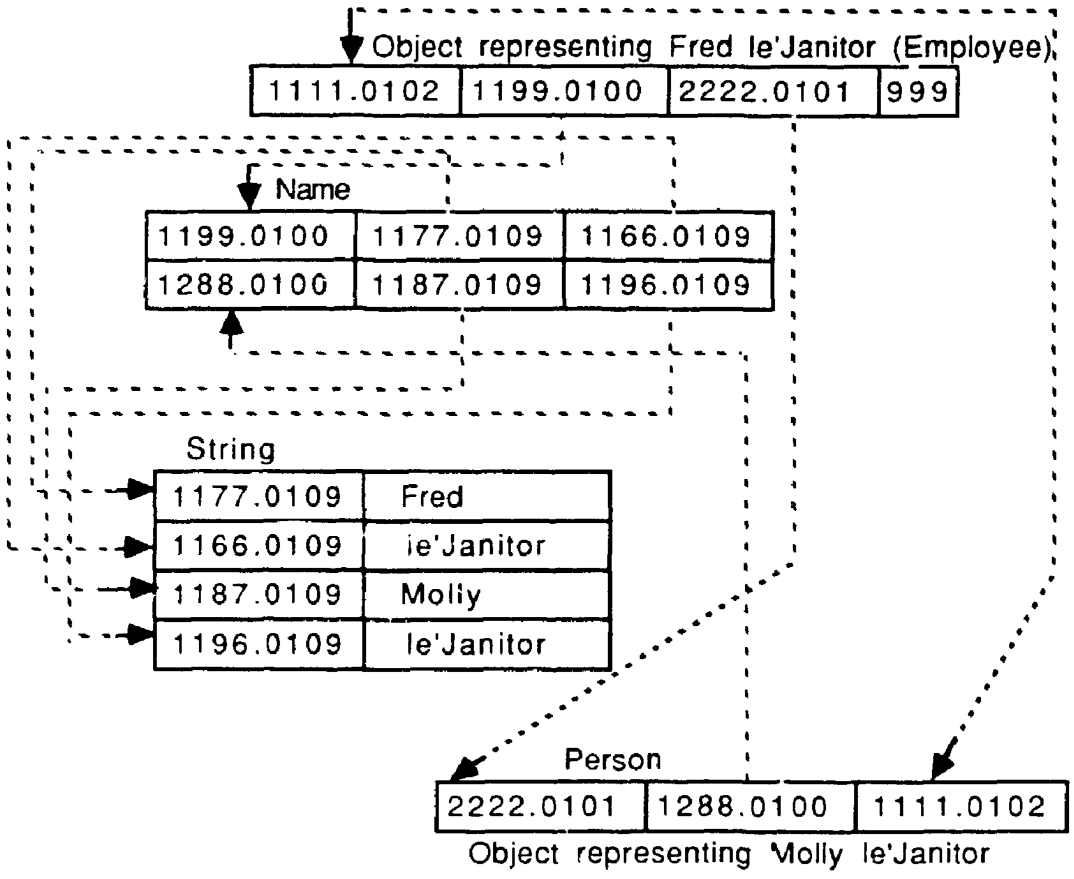
2



application. As the toolset session navigates the network persistent objects, they are read in from the object base. This process will be described in detail in the following chapter.

Figure 5.3: Representing Complex Objects

```
= Name : Object { // ZIM file 100,  Strings are in file 109
     id          firstName,
                 lastName;
}
= Person : Object {  // ZIM file 101
     id          name,
                 spouse;
}
= Employee : Person {  // ZIM file 102
     int         empNum;
}
```



Multi-Valued Attributes

Objects which have attributes which are multi-valued may be represented by using instances of the various collection classes provided by Objective-C as their attribute value. For example (Figure 5.4 a & b), say that Employee objects were to now maintain a set of all of their children. This may be done by simply assigning the instance variable children a set object (or some other collection-type object) as its value. In Objective-C, instances of class Cltn (Collection) and its subclasses - such as Set - maintain their contents as an IdArray, as is shown in the figure. Instances of these collection classes are often referred to as *aggregate* objects.

Persistence Granularity

Most persistent object systems use the object as the unit of persistence. That is, an object is the smallest unit that may be placed in the persistent object store. Typically, which objects are to be saved is specified by describing a class of objects to the system. Instances of that class may then be saved to the object base.

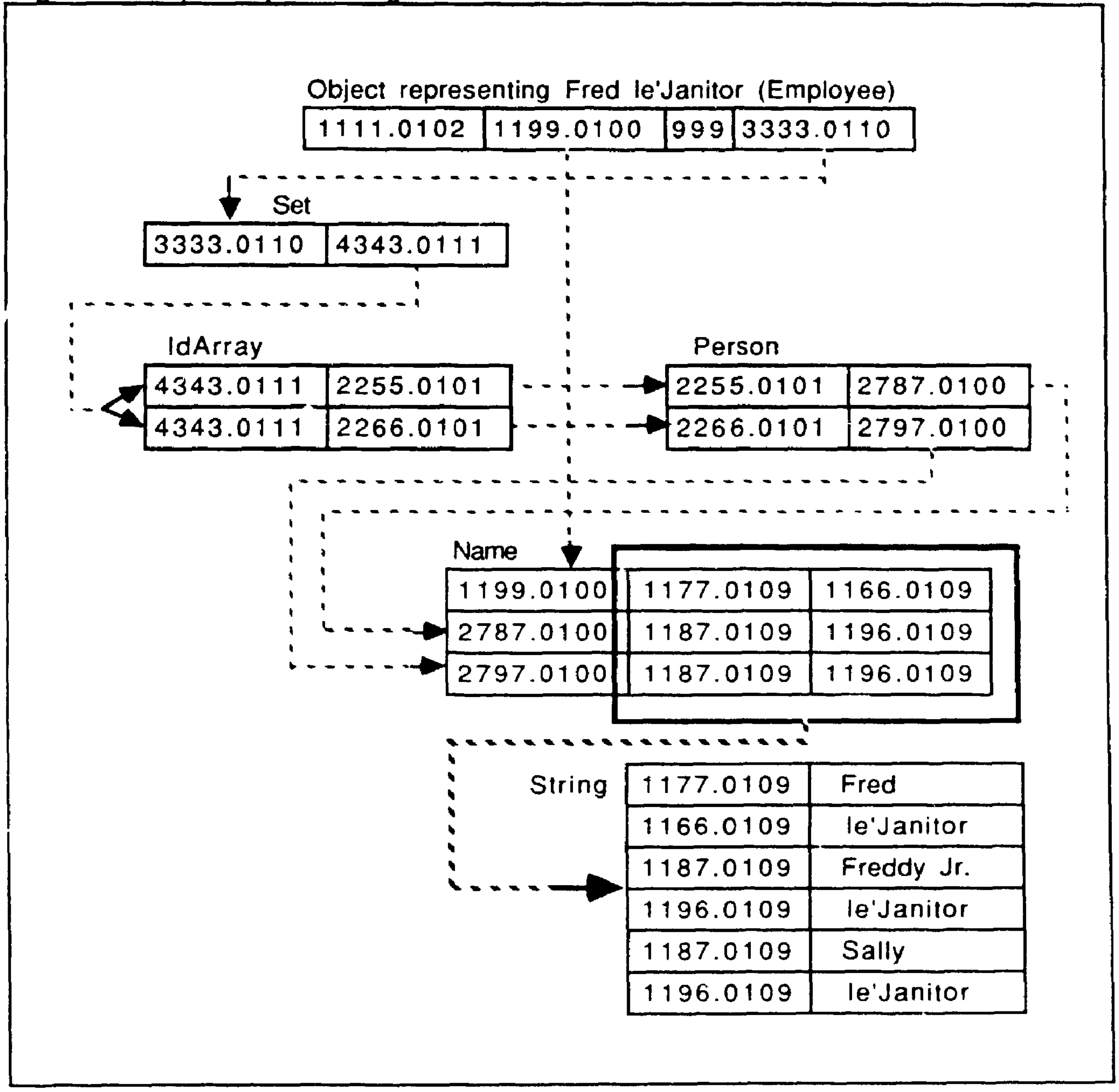
The data model used here allows instance variables as the unit of persistence. This allows for the specification of object classes which save only portions of their instances in the database The portions not saved are assumed to be resolved at run-time by the application. When an object is read from the object base, any non-persistent variables are initialized to nulls; when an object is written to the database, any non-persistent variables are removed before the object is stored.

This approach is motivated by the fact that many of the classes which make up the toolset contain instance variables which have non-persistent instance variables. For example, many objects contain references to SunView structures. These structures represent some run-time, graphical representation of the object. As such, it makes no sense to save these structures to the object base.

Figure 5.4 a): Defining Multi-Valued Attributes

```
= Name : Object { // ZIM file 100,  Strings are in file 109
      id          firstName,
                  lastName;
}
= Person : Object {  // ZIM file 101
      id          name;
}
= Employee : Person {  // ZIM file 102
      int         empNum;
      id              dependents; // Set is in 110, IdArray is in 111
}
```

Figure 5.4 b): Representing Multi-Valued Attributes



6. The Language to Object Server Interface

6.1. The Building Blocks: ZIM and Objective-C

The object server was implemented using two tools: the Objective-C programming language, and the ZIM entity-relational DBMS. The following sections describe these two products in some detail. Many of the features and constraints mentioned below impacted the eventual outcome of this work.

The ZIM Programming Language Interface

ZIM Basics

ZIM is a database management system developed by Zanthe Information Inc., of Nepean Ontario. The full DBMS supports most of the features of the entity-relational data model, although database entities do not have the property of identity. Entities in the database are stored in *entity sets* which are implemented as database files. Any indices on an entity set are stored in the same file.

ZIM provides an integrated environment to build typical database applications. It includes multi-user data access routines, a transaction manager, access control and security, a screen formatting facility, a report writer, and a 4GL programming and query language, with compiler. All of the various facilities are included in the same package, unlike most such products, where many of these facilities are separate. The programming language supports all of the common control structures, procedure and macro abstractions, and recursion.

Why Use ZIM?

ZIM has three features which made it attractive for this research. The first is that the meta-data is available to the programmer as regular data. For example, the definition of an entity type is contained in the two entity sets EntitySets and Fields, and the data in these may be manipulated as would any other data. These two entity sets maintain the following information:

Entity Name	Field Name	Type	Length	Description
EntitySets	entName	char	18	name of entity set
Fields	SN	num	4	sequence number of field in set
Fields	fieldName	char	18	name of field
Fields	ownerName	char	18	name of set which contains this field
Fields	type	char	8	data type of field
Fields	length	num	5	length of field, in bytes
Fields	decimals	num	2	number of decimals
Fields	index	char	6	indicates if an index is to be maintained on this field

The data manipulation language and the data definition language are one and the same It should be noted that manipulating the meta-data does not result directly in changes to the database structure. Once the database schema has been modified, the definitions involved must be converted to their internal ZIM format, and the database files involved reformatted or created. Unlike most DBMS's, however, these changes to the schema may be made on-the-fly by ZIM programs - with the restriction that the database must be accessed in single-user mode. This process is analogous to compiling: first the schema is defined, and then it is converted into a format directly usable by the system.

The second feature is that Zanthe offers, as a product, the ZIM Programming Language Interface (PLI). This utility offers access to the database files from within C (and by extension, Objective-C) programs. It should be noted that the PLI is not an embedded query language, such as ESQL or QUEL. It provides a series of C functions which may be called to perform a number of database utilities. These include: opening and closing of database files; adding, deleting and updating individual database entities (records); adding and deleting index entries, password control for access to the database; and starting, committing and aborting database transactions An Objective-C message protocol for the ZIM PLI is discussed in the following chapter

The third feature provided by ZIM is its portability. The product, including the PLI, is available for a wide range of hardware platforms, including: IBM PCs, IBM mainframes running VM/CMS, DEC VAXs running VMS, and SUN and Apollo workstations. The ability to re-implement the object server on different platforms, if required, is a powerful feature.

The combination of these three features supplies a solid basis for developing experimental database systems. In this case, ZIM provided the basics required to build a complete Objective-C object server. Since the meta-data is available as regular data, it is possible to write Objective-C routines to control the class definitions in the object base. However, to have these changes reflected in the database, routines must be written in the ZIM 4GL, since that functionality is unfortunately not included with the PLI. How this is performed is described in the following chapter.

ZIM Objects (Record Structure)

Objects in a ZIM database are stored as records in a database file. The records are actually ordinary C structures written to disk. These structures may be made up of a fairly restricted set of data types. The following table lists the types supported by ZIM, and their corresponding C data type.

ZIM Type	C Type
int	short
longint	long
vastint	double
char[1]	char (as an array, not a null terminated
alpha	char string)
varchar[2]	struct
varalpha	struct
numeric	char (restricted to digits)
date	double

The structure of these records are defined in the database schema. For example, the following schema entry:

	Seq No	FieldName	EntName	Type	Length		Decimals	Index
>	1	name	Employee	char	30	0	no	
>	2	spouse	Employee	vastint	0	4	no	
>	3	empNum	Employee	longint	0	0	no	

[1] In ZIM, char and alpha fields are identical, except for their treatment when sorting or indexing. For char, case is respected while sorting, while for alpha, case is ignored.

[2] Variable length fields in ZIM are implemented as the following C structure:

```
{
short    length;
char     string[x];
}
```

where x is the maximum length of the field, as specified in the schema entry. When the record is actually written to the database, the field is compacted to the length specified in the structure.

would translate into the following record structure:

```
{
char            null1;
char            name[30];
char            null2;
double          spouse;
char            null3;
long            empNum;
}
```

The 'null' fields are added by ZIM to allow for the storage of explicit nulls for the field values of entities

Each ZIM database file is assigned a unique four digit number when it is created by the DBMS. This number is included in the operating system filename; for example, in UNIX, a ZIM entity could be stored in the file 'zim0105'. This database file number is used by all of the PLI routines which operate on files. For example, to open a ZIM database file using the PLI, the program must know its corresponding number.

ZIM Multi-User Support

Support for concurrent access in ZIM is quite high-level. Using the PLI, the programmer can perform the following functions: start a transaction, abort a transaction, and commit a transaction. Transactions may not be nested: multiple calls to the start transaction function results in only one transaction, which is terminated by the first call to either the abort or commit function. Two-phase locking is done implicitly by the transaction manager, so there is no programmer control over their placing. There is also no control over the granularity of locks: they are maintained only at the page level. As pages are accessed during a transaction, they are locked automatically by transaction manager, and then released when the transaction either commits or aborts Deadlocks are indicated to the application program via an error return code, which may be returned by any function which modifies the database.

More on Objective-C

Objective-C was described briefly in Chapter Two, as an example of an object-oriented language. Recall that it is a *hybrid* language. The *object-oriented methodology* has been grafted onto the imperative C programming language. As a result, it lacks some of the elegance of the Smalltalk language. For example, not all data items in Objective-C are objects: a typical program could include a mix of C primitive data types, C structures, and objects. Furthermore, while the message-passing computational metaphor is supported, C function calls are also allowed, and the two may be mixed freely. Objective-C is, in fact, a C superset. It is implemented as a front-end processor to the standard C compiler, along with some run-time support routines. As a result, it contains all of the functionality of the C language, with additional support for the object-oriented methodology.

Although Objective-C lacks some of the features of the Smalltalk system, it does share some of the same philosophies. For example, the Objective-C compiler comes with a class library which contains many of the basic tools for software development, such as implementations for sets, collections, arrays, points and rectangles. In keeping with the open-system concept of Smalltalk, all of the source code for these classes is provided, and they may be easily modified. Furthermore, the source code for the message-passing, memory management and run-time support routines is also provided.

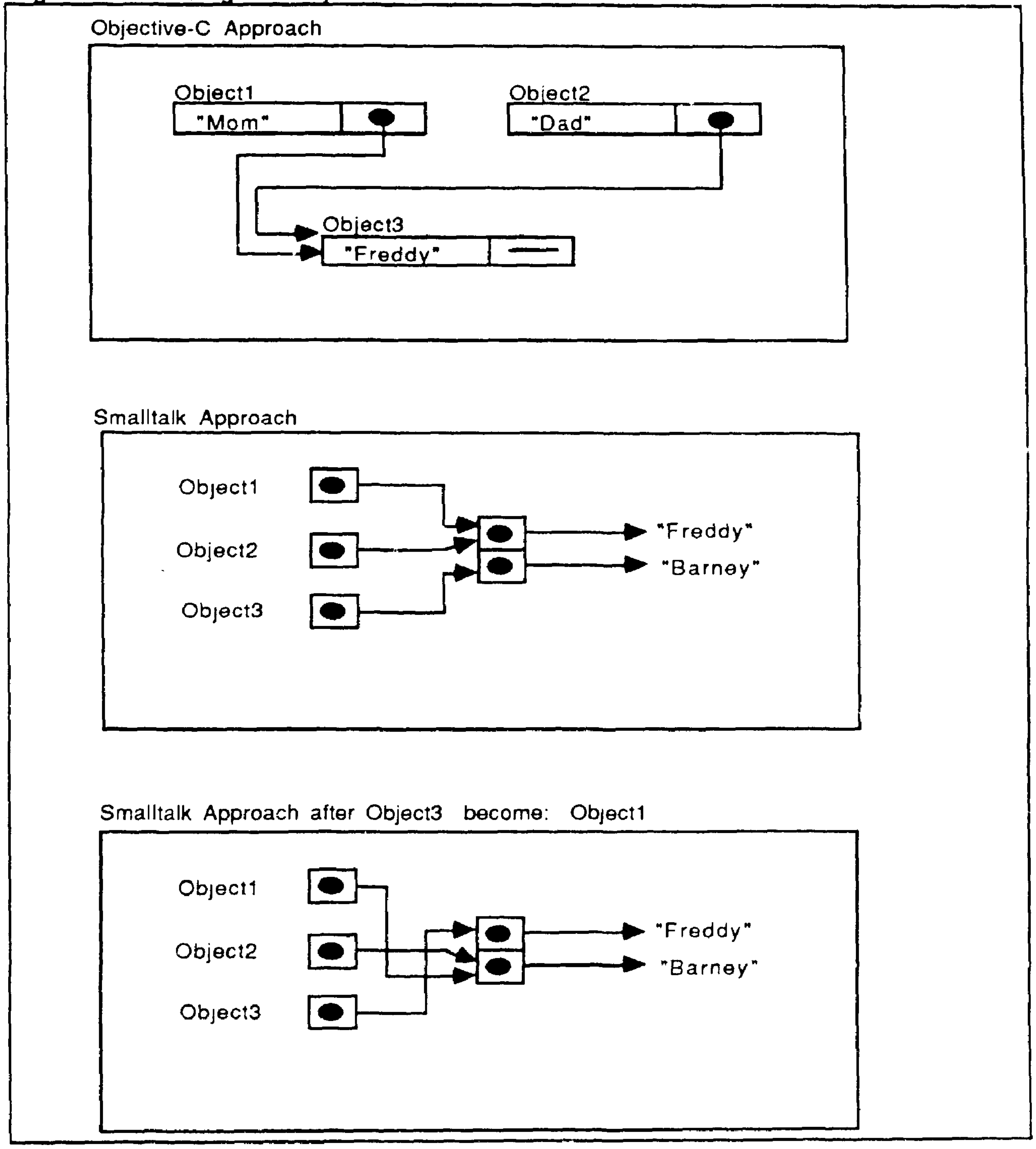
The following sections identify certain aspects of the product that had an impact on the design and implementation of the object server. The topics addressed include: how objects are represented in the language, how memory is managed, how inheritance is supported, how the message-passing computational metaphor is implemented, and the persistent object mechanism provided with the language, and why they were insufficient for the requirements of ARTTisan

Data Representation of Objects

Objects are added to the C language through the addition of one data type: the *id*. Id's are basically pointers to C structures which have been declared using the Objective-C compiler. They are unlike typical C pointers, however, in that they are essentially untyped. The type of an object is known only by itself. Using C pointers to reference objects directly has a number of advantages: it is simple to implement, all of the standard C pointer operations apply, and since there can only be one object per address, they fulfill the identity property of objects.

Using ordinary C pointers to refer to objects, however, poses a problem. In Smalltalk, references to an object are via object-oriented pointers (OOP's). Pointers to objects are actually indices into an object table. The contents of each table entry contains the actual location of the object in memory. Under this approach, there is a layer of indirection between the object and references to it. All objects that refer to the same object share the same OOP to it. The main advantage to this approach is that the *become:* message can be supported. This message can cause an object to be coerced to become an entirely new object, <u>and all references to the previous object are automatically updated</u> to reflect that change. The example shown in Figure 6.1 illustrates the difference between the two approaches. Under the Objective-C approach, if Object1 ("Mom") and Object2 ("Dad") are instances of class Parent, then they may share a child named "Freddy". Smalltalk allows the same flavour of structure sharing, as is shown in the second box. Since Smalltalk uses an object table, however, it extends this functionality to support the *become:* message. For example, say Object1 and Object2 want to swap their child with Object3's child. In other words, "Freddy" will become Object3's child, and "Barney" will become Object1 and Object2's child. Under the Smalltalk approach, it is possible to update all references to an object

Figure 6.1: Using an Object Table



using the become message. Under the Objective-C approach, each reference to the Object 3 would have to be found and updated in order to safely change the value of the object. In general, this cannot be done.

Objects and C Structures

As indicated above, objects in Objective-C are basically C data structures which follow a format known to the Objective-C compiler. What form this structure takes may be best illustrated using an example. Let us define a relatively simple class:

```
= Person : Object {
        char    *name;
        id      spouse;
}
```

This declares (in Objective-C syntax) the class Person, which inherits from class Object. It has two instance variables, name and spouse, of which spouse is actually a reference to another object. An instance of this class would be made up of the following C structure:

```
{
SHR     isa;
char    *name;
id      spouse;
}
```

The important thing to note is the addition of the *isa* pointer. This is a pointer to a structure which is shared by all instances of a class. It is the isa pointer which allows the type of an object to be discerned at run-time The structure referred to by isa has the following format (PPI, 1986):

```
struct _SHARED {
        struct   _SHARED        *isa;
        struct   _SHARED        *clsSuper;
        char                    *clsName;
        char                    *clsTypes;
        short                   clsSizInstance;
        short                   clsSizDict;
        struct   _SLT           *clsDispTable;
}
```

The structure that represents an object includes all of the instance variables defined in its class definition, and in the definition of all of its superclasses. For example, instances of the following class:

```
= Employee : Fruit {
        long    empNum;
}
```

which inherits from the class Fruit, would have the following structure:

```
{
SHARED          *isa;
char            *name;
id              spouse;
long            empNum;
}
```

The instance variables are in the order implied by the inheritance hierarchy.

Unfortunately, the object representation described above is not followed in all cases. This is especially true of the object classes provided with the Objective-C compiler. These are referred to as the Foundation Classes by the Objective-C documentation. Classes which have a non-typical representation include:

- **IdArray**: The class IdArray implements a class where the objects referred to are accessed as indexed array elements, rather than as named instance variables. The structure of an IdArray instance is therefore quite different from the normal case. For example, say you had an IdArray with ten elements, the resulting C structure would be:

```
{
SHARED      *isa;
unsigned    capacity;
id          contents[10];
}
```

Actually, IdArray instances do not declare an array for their contents explicitly. The memory for the array is declared when the object is allocated from the heap. For example, the above structure would be created by simply allocating 48 bytes of storage[1], and then using pointer offsets to index into the array of ids at the end of the structure.

- **Cltn**: The class Cltn (Collection), and its sub-classes, such as Set, Bag, OrdCltn, Stack, and Dictionary provide the various message protocols to implement these well-known data abstractions. They all maintain the objects that they associate in instances of the class IdArray. This, in itself, does not pose a problem. Unfortunately, though, these classes all violate the encapsulation of the IdArray instance that they reference. For example, rather than sending a message to obtain the i^{th} element of the array, they use offsets to point directly into the array.
- **String**: The class String is similar to IdArray, in that its instances allocate an area of storage which is not a named instance variable. In this case, however, it is characters which are being stored, rather than ids.
- **Rectangle**: The class Rectangle has an implementation which defies logic. Essentially, a rectangle is represented as two Point objects: origin and corner. However, rather than implementing the class so that it uses instances of Point in a normal fashion, the points

[1] The number of bytes is calculated as follows:

isa pointer	=	4
capacity	=	4
contents 10@4	=	<u>40</u>
		48

are embedded as structures directly in the Rectangle instance. For example, a reasonable implementation would have defined Rectangles as follows:

```
Rectangle : Object {
        id      origin,
                corner;
}
```

where origin and corner would be instances of class Point. Instead, Rectangles are implemented as follows:

```
Rectangle : Object {
        SHARED      *isa;
        SHARED      *isa1;
        int         originX,
                    originY;
        SHARED      *isa2;
        int         cornerX,
                    cornerY;
}
```

The two extra SHARED references are initialized to the Point class object.

Violating Encapsulation

One of the key concepts in object-oriented programming is that the state of an object is encapsulated. That is, an object may only be manipulated using the operations that it provides. Objective-C allows the programmer to get around this: objects can have their contents accessed directly, as an ordinary C structure. This is not an oversight in the design of the language; it was built into the Objective-C compiler as a feature to allow greater run-time efficiency for those pieces of an application which need it. The execution speed improvement comes from using C function calls instead of message sends, and accessing the contents of an object using pointers instead of messages. However, the dangers of violating encapsulation must be weighed against these greater efficiencies.

In order to gain access to the internal representation of objects of a certain class, the application program must specify a C data type which matches the definition of that class. To facilitate this process, Objective-C provides the compiler directive *@defs()*. For example, a programmer wanted to manipulate instances of the class Person. The following statement would define the C data type required:

```
typedef struct { @defs(Person) } PERSON_TYPE;
```

Then, by casting references to instances of Fruit to pointers of type PERSON_TYPE, the internal representation of Persons may be inspected and modified using ordinary C pointer manipulations. For example, the program could include the following statement:

```
((PERSON_TYPE *)aPerson)->name = "Fred";
```

where aPerson is an instance of class Person.

Not only does Objective-C allow the application developer to violate encapsulation, the Foundation Classes regularly violate this principle, as mentioned in the previous section. For example, the classes which implement the various specializations of collections (stacks, sets, ordered collections, etc.) maintain their contents as instances of class IdArray. IdArrays are arrays of pointers to objects. A strictly object-oriented implementation of these collection classes would require that any manipulations of their contents would be done using the message protocol of the class IdArray. Unfortunately, this was not done. The collection classes use all of the various pointer manipulation tricks available in the C language to access their contents. Encapsulation was violated in order to gain some efficiency.

Memory Management

The objects which represent both classes and their instances are really nothing other than C structures. However, there is a difference in how memory for the two is allocated. Class objects (referred to as *factory objects* in Objective-C documentation) are allocated statically, at compile-time. Their addresses are used to build a *class table* for the executable image, which is used by the language's run-time system.

Instances are allocated at run-time from the heap, using the standard C memory functions *alloc* and *dealloc*. There is, however, two layers of abstraction built around these calls. From the users' point of view, the first layer is a message protocol, implemented by class Object, which provides the basic memory operations to the developer. This message protocol uses the class method *new*, which allocates new instances of a class, and the instance method *free*, which deallocates an object. Note that there is no garbage collection in Objective-C. Objects which are no longer needed must be identified and returned to the heap explicitly by the programmer. Since it is not possible to know *a priori* the location of all references to an object, it is very easy to leave 'dangling pointers' to objects when they are freed. In order to help identify such bugs, when an object is freed, the Objective-C memory management routines set the object's isa pointer to nil before returning the object to the heap. The message-passing routines will raise an error if any message is sent to an object with a nil isa. Unfortunately, this will not help if the memory returned to the heap when the object was freed has been re-allocated before the offensive message send. Under this scenario, bizarre and catastrophic results are to be expected.

The message protocol for memory management calls a number of functions which are part of the Objective-C run-time support library. These functions: _alloc(), _realloc(), and _dealloc(), are not called directly. Instead they are called through pointers to them. For this reason, these function are generally referred to as (*_alloc), (*_realloc) and (*_dealloc). The names of the functions themselves remain hidden. As a result, it is straightforward to replace the memory management provided with the language with a new set of functions, which presumably extend their functionality.

The Message-Passing Mechanism

Objective-C implements the message-passing computational metaphor using a combination of compile-time and run-time facilities. When Objective-C programs are compiled, the methods defined by each class are transformed into ordinary C functions. For example, say that the class Person described above has a method defined as:

```
-name:(char *)aName {
        name = aName;
        return self;
}
```

This method would be compiled to a C function which looks like:

```
(char *)_1_Fruit(self, selector, aName)
        id      self;
        char    *selector;
        char    *aName;
{
        self->name = aName;
        return self;
}
```

The compiler maintains a table which maps the <class, message selector> pair to the the correct function.

Message sends are enclosed in brackets []. These are recognized by the compiler and translated into C function calls to either *_msg()* or *_msgSuper()*. The arguments to these functions are the object id receiving the message, the message selector, and the arguments to the message itself. The C definitions of these functions are as follows:

```
(id)_msg(receiver, selector, ....)
id      receiver;
SEL     selector;

(id)_msgSuper(class, selector, ....)
SHR     class;
SEL     selector;
```

where "...." indicates the additional parameters to the method itself whose number and type vary. For example, the message send:

```
[aPerson name: "Fred"];
```

would compile to the following function call:

```
_msg(aFruit, "name:", "Fred");
```
[1]

The _msg() function provides the basic message sending mechanism. It takes the receiver of the message, along with the message selector, and looks up the address of the correct function. Control is turned over to that routine, without modifying the stack. The _msgSuper() function is similar to _msg(), but it provides support for messages to *super*. Messages to super are treated as a special case, since by definition they can only be called from within an instance method. As a result, self is known to be at a certain location in the caller's stack frame.

Obviously, since a message send results in a search, (which may be proportional to the depth of inheritance) and the overhead of several function calls, they are more expensive than an ordinary function call. They are estimated to be 2 to 2.5 times as expensive as a function call (Cox, 1986). The message kernel has a number of features designed to improve execution speed. The first of these is that the _msg() and _msgSuper() functions are actually implemented

[1] The selector passed to the _msg and _msgSuper function is a pointer to the unique character string contained in the message table maintained for each class. This allows the method lookup routines to use an ordinary pointer comparison (==) to find the implementation, rather than a more expensive string comparison (strcmp).

in assembler, to ensure that they are executing as quickly as possible. The second is that they maintain a cache of the most recently called methods. The cache is maintained as a hash table, keyed on the <class, message selector> pair, which has two fields. The first is a marker which represents the pair, and the second is the address of the function which implements it. For each message that is sent, the hash value is calculated, and the marker is checked to ensure that it matches the current pair. If it does, it is immediately branched to; if it does not, _msg() or _msgSuper() calls the implementation lookup routine *_msgImpFind()*, and the implementation it returns is used to update that location in the hash table. For most applications, the cache hit ratio is 95% (Cox, 1986).

The function _msgImpFind() has the following C declaration:

```
(IMP)
_msgImpFind(refSelf, cls, selector)
id              *refSelf;
SHARED          *cls;
char            *selector;
```

refSelf is a pointer to the object (self) that was originally passed to _msg(). More specifically, it is a pointer to the location of self on the stack. Changing the value of refSelf to another object therefore effectively changes the object which is being sent the message.

If no implementation can be found which matches the selector of the original message send, an error occurs. The error-handling mechanism works as follows: if *_msgImpFind()* cannot find a match for the <class, selector> pair passed to the routine, it sends the message "*doesNotUnderstand: selector*" to the receiver of the message. The default mechanism provided by the class Object results in an fatal error, and the program terminates with a suitable message displayed to the user. In Smalltalk, this mechanism has been used by a number of researchers to provide support for persistent and distributed objects (McCullough, 1987), (Bennett, 1987), and (Merrow and Laursen, 1987). The method "*doesNotUnderstand: message*" has been modified to trigger a read to the object base, for example, when a particular class of object receives a message. Unfortunately, this approach cannot be used with Objective-C. The reason lies in the different parameters to the doesNotUnderstand: method. In Objective-C, all that is passed is the message selector originally sent to the object. In Smalltalk, the entire message is passed. The message is an object which contains both the message selector and all of the parameters passed with the selector. As a result, in the Smalltalk implementation, if the correct receiver object can be found by the doesNotUnderstand: method (by reading it from the object base, for example), the original message can then be sent to it.

To illustrate the above, consider the following example. In both Smalltalk and Objective-C, you have implemented a class called NetworkServerObject, whose sole purpose is to forward any message its instances receive to some Network object. The Network could be, for example, a facility to support distributed objects. In Smalltalk, this would be a straightforward exercise. Just implement a doesNotUnderstand: method for class NetworkServerObject which looks like:

```
doesNotUnderstand: message
        ^Network perform: message selector
                withArguments: message arguments
```

So, for example, the message

```
aNtwkObj keyWord1: parm1 keyWord2: parm2.
```

would be transparently forwarded to the Network object. In Objective-C, you have a problem Since all that is passed to the doesNotUnderstand: is the selector of the message (in this case "keyWord1:keyWord2:"), the parameters are lost. They are somewhere on the stack, but there is no mechanism provided to the applications developer to access them easily.

Just as it is possible to circumvent the encapsulation of objects in Objective-C, it is also possible to bypass the message-passing mechanism. This is done by acquiring the address of the C function which implements a method for a particular class. A message protocol for doing so is provided by the class Object. This can provide some performance improvements when the type of the objects to be addressed is known in advance, with the danger that bizarre results may occur if the function is called with another type of object.

Object Persistence Mechanisms Provided by Objective-C

Objects may be stored on disk using the AsciiFiler routines provided with the language. Under this approach, a complex object, and all of the objects reachable from it may be written out to an ascii file. The internal C representations of all of the object classes defined to the Objective-C compiler have a corresponding ascii representation that is known to the AsciiFiler class.

The protocol provided by the AsciiFiler is based on two messages: storeOn: and readFrom:. The storeOn: message takes a file name as its parameter, and when sent to a complex object, stores the graph of all objects reachable from it in the specified file. The readFrom: message provides the opposite functionality, reading in an object graph from a specified file.

Why They Are Not Enough

If Objective-C already provides a method to save complex objects to disk, what is the motivation for creating an object server for the language? The following are the features which extending the language with database-like facilities is expected to provide:

- The granularity with which objects can be saved and restored is much finer. Under the AsciiFiler approach, only entire object graphs may be made persistent. Therefore, while large objects can be saved into the database, only those pieces which are actually being used by the application need to read into memory.

- Objects which are shared between a number of complex objects are stored only once, thus increasing the integrity of the persistent data. Under the AsciiFiler approach, it is possible to store separate copies, in a number of different files, of an object which is shared. This leads to the possibility of having inconsistent versions of the same logic il object stored in in different files. Basically, once an object has been written to disk using the AsciiFiler approach, it loses its property of identity. The object is only unique within the file in which it is stored, rather than being unique throughout the entire problem domain.
- Much of the work performed by the AsciiFiler is devoted to converting the format of the objects from their internal C representation to some ascii representation. This is avoidable using an object server which supports the primitive C data types.
- Indices on objects will speed their recovery from th^ object base. However, this causes a corresponding penalty when adding and updating objects, due to the overhead of maintaining the indices.
- An object server offers a number of database amenities such as: transactions, indexing and multi-user concurrency control. These will be required to support any CASE environment which intends to meet the needs of development teams, as opposed to individuals.

Linking Classes with the Application

Objective-C is a C super-set. Ultimately, the Objective-C code is changed to ordinary C statements and then compiled to an executable program. Like any other C program, an

Objective-C application must have a function *main()*; in addition, a table of all of the classes used by the application are required to support the run-time messaging kernel. A typical Objective-C program will therefore be made up of a number of files which specify the classes being implemented, and then one additional source file which declares main, and defines the *class table*. The class table maintains the static address of every class used by the application. In order to create this table, the classes which are being used by the application must be linked, and their locations added to the table during compile-time. Since the type of an object is not resolved until run-time, the compiler must be explicitly told which classes to link. There are two mechanisms provided by the Objective-C compiler to do this. These are the @requires statement and the @classes statement.

The @requires statement is placed in the source code of a class definition and specifies the other classes which are to be used by the one being specified. The compiler will then ensure that the classes specified are made available for messaging. Any applications which include the class being defined will also include the classes mentioned in the @requires. For example, if you were defining a class MyClass which was to send messages to the classes String, Dictionary and Set, the definition would look like:

```
@required String, Set, Dictionary;

= MyClass : Object
        {list of instance variables}
```

The @classes statement is placed in the same source code file that contains the function main. The list of classes specified in the @classes statement act as roots of class usage trees, which the compiler uses to build the class table. The compiler will link all of the classes specified in the list, their superclasses, and the classes which they directly use, as specified in their @requires statement. For example, the statement @classes(MyClass) would cause the classes MyClass, Object, String, Set, and Dictionary to all be included in the executable image.

6.2. The Object Server/Language Interface

In the previous chapter, the manner in which objects are represented in the database was described. However, since we are building an object server for the Objective-C language, the next questions are: how are persistent objects represented to the Objective-C application? In addition, how is the relationship between an object in memory, and its representation in the database maintained?

The design of the object server may be described largely in terms of the implementation of two classes: Proxy and ObjectManager. The Proxy class will be described in detail in the following section. The ObjectManager provides the interface between object server and the Objective-C application. It maintains a cache of all of the persistent objects accessed by the application. It provides the capability to read and write persistent objects. In doing so, the ObjectManager utilizes metadata which describes an object's representation, both as a database record and an Objective-C object. In general, it is responsible for communicating between the application and the ZIM PLI routines which provide low level access to the object base. The ObjectManager is described in detail in the following chapter.

Proxy Objects

Persistent objects in the object base are mapped to dynamic Objective-C objects using instances of the class *Proxy*. Proxies are placeholders which represent persistent objects. Proxy objects are similar to the Agent objects described in (Purdy, Schuchardt and Maier, 1987). They are a packaging of the persistent object in the address space of the application; they provide for the caching of objects which have been accessed; and they provide a seamless integration of the application code and the object server.

Proxy objects provide locational transparency: the application program does not know if a persistent object is in memory or on disk. As a result, the programmer does not have to deal with the problems of caching and communicating with the object server. When a message is sent to a Proxy object, the system is responsible for ascertaining whether the persistent object the Proxy represents is already in memory, reading it in if it is not, and then forwarding the message to the real object for execution. This occurs transparently to the application.

When an object is read from the object base, all instance variables which are themselves objects are initialized as instances of class Proxy. Note, however, that a persistent object has only one Proxy object within an application. Multiple references to the same persistent object point to the same Proxy object. The ObjectManager maintains a hash table of all of the active Proxies in order to provide this feature. The ObjectManager returns a Proxy for the object base *root object* when it is opened. Every object in the object base is reachable from this root.

A persistent object is read into memory when its Proxy object first receives a message. This process is illustrated in Figure 6.2 a & b. In this example (which is based on the previous example shown in Figure 5.2), the object representing Fred le'Janitor has been previously sent a message by the application. As a result, the object has been read into memory. Note that it is actually referenced via a Proxy object, which knows the value of Fred's database OID, and points to the object that represents Fred. Fred's spouse instance variable also points to a Proxy instance. However, since no message has yet been sent to the object representing Molly le'Janitor, that Proxy object has nil as the value of its realObject. Once a message has been sent to Fred's spouse, Molly's object is read into memory as well.

The class Proxy has the following Objective-C definition:

```
= Proxy : Object {
        double          oid;
        id       realObject;
        char     beenHere;
}
```

Where oid contains the database surrogate for the persistent object; realObject contains the object being represented by the Proxy; and beenHere is used when creating graphs of persistent objects when writing them to the database. Note that the realObject variable is set to nil if the object has not been read into memory. Intuitively, Proxy objects are an <oid, object> pair. They maintain the mapping from the persistent object's database identifier, and the real object which it represents

The Proxy class has no instance methods. This is similar to the approach taken by (Purdy, Schuchardt and Maier, 1987). However, in their implementation, messages to Agent objects resulted in the *doesNotUnderstand* message being invoked. In the approach taken here, the message-sending routines of the Objective-C language have been altered to intercept messages sent to Proxies. As a result, it is impossible for a Proxy to receive a message.

Figure 6.2 a): Reading an Object (Before)

```
= Person : Object {  // ZIM file 101
      char       name[30];
}
= Employee : Person {  // ZIM file 102
      int        empNum;
      id         spouse;
}
```

Situation 1: The object representing the employee Fred le'Janitor has been read in previously. At that time, its 'spouse' instance variable was initialized to a Proxy object, whose realObject pointer is nil, since Molly le'Janitor is still on disk. Note that Fred's object is referenced via a Proxy object as well

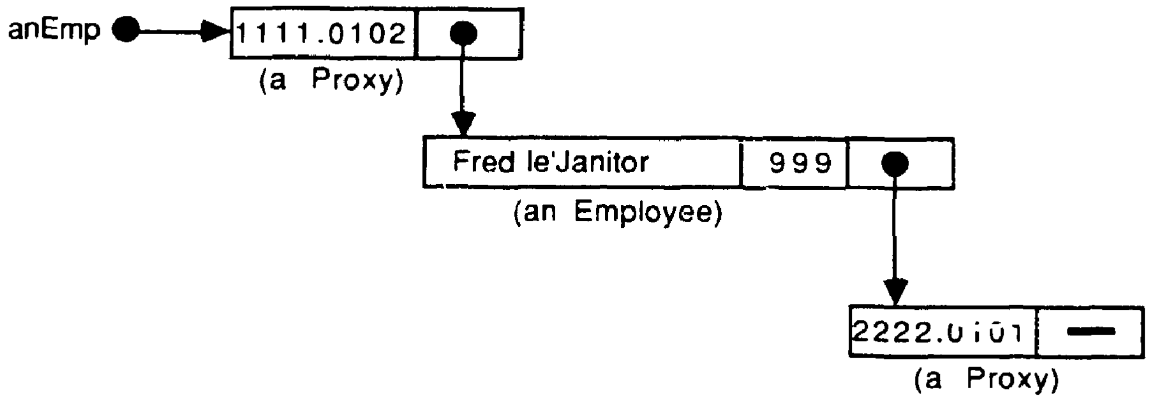
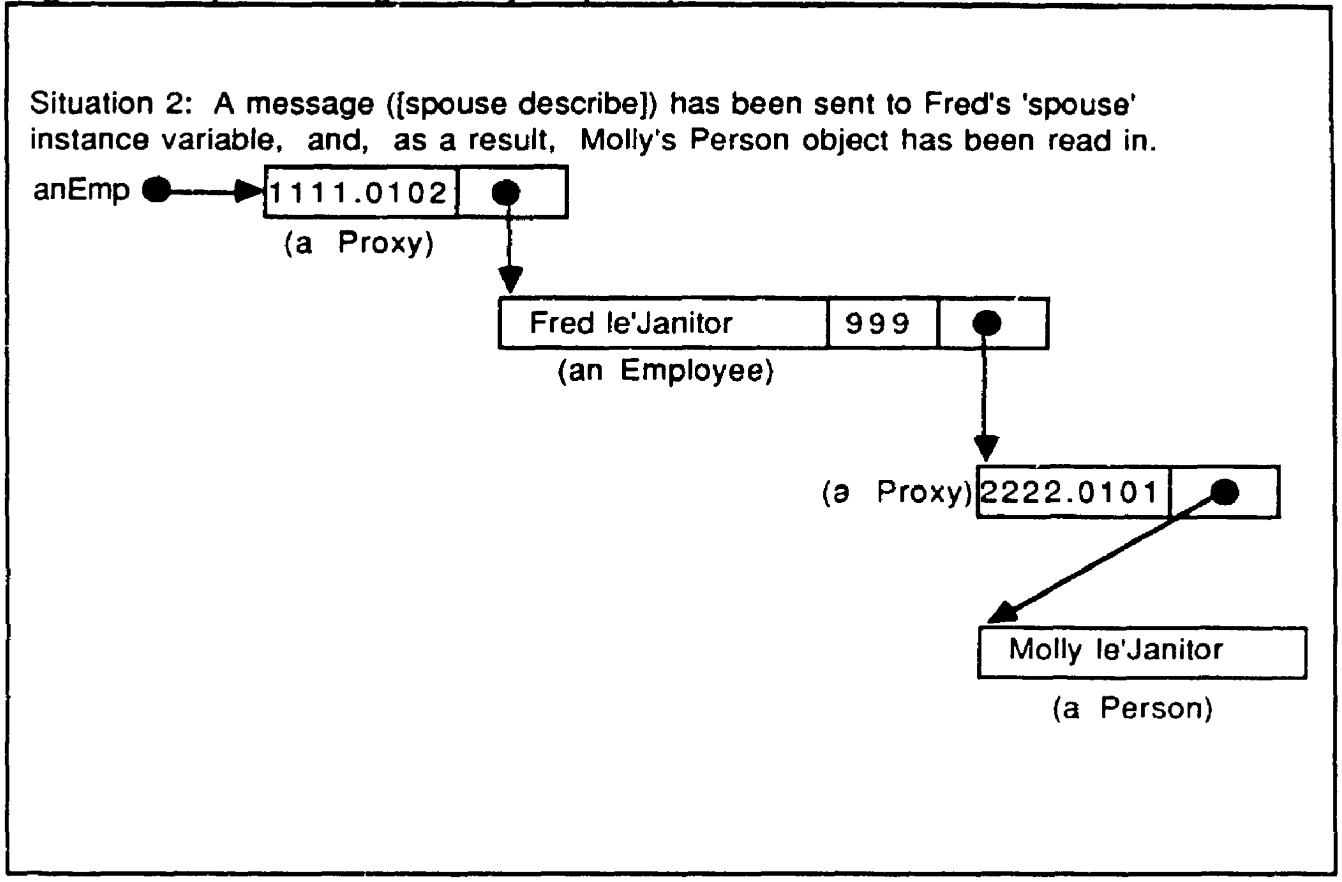


Figure 6.2 b): Reading an Object (After)


Situation 2: A message ([spouse describe]) has been sent to Fred's 'spouse' instance variable, and, as a result, Molly's Person object has been read in.
anEmp
1111.0102
(a Proxy)
Fred le'Janitor
999
(an Employee)
(a Proxy)
2222.0101
Molly le'Janitor
(a Person)

Intercepting Messages: Changes to the Objective-C Messaging Kernel

Recall from the previous chapter that in Objective-C, the methods which implement a class's protocol are compiled to ordinary C functions. When a message is sent at run-time, the messaging kernel looks up the address of the function which implements the method, and branches

to that location. The Objective-C messaging kernel is made up of three functions: *_msg()*, *_msgSuper()*, and *_msgImpFind()*. Message sends in an Objective-C program compile to calls to either _msg() or _msgSuper(). _msg() and _msgSuper() are responsible for finding the address of the function which implements the method, and branching to that location The most recently used methods are maintained in a cache shared by the two functions, which are written in assembler. if the method being called is not found in the cache, then _msgImpFind() is called to search for it. The value returned by _msgImpFind() is used by the calling function to update the cache.

In order to meet the goal of locational transparency of persistent objects, a number of changes were made to the Objective-C message-passing kernel. Before describing those changes in detail, let us turn to the end-result; what happens when a message is sent to a persistent object? To aid us, let us extend the previous example using the classes Employee and Person. Let the class Person implement the following method:

```
-describe {
        printf("%30s", name);
}
```

and the class Employee implement the following method:

```
-describe {
        printf("%30s %d", name, empNum);
        [spouse describe];
}
```

What happens if the message "describe" is sent to the (Proxy) object "anEmp" shown in Figure 6.2? In other words, [anEmp describe] is performed. The sequence of events are as follows:

- First, it must be realized that the application does not have a handle on the persistent object directly. Instead, references to the persistent object are to its unique Proxy instance. AnEmp is pointing to a Proxy, rather than to an Employee. Note that since Proxies are objects, they all reference the same class object through their isa pointer. As a result, it is possible to quickly determine whether or not an object is a Proxy be examining its isa value.

- Message sends result in a call to the function _msg(). This routine checks if the object being sent the message is a Proxy by checking its isa pointer. If it is a Proxy, the routine _msgImpFind() is called immediately. If not, the regular messaging routine is continued with. In this case, since anEmp is a Proxy, the former case holds. Note that _msgImpFind() may be called to service messages to non-persistent objects as well.
- The first action done by _msgImpFind() is to check whether the object being sent the message is a Proxy object. Recall that _msgImpFind() has the following definition:

```
(IMP)
_msgImpFind(refSelf, cls, selector)
        id              *refSelf;
        SHARED          *cls;
        char            *selector;
```

where refSelf is a pointer to the object (self) that was originally passed to _msg(). In this case, (*refSelf) is anEmp, which is a Proxy object. If refSelf points to a Proxy object, refSelf must be changed to the value of the Proxy's realObject. If realObject is nil, then the ObjectManager must be called to read the object from the database. The actual code is as follows:

```
if ((*refSelf)->isa == _ProxyIsa) {
        if (((PROXY_TYPE*)(*refSelf))->realObject != nil)
                (*refSelf) = ((PROXY_TYPE*)(*refSelf))->realObject;
        else {
                ((PROXY_TYPE*)(*refSelf))->realObject =
                        [ObjectManager get: (*refSelf)];
                (*refSelf) = ((PROXY_TYPE*)(*refSelf))->realObject;
        }
}
```

Note that refSelf is pointing to the location of the message receiver on the program stack. When the object pointed to by refSelf is altered, the receiver of the message has been altered. In this case, the Employee object referred to by anEmp is already in memory. As a result, the value of refSelf can be changed using one pointer manipulation. Access to persistent objects which have already been read is therefore quite efficient.

The implementation of the "describe" method for the Employee class is found, and a pointer to it is returned to the _msg() routine.

- _msg() takes the function address returned by _msgImpFind(), and branches directly to that location. In this case, first the C library function printf is performed, and then the message "describe" is sent to the Employee's spouse. The message-passing routine starts all over again. The process is identical to the one described above, with one notable exception. At the time of the message send, the Employee's spouse has not yet been read into memory. As a result, the ObjectManager is requested to do so. The value of the realObject instance variable of the spouse's Proxy is changed to point to the object read. Note that since each database object may have only one Proxy object within an application, this change is reflected by all other references to the spouse object, if any.

Polymorphism and the Message Cache

The previous example may also be used to illustrate a subtle, but potentially dangerous facet of how messages to Proxies must be handled. Recall that the functions _msg() and _msgSuper() share a cache of all of the recently accessed methods. This cache is implemented as a hash table, whose key is the <class, selector> pair which uniquely identifies a method. When the method is not found in the cache, the _msgImpFind() routine is called, and the function address returned by it is used to update the cache slot.

Consider what would happen if polymorphic behaviour was expected of objects represented by Proxies, if the message cache was used as normal. Figure 6.3 illustrates the point. In the previous example, the "describe" message was sent to the Proxy objects representing an Employee and a Person. The Employee's Proxy receives its message first. In the _msg() function, the <class, selector> pair was <Proxy, "describe">. Assume that the cache slot for that pair was empty, so the _msgImpFind() routine was called to look up the address of the implementation. The function address returned _msgImpFind() was the method which actually matches the pair <Employee, "describe">. The address of that function was placed in the cache slot, and the method was then executed.

The "describe" instance method implemented by the Employee class sends the message "describe" to the spouse object which, in this case, is expected to be a Person. Note what would now happen in the _msg() function. Once again, _msg() is called with the <class, selector> pair <Proxy, "describe">. This time, however, a match is found in the cache, and that function is branched to. Unfortunately, the wrong function has just been called, since the implementation of "describe" for the Employee class was found, rather than the implementation for the Person class. Bizarre and dangerous results would result if this problem was not addressed.

In Figure 6.3, then, we see the following:

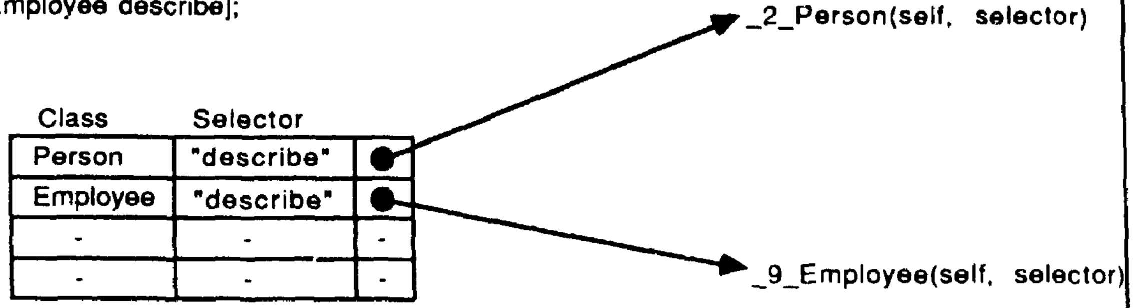
- In situation (A) - which is the normal Objective-C situation - we see that the two classes Person and Employee each have their "describe" method placed in a separate slot in the message cache.

- In situation (B) and (C), we have a more dangerous situation, since both the Employee and Person objects are, in fact, represented by Proxy objects. This confuses the normal message-passing and caching mechanisms.

Figure 6.3: The Objective-C Message Cache

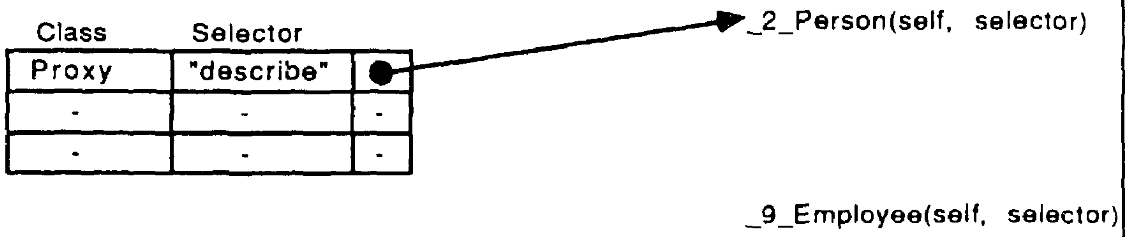
Objective-C Message Cache:

(A) [aPerson describe];
[anEmployee describe];



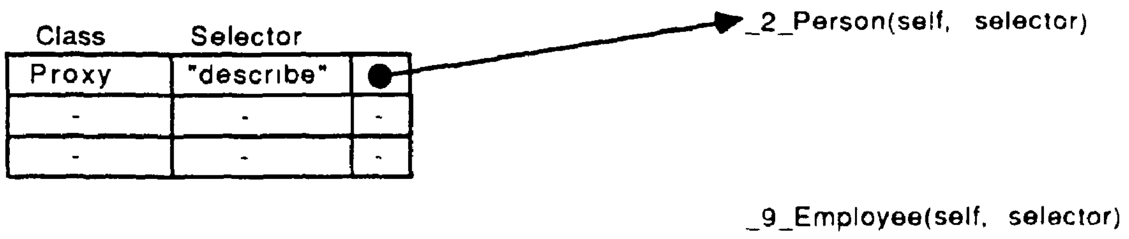
(B) [aPerson describe];
Note that aPerson is actually a Proxy object.

The message "describe" has been sent to a Proxy representing a Person. The cache is updated with the function address returned for the <Proxy, "describe"> pair.



(C) [anEmployee describe];
Note that anEmployee is actually a Proxy object.

The message "describe" has been sent to a Proxy representing an Employee. Since a <Proxy, "describe"> pair is found in the cache, that function is executed and an error results.



In order to handle to handle this problem, the _msg() function has been altered to identify messages to Proxy objects. When such an occurrence is detected, the cache lookup code is bypassed. Instead, the _msgImpFind() routine is called, and the control is turned over to the function address it returns. The return value of _msgImpFind() is not used to update the cache, as is the normal case. As a result, messages to 'ordinary' objects continue to utilize the message cache, while messages to persistent objects always require a call to _msgImpFind() in order to find the address of the specified method. This has performance repercussions, since the search undertaken by _msgImpFind() may be proportional to the depth of the inheritance tree of the receiver object. An 'industrial strength' implementation could maintain a separate cache for Proxy object messages; or, alternatively, the <class, selector> pair used to hash into the cache could be created using the class of the realObject pointed to by the Proxy object.

The Dangers of Violating Encapsulation

One of the features of the Objective-C language is that it allows the violation of the encapsulation principle inherent in the object-oriented paradigm. It is possible to get a handle on the internal state of an object directly. A purist approach, such as the one taken in the Smalltalk language, demands that all object manipulations be done using the operation provided by the object itself. With respect to the implementation of an object server for the language, this approach is a double-edged sword. On one hand, being able to violate encapsulation was a great help in implementing the class Proxy. Since it is impossible to send a message to a Proxy object (all messages are passed along to the persistent object represented by the Proxy), the only way for the object server implementation to modify or access Proxy objects was to do so directly. Essentially, Proxy objects are treated by the object server implementation much like a regular C structure.

On the other hand, applications which insist on violating the encapsulation of persistent objects must be very careful, or disaster will result. Since references to persistent objects are done so via Proxies, any attempt in the application code to point directly to the instance variables of a persistent object must recognize the existence of Proxies, or it will be using incorrect offsets Instead of pointing to an object of class Employee, for example, and offsetting to gain access to the empNum variable, the program would be pointing to a Proxy object, and the same offset would be pointing at the value of the 'realObject' instance variable within the Proxy.

Typically, when an Objective-C program violates the encapsulation of an object, it does so by casting the parameters of a method to be of the expected type. Figure 6.4 provides an example code fragment which shows how a method which violates the encapsulation of an object would have to deal with the Proxy interface. This code would have to replicated in every method which attempted to deal with objects directly.

Figure 6.4. Dealing With Proxies Explicitly

```
@requires SomeClass, ObjectManager, Proxy;
typedef struct { @defs(Proxy) } PROXY_TYPE;
typedef struct { @defs(SomeClass) } SOME_TYPE;
// The method casts its parameter to be an instance of a specific class.
-aMethod:(SOME_TYPE *)parm1 {
        // First, check to see if the parameter is a Proxy object
        if (parm1->isa == Proxy) {
                // Has the object been read from the object base?
                parm1 = (PROXY_TYPE *)parm1;
                if (parm1->realObject == nil)
                        parm1 = (SOME_TYPE *)parm1->realObject;
                else {  // Read the object into memory
                        parm1->realObject = [ObjectManager get: parm1];
                        parm1 = (SOME_TYPE *)parm1->realObject;
                }
        }
        /* Then the method may continue normally.... */
```

7. The ObjectManager

Any application which wishes to use the object server must link the Objective-C class ObjectManager. The ObjectManager provides the major components of the object server's functionality. These include:

- Transferring objects from memory to disk, and vice versa. In order to do this, the ObjectManager accesses metadata which describes the representation of objects.
- Caching objects which have been read from the database. The cache is maintained as a hash table.
- A message protocol which allows the application to start and end ObjectManager sessions.

7.1. The Object Schema

Before the inner workings of the ObjectManager can be described, the concept of the object schema must be explained. The object schema provides the 'data about the data' - or metadata - that the system requires to manipulate the persistent objects.

The format of the metadata required by the object server was largely motivated by the format maintained by ZIM. Recall that a ZIM database schema is described largely in terms of the two entity sets: EntitySets and Fields. The object server schema is described largely in term of two classes: Class and InstVar. How the object server metadata is used to maintain the ZIM database schema is discussed at the end of the chapter.

Defining a Class

The object server must maintain information which allows it to describe both the ZIM and Objective-C represe⁻ᵗation of an object. Since the ZIM DBMS maintains records which are essentially C structures which have been written to disk, why are these representations different?

- ZIM places a 'null' byte before the beginning of each field in the record. This is used by ZIM applications to store explicit null values in the database. These null bytes must be removed upon reading the object from the database, and inserted when the object is written.
- Each instance in the object server must contain that object's OID. This value is kept in the first field in the record which represents the object in the database. When the object is read by an Objective-C application, this OID is used to find the object in the database. Once the object is read, the OID is maintained by the object's Proxy. It is not contained in the object itself.
- Objects in Objective-C have, as their first instance variable, their *isa* pointer. Isa points to the object's factory object. This pointer must be initialized when the object is read, and removed when the object is written.
- References to other objects are handled differently in the persistent and dynamic representations of the object. In Objective-C, object references are via pointers of type *id*. These require four bytes of memory. Persistent objects refer to each other via OIDs, which require eight bytes of storage. When an object is read, its references to other objects are represented by a Proxy instance. When an object is written, its references to other objects are represented by the OID stored within the Proxy.

Recall that all objects are unique instances of some class. This means that the object server must know the classes that it is expected to maintain, and their format. To do so, the class Class is used. Instances of Class correspond one-to-one with the Objective-C classes for which the toolset designer wishes to maintain persistent instances. The class Class is itself an Objective-C class which is known to the ObjectManager. As we shall see, it is an example of a class which has several instance variables which are not persistent. It is defined as follows:

```
= Class : Object {
        id      name,
                superClass,
                instVars,
                factoryObject,
                subClasses,
                dbSet,
                getSel,
                putSel;
        short   classNum;
}
```

Where the instance variables have the following meaning:

name	- The String object which contains the name of the Class (ie. "Object").
superClass	- The Class object which specifies the class's superclass.
instVars	- The ordered collection of all of the instance variables which make up the class. The specification of the class InstVar will be described below.
factoryObject	- The location of the actual Objective-C factory object which is used to create new instances of the class. The factoryObject is not a persistent object, since it is only meaningful at run-time.
subClasses	- The ordered collection of all of the class's subclasses.
dbSet	- An instance of class DBSet, which is used to read and write instances of the class to and from the underlying ZIM database. dbSet is not a persistent object, since it is only meaningful at run-time.
getSel	- The String object which represents the selector which is to be sent to the Class instance in order to have an object of that class read from the database. This is needed to handle the accessing of classes which are a special case.

putSel - The String object which represents the selector which is to be sent to the Class instance in order to have an object of that class written to the database. This is needed to handle the accessing of classes which are a special case.

classNum - The ZIM file number of the database file which contains the class. All instances of the persistent class are held in this file.

The class InstVar is *crucial to understanding the object schema*. Class objects use InstVars to describe both the persistent and dynamic representation of their instances. The implementation of InstVar is as follows:

```
= InstVar : Object {
        id      name,
                partOfClass;
        int     type;
        BOOL    index,
                persistent;
        short   length;
}
```

Where the instance variables have the following meaning:

name - The String object which contains the name of the InstVar (ie. "name").

partOfClass - The instance of Class which this instVar is part of.

type - An enumerated type which indicates the data type of the instance variable (ie. "DB_ID", "DB_INT", "DB_DOUBLE", ...).

index - A Boolean variable which indicates whether or not this variable has a ZIM index based on its value.

persistent - A Boolean variables which indicates whether or not this variable is persistent. For example, in the class Class, dbSet would have a value of NO, while name would have a value of YES.

length - Used to indicate to ZIM how long a character array instance variable is

Supporting Inheritance

The inheritance tree of an object class is described explicitly in the metadata through the Class instance variable *superclass*. At the root of the class hierarchy is the class Object, which has a superclass of nil. Each Class maintains an ordered collection of the Classes which inherit from it. Thus the superClass and subClass links define the inheritance hierarchy An example inheritance hierarchy implemented in this fashion is shown in Figure 7.1. Each Class object maintains an ordered collection of its instance variables. It should be noted that the Class Object defines the instance variable "oid". Since all objects inherit from Object, this ensures that all persistent objects have this instance variable defined.

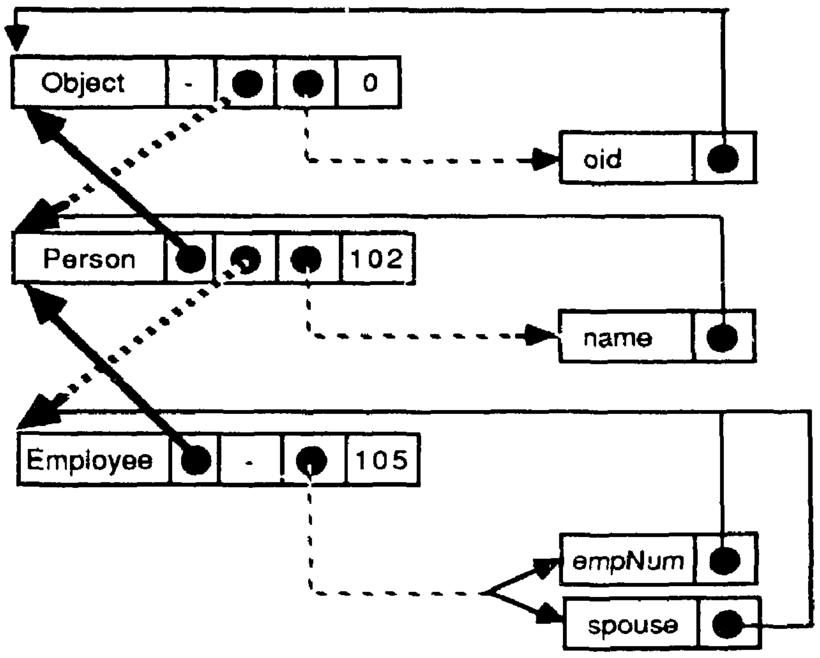
There is a message protocol to query a Class object about its instance variables, made up of two messages:

- *yourInstVars* returns the instance variables specified for this class. It ignores any instance variables defined in its superclasses. For example, the Person class object would respond to this message with a collection containing the InstVar instances for "name" and "spouse". The Employee Class object would return a collection containing only the InstVar instance for "empNum".
- *instVars* returns all of the instance variables defined for a class, including those defined in its superclasses. It is implemented in terms of *yourInstVars*. The array of InstVars returned has the superclass's instance variables first. For example, the Employee class would respond with a collection containing the InstVar instances for "oid", "name", "spouse", and "empNum".

For example, the instVars message would be used to get the list of instance variables which fully describe an object when reading it from the object base. The yourInstVars message could be used

Figure 7.1: Supporting Inheritance

```
= Person : Object {  // ZIM file 101
      char      name[30];
}
= Employee : Person {  // ZIM file 102
      int       empNum;
      id        spouse;
}
```



Key:

━━━▶	superClass
- - - -▶	subClass
───▶	partOfClass (for instance variables)
- - - - -▶	instance variable

in constructing a browser for the object schema, where you would wish to deal with only the instance variables defined in a particular class.

Metadata Access

The object schema is read from the object base by the ObjectManager in response to the *open* message. There are several implementation details to note regarding how this is performed:

- The code to read the object schema doe. not utilize the normal ObjectManager methods for reading objects, since that code relies on the metadata. You cannot use the metadata to read the metadata. Note that this does not imply that the object schema may not be manipulated by applications which use the object server. An example of such an application is the Browser described in the following chapter.
- The Class and InstVar instances which make up the object schema are not represented by Proxy objects within the ObjectManager. It has a direct handle on the persistent objects. This is safe since the ObjectManager never modifies the metadata. The Class and InstVar instances which are read when the ObjectManager is opened are discarded when it is closed. By allowing these objects to be handled directly by the ObjectManager, performance gains are realized, since messages to non-Proxy objects use the message cache.
- The root of the inheritance hierarchy is the class Object. This object has one special property that is hard-wired into the system: it has an Object Identifier (OID) of zero (0)[1].

[1] Actually, it has an OID of 0.0104, where 104 is the classNum of the class Class.

This allows the ObjectManager initialization code to read in that object from the database directly, and then read all of Object's subclasses recursively.

7.2. An Objective-C Protocol for the ZIM PLI

The ZIM PLI provides a number of C functions which may be used to access, add, change and delete records and index entries in ZIM database files[1]. Since all of the instances of an class are maintained in the same ZIM database file, these operations are the equivalent to adding, changing and deleting instances of classes.

Access to the database files is provided by two abstractions: the ZIM entity set (ZESET), and the ZIM index (ZINDEX). A ZIM entity set is essentially a pointer to a C structure which provides a reference to the contents of a ZIM database file. The pointer is returned by the PLI function which opens a file, and it is passed as a parameter to all of the PLI functions which manipulate the records in the file. The ZESET structure maintains a 'current record', which is essentially a pointer to a specific record in the file. This current record pointer may be moved explicitly using a 'get next record' function, or by using an index to locate a object which has a specific value for an instance variable.

Once an entity set has been opened, any indices which exist may also be opened. A set of PLI functions exists which will locate specific records in the file based on a specific value and a Boolean operator (ie. <, <=, =, =>, > or !=). New records may be added, and existing ones updated or deleted. Note that it is up to the program modifying the record values to update any associated indices. Failure to perform these index updates correctly could result in the corruption of the database.

[1] Additional PLI functions are provided to support session control, security, and concurrency control

There is one small, but important detail regarding the ZIM PLI which must be made explicit. When using the PLI. it is recommended that the application open and close the entity sets and indices very frequently. A typical sequence would be:

```
open entity set
open index1
open index2
locate a record using index1
modify the retrieved record
write the updated record
update index1
update index2
close index1
close index2
close entity set
```

It should be noted that closing the entity set has relatively little overhead associated with it. This is because closing the entity set does not normally result in the closing of the underlying file. ZIM maintains a ring of recently accessed entity sets, and a file is physically closed only if necessary. The number of open files allowed is a parameter which the user may control. Closing entity sets and indices does cause any modified buffers to be flushed back to the disk file.

A ZIM database file may be modelled effectively as a pointer to a ZESET structure, with an associated collection of indices. The class DBSet performs this role. Instances of class DBSet are created at run-time, as the different classes in the object schema are requested to read, write or modify their instances. The basic approach is that instances of DBSet are responsible for dealing with accesses to the ZIM database. They retrieve, add, change and delete instances of classes in the database. However, it is the Class objects which are responsible for translating between the persistent and dynamic representations of the objects. There is one DBSet instance for each active Class. DBSet has the following structural definition:

User:	michael
Application:	PrintMonitor
Document:	Michael's Thesis
Date:	Monday, November 14, 1988
Time:	10:21:17 PM
Printer:	3.A16.Plus.Telos.MacLeod,R

Figure 6.2 a): Reading an Object (Before)

```
= Person : Object {  // ZIM file 101
      char      name[30];
}

= Employee : Person {  // ZIM file 102
      int       empNum;
      id        spouse;
}
```

Situation 1: The object representing the employee Fred le'Janitor has been read in previously. At that time, its 'spouse' instance variable was initialized to a Proxy object, whose realObject pointer is nil, since Molly le'Janitor is still on disk. Note that Fred's object is referenced via a Proxy object as well

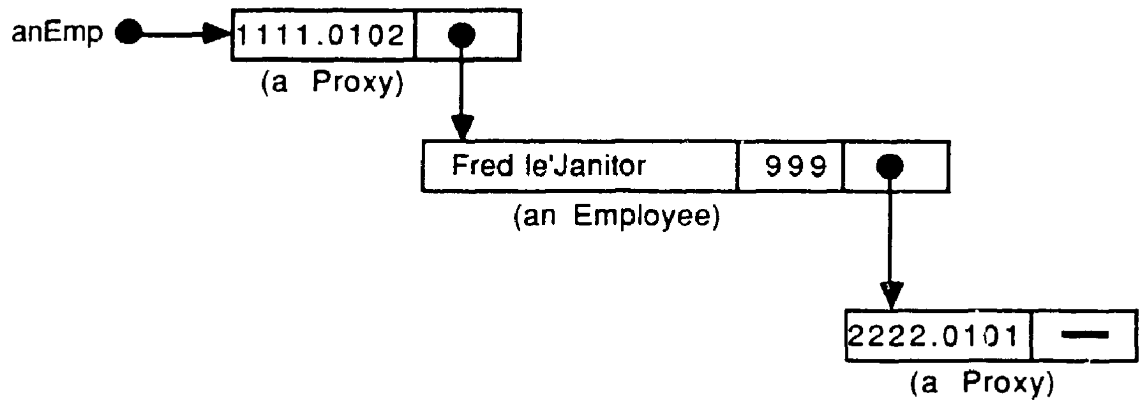
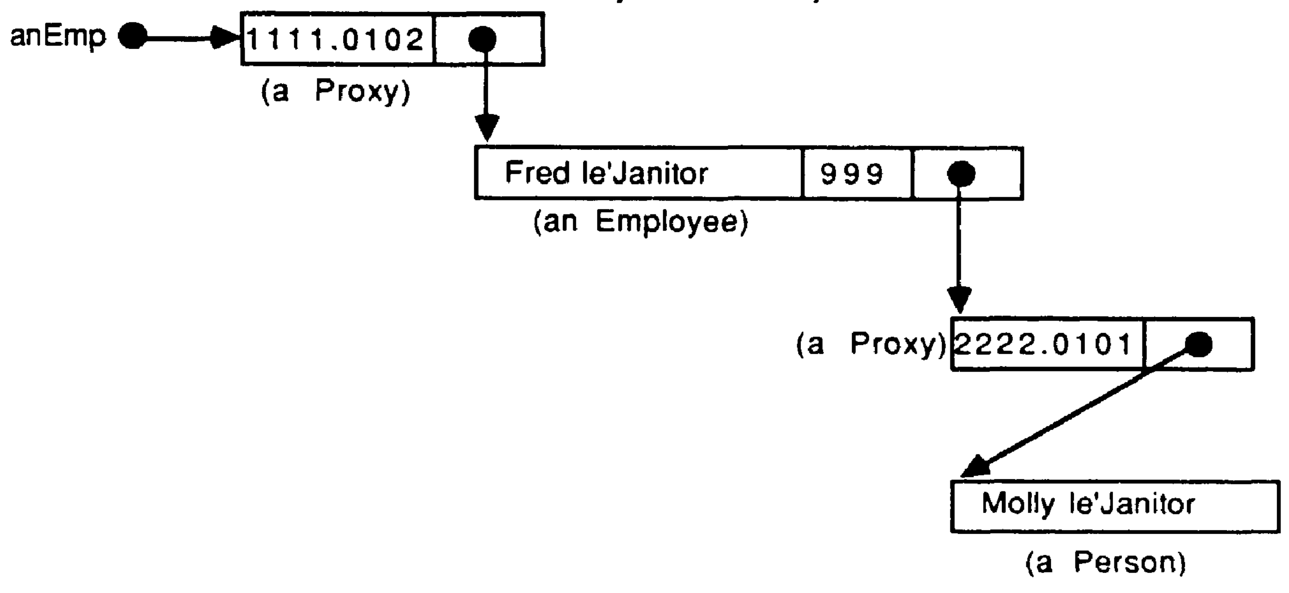


Figure 6.2 b): Reading an Object (After)

Situation 2: A message ([spouse describe]) has been sent to Fred's 'spouse' instance variable, and, as a result, Molly's Person object has been read in.



Intercepting Messages: Changes to the Objective-C Messaging Kernel

Recall from the previous chapter that in Objective-C, the methods which implement a class's protocol are compiled to ordinary C functions. When a message is sent at run-time, the messaging kernel looks up the address of the function which implements the method, and branches to that location. The Objective-C messaging kernel is made up of three functions: *_msg()*, *_msgSuper()*, and *_msgImpFind()*. Message sends in an Objective-C program compile to calls to either _msg() or _msgSuper(). _msg() and _msgSuper() are responsible for finding the address of the function which implements the method, and branching to that location. The most recently used methods are maintained in a cache shared by the two functions, which are written in assembler. If the method being called is not found in the cache, then _msgImpFind() is called to search for it. The value returned by _msgImpFind() is used by the calling function to update the cache.

In order to meet the goal of locational transparency of persistent objects, a number of changes were made to the Objective-C message-passing kernel. Before describing those changes in detail, let us turn to the end-result; what happens when a message is sent to a persistent object? To aid us, let us extend the previous example using the classes Employee and Person. Let the class Person implement the following method:

```
-describe {
        printf("%30s", name);
}
```

and the class Employee implement the following method:

```
-describe {
        printf("%30s %d", name, empNum);
        [spouse describe];
}
```

What happens if the message "describe" is sent to the (Proxy) object "anEmp" shown in Figure 6.2? In other words, [anEmp describe] is performed. The sequence of events are as follows:

- First, it must be realized that the application does not have a handle on the persistent object directly. Instead, references to the persistent object are to its unique Proxy instance. AnEmp is pointing to a Proxy, rather than to an Employee. Note that since Proxies are objects, they all reference the same class object through their isa pointer. As a result, it is possible to quickly determine whether or not an object is a Proxy by examining its isa value.

- Message sends result in a call to the function _msg(). This routine checks if the object being sent the message is a Proxy by checking its isa pointer. If it is a Proxy, the routine _msgImpFind() is called immediately. If not, the regular messaging routine is continued with. In this case, since anEmp is a Proxy, the former case holds. Note that _msgImpFind() may be called to service messages to non-persistent objects as well.
- The first action done by _msgImpFind() is to check whether the object being sent the message is a Proxy object. Recall that _msgImpFind() has the following definition:

```
(IMP)
_msgImpFind(refSelf,  cls,  selector)
        id              *refSelf;
        SHARED          *cls;
        char            *selector;
```

where refSelf is a pointer to the object (self) that was originally passed to _msg(). In this case, (*refSelf) is anEmp, which is a Proxy object. If refSelf points to a Proxy object, refSelf must be changed to the value of the Proxy's realObject. If realObject is nil, then the ObjectManager must be called to read the object from the database. The actual code is as follows:

```
if ((*refSelf)->isa == _ProxyIsa) {
        if (((PROXY_TYPE*)(*refSelf))->realObject != nil)
                (*refSelf) = ((PROXY_TYPE*)(*refSelf))->realObject;
        else {
                ((PROXY_TYPE*)(*refSelf))->realObject =
                        [ObjectManager get: (*refSelf)];
                (*refSelf) = ((PROXY_TYPE*)(*refSelf))->realObject;
        }
}
```

Note that refSelf is pointing to the location of the message receiver on the program stack. When the object pointed to by refSelf is altered, the receiver of the message has been altered. In this case, the Employee object referred to by anEmp is already in memory. As a result, the value of refSelf can be changed using one pointer manipulation. Access to persistent objects which have already been read is therefore quite efficient.

The implementation of the "describe" method for the Employee class is found, and a pointer to it is returned to the _msg() routine.

- _msg() takes the function address returned by _msgImpFind(), and branches directly to that location. In this case, first the C library function printf is performed, and then the message "describe" is sent to the Employee's spouse. The message-passing routine starts all over again. The process is identical to the one described above, with one notable exception. At the time of the message send, the Employee's spouse has not yet been read into memory. As a result, the ObjectManager is requested to do so. The value of the realObject instance variable of the spouse's Proxy is changed to point to the object read. Note that since each database object may have only one Proxy object within an application, this change is reflected by all other references to the spouse object, if any.

Polymorphism and the Message Cache

The previous example may also be used to illustrate a subtle, but potentially dangerous facet of how messages to Proxies must be handled. Recall that the functions _msg() and _msgSuper() share a cache of all of the recently accessed methods. This cache is implemented as a hash table, whose key is the <class, selector> pair which uniquely identifies a method. When the method is not found in the cache, the _msgImpFind() routine is called, and the function address returned by it is used to update the cache slot.

Consider what would happen if polymorphic behaviour was expected of objects represented by Proxies, if the message cache was used as normal. Figure 6.3 illustrates the point. In the previous example, the "describe" message was sent to the Proxy objects representing an Employee and a Person. The Employee's Proxy receives its message first. In the _msg() function, the <class, selector> pair was <Proxy, "describe">. Assume that the cache slot for that pair was empty, so the _msgImpFind() routine was called to look up the address of the implementation. The function address returned by _msgImpFind() was the method which actually matches the pair <Employee, "describe">. The address of that function was placed in the cache slot, and the method was then executed.

The "describe" instance method implemented by the Employee class sends the message "describe" to the spouse object which, in this case, is expected to be a Person. Note what would now happen in the _msg() function. Once again, _msg() is called with the <class, selector> pair <Proxy, "describe">. This time, however, a match is found in the cache, and that function is branched to. Unfortunately, the wrong function has just been called, since the implementation of "describe" for the Employee class was found, rather than the implementation for the Person class. Bizarre and dangerous results would result if this problem was not addressed.

In Figure 6.3, then, we see the following:

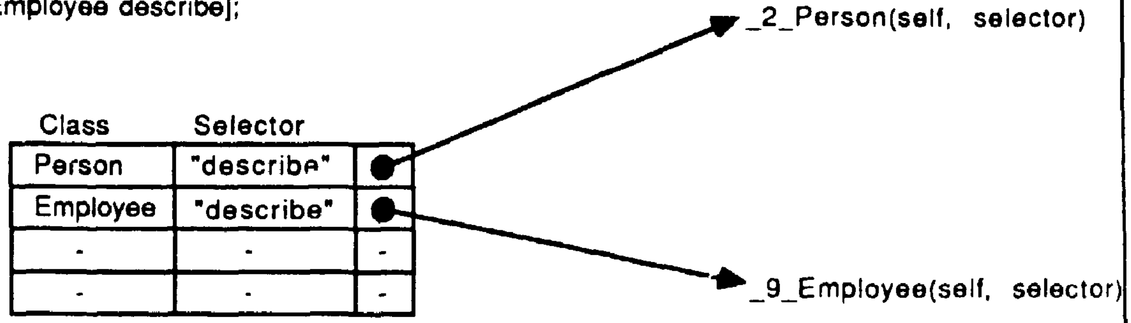
- In situation (A) - which is the normal Objective-C situation - we see that the two classes Person and Employee each have their "describe" method placed in a separate slot in the message cache.

- In situation (B) and (C), we have a more dangerous situation, since both the Employee and Person objects are, in fact, represented by Proxy objects. This confuses the normal message-passing and caching mechanisms.

Figure 6.3: The Objective-C Message Cache

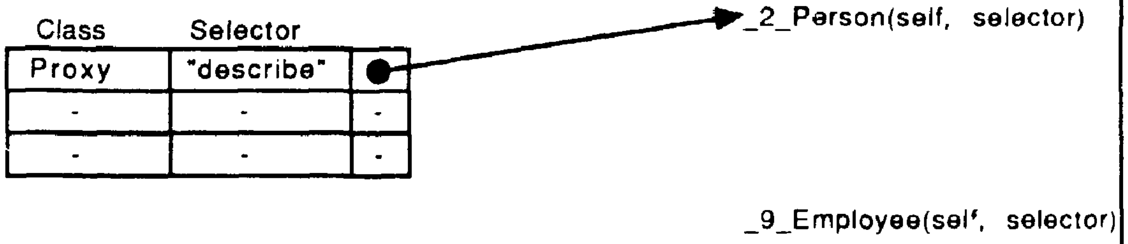
Objective-C Message Cache:

(A) [aPerson describe];
[anEmployee describe];



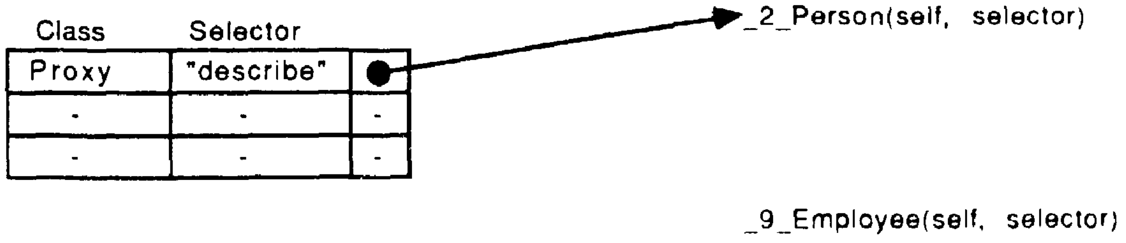
(B) [aPerson describe];
Note that aPerson is actually a Proxy object.

The message "describe" has been sent to a Proxy representing a Person. The cache is updated with the function address returned for the <Proxy, "describe"> pair.



(C) [anEmployee describe];
Note that anEmployee is actually a Proxy object.

The message "describe" has been sent to a Proxy represen[illegible]n Employee. Since a <Proxy, "describe"> pair is found in the cache, that function is executed and an error results.



In order to handle to handle this problem, the _msg() function has been altered to identify messages to Proxy objects. When such an occurrence is detected, the cache lookup code is bypassed. Instead, the _msgImpFind() routine is called, and the control is turned over to the function address it returns. The return value of _msgImpFind() is not used to update the cache, as is the normal case. As a result, messages to 'ordinary' objects continue to utilize the message cache, while messages to persistent objects always require a call to _msgImpFind() in order to find the address of the specified method. This has performance repercussions, since the search undertaken by _msgImpFind() may be proportional to the depth of the inheritance tree of the receiver object. An 'industrial strength' implementation could maintain a separate cache for Proxy object messages; or, alternatively, the <class, selector> pair used to hash into the cache could be created using the class of the realObject pointed to by the Proxy object.

The Dangers of Violating Encapsulation

One of the features of the Objective-C language is that it allows the violation of the encapsulation principle inherent in the object-oriented paradigm. It is possible to get a handle on the internal state of an object directly. A purist approach, such as the one taken in the Smalltalk language, demands that all object manipulations be done using the operation provided by the object itself. With respect to the implementation of an object server for the language, this approach is a double-edged sword. On one hand, being able to violate encapsulation was a great help in implementing the class Proxy. Since it is impossible to send a message to a Proxy object (all messages are passed along to the persistent object represented by the Proxy), the only way for the object server implementation to modify or access Proxy objects was to do so directly. Essentially, Proxy objects are treated by the object server implementation much like a regular C structure.

On the other hand, applications which insist on violating the encapsulation of persistent objects must be very careful, or disaster will result. Since references to persistent objects are done so via Proxies, any attempt in the application code to point directly to the instance variables of a persistent object must recognize the existence of Proxies, or it will be using incorrect offsets. Instead of pointing to an object of class Employee, for example, and offsetting to gain access to the empNum variable, the program would be pointing to a Proxy object, and the same offset would be pointing at the value of the 'realObject' instance variable within the Proxy.

Typically, when an Objective-C program violates the encapsulation of an object, it does so by casting the parameters of a method to be of the expected type. Figure 6.4 provides an example code fragment which shows how a method which violates the encapsulation of an object would have to deal with the Proxy interface. This code would have to replicated in every method which attempted to deal with objects directly.

Figure 6.4: Dealing With Proxies Explicitly

```
@requires SomeClass, ObjectManager, Proxy;
typedef struct { @defs(Proxy) } PROXY_TYPE;
typedef struct { @defs(SomeClass) } SOME_TYPE;
// The method casts its parameter to be an instance of a specific class.
-aMethod:(SOME_TYPE *)parm1 {
        // First, check to see if the parameter is a Proxy object
        if (parm1->isa == Proxy) {
                // Has the object been read from the object base?
                parm1 = (PROXY_TYPE *)parm1;
                if (parm1->realObject == nil)
                        parm1 = (SOME_TYPE *)parm1->realObject;
                else {  // Read the object into memory
                        parm1->realObject = [ObjectManager get: parm1];
                        parm1 = (SOME_TYPE *)parm1->realObject;
                }
        }
        /* Then the method may continue normally.... */
```

7. The ObjectManager

Any application which wishes to use the object server must link the Objective-C class ObjectManager. The ObjectManager provides the major components of the object server's functionality. These include:

- Transferring objects from memory to disk, and vice versa. In order to do this, the ObjectManager accesses metadata which describes the representation of objects.
- Caching objects which have been read from the database. The cache is maintained as a hash table.
- A message protocol which allows the application to start and end ObjectManager sessions.

7.1. The Object Schema

Before the inner workings of the ObjectManager can be described, the concept of the object schema must be explained. The object schema provides the 'data about the data' - or metadata - that the system requires to manipulate the persistent objects.

The format of the metadata required by the object server was largely motivated by the format maintained by ZIM. Recall that a ZIM database schema is described largely in terms of the two entity sets: EntitySets and Fields. The object server schema is described largely in term of two classes: Class and InstVar. How the object server metadata is used to maintain the ZIM database schema is discussed at the end of the chapter.

Defining a Class

The object server must maintain information which allows it to describe both the ZIM and Objective-C representation of an object. Since the ZIM DBMS maintains records which are essentially C structures which have been written to disk, why are these representations different?

- ZIM places a 'null' byte before the beginning of each field in the record. This is used by ZIM applications to store explicit null values in the database. These null bytes must be removed upon reading the object from the database, and inserted when the object is written.
- Each instance in the object server must contain that object's OID. This value is kept in the first field in the record which represents the object in the database. When the object is read by an Objective-C application, this OID is used to find the object in the database Once the object is read, the OID is maintained by the object's Proxy. It is not contained in the object itself.
- Objects in Objective-C have, as their first instance variable, their *isa* pointer. Isa points to the object's factory object. This pointer must be initialized when the object is read, and removed when the object is written.
- References to other objects are handled differently in the persistent and dynamic representations of the object. In Objective-C, object references are via pointers of type *id*. These require four bytes of memory. Persistent objects refer to each other via OIDs, which require eight bytes of storage. When an object is read, its references to other objects are represented by a Proxy instance. When an object is written, its references to other objects are represented by the OID stored within the Proxy

Recall that all objects are unique instances of some class. This means that the object server must know the classes that it is expected to maintain, and their format. To do so, the class Class is used. Instances of Class correspond one-to-one with the Objective-C classes for which the toolset designer wishes to maintain persistent instances. The class Class is itself an Objective-C class which is known to the ObjectManager. As we shall see, it is an example of a class which has several instance variables which are not persistent. It is defined as follows:

```
= Class : Object {
        id      name,
                superClass,
                instVars,
                factoryObject,
                subClasses,
                dbSet,
                getSel,
                putSel;
        short   classNum;
}
```

Where the instance variables have the following meaning:

name	- The String object which contains the name of the Class (ie. "Object").
superClass	- The Class object which specifies the class's superclass.
instVars	- The ordered collection of all of the instance variables which make up the class. The specification of the class InstVar will be described below.
factoryObject	- The location of the actual Objective-C factory object which is used to create new instances of the class. The factoryObject is not a persistent object, since it is only meaningful at run-time.
subClasses	- The ordered collection of all of the class's subclasses.
dbSet	- An instance of class DBSet, which is used to read and write instances of the class to and from the underlying ZIM database. dbSet is not a persistent object, since it is only meaningful at run-time.
getSel	- The String object which represents the selector which is to be sent to the Class instance in order to have an object of that class read from the database. This is needed to handle the accessing of classes which are a special case.

putSel	- The String object which represents the selector which is to be sent to the Class instance in order to have an object of that class written to the database. This is needed to handle the accessing of classes which are a special case.
classNum	- The ZIM file number of the database file which contains the class. All instances of the persistent class are held in this file.

The class InstVar is crucial to understanding the object schema. Class objects use InstVars to describe both the persistent and dynamic representation of their instances. The implementation of InstVar is as follows:

```
= InstVar : Object {
        id      name,
                partOfClass;
        int     type;
        BOOL    index,
                persistent;
        short   length;
}
```

Where the instance variables have the following meaning:

name	- The String object which contains the name of the InstVar (ie. "name").
partOfClass	- The instance of Class which this instVar is part of.
type	- An enumerated type which indicates the data type of the instance variable (ie. "DB_ID", "DB_INT", "DB_DOUBLE", ...).
index	- A Boolean variable which indicates whether or not this variable has a ZIM index based on its value.
persistent	- A Boolean variables which indicates whether or not this variable is persistent. For example, in the class Class, dbSet would have a value of NO, while name would have a value of YES.
length	- Used to indicate to ZIM how long a character array instance variable is.

Supporting Inheritance

The inheritance tree of an object class is described explicitly in the metadata through the Class instance variable *superclass*. At the root of the class hierarchy is the class Object, which has a superclass of nil. Each Class maintains an ordered collection of the Classes which inherit from it. Thus the superClass and subClass links define the inheritance hierarchy. An example inheritance hierarchy implemented in this fashion is shown in Figure 7.1. Each Class object maintains an ordered collection of its instance variables. It should be noted that the Class Object defines the instance variable "oid". Since all objects inherit from Object, this ensures that all persistent objects have this instance variable defined.

There is a message protocol to query a Class object about its instance variables, made up of two messages:

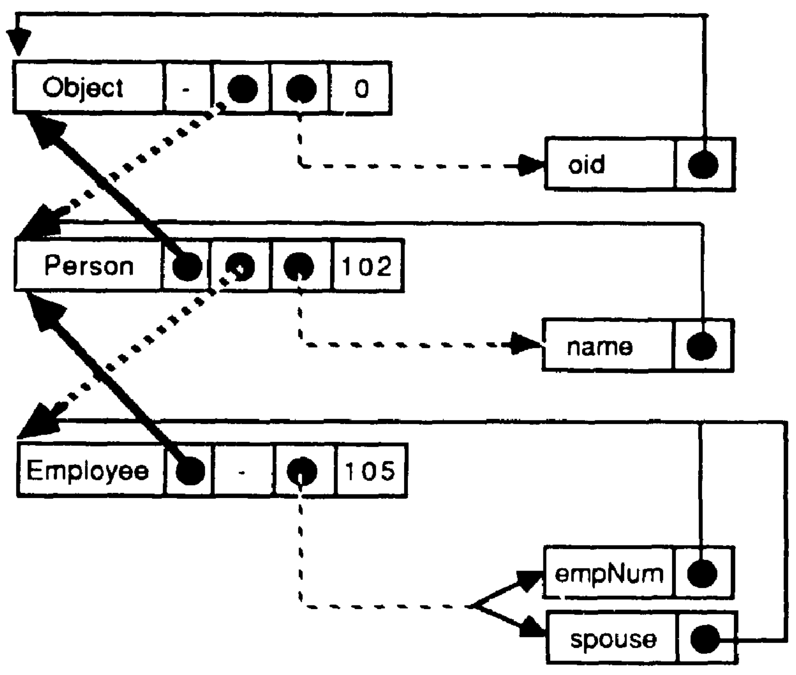
- *yourInstVars* returns the instance variables specified for this class. It ignores any instance variables defined in its superclasses. For example, the Person class object would respond to this message with a collection containing the InstVar instances for "name" and "spouse". The Employee Class object would return a collection containing only the InstVar instance for "empNum".
- *instVars* returns all of the instance variables defined for a class, including those defined in its superclasses. It is implemented in terms of *yourInstVars*. The array of InstVars returned has the superclass's instance variables first. For example, the Employee class would respond with a collection containing the InstVar instances for "oid", "name", "spouse", and "empNum".

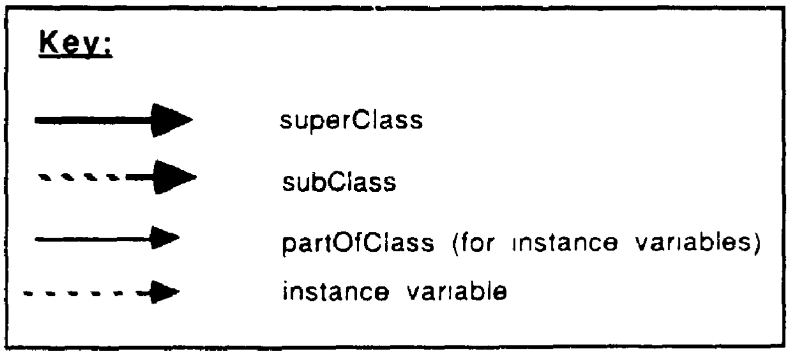
For example, the instVars message would be used to get the list of instance variables which fully describe an object when reading it from the object base. The yourInstVars message could be

Figure 7.1: Supporting Inheritance

```
= Person : Object {  // ZIM file 101
        char        name[30];
}

= Employee : Person {  // ZIM file 102
        int         empNum;
        id          spouse;
}
```





used in constructing a browser for the object schema, where you would wish to deal with only the instance variables defined in a particular class.

Metadata Access

The object schema is read from the object base by the ObjectManager in response to the *open* message. There are several implementation details to note regarding how this is performed:

- The code to read the object schema does not utilize the normal ObjectManager methods for reading objects, since that code relies on the metadata. You cannot use the metadata to read the metadata. Note that this does not imply that the object schema may not be manipulated by applications which use the object server. An example of such an application is the Browser described in the following chapter.
- The Class and InstVar instances which make up the object schema are not represented by Proxy objects within the ObjectManager. It has a direct handle on the persistent objects. This is safe since the ObjectManager never modifies the metadata. The Class and InstVar instances which are read when the ObjectManager is opened are discarded when it is closed. By allowing these objects to be handled directly by the ObjectManager, performance gains are realized, since messages to non-Proxy objects use the message cache.
- The root of the inheritance hierarchy is the class Object. This object has one special property that is hard-wired into the system: it has an Object Identifier (OID) of zero (0)[1].

[1] Actually, it has an OID of 0.0104, where 104 is the classNum of the class Class.

This allows the ObjectManager initialization code to read in that object from the database directly, and then read all of Object's subclasses recursively

7.2. An Objective-C Protocol for the ZIM PLI

The ZIM PLI provides a number of C functions which may be used to access, add, change and delete records and index entries in ZIM database files[1]. Since all of the instances of an class are maintained in the same ZIM database file, these operations are the equivalent to adding, changing and deleting instances of classes.

Access to the database files is provided by two abstractions: the ZIM entity set (ZESET), and the ZIM index (ZINDEX). A ZIM entity set is essentially a pointer to a C structure which provides a reference to the contents of a ZIM database file. The pointer is returned by the PLI function which opens a file, and it is passed as a parameter to all of the PLI functions which manipulate the records in the file. The ZESET structure maintains a 'current record', which is essentially a pointer to a specific record in the file. This current record pointer may be moved explicitly using a 'get next record' function, or by using an index to locate a object which has a specific value for an instance variable.

Once an entity set has been opened, any indices which exist may also be opened. A set of PLI functions exists which will locate specific records in the file based on a specific value and a Boolean operator (ie. <, <=, =, =>, > or !=). New records may be added, and existing ones updated or deleted. Note that it is up to the program modifying the record values to update any associated indices. Failure to perform these index updates correctly could result in the corruption of the database.

1 *Additional PLI functions are provided to support session control, security, and concurrency control*

There is one small but important detail regarding the ZIM PLI which must be made explicit. When using the PLI, it is recommended that the application open and close the entity sets and indices very frequently. A typical sequence would be:

```
open entity set
open index1
open index2
locate a record using index1
modify the retrieved record
write the updated record
update index1
update index2
close index1
close index2
close entity set
```

It should be noted that closing the entity set has relatively little overhead associated with it. This is because closing the entity set does not normally result in the closing of the underlying file. ZIM maintains a ring of recently accessed entity sets, and a file is physically closed only if necessary. The number of open files allowed is a parameter which the user may control. Closing entity sets and indices does cause any modified buffers to be flushed back to the disk file.

A ZIM database file may be modelled effectively as a pointer to a ZESET structure, with an associated collection of indices. The class DBSet performs this role. Instances of class DBSet are created at run-time, as the different classes in the object schema are requested to read, write or modify their instances. The basic approach is that instances of DBSet are responsible for dealing with accesses to the ZIM database. They retrieve, add, change and delete instances of classes in the database. However, it is the Class objects which are responsible for translating between the persistent and dynamic representations of the objects. There is one DBSet instance for each active Class. DBSet has the following structural definition:

```
DBSet : Object {
        ZESET       *entitySet;
        id          indexes;
        int         mode.
                    status;
        id          class;
        char        buffer[MAX_BUFFER_SIZE];
        int         buffLen;
}
```

Where the instance variables have the following meanings:

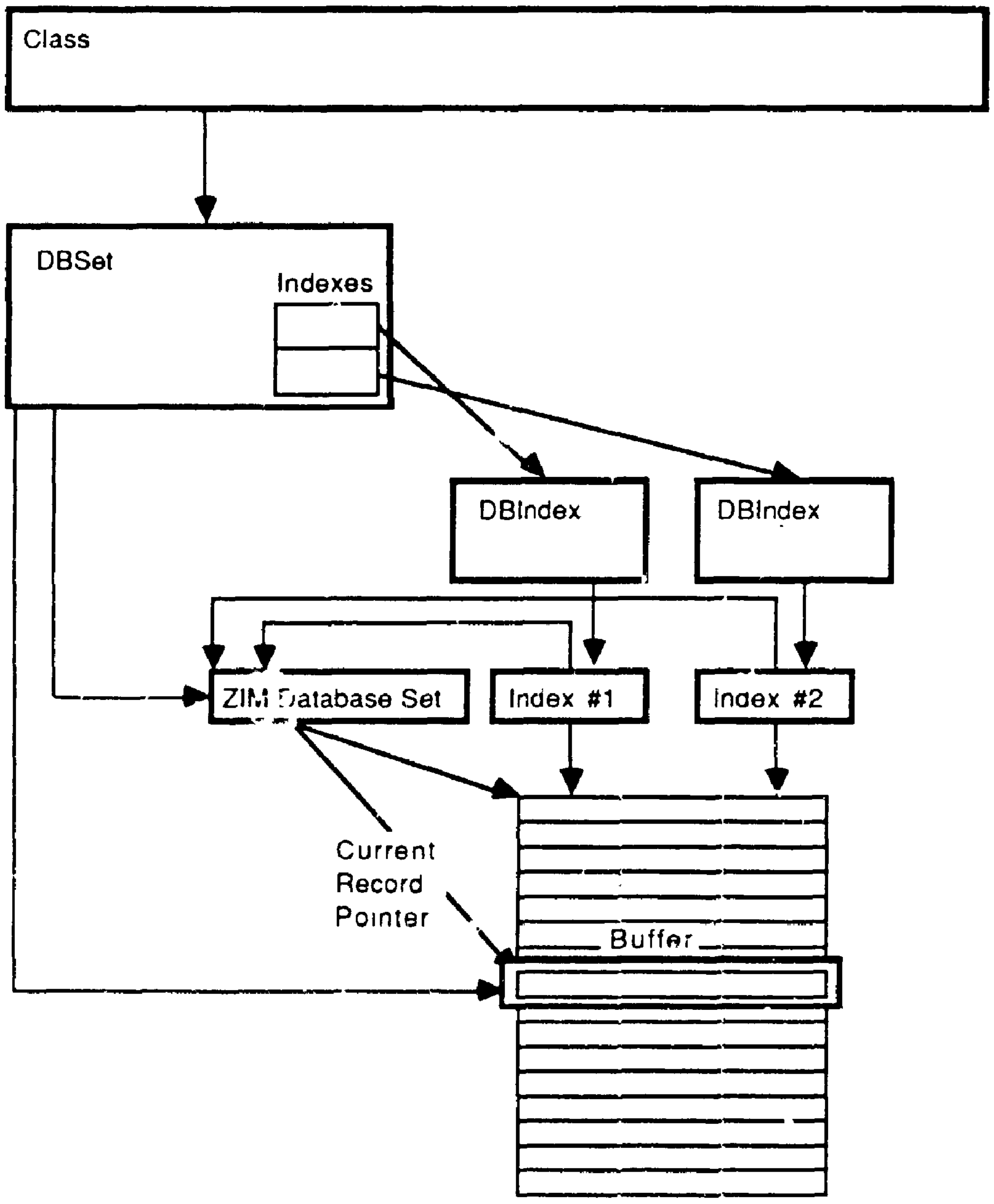
entitySet	- A pointer to the ZIM entity set structure returned when the ZIM databas‹ .ile is opened.
Indexes	- An ordered collection of the DBIndex instances for all of the indices for this file. The class DBIndex is described below.
mode	- When a ZIM database file is opened, its access mode must be specified. The two possible values here are read-only or update.
status	- Indicates whether the ZIM database file is presently opened or closed.
class	- The Class instance which owns this DBSet object.
buffer	- The buffer into which records from the database are read.
buffLen	- When writing a record, how many bytes are to be written. When reading a record, how many bytes were read.

The "buffer" variable allocates a storage location through which all records pass when being read or written.

The class DBIndex associates a ZINDEX structure pointer with the instance variable upon which it is built. The structural definition of the class is as follows:

Figure 7.2: Manipulating the ZIM Database from Objective-C

Class objects know how to read their instances from the object base. Interactions with the database are handled through DBSets and DBIndexes, which provide an Objective-C message protocol for dealing with ZIM PLI sets and indices.



```
DBIndex : Object {
        ZINDEX      *index;
        id          ownerDBSet,
                    instVar;
        int         keyType,
                    status,
                    indexNum,
                    keyLength,
                    keyDecimals;
}
```

Where the instance variables have the following meaning:

index	- *A pointer to the ZIM index structure which this DBIndex represents*
ownerDBSet	- The DBSet object which owns this index.
instVar	- The InstVar object which this index is built on.
keyType	- Indicates which ZIM data type of the field on which the index is built.
status	- Indicates whether the ZIM index is presently opened or closed.
indexNum	- The index number of the this index in the ZIM entity set. For example, the first index defined on an entity set would have an indexNum = 1
keyLength	- The length of the field on which this index is built. It has a value of zero for numeric data types.
keyDecimals	- The number of implied decimal places for a numeric data type.

Using this approach, a message protocol may then be implemented for the ZIM PLI functions which allows the instances of a class to be retrieved, created and modified using a DBSet object. DBSets are responsible for ensuring that whenever an operation is performed an a persistent object, all of the indices are updated correctly as well

7.3. Initialization: The Database Root Object

Applications which use the ObjectManager do so by linking the ObjectManager class into the program. When the program is started, the class message *open* is sent to the ObjectManager. This results in the following steps:

1) The ZIM database is opened. This requires a call to a ZIM PLI function.
2) Memory for the object cache is allocated. The object cache is described in detail in the following section.
3) The object schema is read. The schema is the class hierarchy of all of the persistent classes defined for this object base.
4) The database root object is read and its Proxy returned to the application.

The database root object requires some description. All objects in the object base are reachable from this object It is presently implemented as an instance of an ordered collection with two entries[1] The first entry represents the root of the class hierarchy. The second entry is the root object for the objects in the application domain. Every object stored in the object base by the application is reachable from this root.

There are two constants defined to represent these entries in this root ordered collection. They are SCHEMA and APPLICATION. Since most clients of the object server would be interested in only the application objects, the protocol for opening the ObjectManager returns that object For example, for a program that was only interested in accessing the application data could use the following piece of code to retrieve the highest-level application object:

```
rootObject = [ObjectManager open];
```

[1] There are no *a priori* restrictions on the type of the root object, so the toolset developer is free to place any object there.

The application root object may be changed by sending the message *applicationRoot:* [1] to the ObjectManager. Changing the root object must be done with great care, since it is possible to inadvertently lose objects in the database if there is no path to them from the root.

As an example to illustrate the concept of a root object, recall the design of the ARTTisan toolset described in Chapter One. The toolset is implemented as a number of integrated tools which manipulate designs which are represented as a hierarchy of notations. At the top level of the design hierarchy was a System object. Every object in the design is reachable from this System object. Given this, one can draw a number of different toolset implementation scenarios which would result in different objects at the root. In a single-user, one design per object base environment, the application root would be the System object itself. In a single-user, multiple design environment, the application root would be a collection of System objects. Thus the root would be (for example) a Set, OrdCltn or Dictionary object. In a multi-user, multiple design environment, where each user owned a number of designs, the application root could be a collection of some hypothetical User objects, each of which, in turn, referenced a collection of designs owned by that user.

7.4. Records to Objects

Recall that objects are read from the database one at a time, as they receive messages The algorithm for converting from an object's persistent and dynamic representation is straightforward for most classes of objects. Simply allocate the memory for the object and then iterate through the instance variables of the object - whose type was described in the object

[1] Some examples of modifying the root objects:

```
[ObjectManager applicationRoot: someObject],
[ObjectManager schemaRoot: someObject];
[ObjectManager root: someOrdCltn];
```

schema - copying the data from one representation to the other. There are only a few minor issues that must be addressed when implementing this approach:

- Allocating memory for an object in Objective-C typically involves sending the message *new* to the class (factory) object of the desired type. When allocating the storage for a persistent object being read from the database this approach will not work in general. This is due to the fact that it is quite common to place initialization code in a class's new method. However, reading an object from the database is not the same as creating a new one. You are merely re-activating an existing object.

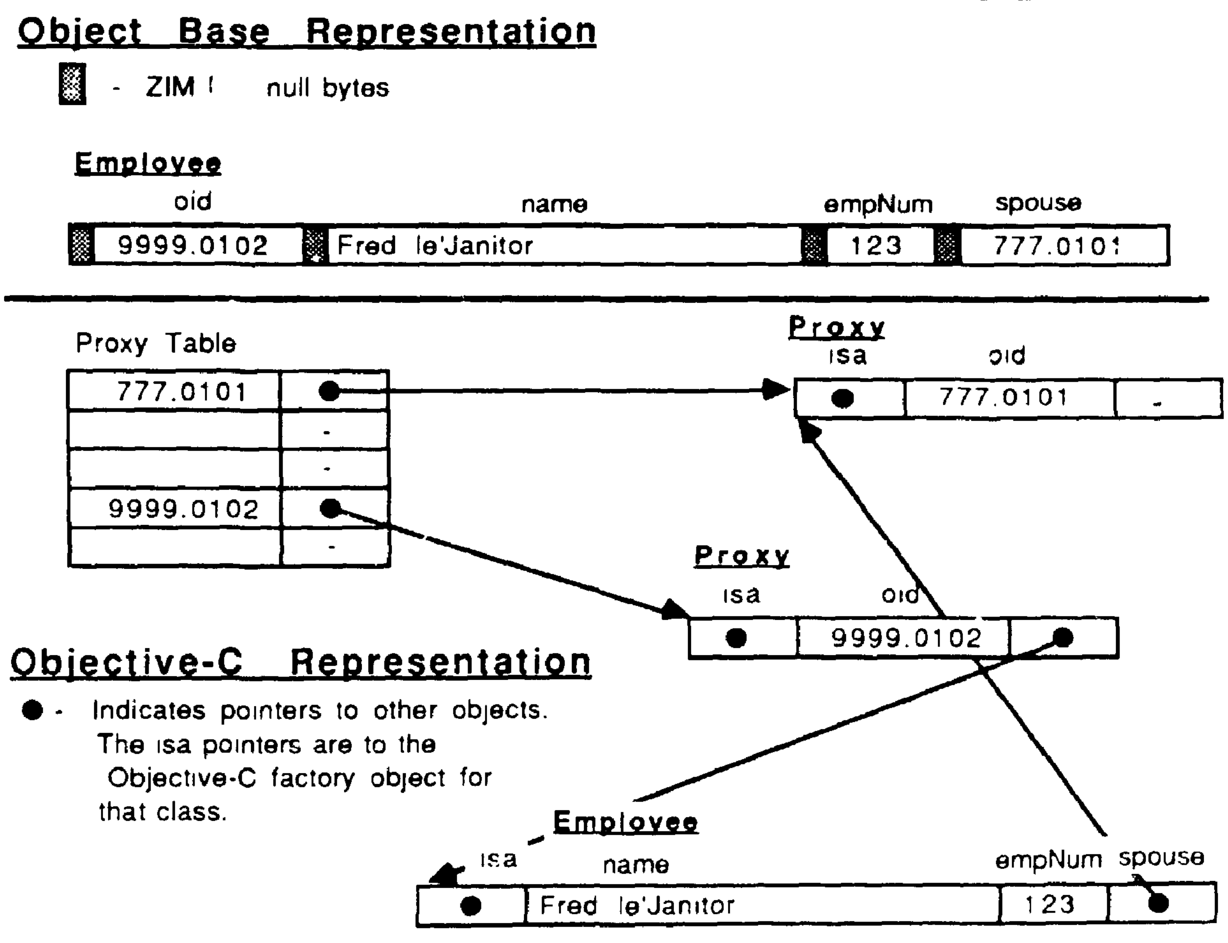
  When a new object is created, the memory for it is allocated using the Objective-C kernel function (*_alloc) In order to ensure that no initialization code is being invoked, this function is called directly.
- The instance variables for an object include all of those defined by the object's superclasses as well.
- Instance variables which are not persistent must be skipped over.
- Numeric data types (such as short, long, int, double, etc.) must be aligned at an even memory location in both the Objective-C and object server representations. This is a result of the SUN's machine architecture.
- ZIM does not support all of C's numeric data types. It is restricted to short, long and double. Appropriate type casting must be used to ensure that the data is in the correct format in both representations.
- Instance variables of type id - which represent references to other persistent objects - are converted to Proxy objects when they are read and to OIDs when they are written. Note that there is only one Proxy object per persistent object. When an object is being

Figure 7.3: Reading an Object

In this example, the Employee object 9999.0102 is read from the object base. It contains a reference to 777.0101. If a Proxy object for 777.0101 already exists, it is found in the object cache, and its location becomes the value of the Employee's spouse variable. If a Proxy does not already exist, one is created with an OID of 777.0101, it is added to the cache, and then it is assigned to the spouse variable.

```
= Person : Object {  // ZIM file 101
      char       name[30];
}

= Employee : Person {  // ZIM file 102
      int        empNum;
      id         spouse;
}
```



read, and one of its instance variables is being assigned a Proxy, the Proxy cache in the ObjectManager is checked to see if that has already been created. For example, (see Figure 7.3) say an Employee object is being read whose spouse is represented by the Object Identifier (OID) 777.0101. First the cache is checked to see if a Proxy with that OID has already been created. If it has, the spouse instance variable of the Objective-C representation of the object is assigned to the Proxy. If it has not, a new Proxy is created, with its OID initialized to 777.0101, the Proxy is added to the cache, and then the spouse variable for the object is assigned to the new Proxy.

To illustrate what occurs when an object is read from the database, let us extend the example described in Figure 6.2, where the "describe" message is sent to a Person object which has not been read from the database. The sequence of events is as follows:

- The fact that the object has not yet been read in is detected in the _msgImpFind() function. The message has been sent to a Proxy which has a realObject of nil. The code in _msgImpFind() is:

```
if ((*refSelf)->isa == _ProxyIsa) {
        if (((PROXY_TYPE*)(*refSelf))->realObject != nil)
                (*refSelf) = ((PROXY_TYPE*)(*refSelf))->realObject;
        else {
                ((PROXY_TYPE*)(*refSelf))->realObject  =
                        [ObjectManager  get:  (*refSelf)];
                (*refSelf) = ((PROXY_TYPE*)(*refSelf))->realObject;
        }
}
```

The ObjectManager is instructed to fetch the object represented by the Proxy from the database, and return it so that the Proxy's realObject instance variable may be modified accordingly

- When the ObjectManager receives the get: message, it looks up the Class object in the object schema for the class of object represented by the Proxy. Recall that OIDs contain an indicator of the class of the object represented by the Proxy. The message

  [proxyClass get: Proxy]

  is then sent to the Class object. The key thing to note is that Classes are responsible for knowing how to retrieve their instances from the object base.
- The Class instance method get: performs the following: it ensures that the Objective-C factory object for the class is available[1]; if an instance of DBSet has not yet been created for the class, one is; the DBSet is opened; the class's "getSelector" is then performed to retrieve the object from the database; and the DBSet is closed. Recall that each class has two instance variables named "getSelector" and "putSelector". The methods named by these two variables specify how instances of that class are to be read from the object base. The default getSelector is named "readDB:", and it is this routine to which the following description applies.
- The readDB: method takes the Proxy to be read as its parameter. It, in turn, sends the message

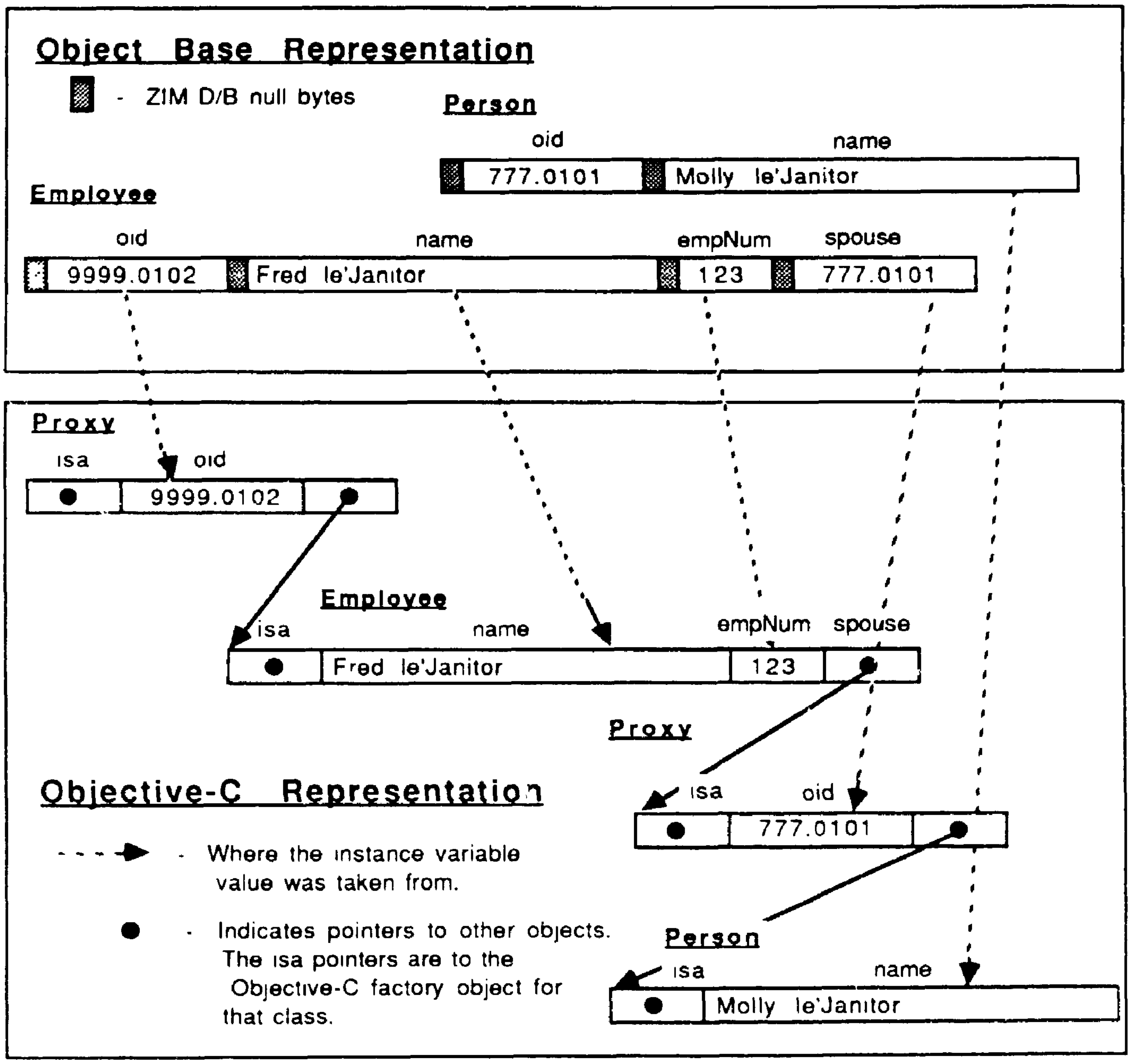
  [dbSet get: aProxy]

  to the DBSet associated with this class.

[1] This is equivalent to ensuring that the application has linked the class of the object which is to be read from the object server. It is possible, for example, to have defined a class Person in the object schema and then not link the Objective-C code for Person with the application A fatal error will result if this situation occurs.

Figure 7.4: Representing Persistent Objects

```
= Person : Object {  // ZIM file 101
      char        name[30];}
= Employee : Person {  // ZIM file 102
      int         empNum;
      id          spouse;}
```



- The DBSet object uses the OID provided by the Proxy to retrieve the desired object from the ZIM database file for the class. A pointer to the record retrieved from the database is returned to the Class method readDB:.
- The instance variable values of the retrieved record are then used to create the Objective-C representation of the object, as described at the beginning of the section Once the object has been created, it is added to the object cache (described later in this chapter), and then returned.
- Eventually, the newly-read object is returned to _msgImpFind(), where it is assigned to the realObject variable of the Proxy which originally received the message. The object is used to modify the (*refSelf) value, and from there on, the Objective-C message-passing routines operate as normal.

Figure 7.3 illustrates the persistent and dynamic representations of the objects involved in this example.

Other Object Representations and Their Access Methods

Unfortunately, the instances of all classes cannot be read using the same mechanism. This is due to the fact that certain classes have representations which are non-standard, as described in the previous chapter. In addition, the toolset developer may wish to define classes which, for some reason, cannot be accessed in the normal fashion. Each of these classes require a different access method or the specification of an additional storage type, some also required a specific storage model. The following sections describe these classes, their representation in the ZIM database, and the access methods required to read their instances.

Custom-Defined Access Methods

Each Class object has "getSelector" and "putSelector" instance variables. The default values for these two variables are "readDB:" and "writeDB:" respectively. These name instance methods of the class Class which represent the standard method for reading and writing instances of a class from the object base. They each expect a Proxy as their parameter. Failure to conform to this will probably result in disaster.

By placing the names of different methods in these slots, different Classes may implement different access methods. For example, the following classes use the following methods:

IdArray	readDBIdArray: writeDBIdArray:
String	readString: writeString:
User-Defined	readByClass: writeByClass:

These new access methods will be described in greater detail below.

Using these instance variables allows the toolset developer, if he desires, to create new access methods for classes which have some non-standard representation. To do so, he will require some knowledge about the object server. The source code for the other access methods is available in the Class implementation to provide some guidelines.

In order to implement such a class, the Class object should have a "getSelector" of "readByClass:" and a "putSelector" of "writeByClass:". These two methods have the following implementation:

```
-readByClass: aProxy {
        [factoryObject readInstance: aProxy for: self];
}
```

```
-writeByClass: aProxy {
        [factoryObject writeInstance: aProxy for: self];
}
```

Note that each of these expect a Proxy as a parameter. In addition, the developer must write the access method code as well, in the Objective-C code which implements the class. The access method code must be implemented in terms of two class methods, which must have the following definitions:

```
+readInstance: aProxy for: aClass
+writeInstance: aProxy for: aClass
```

The "readInstance:for:" method must return the Proxy passed to it, with its realObject instance variable assigned to the object read from the database. The "writeInstance:for:" method must return nil. The "aClass" parameter passed to both of these methods is the Class object in the object schema for the class being accessed.

IdArray

Recall that the class IdArray implements a class where the objects referred to are accessed as indexed array elements, rather than as named instance variables[1]. All of the examples discussed thus far have stated that all instances of a class are contained within one ZIM database file. This is not the case for instances of IdArray, which require two files to fully describe their instances. Figure 7.5 illustrates the point. It shows an IdArray in its Objective-C representation and its format in the database. The points to note are: the named instance

1 This is described fully in Section 6.1.

variable capacity is placed in one file, and the indexable portion of the object are in the second. In the second file, each non-nil array element and its location in the array are written.

Figure 7.5: Representing IdArrays

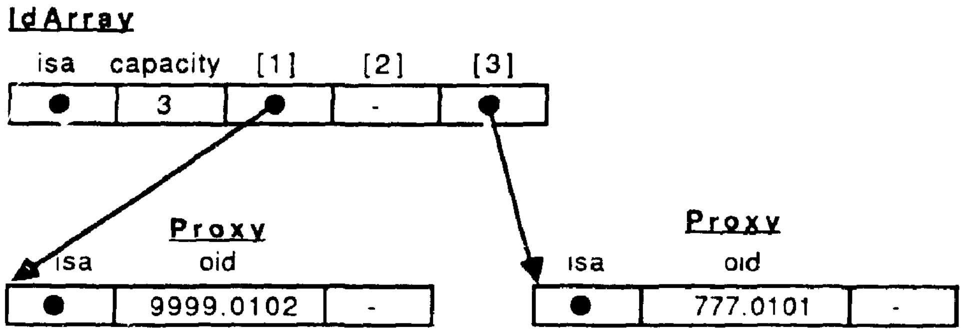
Objective-C Representation

● - Indicates pointers to other objects. The isa pointers are to the Objective-C factory object for that class.

In this example, the following lines of code have been used to create an instance of IdArray.

```
myArray = [IdArray new: 3];
[myArray at: 1 put: anObject1];
[myArray at: 3 put: anObject2];
```

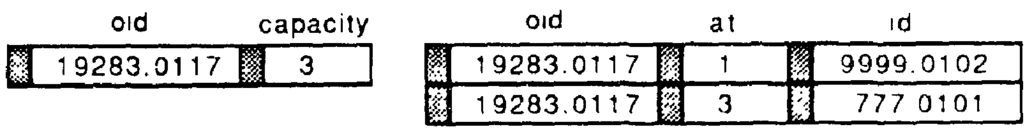
The two objects inserted into the array are, in fact, Proxies representing persistent objects stored in the object base.



Object Base Representation

▨ - ZIM D/B null bytes

When the IdArray is written to the object base, the indexed portion of the array is written to a separate file. Only the non-nil entries are written, along with their location in the array



Recall that each Class has the instance variables "getSelector" and "putSelector" which take a Proxy as a parameter and return the specified object from the database. In order to read an instance of IdArray, the Class object for IdArray uses the method "readDBIdArray:"; to write an instance of IdArray, the method "writeDBIdArray:" is used.

Since IdArray has a specialized access method, it cannot be sub-classed in the object server.

Collections

The class Cltn and its subclasses such as Set, Bag, OrdCltn and Stack all use an instance of IdArray to contain their elements. However, since the code that implements these data abstractions routinely violate the encapsulation of this array, it is not possible to use a Proxy to refer to it. When a collection is read from the database, the IdArray which contains the contents must be created and its memory location given explicitly to the collection using it. The collection object itself, of course, is referred to via a Proxy object.

Since the collection classes specify additional named instance variables, and their contents are implemented as an IdArray, their instances are spread over three ZIM database files.

The manner in which collection classes read their contents is of special note. Each of the collection classes keep the IdArray which contains their references in a named instance variable called 'contents'. In order to allow the collection classes to be sub-classed, this instance variable has a specific data type, called DB_CLTN. Essentially, the DB_CLTN type indicates to the Class that this instance variable contains an IdArray. When an instance variable of this type is read from or written to the object base, it is treated as such.

The advantage to this approach is that it allows Collection classes to be sub-classed. When a specific access method is written, as was the case with IdArray, that class can only be specialized if the subclass does not add any instance variables. A subclass may be specified which modifies the behaviour, but not the structure of the class. When a new type is introduced, it is the type of the instance variable which specifies how it is accessed, rather than some specific piece of code.

String

Instances of the class String are similar to IdArrays in that they maintain a memory area which is indexed into rather than being represented by named instance variables. In this case, however, the contents are composed of a null-terminated C string.

The access method implemented for the String class takes advantage of the ZIM "varchar" data type described in the previous chapter. This data type will store a variable-length array of characters in the following structure:

```
{
short   length;
char    contents[x];
}
```

where x is the maximum length of the field, as specified in the database schema entry. When the record is actually written to the database, the field is compacted to the length specified in the structure. When the record is read from the database, the character array is padded with spaces (not nulls) to its maximum length. The access code then strips the trailing spaces and uses the res[illegible]ng C string to create an instance of String.

In order to utilize this access method, the String Cla s object has a "getSelector" of "readString:" and a "putSelector" of "writeString:".

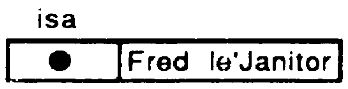
Figure 7.6: Representing Strings

Objective-C Representation

In this example, the following has been used to create an instance of String.

myString = [String str: "Fred le'Janitor"];

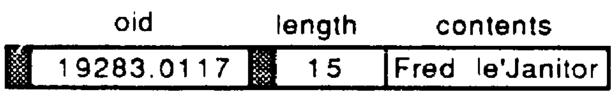
Note that the contents of the String object is a C null-terminated st'ing.



Object Base Representation

▨ - ZIM D/B null bytes

When the String is written to the object base, the contents of the object is placed in a ZIM variable-length character field. Only the specified number of characters are written to the database.



7.5. Writing Objects to the Object Base

Recall that the object base is essentially one large complex object. Every item in the database is reachable from the database root object. Applications which use the object base do so by navigating through the directed graph of objects which is of interest to the user. This navigation is performed by sending messages to the objects which are to be manipulated by the toolset. Of course, these messages are actually being sent to Proxy objects, and the database reads are therefore entirely transparent to the application. Writing objects back to the object base, however, is performed explicitly. The application code must send the "save:" message to the ObjectManager. For example, the following statement would save every object read since the ObjectManager was opened back to the object base:

[ObjectManager save: rootObject];[1]

where *rootObject* was the object returned when the ObjectManager was opened.

When an object is saved, every object reachable from that object is saved at the same time. The directed, cyclical graph of objects reachable from the root object is saved at the same time. The steps involved with performing this are straightforward.

- Given a root object, create a collection of every object of a persistent class reachable from that root. The collection is composed entirely of Proxy objects. The graph of objects to be saved is created by the ProxyGraph class method "over:". For example, the message

  [ProxyGraph over: anObject];

  will return an IdArray which contains the Proxy of every persistent object reachable from *anObject*.
- In order to handle cycles, the graph walk procedure uses the "beenHere" instance variable of the Proxies. Whenever an object is added to the collection, its Proxy's "beenHere" variable is assigned YES. If this Proxy is encountered later in the graph traversal, it is ignored.
- The graph walk procedure uses the object schema data to access the contents of the objects. For objects with named instance variables, the algorithm iterates through each

[1] There is also a "saveContentsOf:" message which will save every object reachable from the contents of some collection, but which does not save the collection object itself. For example, if empSet is a Set of Employees, the following will save the set:

[ObjectManager saveContentsOf: empSet],

of the variables. For instances of IdArray, the algorithm iterates through each element in the array. One key thing to note is that the traversal stops whenever it encounters an instance variable which has been defined to be not persistent **or** is not of a persistent class.

- Iterate through the array of Proxy objects, saving each one in turn to the object base. This process relies on the fact that each Class knows how to write its instances to the object base. Each Proxy is sent to its class as a parameter to a "store:" message; the method which implements "store:" then uses the Class's "putSelector" to write the object to the database. The process is optimized by opening the underlying ZIM database file for each class only once, and then closing all of the files after the objects have been written. As each object is written, its "beenHere" variable is reset to NO.
- Free the IdArray instance which contained the graph of Proxies.

7.6. The Proxy Table

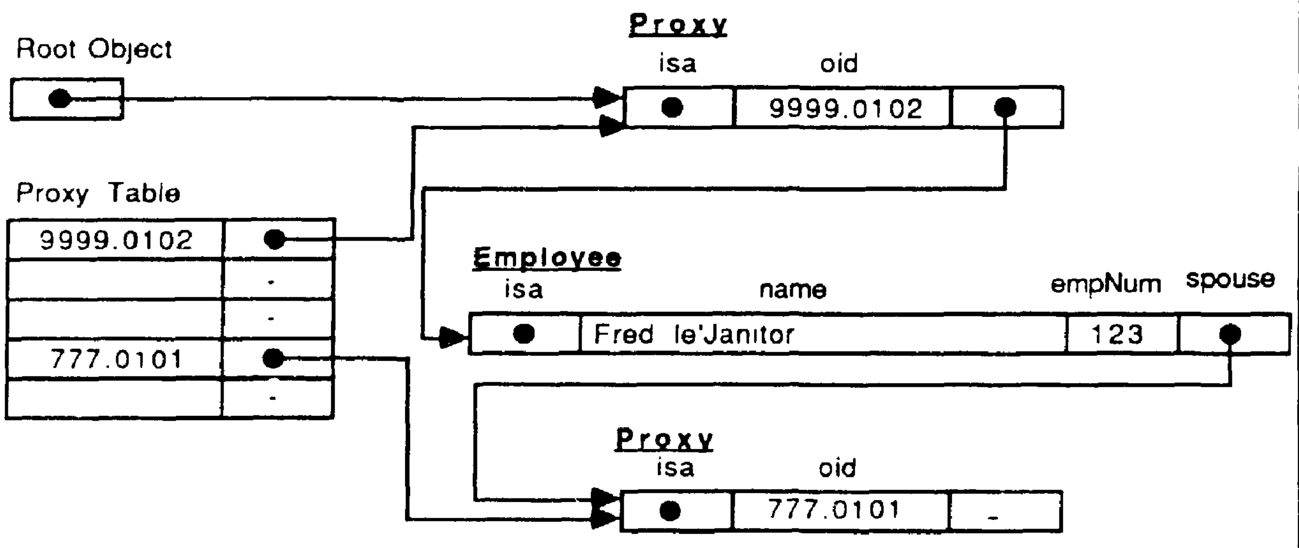
When the ObjectManager is opened, it returns a Proxy for the database root object. Every Proxy created afterwards is the result of an object being read which refers to a second object. These reads occur whenever a message is sent to a Proxy object which has a realObject of nil For example, in Figure 7.3 the Proxy for the Person object with an OID of 777.0101 was created when the Employee object with an OID of 9999.0102 was read. Whenever a persistent variable of type id is encountered when an object is read, the Proxy for the referenced object is placed in the correct offset in the object being read. However, it is critical that there is only ever one Proxy for each persistent object, so that all references to the persistent object remain consistent To ensure that this is the case, the ObjectManager maintains a *proxy table*, which maps the oid of the persistent object to the location of the Proxy in memory. Whenever an object reference is encountered when reading an object, the following events occur:

- If the OID of the instance variable is 0.0, then the variable is assigned nil. When writing an object, any nil references are assigned 0.0.
- If a Proxy already exists for that OID, it is found in the proxy table by the ObjectManager and the variable is assigned to it. Note that this ensures that all references to the same persistent object are via the same Proxy object. Also, if the object had already been read, then this is reflected by the realObject value of this unique Proxy.
- If no Proxy exists for that OID, a new one is created. Its oid value is assigned to the OID of the object being referred to, and its realObject value is initialized to nil. This new Proxy is then added to the proxy table kept by the ObjectManager. This table is used by the ObjectManager to ensure that any subsequent references to the same object are resolved using the same Proxy instance.

Figure 7.7: The Proxy Table

Every reference between persistent objects is handled via a Proxy object. The ObjectManager maintains a table of these Proxies ordered by their oid. Whenever a persistent object is being read from the object base, the inter-object references are resolved using this table.

Note also that the root object is actually nothing other than a pointer to a Proxy.



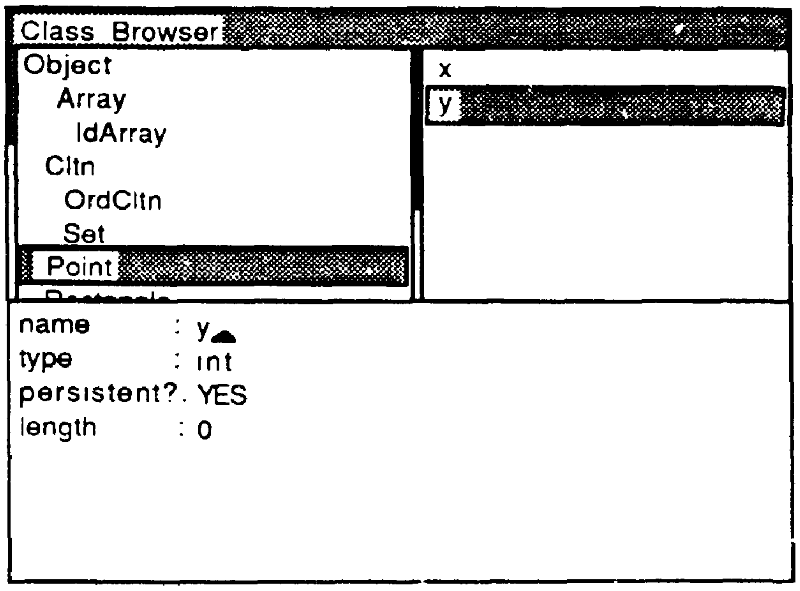
The proxy table is implemented as an instance of class HashTable which, in turn, uses the class Set to maintain its elements. This extra level of abstraction is required since Sets send messages to their elements and it is impossible for a Proxy object to receive a message. Even more importantly, any messages received by a Proxy object will result in the object that it represents being read from the object base. If the proxy table inadvertently sent messages to its elements, it would quickly result in the entire database being read into memory. However, the key for the cache elements must be the oid of the Proxy objects, since that is how they are retrieved by the ObjectManager.

7.8. A Browser for the Schema

In order to maintain the object server's metadata, a browser for the class hierarchy has been provided. The browser essentially provides a user interface for the definition of the classes whose instances are to be maintained by the object server. It provides the ability to create and destroy instances of the classes Class and InstVar.

The browser is of interest for two reasons: first, it provides a tool to inspect and modify the class hierarchy used by the object server; and second, it serves as an example application in the use of the object server itself. The browser is written in Objective-C, and demonstrates that the goal of transparency has been fulfilled.

Figure 7.8: The Class Browser



```
name        : y
type        : int
persistent?. YES
length      : 0
```

Figure 7.8 illustrates the format of the browser window. It is implemented as a SunView window with three panes. The first pane (top left) shows the class hierarchy. The inheritance tree of the classes defined to the system is indicated by the indentations. For example, class *Array* inherits from *Object*, and class *IdArray* inherits from *Array*. The second pane lists the instance variables for the selected class. In the example shown, the class *Point* has been selected, and its instance variables *x* and *y* are listed in the pane. The third pane allows the user to specify the name, type, and length of the selected instance variable. In addition, the user can specify whether or not the individual instance variable is persistent. Non-persistent instance variables are not store n the object base, and are initialized to nil when the object is read.

At present, classes may be only added and deleted using the browser. New classes are added to the class hierarchy by selecting a class and then choosing 'add subclass' from the pop-up menu. Instance variables for a class are added by selecting a class and then choosing 'add instance variable' from the pop-up menu. Classes are deleted by selecting a class and then choosing 'delete class' from the pop-up menu. Changes to classes are a tricky problem, since modifying the underlying data to reflect changes in the class specification is a non-trivial task. While the functionality of the present implementation is rudimentary, it provides a good demonstration of what a browser for the object server schema should look like.

In order to cause a change in the underlying ZIM database schema, the browser must communicate the desired additions and deletions to the ZIM environment. Recall that the ZIM schema is largely described by two entity sets, Ents and Fields. The relationship between Class and Ents should be obvious. When a class is added or deleted, a corresponding entry in Ents is created or destroyed. Similarly, when a instance variable is created or destroyed, that change must be reflected in the corresponding entry in the Fields entity. Since ZIM's schema information may be accessed through the PLI, these changes can be made in a straightforward manner. Unfortunately, changing the schema information alone does not cause an entity to be created or destroyed by ZIM. The metadata must be acted upon by the ZIM system in order to have the necessary database files acted upon. For example, once an entity set has been defined in the Ents and Fields entity sets, the ZIM command *create* must be issued to actually create the necessary ZIM database file.

In order to have the additions and deletions to the class hierarchy reflected in the underlying ZIM database, the following approach is used:

- The browser is actually called from inside ZIM. The ZIM environment is entered from UNIX, and then the command

  ```
  system 'dbBrowser'
  ```

  is issued.
- A ZIM entity set known as ModifiedClasses has been defined. The entity has two fields, namely *className* and *actionCode*. *ClassName* refers to the name of the class being manipulated by the browser. *ActionCode* is either 'A' for add, or 'D' for delete.
- Inside the browser, whenever a class is added its name is added to Ents, and its instance variables (including those defined by its superclasses) are added to Fields. Finally, a record is added to ModifiedClasses indicating that a class has been added. When a class is deleted, a record is added to ModifiedClasses indicating this as well
- When the user quits the browser, control is returned to the ZIM environment. ZIM checks the ModifiedClasses to see if there are any records. If so, the changes are acted upon

7.9. Using the Object Server

In order to use the object server, the application developer should follow these steps:

1) Write the application in Objective-C, just as usual. The only constraint is that the encapsulation of the objects should be respected if the persistent objects are to be treated completely transparently. If, for performance reasons, encapsulation must be violated, the Proxy objects must be recognized and dealt with explicitly. A method which violates the encapsulation of its parameter(s) would have to recognize that they could be Proxies instead of whatever class of object is normally expected, as discussed in Chapter Six. For an example of the necessary code, refer to Figure 6.4.

2) After the system is working without persistent objects, the next step is to integrate the object server with the application. In order to do this, the developer must first identify which classes are to have persistent instances. Next, the developer must decide which instance variables are to be persistent within those classes. After these decisions are made, the browser may be used to add the classes to the object server. In order to create a new object server for this particular application, copy all of the files in the directory "/usr/sirius/milink/zim3.0/caseDB/basic" into a new directory. To open a browser on the object server's schema, enter "dbBrowser" from UNIX.

3) Once the classes have been defined to the object server, the last step involves linking the ObjectManager and modifying the application's source code to open and close the object server. Typically, these changes would be isolated to the *main()* function of the application. For an example of the code fragments required, see Figure 7.9. The *rootObject* returned to the application by the *open* message is the root of the application objects in the object server. If the system wants to deal with the schema objects, it should send the message *schemaRoot* to the ObjectManager.

Figure 7.9: Using the Object Server

```
@requires ObjectManager;
main() {
        id          rootObject;
        rootObject = [ObjectManager open];


        /* Insert application code here */


        [ObjectManager save: rootObject];
        [ObjectManager close];
```

8. Conclusions

8.1. Meeting the Goals

The primary goals of this work were to support the following:

- To provide a persistent object store for the ARTTisan toolset.
- To integrate the object server with the Objective-C language.
- To evaluate the effectiveness of the object-oriented approach as an implementation vehicle for CASE environments.

Viewed from the vantage points described above, the present implementation of the object server is a success, since the goals were accomplished. However, a thesis is a learning process, and there are a number of improvements which could be made to the design and implementation of the object server. These extensions to the work are described in the next section, entitled *Future Research*.

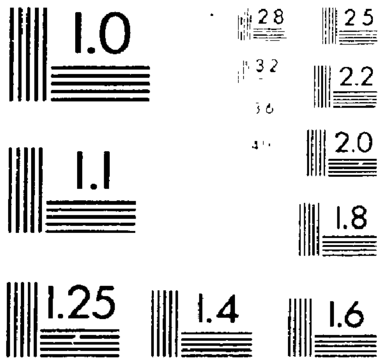
Persistent Objects

The number one goal of this research was, of course, to provide a working persistent object store for the ARTTisan toolset. This goal has been met. The system described in this work provides for the persistent storage and retrieval of the objects manipulated by the toolset. Underlying this goal, of course, is the requirement that the object server provide satisfactory performance. Performance here can be broadly measured in terms of two parameters: the time required to read the objects from the object server, and the time required to save the objects back to the object base. While the object server has not yet been used with the entire toolset, the following observations have been made:

- Reading in individual objects seems to provide adequate performance. The toolset may be viewed as a graphical editor for the persistent objects stored by the server Objects are typically read when the user performs some action that specifies that a particular object is of interest. For example, the user may click a mouse button while pointing to a graphical representation of an object. The response to read a typical diagram from the database is (subjectively at least) acceptable. The time lag to read such a diagram is typically one to two seconds.
- Writing out collections of objects admittedly takes some time. After running some time trials, a mean time of 2.5 minutes to save 3000 new objects to the object base is noted While not unacceptably slow, this time is not blindingly fast either. To a large degree, there is a trade-off being made between retrieval time and storage time for the persistent objects. When an object is being written to the database, the ZIM index associated with the object's OID must be updated. This takes some added time, over and above the time required to write the object to the server. However, the indices provide improved performance when reading the objects from the server, since indexed retrieval is faster.

In order to support persistent objects, an object-oriented data model was used. The model supports the notions of object identity, an extensible type system and inheritance. Complex objects and aggregate objects are supported as well. One uncommon feature of the data model is that individual instance variables of persistent classes may be declared as unpersistent. These instance variables are not saved to the object base, and are initialized to nils when they are read This feature was motivated by the application: there are classes for which this approach is required.

3 of/de 3



The object server implementation illustrates the utility of using a commercially available database as the back-end for the system. Using the ZIM PLI resulted in a number of advantages, including:

- Since the ZIM PLI took care of indexing, storage, retrieval, physical file access, and database consistency, there was less coding to perform, and the resulting tool contains fewer errors as a result.
- The ZIM PLI offered low-level access to the data. The object server would have been difficult, if not impossible, to implement using an embedded database language. This is because of the added consistency checks and compile-time type checking implicit in the use of such tools.
- ZIM offers portability as an avenue for future enhancements to the system. It is presently available on a large number of different hardware platforms and operating systems The PLI is also available on these platforms. As a result, porting the object server to different environments should be greatly simplified.
- The ZIM PLI also offered access to the database schema data. This allowed the creation of routines which modified the system's metadata.

The present object server implementation could be improved with respect to its use of the PLI. The primary drawback of the current approach results from the design decision to place all instances of a class in a separate ZIM database file This approach was motivated by the belief that it was more efficient and that it offered the ability to use the ZIM query language to examine the contents of the object base. Separating the object store into separate files poses a number of constraints:

- It prohibits the clustering of objects on disk based on expected usage patterns. Instead, objects are clustered based on their class, a strategy which yields few benefits in an application which is graph-oriented and where there is no query language to be supported.
- Maintaining consistency between the object server's metadata and ZIM's schema was found to be a problem. It is certainly not impossible to maintain such consistency, but it requires a lot of code to check, and is a constant source of errors.
- Because of the operating system limit on the number of files opened by an application, spreading the object repository over a group of files occasionally caused errors as well. The problem is certainly not difficult to fix through the use of ZIM configuration parameters, but it is ancther variable to concern yourself with.

An alternative object server implementation, which uses a minimum number of ZIM database files is described under *Future Research*.

Integration With Objective-C

The second major goal of this research was to provide a persistent object service which was essentially transparent to the toolset code. This requires that persistent objects be treated in an identical fashion to dynamic objects. The toolset code should not be required to test whether an object is on disk or in memory; persistent objects should be syntactically identical to dynamic ones; and persistent objects should not be required to be a subclass of some special class.

These goals have been met by the current object server implementation through the use of the Proxy interface. The illusion of the one-level store is fully supported - the application code does not know, or care, if an object has been read into memory. In addition, since the messaging kernel of the Objective-C language has been modified to identify messages to Proxy obiects, this interface provides good performance[1]. There is relatively little overhead in sending a message to a persistent object, once it has been read into memory.

Persistent objects are also fully consistent with dynamic Objective-C objects. This means, for example, that the object server does not use a different type system than the Objective-C language. In addition, the object server provides support for a large subset of the C primitive types and most of the Objective-C Foundation Classes. Those that were not implemented could be added easily to the system. They are not provided at the present time simply because they were not used by the toolset code, so there was little motivation to implement them. The exception to this is that there is no support for the storage of arbitrary C structures and unions. Again, however, the present toolset implementation does not require the persistent storage of these types. Typically, any references to a C structure are not persistent, and this is handled by the object server by specifying that the instance variable in question is not persistent

Effectiveness of the Object-Oriented Approach

The Objective-C language, and the object-oriented approach that it entails has been the implementation vehicle for the ARTTisan toolset. While there has certainly been a learning curve involved with adopting this methodology, it has been, on balance, a success. The object-oriented approach offers a number of key advantages for implementing a CASE environment It is an effective approach for the modeling of complex data types, such as the directed graphs which form a large part of the application. Its emphasis on encapsulation and modularity provides for increased reliability, minimized code bulk and increased code re-use. Its major drawback is the lack of a generally accepted and coherent design methodology. Object-oriented design is still

[1] Note, however, that no changes were required to the Objective-C compiler.

very much a hit-and-miss affair. It seems that there is really no substitute for the subjective application of a designer's experience at the present time.

The Objective-C language itself has a number of advantages and disadvantages. Its strong points include: a fairly consist implementation of the object-oriented paradigm, closely modeled after the Smalltalk model; a reasonably extensive library of classes; the ability to link to different packages; the ability to violate the encapsulation of an object at the programmer's discretion; and reasonably fast execution speeds. Its major disadvantages include: the ability to violate the encapsulation of an object at the programmer's discretion[1], and the Foundation class implementations which do so as the rule, rather than the exception; the lack of a cohesive development environment, such as Smalltalk's; and the lack of a reasonable memory management scheme.

8.2. Future Research

Object-oriented databases, object servers and database programming languages based on the object-oriented paradigm are all currently active areas of research. As such, this work represents but a first step. There are numerous extensions to, or additional avenues suggested by this research. The following sections cover a number of possible research areas which could follow this work.

Miscellaneous

The following describes a number of improvements which could be made to bring the system from a 'proof of concept' to a more complete and robust state. These enhancements

[1] The inclusion of 'violating encapsulation' as both a strength and weakness of Objective-C was intentional This is a double-edged sword which, if used without discretion, will negate many of the benefits of using an object-oriented language. That said, however, access to the internals of objects made the implementation of the object server simpler and more efficient.

represent incremental improvements to the existing implementation, rather than the major changes described in later sections. They include:

- Improved consistency checking between the object server and the language. This could include the creation of a automatic schema translator to generate Objective-C class definitions from the object server schema. Translating from the Objective-C definition to a schema representation is not feasible, since the schema requires more information than is available. For example, each instance variables must be defined to be either persistent or unpersistent.
- Improved consistency checking between the object server and ZIM. Primarily, this implies the ability to handle class modifications. When a persistent class is modified, two things must occur: the class definition must be changed in Objective-C, in the object server, and in ZIM, and any existing instances of the class must be modified to reflect the new definition.
- Garbage collection of the persistent objects must be provided. This is necessary since there is no way, at present, to physically delete an instance of a persistent class from the object base[1]. It is, however, possible to logically delete an object by de-referencing it. A simple mark-and-sweep garbage collection algorithm could be implemented, using the database root object as a starting point.

[1] It could be possible, however, to modify the behaviour of the free message to delete instances from the object base.

Multi-User Issues

One of the most important extensions for this work is in the area of multi-user access to the persistent data. Before presenting a strategy to solve this problem, let us describe the issues involved in extending the object server to support multi-user access to the toolset.

Software Design Transactions

In typical database applications, transactions typically have a duration of a few seconds or less In a software design environment, however, transactions may last hours, or even days. For example, a typical software design transaction for a user of the ARTT toolset would consist of the following steps:

- The user logs on and starts up the toolset.
- He then selects the system that he wishes to work on from the set of system icons on the desk top. He then enters the top-level design tool. Note that several users may want to work on the same system concurrently. However, it should be possible to lock specific system components for the exclusive use of a single toolset user. These locks may have a duration of days.
- Once he enters the system, he adds, changes, deletes and browses the various objects that make up the design. These actions may result in additional tools being activated.
- At any point in time, the user may wish to save his changes back to the object base. He may wish to save his work either because some logical point in the design process has been reached or merely to protect himself against the possibility of the workstation or network going down. Note that saving a work-in-process is not the same as committing a design to the object base.
- Periodically, he will terminate a session, typically after saving his work once again.

- At some point, a commit is issued and the design is identified as once again being in a consistent state and available to others for use.

This scenario implies a number of important points regarding the nature of software design transactions.

Duration: Design transactions may last a long time, typically several hours or days Therefore, the traditional solutions for controlling multi-user access of queuing users making competing requests for database access will not work. Obviously, queuing a user for an extended period while someone else is manipulating the desired object is not a viable mechanism for providing concurrency control.

Consistency: The overall consistency of the data committed to the database during a transaction cannot be checked by the DBMS; designs can and must be saveable in an intermediate (and hence, inconsistent) state. As long as the data is consistent at the individual object level, it can be written. The consistency of complex objects is the responsibility of the tools themselves. This is quite different from traditional database applications, where only completed, and presumably consistent transactions are written to the database.

Note, that to a large degree, intermediate states must be saved to the object base because of the long duration of software design transactions. The tool users will want to save intermediate states of their design to protect themselves from workstation of network failure.

Volume: Software design transactions touch (or have the potential to touch) large volumes of data of many different types. However, designers do not necessarily want write access to all of the data that they wish to see. Often, much of the data that they access is to browse other portions of the system under design.

Locking: Unlike traditional database transactions, the inability to lock an object should not imply the failure of the entire transaction. Hours of work could be lost, if this approach was followed. However, the user should be informed that he cannot access a certain object because someone else has already accessed it.

Performance: The duration and volume of data accessed by software design transactions makes the caching of objects a requirement for performance reasons. Reading objects from the database every time they are accessed by the tools would be too expensive in terms of disk I/O and network traffic. However, the object manager must know which objects have been cached in order to ensure that the different users are not manipulating the same object.

In addition, the implementation of the object server presented in this thesis has an impact on the nature of the design transactions, as they would be supported by ARTTisan. Data is read from the object base implicitly, while they are written explicitly. Objects are fetched from the object base as the toolset references them by sending a message to them. Objects are written, however, when a complex object is explicitly sent to the object manager for saving.

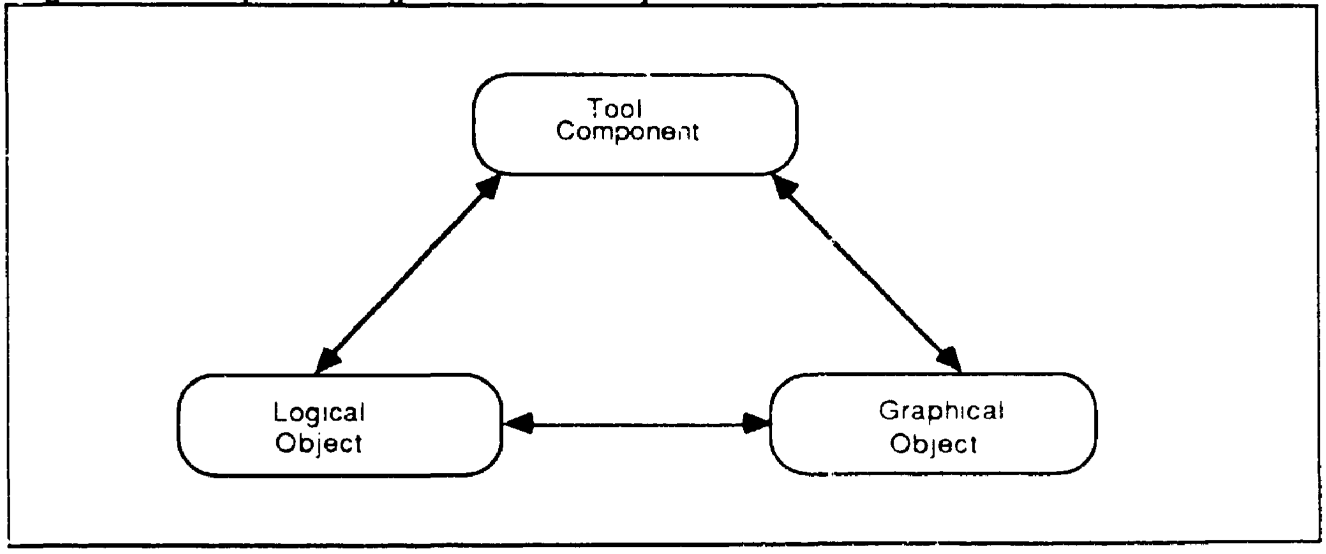
As a result, reading a complex object can be a very long operation, since it may involve many small reads as the object components are read on an 'as needed' basis. Write operations, on the other hand, would typically be of a shorter duration, with a (potentially) large number of objects being stored at one time.

The Granularity of Object Access: Defining Composite Objects

In the present implementation, individual objects are read from the object base as they receive messages. While this works reasonably well, there are a number of factors which motivate extending this approach:

1) Often, the toolset developer will know that a certain group of objects would be accessed together. For example, recall that the toolset objects are represented by a 'tool' object, which references a 'logical' object and a 'graphical' object. This relationship is shown in Figure 9.1. In virtually all cases, when the tool object is read, the graphical object associated with it would be read soon after. This is due to the fact that usually if a tool component object is accessed, it will be shown to the user by the toolset It would be useful to anticipate this, and read the two objects as a unit. There should be a mechanism to specify a larger unit than individual instances as the unit which is to be read from the object base. This unit is referred to here as a 'composite object'

Figure 8.1 Representing ARTTisan Objects



2) For the purposes of multi-user access to the object base, the reading and caching of single instances of objects presents a problem. Since objects are read one at a time, and the reads are essentially event-driven, it may take a long time to read in those portions of a large, complex object that the user wishes to access. In the meantime, another user could be following another path through the object base to the same complex object. At some point, a collision will occur, and one user will lock an object that the other is interested in The problem is that, since the unit of access is single objects, there is no reasonable place to test to see if a object is already locked, without testing every single object in the code What is needed is some way to identify larger groups of objects which may be locked as a unit, and have the toolset code test to see of they are locked before they are accessed. For example, it would be an onerous task to write code which explicitly locks individual instances of class Point, but it would make sense to explicitly lock an entire state machine diagram when it is read by the toolset.

3) As an extension (or even an alternative) to locking composite objects, they could also be the unit of versioning in the object base. Maintaining a sequence of versions of the major components of software designs has long been recognized as an important feature offered by CASE environments. The definition of the units which the application wishes to version is a key facet of providing such support. A second interesting point regarding versioning is that it has been suggested as a mechanism for providing concurrency control The basic idea is to create new versions of objects when they are written to the object base, rather than changing them in place. Since objects are never changed, only added, concurrent access to them is greatly simplified.

The challenge then, is to define a mechanism which will allow the toolset developer to specify 'composite' objects which may then be read and cached as a single unit, and which will also act as the unit which individual users may lock and/or version. The present data model supports the notion of complex objects as large networks of inter-related objects, where the relationships are based on the identity of the objects involved. At the limit, the entire database is reachable from the root object, and thus may be viewed as a single, if huge, complex object Composite objects may be viewed as providing the ability to define objects whose granularity fall between these two extremes.

Once a mechanism has been provided whereby the object server users may specify complex objects which may read as a unit, the next step would be to store these inter-related objects together in order to improve performance. Disk search times are one of the major limitations on database retrieval, and clustering related objects physically together is one approach which has been suggested for improving performance (Hornick and Zdonik, 1987) (Banerjee, et al, 1987).

Making Objective-C a Database Programming Language

One of the most interesting avenues of programming language research has been in the area of database programming languages. Researchers in this field view the separation of databases and programming languages to be artificial Instead, they propose the inclusion of persistence and multi-user access to shared data (two of the major roles of database) in general-purpose programming languages. This approach is in stark contrast to the database/programming language dichotomy which is the norm today. For example, to use the latest in relational DBMS technology - the SQL query language standard - the applications developer must write the control logic in a third generation language, and restrict his use of the database for fetching and storing relational tuples Complex objects are not supported: they must be constructed by the selects, projects and joins contained in the application programs.

An interesting extension to the research presented in this thesis would be to extend the object server implementation to the development of Objective-C as a database programming language Under this approach, all objects would be at least capable of being written to a persistent store[1]; however, which objects are actually saved would be under program control. The object server presented in this work is a large step in this direction. It provides the transparent storage and retrieval of objects defined to the application. It does not go quite far enough to be accepted as a database programming language. The major missing piece is the difference between the memory management for persistent and dynamic objects.

A true object-oriented database programming language would be an ideal development vehicle for the creation of CASE environments. The 'one-level store' abstraction would be a natural part of such a language. Therefore, programmers would not have to concern themselves with the location of objects. Disk-resident objects would be treated identically to dynamic ones. Objects would be read from the persistent store in a transparent manner. However, the real advantage to the extensions proposed below are:

- The representation of persistent objects would be as close as possible to their dynamic representation A great deal of computing muscle is expended with the current implementation translating between the two representations.

[1] This means that all classes defined to the application are persistent. This is contrast with the current implementation, where only specified classes are capable of being written to a persistent store.

- The message passing kernel of the language would be modified such that pointers to persistent and dynamic objects would be identical in format[1] Whether the receiver of the message was dynamic or persistent would be resolved when the message was sent.
- All of the objects in the persistent store would be saved in a single ZIM database file This approach has a number of advantages:
  1) There is no requirement to maintain consistency between the ZIM schema and the object server's metadata. Nor do objects have to conform to a ZIM record format. This is a major benefit, since maintaining this consistency was found to be problematic.
  2) It is possible to implement clustering strategies, where the components of large, complex objects are stored contiguously in the object base This could allow major performance improvements, since objects which are likely to be used together would be stored and retrieved from the persistent object store as a unit. Disk search times would be reduced as a result.

The following sections describe a design for extensions to Objective-C which would make it a true database programming language.

Buffer and Segment Management in a Single Object Repository

The advantage to storing all of the objects in the object store in a single ZIM database file is that it allows greater flexibility in the implementation. For example, since there would no longer be a requirement to store definitions in the ZIM schema, all of the constraints placed on metadata management would be up to the implementer. As a result, not all instances of a class would be

[1] Under the present implementation, references between dynamic objects are via regular 4-byte pointers while references between persistent objects are represented as 8-byte doubles

necessarily stored together. Objects could be clustered on a more application-specific basis. For example, all of the sub-components of a toolset diagram could be stored together in a single segment. This would minimize the number of disk head movements required to retrieve a complex object from the object base.

The suggested implementation would be as follows:

- As far as the ZIM database schema was concerned, the object repository would have two fields

(1)	the segment number	(4-byte int, indexed)
(2)	the segment	(1000-byte char)

  All persistent objects would be written directly into segments. Segments would also contain a reference to the next segment - for objects which span multiple segments - and a pointer to the next available offset within the segment for an object to be placed. Segment compaction would be performed by the garbage collector. Obviously, variable-length objects such as Strings and Arrays will require special handling. Since they no longer have to match the structure of a ZIM record, the format of persistent objects is also under the control of the implementer. The persistent format of a class's instances should be identical to its dynamic representation. Any additional information required by the object server could be placed before the object's offset in the segment as an 'object header'. For example, it may be useful to store such things as the object's OID, its lock status, its owner, its version, and a byte for use by the mark-sweep garbage collector.

- A second ZIM database file would contain the persistent object table. The table entries would consist of:

(1)	the Object Identifier	(4-byte int, indexed)
(2)	the object's location [segment # + offset]	(4-byte int)

Objects would be contained in one or more segments. The object's physical location would consist of a 19-bit segment number and a 13-bit offset[1]. A persistent object table is required in order to allow for the movement of objects within and between segments. For example, if the object base garbage collector is to compact the storage reclaimed within segments, it will be necessary to move objects within them.

- Since ZIM concurrency control mechanism implicitly locks at the page level, manipulating segments of this size yields a segment locking strategy. In addition, the transaction commit and rollback scheme supported by ZIM works at the page level, so consistency is maintained at the segment level. Lastly, ZIM's buffer management is also based on pages.

Implementing Persistent Objects Using an Object Table

The key to turning Objective-C into a database programming language is the introduction of an object table into the language's memory management. This approach is based on the Smalltalk language model. The contents of the object table would be based on the persistent object table described above. At run-time, references from one object to another are actually handled as offsets into the object table. All objects - both persistent and dynamic - are referenced via the object table[2]. The contents of the object table contain either the object's

[1] This scheme was motivated by the following: ZIM databases may be configured to have page sizes of 1K, 2K or 4K. Since we want to use offsets into segments to form part of the address of the object, we must allow for a maximum offset of 4096. This translates to 2^{12}. This leaves 19 bits of a 32-bit integer to represent the segment number.
[2] For dynamic Objective-C objects, re-write (*_alloc), (*_realloc), (*_dealloc), and (*_copy) functions so that the memory model recognizes the added level of indirection provided by the object table.

location in memory, or the object's location in the object base. As persistent objects are read into buffers by the ObjectManager, their location would be updated in the object table. The message passing mechanism would require modification to recognize this additional indirection.

The addition of an object table would also require a modification of the code generated by the Objective-C compiler. Presently, each message is passed the variable *self*, where self is memory address of the message receiver. Under this scheme, self could be either the OID of the receiver in the object table, or its memory location. Both of these alternatives pose a problem. If self is assigned the value of the table offset, the compiler will handle the locations of instance variables incorrectly. At present, all instance variable references in Objective-C code is compiled to *self->instVarName*. If self is an object table offset, offsets generated in this way will be incorrect. If self is passed as the actual address of the receiving object, then assignments of self will be incorrect, since the object's address, rather than its OID would be used.

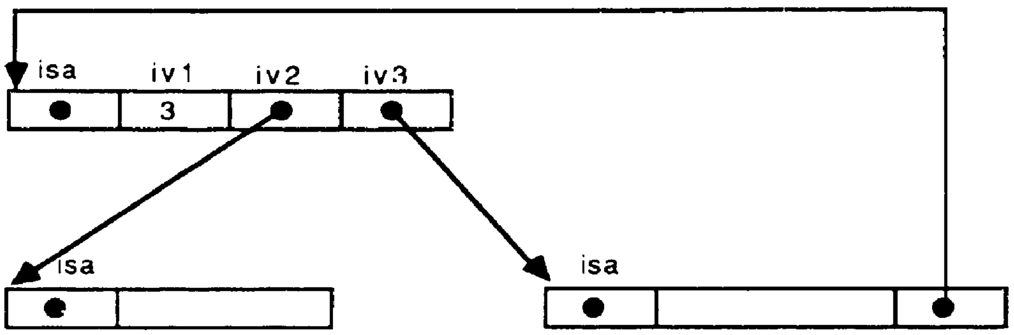
One possible solution could be to add an additional parameter to the functions which are generated by Objective-C to implement methods. This new parameter would be called *location*, and would be the actual memory location of the object in question. Self would contain the object's OID, and location would contain the object's address. Instance variable references would now compile to *location->instVarName*, rather than *self->instVarName*. This approach would ensure that message sends to self would be handled correctly.

The object table entries would contain a variable which indicates whether an object is located on the heap or in the ObjectManager. Note that the entire object table would not be read at start up. Instead, entries would be read as references to them were actually made available to the application as segments are read[1].

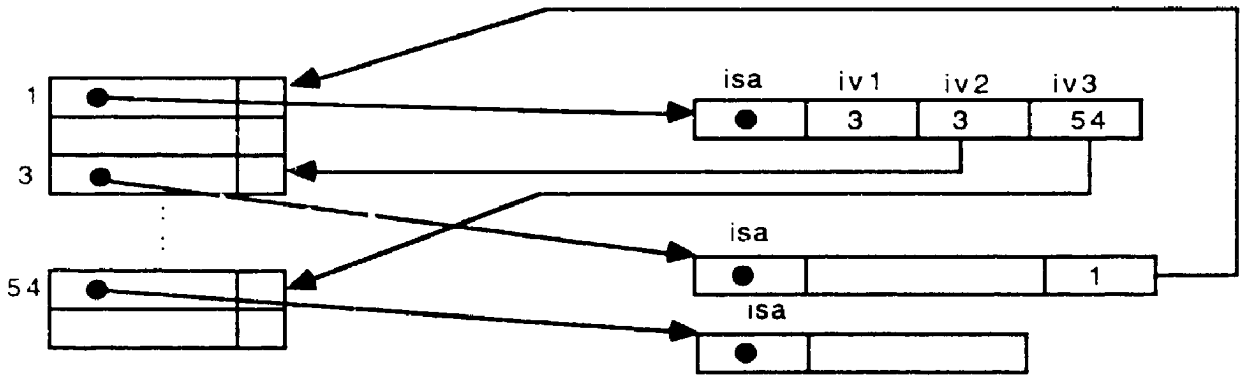
[1] How the object table is actually implemented in memory would be a critical implementation detail. While OIDs are described here as offsets into the object table, this scheme requires an array as large as the highest object identifier number, which could be 2^{31}. Alternative implementations could be based on hash tables or B^*-trees.

Figure 9.2: Object Table Memory Management

All references between objects under the current memory management model are direct - ie. through normal C pointers



Introducing an object table gives an added layer of indirection in the object references. References to other objects are now via offsets in the object table. The message-passing kernel must be modified to recognize this.



The violation of encapsulation which is allowed by the Objective-C compiler would be even more fraught with danger, since the applications developer would also have to explicitly recognize this added level of indirection. This is in addition to the present requirement that any developer who wishes to violate an object's encapsulation must test whether or not the object has been read into memory.

The objects managed by the system would physically reside in two locations: either on the heap as dynamically created objects, or in the object base buffers managed by the ObjectManager. New objects would be allocated on the heap, and their OIDs and locations added to the object table. New OIDs would be allocated sequentially and would not be re-used. All of the classes defined to the application would be capable of being written to the persistent store. Whenever the client application issued a commit, all of the objects reachable from the root object would be written to the object base. Any objects which are preser 'ly located in the heap would be copied to the ObjectManager's buffers, and their dynamically allocated memory returned to the heap. Their new location would then be registered in the object table[1].

When a message was sent to a persistent object which was not in memory, its segment would be read. All of the objects in the segment would have their locations in the object table update to point to their location in the ObjectManager's buffers. All of the objects referenced by the objects in the segment would have their OIDs added to the object table. Thus, objects which have no possibility of being accessed by the application would not be referenced by it. When a segment was committed to the object base, the object's memory location would have to be replaced in the object table by its location in the persistent store.

Using this approach, it would be relatively straightforward to implement a virtual memory for Objective-C. As the buffers of the ObjectManager became full, the segments they contain

[1] Note that this implies that the *becomes:* message must be supported in this implementation. Recall that this message modifies the memory location pointed at by an entry in the object table. In order for this message to work, <u>all</u> objects must be referenced via an object table.

could be written to a file in the user's working directory. The objects contained in the swapped segments would have a toggle in their object table entry modified to reflect this change. New messages to swapped out objects would result in their segments being returned to memory.

Another possible extension to the object server's functionality which is made possible by these proposed changes would be to implement a distributed object server architecture. ZIM allows different database files to be located in separate directories. The location of each database file is contained in a file called *areas.zim*. Using SUN's Network File System, the database files could be located anywhere on the network. Under the normal ZIM environment, this allows the vertical partitioning of the database. All records of a certain entity set may be stored on the same physical file system. Using the approach discussed here, each file could represent a separate object repository. Instances of any number of classes could be contained in each of these repositories, thus providing a truly distributed environment. One could, for example, easily envisage the different toolset users maintaining private design information on their own local object stores, with shared objects maintained in one or more shared object bases.

References

(Aho, Hopcroft and Ullman, 1983) Aho, A., Hopcroft, J., Ullman, J., Data Structures and Algorithms. Addison-Wesley Publishing Co., Don Mills, Ont., 1983.

(Andrews and Harris, 1987) Andrews, T., Harris, C. "Combining Language and Database Advances in an Object-Oriented Development Environment". OOPSLA 1987 Proceedings, p. 430-440.

(Atkinson and Buneman, 1987) Atkinson, M.P., Buneman, O.P., "Types and Persistence in Database Programming Languages", *ACM Computing Surveys*, vol 19, no. 2, June, 1987, p. 105-190.

(Atwood, 1985) Atwood, T., "An Object-Oriented DBMS for Design Support Applications", COMPINT 1985, p. 299-307.

(Banjeree, et al, 1987) Banjeree, J., Chou, H., Garza, J., Kim, W., Woelk, D., Ballou, N., Kim, H. "Data Model Issues for Object-Oriented Applications" *ACM Transactions on Office Information Systems*, Vol. 5, No. 1, January 1987, p. 3-26.

(Barbedette and Richard, 1986) Barbedette, G., Richard, P. "VOOD: The Verso Object-Oriented Data Model". Research Report #580, Institut National de Recherche en Informatique at en Automatique, France, November 1986.

(Baroody and DeWitt, 1981) Baroody, A., DeWitt, D., "An Object-Oriented Approach to Database System Implementation". *ACM Transactions on Database Systems*, December, 1981, Vol. 6, No. 4, p. 576-601.

(Belkhatir and Estublier, 1986) Belkhatir, N., Estublier, J., "Experience With a Data Base of Programs", COMPSAC 1986 Proceedings, p. 84-91.

(Bennett, 1987) Bennett, J. "The Design and Implementation of Distributed Smalltalk" OOPSLA 1987 Proceedings, p. 318-330.

(Bernstein, 1987) Bernstein, P. "Database System Support for Software Engineering - An Extended Abstract". Proceedings of the Ninth International Conference on Software Engineering, April, 1987, p. 166-178.

(Bigelow, 1988) Bigelow, J. "Hypertext and CASE" *IEEE Software*, March, 1988, p. 23-27.

(Bloom and Zdonik, 1987) Bloom, T., Zdonik, S. "Issues in the Design of Object-Oriented Database Programming Languages". OOPSLA 1987 Proceedings, p. 430-440.

(Booch, 1986) Booch, G. "Object-Oriented Development". *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 2, February, 1986, p. 211-221

(Borgida, Mylopolous and Wong, 1984) Borgida, A., Mylopolous, J., Wong, H.K.T. "Generalization/Specialization as a Basis for Software Specification". In Brodie, M., Mylopolous, J., Schmidt, J. (ed.) On Conceptual Modelling, Springer-Verlag Berlin, Germany, 1984.

(Brodie, 1981) Brodie, M., "On Modelling Behavioral Semantics of Databases". Proc. 7th International Conference on Very Large Databases, Cannes, France, September, 1981.

(Brodie, 1984) Brodie, M. "On the Development of Data Models". In Brodie, M., Mylopolous, J., Schmidt, J. (ed.) On Conceptual Modelling, Springer-Verlag, Berlin, Germany, 1984.

(Brodie and Ridjanovic, 1984) Brodie, M., Ridjanovic, "On the Design and Specification of Database Transactions". In Brodie, M., Mylopolous, J., Schmidt, J. (ed.) On Conceptual Modelling, Springer-Verlag, Berlin, Germany, 1984.

(Brooks, 1987) Brooks, F.P., "No Silver Bullet - Essence and Accidents of Software Engineering", *IEEE Computer*, April, 1987, p. 10-19.

(Chifosky, 1988) Chifosky, E. "Software Technology People Can Really Use". *IEEE Software*, March, 1988, p. 8-10.

(Chikofsky and Rubenstein, 1988) Chikofsky, E., Rubenstein, B. "CASE: Reliability Engineering for Information Systems". *IEEE Software*, March, 1988, p. 11-16.

(Cockshot, et al, 1984) Cockshot, W.P., Atkinson, M.P., Chisholm, K.J., Bailey, P.J., Morrison, R., "Persistent Object Management System", Software Practice and Experience, vol. 14, 1984, p. 49-71.

(Copeland and Khoshaflan, 1985) Copeland, G., Khoshafian, S. "A Decomposition Storage Model". Proceedings of ACM-SIGMOD 1985 International Conference on the Management of Data, 1985, p. 268-279.

(Cox, 1984) Cox, B. "Message/Object Programming: An Evolutionary Change in Programming Technology". *IEEE Software*, January 1984, p. 50-61.

(Cox, 1986) Cox, B., Object-Oriented Programming: An Evolutionary Approach, Addison-Wesley Publishing Company, Don Mills, Ontario, 1986.

(Cox and Schmucker, 1987) Cox, B., Schmucker, K. "PRODUCER: A Tool for Translating Smalltalk-80 to Objective-C". OOPSLA 1987 Proceedings, p. 423-429.

(Cureton, 1988) Cureton, B. "The Future of Unix in the CASE Renaissance". *IEEE Software*, March, 1988, p. 18-22.

(Date, 1983) Date, C.J., An Introduction to Database Systems, Volume II, Addison-Wesley Publishing Company, Don Mills, Ontario, 1983.

(Decouchant, 1986) Decouchant, D. "Design of a Distributed Object Manager for the Smalltalk-80 System". OOPSLA 1986 Proceedings, p. 444-452.

(Digitalk, 1986) Digitalk Inc., Smalltalk/V Tutorial and Programming Handbook. Digitalk Inc., Los Angeles, CA, 1986.

(Duff, 1986) Duff, C. "Designing an Efficient Language". *Byte*, August, 1986, p. 211-224.

(Elmasri and Navathe, 1984) Elmasri, R., Navathe, S., "Object Integration in Logical Database Design". Proc. IEEE COMPDEC Conference, 1984.

(Elmasri, Weeldreyer and Hevner, 1985) Elmasri, R., Weeldreyer J., Hevner, A., "The Category Concept: An Extension to the Entity-Relational Model". In Data & Knowledge Engineering 1, North-Holland Pub., 1985.

(Emeraude, 1987) "Emeraude: General Presentation". 1987

(Fishman, et al, 1987) Fishman D., et al "Iris: An Object-Oriented Database Management System". *ACM Transactions on Office Information Systems*, Vol. 5, No. 1, January 1987, p. 48-69.

(Furtado and Casanova, 1985) Furtado, A., Casanova, M. "Updating Relational Views" In Kim, W., Reiner, D., Batory, D. (ed.), Query Processing in Database Systems, Springer-Verlag, Berlin, Germany, 1985.

(Gallo, Minot and Thomas, 1986) Gallo, F., Minot, R., Thomas, I., "The Object Management System of PCTE as a Software Engineering Database Management System". Proc. COMPSAC '86, IEEE, 1986.

(Goldberg and Robson, 1983) Goldberg, A., Robson, D., Smalltalk-80: The Language and its Implementation, Addison-Wesley Publishing Company, Don Mills, Ontario, 1983.

(Hammer and McLeod, 1981) Hammer, M., McLeod, D., "Database Description with SDM A Semantic Database Model". *ACM Transactions on Database Systems*, September, 1981, Vol. 6, No. 3.

(Hornick and Zdonik, 1987) Hornick, M., Zdonik, S. "A Shared, Segmented Memory System for an Object-Oriented Database". *ACM Transactions on Office Information Systems*, Vol. 5, No. 1, January 1987, p. 70-95.

(Katz, 1984) Katz, R. "Transaction Management in the Design Environment" In Gardain, G., Gelenke, E. New Applications of Data Bases, Academic Press, Toronto, 1984.

(Khoshafian, et al, 1987) Khoshafian, S., Copeland, G., Jagodits, J., Boral, H., Valduri, E. "A Query Processing Strategy for the Decomposed Storage Model". Proceedings of the Third International Conference on Data Engineering, Feb., 1987, p. 636-643.

(Kim, et al, 1987) Kim, W., Banjeree, J., Chou, H., Garza, J, Woelk, D. "Composite Object Support in an Object-Oriented Database System" OOPSLA 1986 Proceedings

(Lalonde, Thomas and Pugh, 1986) LaLonde, W., Thomas, D., Pugh, J. "An Examplar Based Smalltalk". OOPSLA 1986 Proceedings.

(Liskov, 1988) Liskov, B., "Data Abstraction and Hierarchy", OOPSLA 1987: Addendum to the Proceedings. SIGPLAN Notices, vol. 23, Number 5, May, 1988.

(Lorie, et al, 1985) Lorie, R., Kim, W., McNabb, D., Plouffe, W., Meier, A. "Supporting Complex Objects in a Relational System for Engineering Databases". In Kim, W., Reiner, D., Batory, D. (ed.), Query Processing in Database Systems, Springer-Verlag, Berlin, Germany, 1985.

(MacLennan, 1982) MacLennan, B. "Values and Objects in Programming Languages". *SIGPLAN Notices*, Vol. 17, No. 12, December, 1982, p. 70-79.

(Maier, et al, 1986) Maier, D., Stein, J., Otis, A., Purdy, A., "Development of an Object-Oriented DBMS". OOPSLA 1986 Proceedings, p. 472-482.

(Mark and Roussopoulos, 1986) Mark, L., Roussopoulos, N. "Metadata Management". *IEEE Computer*, December, 1986, p. 26-36.

(Martin, 1988) Martin, C. "Second-Generation CASE Tools: A Challenge to Vendors". *IEEE Software*, March, 1988, p. 46-49.

(McCullough, 1987) McCullough, P. "Transparent Forwarding: First Steps". OOPSLA 1987 Proceedings, p 331-341.

(Merrow and Laursen, 1987) Merrow, T., Laursen, J. "A Pragmatic System for Shared Persistent Objects" OOPSLA 1987 Proceedings, p. 103-110.

(Nierstrasz and Tsichritzis, 1985) Nierstrasz, O., Tsichritzis, D. "An Object-Oriented Environment for OIS Applications". Proceedings of the 1985 Conference on Very Large Database Systems, 1985, p. 335-345.

(Neuhold, 1986) Neuhold, E. "Objects and Abstract Data Types in Information Systems". In Steel, T., Meersman, R. Database Semantics, IFIP, 1986, p. 1-12.

(Penney and Stein, 1987) Penney, D.J., Stein, J. "Class Modification in the GemStone Object-Oriented DBMS". OOPSLA 1987 Proceedings, p. 111-117.

(Power and Weiss, 1988) Power, L., Weiss, Z., eds., OOPSLA 1987: Addendum to the Proceedings. SIGPLAN Notices, vol. 23, Number 5, May, 1988.

(Purdy, Schuchardt and Maier, 1987) Purdy, A., Schuchardt B., Maier, D. "Integrating an Object Server with Other Worlds". *ACM Transactions on Office Information Systems*, Vol. 5, No. 1, January 1987, p. 27-47.

(Rentsch, 1982) Rentsch, T. "Object Oriented Programming". *SIGPLAN Notices*, Vol. 17, No. 9, September, 1982, p. 51-56.

(Rumbaugh, 1987) Rumbaugh, J. "Relations as Semantic Constructs in an Object-Oriented Language". OOPSLA 1987 Proceedings, p. 466-481.

(Skarra and Zdonik, 1986) Skarra, A., Zdonik, S. "The Management of Changing Types in an Object-Oriented Database". OOPSLA 1986 Proceedings, p. 483-495

(Smith and Zdonik, 1987) Smith. K., Zdonik, S. "Intermedia: A Case Study of the Difference Between Relational and Object-Oriented Database Systems". OOPSLA 1987 Proceedings, p. 452-465.

(Stefik and Bobrow, 1985) Stefik, M., Bobrow, D. "Object-Oriented Programming: Themes and Variations". *The AI Magazine*, 1985, p. 40-62.

(Stein and Maier, 1988) Stein, J., Maier, D. "Concepts in Object-Oriented Data Management". *Database Programming and Design*, April, 1988, p. 58-67

(Stonebraker, Anton and Hanson, 1987) Stonebraker, M., Anton, J., Hanson, E. "Extending a Database Ssytem with Procedures". *ACM Transactions on Database Systems*, Vol. 12, No. 3, September 1987, p. 350-376.

(Stonebraker and Rowe, 1986) Stonebraker, M., Rowe, L. "The Design of POSTGRES" ACM, 1986. p. 340-355.

(Stroustrup, 1986) Stroustrup, B. "An Overview of C++". *SIGPLAN Notices*, Vol. 21, No 10, October, 1986, p. 7-18.

(Stroustrup, 1988) Stroustrup, B., "What is Object-Oriented Programming?", *IEEE Software*, May, 1988, p. 10-20.

(Symonds, 1988) Symonds, A. "Creating a Software-Engineering Knowledge Base" *IEEE Software*, March, 1988, p. 50-56.

(Tsichritzis, 1981) Tsichritzis, D. "Integrating Data Base and Message Systems" IEEE, 1981, p. 356-362.

(Tsichritzis and Lochovsky, 1982) Tsichritzis, D., Lochovsky, F. Data Models. Prentice-Hall, Englewood Cliffs, N.J., 1982.

(Wiederhold, 1984) Wiederhold, G. "Knowledge and Database Management" *IEEE Software*, December, 1984, p. 63-73.

(Wiederhold, 1986) Wiederhold, G. "Views, Objects, and Databases" *IEEE Computer*, December, 1986, p. 37-44.

(Wile and Allard, 1986) Wile, D., Allard, D., "Worlds: an Organizing Structure for Object-Bases", COMPSAC 1986 Proceedings, p. 16-26.

(Zaniolo, et al) Zaniolo, C., Ait-Kaci, H., Beech, D., Cammarata, S., Kerschberg, L., Maier D. "Object-Oriented Database Systems and Knowledge Systems"

(Zanthe, 1986) Zanthe Information Inc. ZIM Programming Language Interface, June, 1986.

(Zdonik, 1984) Zdonik, S. "Object Management System Concepts". Proceedings of the Second ACM-SIGOA Conference on Office Information Systems, 1984, p. 13-19.

(Zdonik, 1986) Zdonik, S., "Maintaining Consistency in a Database with Changing Types". Proceedings of the Object-Oriented Programming Workshop, June, 1986, p. 120-127.

(Zintz, 1988) Zintz, W. "Moving From C to Object-Oriented C". *UnixWorld*, May, 1988, p. 60-66.

END

18·02·90

FIN