1

**National Library
of Canada**

**Bibliothèque nationale
du Canada**

Canadian Theses Service

Service des thèses canadiennes

**Ottawa, Canada
K1A 0N4**

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

APPLICATION PROGRAMMER CONTROLLED SHARED, PERSISTENT OBJECTS

by

James A.Q. Stringham

**A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of
Master of Computer Science**

**School of Computer Science
Carleton University
Ottawa, Ontario
August 1st, 1990
© copyright
1990, James A.Q. Stringham**

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

ISBN 0-315-60482-4

**The undersigned recommend to
the Faculty of Graduate Studies and Research
acceptance of the thesis,**

**APPLICATION PROGRAMMER CONTROLLED
SHARED, PERSISTENT OBJECTS**

submitted by

James A.Q. Stringham. B.A.Sc.

**in partial fulfillment of the requirements for
the degree of Master Of Computer Science**

A handwritten signature in cursive script, appearing to read "D.A. Thomas", written over a solid horizontal line.

Thesis Co-Supervisor

A handwritten signature in cursive script, appearing to read "J. R. Pugh", written over a solid horizontal line.

Thesis Co-Supervisor

A handwritten signature in cursive script, appearing to read "J. Fiala", written over a solid horizontal line.

Chairman, School of Computer Science

**Carleton University
August 31st, 1990**

Abstract

Our objectives in this thesis are to provide support for shared, persistent objects, and to give the application programmer control over how these objects are implemented and used.

Our second objective is novel. The support for sharing and persistence must be flexible and adaptable; the programmer must not be limited to a fixed set of sharing and persistence mechanisms.

We describe an experimental prototype persistent object system for Smalltalk that adds persistence operations to volatile objects, supports composite references to define the boundaries of persistent objects and to cluster persistent objects on disk, and uses transactions to control shared access to persistent objects on disk. Our prototype has a distinct advantage over other systems: the application programmer can select, change or augment the persistence operations and transactions to suit the application. We demonstrate this advantage by subclassing the Transaction class to create new transaction mechanisms.

Acknowledgements

I would like to thank Professor Dave Thomas for his persistent inspiration, encouragement and guidance throughout this thesis, and Professor John Pugh for his kind, constructive criticism during the final writing stages.

This research was supported by Object Technology International (OTI) and the Rockwell International Science Center (RISC). I am indebted to them for their assistance, and in particular, to Dr. David Morley of RISC.

I enjoyed collaborating with Thong Nguyen on Sticky, at Carleton and through long distance email ;-)

I am grateful to my colleagues at OTI, John Duimovich, Dan Lesage, Jim McGugan, and Mike Milinkovich, for generously sharing their insights into the mysteries of Smalltalk and database systems.

Finally, and most especially, my thanks to Suzanne for her love and support. I wish her luck on her thesis, and hope I can be half as much of a help to her as she was to me.

Table of Contents

1	Introduction	1
1.1	Motivation	1
1.2	Sticky	4
1.3	Example Application	5
1.4	Thesis Organization	8
2	Related Research	9
2.1	Persistent Objects	9
2.1.1	Object-Oriented Database Systems	9
2.1.2	Persistent Object Systems	11
2.2	Object Clustering	13
2.3	Composite Objects	15
2.3.1	Part-Whole Hierarchies	17
2.3.2	Part-Of Relations	18
2.3.3	Composites in ORION	19
2.4	Transactions and Concurrency Control	21
2.4.1	Timestamp Ordering	25
2.4.2	Locking	26
2.4.3	Co-operative Sharing	27
2.5	Comparison of Systems	28
3	Persistent Objects	30
3.1	Adding Persistence Operations	30
3.2	Object Unloading and Loading	33
3.3	Object Faults	37
4	Composite Objects	40
4.1	Propagation of Persistence	41

4.2	Composite Clustering Policies	44
4.3	Part Variables	49
5	Transactions	56
5.1	Crash Recovery Schemes	57
5.2	Optimistic Concurrency Control	58
5.2.1	Kung and Robinson's Method	61
5.2.2	Rahm's Method	64
5.3	Optimistic Transactions	65
5.3.1	Transaction	65
5.3.2	Object Controller	68
5.4	Transaction Subclasses	71
5.4.1	SaveTransaction	71
5.4.2	LargeTransaction	72
5.4.3	InformedTransaction	74
5.5	Integration with Secondary Storage Garbage Collection	77
6	Conclusion	81
6.1	Future Work	82
7	References	84

List of Figures

Figure 1.1 CAD application environment	6
Figure 1.2 CAD schematic	7
Figure 2.1 Bicycle component hierarchy	16
Figure 2.2 Design loaded from disk	22
Figure 2.3 Design changed	22
Figure 2.4 Inconsistent update	22
Figure 2.5 Design loaded into two workstations	23
Figure 2.6 Design saved from workstation #1	23
Figure 2.7 Design saved from workstation #2	23
Figure 2.8 Comparison of systems	29
Figure 3.1 Volatile point	32
Figure 3.2 Point becomePersistent	32
Figure 3.3 aLine holds references to persistent points	34
Figure 3.4 Line unloaded	36
Figure 3.5 aLine loaded back into m in memory	36
Figure 3.6 Proxy shortcircuited	37
Figure 3.7 aLine keeps disk references	38
Figure 4.1 Limited number of backward pointers	45
Figure 4.2 A collection of backward pointers	46
Figure 4.3 Replacing Old component	47
Figure 4.4 Old no longer a component	47
Figure 4.5 New now a component	47
Figure 4.6 New assigned to composite	47
Figure 4.7 Component collection	51

Figure 4.8 Signal connecting microprocessor to nand gate	53
Figure 4.9 New signal is not persistent	54
Figure 4.10 Persistence propagated to signal and components, but not to selected	55
Figure 5.1 Optimistic transaction phases	62
Figure 5.2 Scheduling conditions	62
Figure 5.3 Unnecessary rollback	62
Figure 5.4 Design loaded into two workstations	76
Figure 5.5 Design saved, invalid on workstation #2	76
Figure 5.6 Design loaded into workstation #2	76
Figure 5.7 aLine adds a new point in main memory	79
Figure 5.8 aPoint on disk is garbage collected	79
Figure 5.9 System crashes, aLine on disk is inconsistent	80
Figure 5.10 Reference Set prevents aPoint on disk from being garbage collected	80

1 Introduction

1.1 Motivation

Our objectives in this thesis are to provide support for *shared, persistent* objects¹, and to give the application programmer control over how these objects are implemented and used. There is a need to model complex state and behaviour in software applications for Computer Aided Design (CAD), Computer Aided Software Engineering (CASE), and Office Information Systems (OIS) [Maier 1989], [Nierstrasz and Tschritzis 1989]. These applications also require computing environments that provide access to large quantities of shared, persistent data. Object-oriented programming languages have the expressive power needed to model complex state and behaviour. However, object-oriented programming environments do not provide adequate support for shared, persistent objects.

Commercial database systems provide access to shared, persistent data. The data is stored in a database, on a secondary storage device such as a hard disk. Shared access to the data is controlled by a database management system (DBMS) through transactions.

Commercial database systems have been designed to support applications such as payroll processing, in which the data is uniform and fits into simple record data structures. By contrast, in applications for CAD, CASE, and OIS, the data is varied and requires deeply nested, complex data structures. Copeland and Maier argue that commercial database systems do not provide adequate support for these kinds of applications [Copeland and Maier 1984], for the following reasons:

¹ A persistent object is an object that survives until it is no longer needed, often outliving the process that creates it. A shared, persistent object is a persistent object that is shared between several users in separate workspaces.

1) Most database systems have limited type definition facilities. They furnish a fixed set of simple types that cannot be augmented with new simple types or operations on the existing types. In addition, they have limited constructors for more complex types.

2) Database data structures do not match the complexity and variety of the application data structures. Consequently, the application data must be simplified or transformed into a format suitable for the database. In addition, changes to the structure of the data in the application often require a reconfiguration of the database.

3) Application programmers may have to contend with up to three separate languages: a data definition language to define the schema of the database, a data manipulation language to add, delete, request, and update data in the database, and a general purpose language to process the retrieved data in the application. The language paradigms and data types of the data manipulation language and the general purpose language are often different. These differences constitute an *impedance mismatch*.

Relational databases have come under particular scrutiny in hypermedia and CAD applications. Smith and Zdonik have found that flattening a hypermedia system's complex, hierarchical data structures into relations is awkward [Smith and Zdonik 1987]. Maier argues that relational databases are too slow for CAD, because they are too slow at fetching and updating individual records [Maier 1989]. In particular, because key values are logical pointers to entries in the database, at least one address translation is required to access the desired tuple. In contrast, objects in memory have direct pointers to other objects; memory objects are addressable by their identity as opposed to their state. Sutherland states that current CAD systems can only access on the order of 10 objects/second from the database, when performance of 1000 objects/second is required [Sutherland 1989].

Commercial database systems have another limitation: they provide a fixed set of transaction mechanisms, and these mechanisms cannot be changed or augmented by the application programmer. Traditional applications such as payroll processing require only a few, general transaction mechanisms. However, applications for CAD, CASE and OIS require specialized transaction mechanisms that are particular to the application [Skarra and Zdonik 1989], [Chrysanthis and Ramamritham 1990]. We argue that the support for sharing and persistence in these applications must be adaptable. The application programmer must be able to tailor the support to fit the application.

In response to the inadequacies of commercial database systems, researchers have added object-oriented facilities to database systems. An *object-oriented database system* supports user defined abstract data types, user defined operations on data types, and data identity, while often retaining common database facilities such as query processing.

Nevertheless, object-oriented database systems do not correct all of the shortcomings found in commercial systems. For example, if the data manipulation language and the general purpose object-oriented programming language are not the same, there will still be an impedance mismatch between them. In addition, although some of these systems provide specialized transaction mechanisms, most do not give the application programmer the ability to change or augment them.

Bloom and Zdonik discuss an alternative approach: add the database notions of persistence and sharing to the general purpose object-oriented programming language [Bloom and Zdonik 1987]. Using this approach, the support for sharing and persistence is seamlessly integrated into the environment. There is no impedance mismatch because only one representation is used. We call this addition to the object-oriented programming environment a *persistent object system*.

Persistent object systems use two approaches to implement shared, persistent objects. One approach is to modify the underlying mechanisms that support objects. This hides the support for sharing and persistence from the application programmer. The second approach is to add persistence operations to volatile objects, and to use transactions to control shared access to persistent objects on disk. This gives the programmer control over how the support for sharing and persistence is implemented and used.

In this thesis, we describe an experimental prototype persistent object system that uses the second approach. By using this approach, we satisfy both of our objectives: we provide support for shared, persistent objects, and we give the application programmer control over how these objects are implemented and used.

Our second objective is novel. In other systems, the application programmer is limited to a fixed set of sharing and persistence mechanisms. This is inadequate; the support for sharing and persistence must be flexible and adaptable so that the programmer can tailor the support to fit the application. In our system, the programmer can select, change or augment the persistence operations and transactions to meet the specific requirements of the application. This gives our prototype a distinct advantage over existing object-oriented databases and persistent object systems.

1.2 Sticky

Smalltalk/V [Digitalk 1988] is an example of an object-oriented programming environment that lacks support for shared, persistent objects. It runs in a private, single user workspace. The workspace can accommodate only a limited number of objects at any time. Objects in the workspace are volatile; they are lost if the system crashes. Smalltalk/V

has no facilities for coordinating the sharing of objects between multiple workspaces, although there are two ways to make objects persist on disk. One way is to save the image; this stores a snapshot of all the objects in memory. However, objects stored on disk in this way cannot be shared, because the objects can only be restored all together, by loading the entire image. Another way is to use the Smalltalk/V loader/dumper facility; this stores objects on disk singly or in groups. Unfortunately, the objects lose their property of identity in the process: the same object can be saved to several files, and subsequently reloaded from each, resulting in a new instance of the same object each time.

This thesis describes aspects of an experimental prototype persistent object system; the system is called Sticky, to denote that objects stick around until they are no longer needed. Sticky runs on a single workstation, and is integrated with a secondary storage garbage collector, called Creepy [Nguyen 1989]. The prototype is implemented in Smalltalk/V, but the underlying concepts can be applied to other object-oriented programming environments.

1.3 Example Application

We use the following example application to aid in the exposition of our prototype. A team of electronics engineers use a CAD toolkit, written in Smalltalk, to lay out circuit board schematics. Each engineer has a workstation, running Smalltalk augmented with the Sticky persistent object system. The workstations are connected by a network to a common object repository (Figure 1.1). The engineers use the object repository to store shared, persistent objects, such as design schematics, specifications, and reports.

Figure 1.1 CAD application environment

Figure 1.2 CAD schematic

The object repository has a hard disk for storing objects, an object controller to control access to the objects on disk, and stable storage to provide data integrity (see [Lampson et al. 1981] for a detailed description of stable storage). The persistent object system in each workstation communicates with the object controller to read objects from the repository, and to write them back into the repository.

A typical design session for an engineer involves retrieving a design from the object repository, making changes to the design by adding, removing and modifying objects, and storing the design in the object repository. A session may last for several hours. During this time, other engineers may access the same design to look at it or to modify it; some may even work on the design in a co-operative effort. Figure 1.2 illustrates the objects found in a typical CAD schematic. A circuit board design includes semiconductor devices, such as microprocessors and gate arrays, and the signals that connect these devices.

1.4 Thesis Organization

In the next chapter, we survey related work on object-oriented databases and persistent object systems, and compare their approaches for providing support for shared, persistent objects.

We then present three different aspects of our prototype persistent object system:

- 1) How the prototype provides support for persistence, including how volatile objects become persistent, and how persistent objects are loaded and unloaded (Chapter Three).
- 2) How the prototype provides support for propagating persistence to volatile objects and for clustering objects on disk to make object loading and unloading more efficient (Chapter Four).
- 3) How the prototype provides support for sharing persistent objects (Chapter Five).

2 Related Research

Object-oriented programming environments are handicapped by their poor support for objects that are persistent and shared. Systems that have been designed to address this problem fall into two categories: object-oriented database systems and persistent object systems. In this chapter, we survey systems from both categories and compare their approaches to providing support for shared, persistent objects.

2.1 Persistent Objects

Essentially, an object is made persistent by storing it in a non-volatile storage medium such as a hard disk. Of course, some means are required to create, retrieve, update, restore and reclaim these disk-based objects. The approaches used in object-oriented database systems and persistent object systems are fundamentally different. For this reason, they are considered separately below.

2.1.1 Object-Oriented Database Systems

Object-oriented database systems support user defined abstract data types, user defined operations on data types, and data identity. Objects are made persistent by defining appropriate abstract types in the data definition language, and then creating object instances in the database using the data manipulation language. The data manipulation language is also used to retrieve, update and delete the objects in the database. As most data

manipulation languages are not computationally complete, the processing of the retrieved objects is performed in a general purpose object-oriented language, such as Smalltalk/V. Object-oriented database systems do not hide the two levels of store; objects must be explicitly requested from the database before they can be used in memory.

Some object-oriented database systems are derived from existing database systems, so it is not unusual that they bear a strong resemblance to their forbears. POSTGRES [Stonebraker and Rowe 1986], [Rowe 1986], [Stonebraker 1986] is a descendent of the INGRES relational database management system. POSTGRES extends the relational model with user defined data types and data type operations. Complex objects¹ are represented as procedures that access multiple tuples.

EXODUS [Carey et al. 1986], [Carey et al. 1988] is a toolkit of kernel database facilities that can be used to construct application-specific database systems. Its data definition language and data manipulation language are based on the language C. EXODUS supports user defined data types (including complex objects) and data type operations. ObServer [Skarra et al. 1986] is a persistent store for untyped byte sequences. ObServer has been used as the backend for an object-oriented database and as an object store for an object-oriented graphics system, called GARDEN. Mneme [Moss and Sinofsky 1988] is an interface to a persistent heap; object servers such as ObServer and EXODUS are used to implement the persistent heap. Client object-oriented programming environments such as Owl, Smalltalk-80 and CLU, access the heap through a procedural interface. ORION [Kim et al. 1987a], [Kim et al. 1987b], [Kim et al. 1988], [Kim et al. 1989] is an object-oriented database system implemented in Common LISP. ORION has been used to support applications written in Proteus, an expert system shell.

¹ A complex object is an object that contains references to other objects, as opposed to primitive objects, such as integers and character strings, that do not contain references.

Object-oriented and database concepts are merged in GemStone [Maier et al. 1986]. GemStone's proprietary object-oriented language, OPAL, is used for data definition, manipulation and computation. GemStone possesses some of the characteristics of a persistent object system: OPAL objects are persistent and shared. The Stone object memory presents the illusion of a one-level store for all OPAL objects. Yet, like the object-oriented databases above, GemStone is founded on a data model that has been extended to support complex objects, and supplies agent interfaces to support applications written in client languages such as C and Pascal.

2.1.2 Persistent Object Systems

Persistent object systems have their origins in systems that extend the address space of programming environments to include secondary storage. The Persistent Object Management System [Cockshot et al. 1984] extends the Algol heap to secondary storage, making it persistent. The Large Object-Oriented Memory [Kaehler 1986] extends a 16-bit Smalltalk-80 address space to 32 bits by making objects on disk addressable with 32-bit pointers. The Persistent Memory [Thatte 1986] supports a uniform memory abstraction for both primary and secondary storage in a Lisp environment, giving the illusion of one level of persistent memory. These systems do not meet the requirements of a persistent object system however, because they only support persistent objects; they do not support shared, persistent objects.

Persistent object systems use a combination of two techniques to implement shared, persistent objects: one technique is to modify the underlying mechanisms that support objects, and the other is to add persistence operations to volatile objects and use

transactions to control shared access to persistent objects on disk. Persistent object systems can be distinguished by how much they rely on each technique.

Avance [Bjornerstedt and Britts 1988] uses lower level mechanisms in its virtual machine to make objects shared and persistent. Avance supports two kinds of objects: packets, and datatype values. Packets are persistent and sharable, whereas datatype values are only private, local values. VBASE [Andrews and Harris 1987] is an object-oriented development environment for an object-oriented extension of the C language. VBASE also uses low-level mechanisms to make all abstract data types defined in its Type Definition Language (TDL) and created in its C Object Processing (COP) language automatically persistent and sharable. The persistent objects of Avance and VBASE are straightforward to use. Nevertheless, it is difficult for the application programmer to tailor the persistence facilities to suit the application, because he does not have access to the lower level mechanisms in the programming environment.

Other persistent object systems add dedicated persistent container objects to the environment. Volatile objects placed in the containers become shared and persistent as a result. Trellis/Owl [O'Brien et al. 1986] is a strongly typed, multiple-inheritance, object-oriented programming language, which uses database collections to store volatile objects. The database collections are preserved in an object repository. In Coral3 [Morrow and Laursen 1987], instances of SharedObjectHolder and SharedCollection act as persistent containers for volatile Smalltalk-80 objects. The approach taken in Trellis/Owl and Coral3 makes no attempt to hide the two levels of object store. The programmer must extract the volatile objects from their persistent containers before they can be used, and must put them back into the containers when he wants them to be stored on disk.

The Shared, Persistent Object Store [Low 1988] for Smalltalk-80 adds persistence operations directly to the volatile objects. Persistence methods are implemented in the class

Object, so that all objects inherit the operations. Arjuna [Dixon et al. 1989] supports shared, persistent objects in a distributed C++ programming environment. Arjuna also adds the persistence operations to its root class, StateManager. The approach taken in these two systems is flexible; the operations can easily be modified in the programming environment to suit the application. However, attaching the persistence operations to all objects is unnecessary, and imposes penalties on objects that do not need to be persistent. For example, if each persistent object requires an extra instance variable to store its current timestamp, all objects, including volatile objects, must make room for this variable.

2.2 Object Clustering

The time taken to access an object in memory is the same for all objects. However, this is not true of an object on disk. An object on disk must be loaded into memory through a disk request before it can be used. The time taken to complete a disk request can be divided into three parts: *seek time*, *latency time* and *transfer time*. The seek time is the time the disk head takes to move to the appropriate track on the disk. The latency time is the time the disk takes to rotate to the correct block in the track. The transfer time is the time required to transfer the object from disk to memory. Note that the seek time and latency time vary according to the starting position of the disk head and the location of the requested object. Therefore, the time taken to access several objects on disk is greatest when each object is loaded in a separate disk request and each disk request accesses a different part of the disk.

Fetching objects that are frequently used together in one disk request reduces the number of disk requests, and thereby improves the performance of the application. Working on this principle, ObServer uses a primitive locality of use technique. Objects

unloaded to disk at the same time are stored next to each other using a first free fit algorithm, under the assumption that they will be used at the same time when brought back into memory.

Arjuna and ORION group objects on disk according to their class. Arjuna stores all instances of a class in separate ObjectStores on disk; ORION stores all instances of a class in the same section on disk. Grouping instances according to their class reflects the conventional database definition for class, whereby a class not only describes instances that have the same behaviour, but holds all the instances as well. This technique makes it straightforward to find an instance, by starting at its class. However, it does not necessarily make disk requests more efficient, as applications rarely use objects of only one class at a time.

Trellis/Owl database collections and EXODUS files put a less restrictive constraint on the objects they hold: all instances must be of a given class, or of one of its subclasses. Thus, a group of arbitrary instances could be stored together if the given class was the superclass of all classes; in Smalltalk/V this would be the class Object.

Recognizing that the application programmer often knows which objects will be used together, several conventional database systems give the programmer a mechanism to specify how objects should be clustered on disk. For example, in the DTBG CODASYL model, this mechanism is called *placement near* [Korth and Silberschatz, 1986]. GemStone and VBASE provide similar mechanisms for clustering. In ORION, this mechanism is used to group objects from several classes in the same section on disk.

O'Brien et al. discuss several clustering mechanisms that could be used to override the default grouping of objects by class in Trellis/Owl [O'Brien et al. 1986]. For example, the clustering could be specified in the type definition of the objects, or in the definitions of the

database collections. However, their initial approach is based on the notion of major and minor objects; minor objects will be grouped with the major object that references them.

2.3 Composite Objects

The programmer can fine-tune the performance of his persistent object application by specifying the clustering of objects on disk. Yet, Manola argues that programmers find it hard to use explicit clustering mechanisms [Manola 1987]. Fortunately, it is possible to exploit knowledge of the relationships between objects to determine how they should be clustered, thereby removing this burden from the programmer.

Objects contain references to other objects. These references often imply a certain relationship between the objects. For example, a reference may imply that an object is *made-up-of* another object, or the converse, that an object is *part-of* another object. These relationships are found in the real world: a bicycle is made up of components such as a frame, a saddle, handlebars, wheels, pedals, brakes, gears, and a chain. A component, in turn, may be made up of components: a bicycle wheel is made up of a hub, spokes, a rim and a tire. This breaking down of entities into components continues until a group of basic components is reached. (For all practical purposes, there are no sub-components to these basic components, as it is not useful to reduce entities to, say, their subatomic particle components). A hierarchy of components is revealed by traversing the *made-up-of* relationships from the bicycle to its components and in turn traversing their *made-up-of* relationships to their components until all basic components are reached (Figure 2.1).

Figure 2.1 Bicycle component hierarchy

The notion of composite entities made up of components proves to be useful when applied to objects. Kim et al. define a *composite object* as an object that is made up of a hierarchy of component objects [Kim et al. 1989]. Each composite object has a special object at the top of the hierarchy, known as the root; for example, the bicycle would be the root of a bicycle composite. This definition restricts the hierarchy to constituting only one composite; if the composite becomes a component of another composite, the root changes, and only one composite is left. For example, when a wheel composite is added to a bicycle composite, the wheel is no longer considered a composite. This definition is too restrictive. In this thesis, any non-leaf object in the component hierarchy is the root of a composite. The composite is defined by the subtree of the hierarchy under the non-leaf object. Thus, a wheel composite will remain a composite, even when it becomes a component of a bicycle composite.

Access to part of a composite object often precedes access to other components, so retrieving the entire composite object at one time reduces disk traffic. Thus, identifying composite objects provides another means for specifying how objects should be clustered on disk, without imposing the burden of an explicit cluster command on the programmer.

Few systems currently provide explicit support for composite objects. The ThingLab constraint-oriented simulation system adds support for part-whole hierarchies to Smalltalk-76 [Borning 1981]. Blake and Low add support for part-whole hierarchies to Smalltalk-80 [Blake and Low 1987]. Part-of relationships are explicit in VBASE [Andrews and Harris 1987] and the Data Structure Manager [Rumbaugh 1987] [Rumbaugh 1988]. ORION [Kim et al. 1987b] [Kim et al. 1989] supports composite objects, but requires user intervention to exploit them for clustering on disk.

2.3.1 Part-Whole Hierarchies

The ThingLab constraint-oriented simulation laboratory [Borning 1981] provides explicit support for part-whole hierarchies in Smalltalk. Part-whole hierarchies correspond to our definition of composite objects.

In ThingLab, the class of the root of a composite object contains a part description for each component, including the component's name, the component's class, and the constraints that the component must satisfy. Internally, each component is represented by an instance variable, but this direct reference to the component is never passed to other objects. Instead, a symbolic reference to the component is passed. For example, the symbolic reference to the x coordinate of one of the endpoints of a line is `point1 x`. The symbolic reference is applied to the root of the composite to access the component. Thus,

all access to the component passes through the root. This scheme allows the root to control which messages are sent to the component.

The part-whole hierarchies added to Smalltalk-80 by Blake and Low are similar to ThingLab's [Blake and Low 1987]. Again, the root of a composite must control access to its components to preserve the integrity of the composite. A component can only be accessed by sending a compound message to the root. For example, the x coordinate of a line's endpoint is reached with the selector `point1.x`. The root can censor a compound message before delivering it to the specified component, and can censor the component's reply.

In both of these implementations, the composite hierarchy is defined by references from the root to the components.

2.3.2 Part-Of Relations

Rumbaugh argues that instance variables are used to imply several different relationships between objects, and that these relationships should be explicit [Rumbaugh 1987]. He proposed an extension to the object-oriented model, called the *object-relation* model, and implemented it in the Data Structure Manager (DSM). Relations are provided as a syntactic extension, similar to the support given classes in Smalltalk. Relations are bi-directional references between objects:

"Use of a relation can reduce the complexity of an algorithm, because a relation can inherently be traversed in either direction, while an instance variable can be followed only in the 'forward' direction. We ... consider instance variables as simply implementation optimizations for cases where traversal in the 'reverse' direction is not needed."²

DSM's composite root objects and their component objects are connected by part-of relations. VBASE also supports bi-directional relationships between objects. However, it does not take advantage of part-of relationships to determine how objects should be clustered on disk.

2.3.3 Composites in ORION

Attributes (object references) in ORION are flagged to distinguish two types of references: weak, and composite. A weak reference has no special semantic significance, whereas a composite reference is a reference from an object to one of its component objects.

ORION supports four types of composite references:

- 1) independent exclusive
- 2) independent shared
- 3) dependent exclusive
- 4) dependent shared

² J. Rumbaugh, "Relations as Semantic Constructs in an Object-Oriented Language", *Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference*, Orlando, Florida, October 1987, p. 473.

A component that exists independently of its composite is referenced using an independent reference. Conversely, a component whose existence depends upon its composite is referenced through a dependent reference. A component that is restricted to be part of only one composite is referenced through an exclusive reference. A component that may be part of more than one composite is referenced through a shared reference. The combinations of these two binary modes result in the four types of composite references listed above. The user manipulates objects with these distinct composite reference types according to a set of formal rules. For example, a component that is referenced by an exclusive reference cannot be referenced by a shared reference.

Composite references to components are matched by named, backward pointers in the component. When creating a component, the user must specify the component's parent (the object immediately above the component in the component hierarchy).

ORION does not have a secondary storage garbage collector, so dependent composite references are used to make object destruction more efficient: a dependent component is destroyed when its composite is destroyed. If a component is referenced as dependent shared, then the destruction of one of its composites will only bring about the destruction of the component if it is not currently shared by another composite.

Composite objects can be used as the unit of transfer to and from disk, but only if the user has issued an explicit *Cluster* message to specify that the classes of the objects in the composite should reside in the same section on disk. Thus, ORION does not completely remove the burden of specifying what objects should be clustered from the application programmer. If a component is shared, it is clustered with the first composite in its list of backward pointers.

2.4 Transactions and Concurrency Control

Systems that support shared, persistent objects must provide a mechanism for governing shared access to those objects. Two examples drawn from the application described in chapter one demonstrate the need for such a mechanism:

Example 1: An engineer edits a circuit board design consisting of two semiconductor devices, and two signals (Figure 2.2). A new device is added and one signal is moved (Figure 2.3). In the midst of saving the altered design to disk, the system crashes, wiping out the design in volatile memory. The addition of the new device has been saved to disk, but the change to the signal has not (Figure 2.4). The design on disk is *inconsistent*, "... the state of the system no longer reflects a real state of the world that the database is supposed to capture."³

Example 2: Two engineers edit a design concurrently (Figure 2.5). The first engineer moves a signal to connect it to a different pin, and then saves the change to disk (Figure 2.6). The second engineer does not see the disk update, and moves the same signal to another pin. This change is saved to disk, obliterating the first update (Figure 2.7). Once again, the design on disk is inconsistent, because an update has been lost.

³ H. Korth and A. Silberschatz, *Database System Concepts*, McGraw-Hill, New York, 1986, p. 331.

Figure 2.2 Design loaded from disk

Figure 2.3 Design changed

Figure 2.4 Inconsistent update

Figure 2.5 Design loaded into two workstations

Figure 2.6 Design saved from workstation #1

Figure 2.7 Design saved from workstation #2

To prevent inconsistencies from occurring, most systems regulate access to persistent objects through *transactions*. Transactions are designed to take objects on disk from one consistent state to another. The user manipulates transactions with three commands: begin, commit, and abort. The user must begin a transaction before objects on disk can be read or updated. The user ends the transaction by either committing or aborting the transaction. Committing the transaction indicates the user's desire to save all updates made during the transaction to disk. Aborting the transaction causes all updates to be discarded; none of the updates are reflected on disk.

Transactions have two important properties. First, transactions are *atomic*: either all updated objects are saved to disk, or none at all. This property prevents objects on disk from entering an inconsistent state due to a failure such as a system crash. Crash recovery schemes are used to ensure this property. Second, the concurrent execution of transactions is *serial equivalent*: the execution has the same effect on the output and on the objects on disk as some serial execution of the same transactions (see [Korth and Silberschatz 1986] for a detailed explanation of serial equivalent schedules). This property prevents the concurrent execution of transactions from causing the objects on disk to enter an inconsistent state. Concurrency control methods are used ensure this property.

Bernstein and Goodman argue that concurrency control methods that enforce serial equivalent transaction execution can be divided into two broad categories: timestamp ordering and locking [Bernstein and Goodman 1981]. Amongst the concurrency control methods that are not guaranteed to be serial equivalent, there are several that support cooperative sharing.

2.4.1 Timestamp Ordering

Timestamp ordering methods use timestamps to impose a serial equivalent order on transaction execution. Each transaction is assigned a unique timestamp; each persistent object has one read and one write timestamp. The read timestamp corresponds to the timestamp of the last transaction that successfully loaded the persistent object from disk. The write timestamp corresponds to the timestamp of the last transaction that successfully updated and unloaded the persistent object to disk. Timestamp ordering methods can be divided into two categories: pessimistic, and optimistic.

Under pessimistic timestamp ordering schemes, as each persistent object is loaded or unloaded, the appropriate timestamp is compared to the timestamp of the transaction. If the transaction's timestamp is not as current as the object's, it is not serial equivalent. The invalid transaction is rolled back and restarted to preserve the serial equivalent execution.

Under optimistic timestamp ordering schemes, the comparison of timestamps is deferred until the transaction is committed. It is assumed that contention for shared objects is rare, so transactions will not have to be rolled back frequently.

GemStone, and a Shared, Persistent Object Store use optimistic timestamp ordering methods. Avance uses time stamped versioning, allowing the user to retrieve older versions of objects if desired. Instead of timestamps, Trellis/Owl compares snapshots of objects to detect changes to them over time; snapshots serve the same purpose as timestamps.

2.4.2 Locking

Locking methods impose a serial equivalent order on transaction execution by enforcing mutually exclusive access to objects. A transaction must acquire the appropriate read or write lock on an object before it can proceed. If another transaction has a conflicting lock on the object, the former transaction must wait until the lock is relinquished. Locking methods are regarded as pessimistic: it is assumed that contention for shared objects occurs too frequently to permit the overhead of aborts and restarts found in timestamp ordering. Instead, locking imposes the overhead of locking each object that is accessed as the transaction proceeds. Contention for objects can lead to deadlock; most systems provide either a deadlock detection and recovery scheme, or a deadlock prevention scheme.

POSTGRES, based on the relational data model, supports two-phase locking on tuples. To lock an object in POSTGRES, the transaction locks the procedure which accesses an object's component tuples. This lock does not prevent a lock from being taken out against a component tuple by another transaction. The consistency of the database can only be maintained by ensuring that component tuples are never accessed through any path other than the object's representative procedure.

EXODUS supports two-phase locking on storage objects and byte ranges. ORION provides locking at several levels: a lock on a class object locks all its instances; a lock on a composite object locks all its component objects. In Arjuna, instances of the LockManager class govern two-phase locking on persistent objects. In Coral3, persistent object holders respond to the whileLocked message by performing a block while the object is locked for exclusive use. Coral3 does not, however, have deadlock detection.

2.4.3 Co-operative Sharing

Skarra and Zdonik argue that the design process requires new transaction models for co-operative work; these transaction models do not necessarily preserve the serial equivalence of transaction execution [Skarra and Zdonik 1989]. Co-operative sharing modes are used when several users wish to examine and alter an object at the same time. For example, displaying a CAD object on several workstations at one time using a co-operative sharing mode would allow an engineer at any station to change the object. The change would then be broadcast immediately to the other workstations.

POSTGRES uses alerters to support co-operative object sharing. In addition to the read and write locks used by POSTGRES to implement two-phase locking, a process can obtain a T lock on a tuple. When another process updates the T locked tuple, the former process is alerted by the database system. The former process can then poll the database for the updated tuple.

ObServer provides primitives for two concurrency models: serial equivalent and optional read. The serial equivalent model is implemented using two-phase locking. The optional read model does not require read locks, and consequently does not guarantee that transaction execution is serial equivalent. ObServer also provides triggers; triggers perform the same function as alerters in POSTGRES.

Mneme allows the application programmer to choose a concurrency control method that is appropriate to a given object. Two-phase locking, timestamp ordering, and non-serial equivalent update methods are implemented in Mneme policy modules. Objects are grouped into pools on disk to designate which policy should control them.

2.5 Comparison of Systems

The table in Figure 2.8 compares the systems examined in this chapter. The first column classifies the systems on a scale. The ends of the scale are defined by the two different approaches to supporting shared, persistent objects: object-oriented database systems, and persistent object systems. Although some of the systems are straightforward examples of one or the other approach, most of the systems are a combination of the two; this is reflected by their positions on the scale. For each of the remaining columns, each entry indicates whether the system in the corresponding row supports the given feature, indicated by a Y for yes, or does not support it, indicated by an N for no. If it was not possible to determine whether the feature was supported, this is indicated by a dash.

y = yes n = no - = unknown	Persistent Object Service	O-O DataBase	Time-Stamp	Locking	Co-operative	Clustering	Composites
Postgres	—+—	—+—	n	y	y	-	-
ObServer	—+—	—+—	n	y	y	n	-
EXODUS	—+—	—+—	n	y	-	y	-
Mneme	—+—	—+—	y	y	y	y	-
Orion	—+—	—+—	n	y	-	y	y
Gemstone	—+—	—+—	y	n	-	y	-
Avance	—+—	—+—	y	n	-	-	-
VBASE	—+—	—+—	n	y	-	y	-
Coral3	—+—	—+—	n	y	-	-	-
Trellis/Owl	—+—	—+—	y	n	-	y	n
Arjuna	+—	—+—	n	y	-	n	-
Shared, Persistent Ob. Store	+—	—+—	y	n	-	-	-

Figure 2.8 Comparison of systems

3 Persistent Objects

Persistent object systems use two approaches to implement shared, persistent objects: one approach is to modify the underlying mechanisms that support objects, and the other is to add persistence operations to volatile objects and to use transactions to control shared access to persistent objects on disk. Adding persistence operations to the programming environment allows the application programmer to select, modify or augment the implementation of persistence to suit the application. For this reason, we use this approach to make objects persistent in our prototype. This chapter details the load and unload persistence operations, and how these operations are added to volatile objects.

3.1 Adding Persistence Operations

Persistence operations can be added to volatile objects in several ways. One approach is to add the operations to the class at the top of the class hierarchy, so that all classes inherit them. Low used this approach, adding persistence protocols to the class `Object` in the Shared, Persistent Object Store [Low 1988]. A disadvantage of this approach is that all instances must carry instance variables that pertain to persistence, whether these variables are used or not. For example, in Low's system, all instances reserve space for a timestamp and a unique hash value.

Another approach is to implement the operations in an abstract persistence class. Subclasses of the persistence class inherit these operations, whereas other classes do not. This approach is suitable for applications in which persistence can be added on a class basis. However, for applications in which some instances of a class are persistent and

others are volatile, this approach encourages the development of parallel class hierarchies - one for volatile objects, one for persistent objects - that describe objects with identical operations.

A more flexible approach is to add persistence operations on an instance basis. Pascoe describes an approach for surrounding an object with side-effect or access-control operations [Pascoe 1986]. The operations are implemented in a class of *encapsulators*. An encapsulator encloses another object by adding a layer to the object's message boundary. The encapsulator intercepts messages sent to the enclosed object, forwards the messages to the enclosed object after performing a pre-action, and returns the result after performing a post-action.

We use this approach to add persistence operations to volatile objects in our prototype, enclosing volatile objects in persistence encapsulators. A persistence encapsulator intercepts all messages sent to the enclosed object. Messages that are particular to persistence are accepted and performed by the encapsulator. All other messages are forwarded directly to the enclosed object. We refer to the enclosed object as a persistent object, because the persistence operations are applied to it, and not to the encapsulator.

To enclose a volatile object in a persistence encapsulator, the volatile object is sent the `becomePersistent` message:

becomePersistent

```
(PersistentObject illegalClasses includes: self class)
ifTrue: [^self error: self class printString, ' cannot become persistent'].

^self become: (PersistentObject new: self).
```

The `become:` method redirects all references from the volatile object to the persistence encapsulator, a new instance of the class `PersistentObject`. In Figure 3.1, two objects hold

references to a volatile point object. Sending `becomePersistent` to the point object redirects the two references to the new encapsulator (Figure 3.2).

Adding persistence operations on an instance basis is more flexible than the class based approach, because the operations can be added to an instance of any class at any time without imposing the operations on all the instances of the same class. Nevertheless, it is straightforward to add the operations to all new instances of a class, by putting `becomePersistent` in an initialize instance method.

Figure 3.1 Volatile point

Figure 3.2 Point becomePersistent

3.2 Object Unloading and Loading

Smalltalk objects contain references to other objects; an instance variable is a pointer to the address of an object in memory. As each address is unique, the address also serves as a unique identifier for the object. This unique identifier gives the object the property of *object identity*. In our prototype, we extend this property to objects on disk by using an object's disk address as its unique identifier on disk.

References from objects on disk to objects in memory pose two problems. First, an object in memory is volatile. If the system crashes, an object on disk that held a reference to an object in memory is left with a dangling reference that cannot be resolved. Second, objects in memory are moved by the garbage collector to compact fragmented memory. The garbage collector corrects object references in memory to reflect these changes. However, the frequency of garbage collection in memory coupled with the cost of loading and unloading objects on disk makes it impractical to correct the memory references held by objects on disk.

Therefore, we do not allow objects on disk to hold references to objects in memory. Before a persistent object is unloaded to disk, each reference to an object in memory must be replaced by a reference to an object on disk, or replaced by nil (in effect, removing the reference). For example, in Figure 3.3, the references to two persistent points held by a persistent line must be resolved to references for the point objects on disk.

Figure 3.3 `aLine` holds references to persistent points

A persistent object is unloaded by sending the `unload` message to the object. This message is intercepted by the persistence encapsulator, which makes a copy of the persistent object in a format appropriate for storage on disk, and then writes the copy to disk. The encapsulator then releases the enclosed instance, which is later reclaimed by the garbage collector since the encapsulator held the only reference to it.

The persistence encapsulator is left in memory to serve as a proxy for the persistent object on disk. Proxy objects have been used to stand-in for persistent objects in systems such as LOOM [Kaehler 1986], and Low's Shared, Persistent Object Store, and to stand-in for distributed objects in Distributed Smalltalk [Bennett 1987].

Thus, each `PersistentObject` instance has two roles; its role depends upon the location of its persistent object. When the persistent object is resident in memory, the `PersistentObject` instance is its encapsulator. The encapsulator intercepts messages sent to the persistent object. A message that pertains to a persistence operation, such as `unload`, invokes the appropriate method in the encapsulator. Messages that are not understood by

the encapsulator are caught by **doesNotUnderstand:** and forwarded to the persistent object.

When the persistent object is resident on disk, the **PersistentObject** instance is its proxy. The proxy holds the only reference from memory to the object on disk. Upon receipt of a message that it does not understand, the proxy causes the object on disk to be loaded into memory, and then forwards the message to the loaded object. When an object is loaded into memory from disk, it may contain references to instances on disk. These are resolved to the proxies representing the referenced objects. If a proxy for a referenced object on disk does not exist, a new one is created.

In Figure 3.4, the persistent line object has been unloaded to disk; its proxy holds the only reference from memory to the line on disk. The volatile line instance is no longer referenced, so it is reclaimed by the garbage collector. The point instances and their encapsulators are also reclaimed because they are only referenced by the line instance. In Figure 3.5, the persistent line is loaded into memory to respond to a message received by its proxy. The line contains references to two point objects on disk. The proxies for these points on disk were previously reclaimed by the memory garbage collector, so new proxies must be created.

Figure 3.4 Line unloaded

Figure 3.5 aLine loaded back into main memory

3.3 Object Faults

In our prototype, objects in memory reference disk objects indirectly, through a proxy. There is one proxy in memory for every object on disk that is still referenced by other objects in memory. Since the number of objects on disk is much greater than the number of objects in memory, it is conceivable that over time, memory will become cluttered with proxies.

Thomas et al. describe an efficient implementation of transparent forwarding that could be used to solve this problem [Thomas et al. 1987]. The memory garbage collector is modified to short circuit indirect references, replacing references to a proxy with references to the object on disk. For example, an object holds a reference to the proxy of a line object on disk in Figure 3.4. The garbage collector replaces the reference to the proxy with a reference to the line object on disk. The proxy is left with no references to it in memory, so it is reclaimed by the garbage collector (Figure 3.6).

Figure 3.6 Proxy shortcircuited

This allows objects in memory to hold direct references to objects on disk. An appropriate choice of addresses prevents the range of disk addresses from intersecting the range of memory addresses. Thus, sending a message through a disk reference causes a memory address fault. A memory address fault caused by the use of an object's disk address is called an *object fault*. The fault service routine creates a proxy for the object on disk, and forwards the message to the proxy. The proxy then causes the object on disk to be loaded to respond to the message.

This scheme has an additional advantage: when an object is loaded from disk, its disk references are only resolved to proxies if the proxies already exist in memory. Thus, there is no need to create proxies for objects until they are requested through a disk reference. For example, when the line object is loaded from disk, the disk references for the two point objects are left intact (Figure 3.7).

Figure 3.7 aLine keeps disk references

Resolution of a disk reference can be delayed until the reference is used. However, if a request is made through a disk reference for an object that has already been loaded, then the reference must be resolved to the proxy of the object in memory. To solve this problem, a table is kept to record the objects which have been loaded. The fault service routine uses this table to resolve subsequent requests for the object.

4 Composite Objects

A composite object is made up of a hierarchy of component objects. Objects in the hierarchy are related to each other according to their position. Specifically, an object is *made-up-of* objects directly below it in the hierarchy and *part-of* the object directly above it in the hierarchy.

For the purpose of supporting composite objects in our prototype, we distinguish between two types of object references. Using the terms proposed by Kim et al., we call these two types of references *weak* and *composite* [Kim et al. 1989]. A weak reference has no special semantic significance, whereas a composite reference is a reference from an object in a composite to an object directly below it in the hierarchy. Consider a bicycle composite based on the component hierarchy in Figure 2.1. The reference from the root of the composite, bicycle, to its frame is a composite reference, because the frame is directly below the bicycle in the hierarchy. A reference from the bicycle object to an object outside the hierarchy, such as a reference to the bicycle's owner, is a weak reference.

There are several composite objects in the CAD schematic example described in Chapter One. A printed circuit board design is a composite made up of semiconductor devices and the signals that connect these devices. In turn, the components are also composites: a semiconductor device is represented as a composite made up of an integrated circuit carrier body and the pins that connect the body to the circuit board.

The CAD composites are based on physical decompositions. A composite could also be based on a logical view of an entity. For example, Holtz suggests that contractors may have different views of a building [Holtz 1989]. Each view is based on a functional subsystem: plumbing, electrical, mechanical, structural or architectural. Each of these

subsystems is a composite of components. For example, the structural subsystem is represented as a composite of beams, columns and connections.

In this chapter, we argue that composite references should be used as the basis for propagating persistence from persistent to volatile instances. In Chapter Two, we also argued that composite objects should be used as the basis for clustering objects on disk. Here, we examine how the implementation of composite objects restricts the policies that can be used for clustering composites. We then describe our implementation of composite references in the prototype.

4.1 Propagation of Persistence

Encapsulators and proxies provide persistence operations for volatile objects. A volatile object is enclosed by a persistence encapsulator when it is sent the `becomePersistent` message. From the programmer's perspective however, this does not explain how persistence can be ascribed to volatile objects in an application. Participants in a workshop on object-oriented databases put the question as follows: "The fundamental question addressed here was whether persistence is defined implicitly by reachability or explicitly by user action."⁴

Persistence defined implicitly by reachability means that all objects that can be reached from a persistent object must become persistent as well. Persistence defined explicitly by user action means that the user must designate exactly which objects are to become

⁴ M. Caruso and S. Zdonik (coordinators), "Report on the Object-Oriented Database Workshop: Semantic Aspects", *Addendum to the Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference*, Orlando, Florida, October 1987, p. 48.

persistent. The workshop participants asserted that, "...reachability implies the need for garbage collection while explicit action implies the potential for dangling references."⁵

We prevent these dangling references since disk instances are not allowed to retain pointers to memory. Thus, before a persistent object is unloaded, each reference to an object in memory must be removed or replaced by a reference to a disk object. Resolving a reference to a persistent object in memory is straightforward: the memory reference is replaced with the persistent object's disk reference. Resolving a reference to a volatile object is more difficult; we considered four approaches:

- 1) Make the referenced object persistent, assigning it a disk reference.
- 2) Replace the reference with nil or some other disk reference.
- 3) Ask the user to decide.
- 4) Make the referenced object persistent only if it is referenced by a composite reference. Otherwise, replace the reference with nil or some other disk reference.

The first approach passes persistence on to all objects that can be reached from a persistent object. This ensures that no references are lost, but causes objects to be persistent that do not need to be. For example, some volatile instances may be useful only in one user's environment. These objects are not valid in another user's environment, and should not be preserved on disk. As a result, this approach may degrade performance of an application because unnecessary objects become persistent.

The second approach replaces the reference to an object that the user has not explicitly designated as persistent. This places an onerous burden on the user who must ensure that

⁵ Ibid.

the necessary objects are persistent before unloading objects; otherwise, the references will be lost.

The third approach requires the user to explicitly designate which objects are persistent when the object is unloaded. It also places a burden on the user; deferring the task until the objects are unloaded makes it no less onerous.

The fourth approach passes persistence on to all objects that can be reached from a persistent object through composite references; in other words, persistence is passed on to the components of a persistent composite. This approach makes intuitive sense: if a bicycle composite is unloaded to disk, it should not lose its wheels in the process just because the wheels are represented by volatile objects. However, it is unnecessary to propagate persistence to objects referenced by weak references, since these may not need to be persistent. For example, if the bicycle composite is used in a simulation, it may contain references to its state in the simulation. It is up to the user to decide whether this state should be retained on disk.

The fourth approach has the advantage of reducing the number of objects that must be designated by the user. The user designates the root of the composite, and persistence is automatically propagated to the components. Another benefit is that persistence does not propagate to unnecessary objects. We use this approach in our prototype because it gives the user control over persistence without imposing a heavy burden, and it minimizes the number of superfluous instances that become persistent.

4.2 Composite Clustering Policies

Loading persistent objects from disk on demand imposes a performance cost for each object loaded. Identifying objects that are used together and retrieving them with one disk request improves performance by reducing disk I/O.

Access to part of a persistent composite often precedes access to its other components, so loading an entire composite at the same time will reduce the number of disk requests. To load objects at the same time, they should be clustered together on disk. When a composite is unloaded, its components must be identified and clustered together. This is straightforward for physical composites, in which the components are exclusive to the composite. This is not necessarily the case for logical composites, in which some components may be shared with other composites.

For example, consider the architectural subsystem of a building. In a CAD application, the building can be represented as a composite composed of stories; each storey is composed of rooms. The rooms in turn consist of a floor, a ceiling, and walls. Neighbouring rooms share walls in this logical decomposition. Note that the physical decomposition of this same building is different: if two adjacent rooms are separated in an earthquake, one of the rooms will be missing a wall.

Shared components pose problems for clustering the components of a composite. If a persistent object is added to a persistent composite, then a policy of moving the new component to a disk location that is close to its composite should reduce disk traffic in the long-run. However, if the new component is already part of another persistent composite, then the policy may result in moving the component to another location every time it is unloaded. Instead of reducing disk traffic, this policy could increase it.

The best clustering policy for a shared component is to store the component with the composite that accesses it most frequently. This policy requires mechanisms for determining which composites share the component, and of those composites, which one uses the component most frequently.

One mechanism for determining which composites share a component is to make the composite reference bi-directional, adding a backward pointer from component to composite. The composites that share the component can be found by traversing the component's backward pointers. Backward pointers can be stored in either a fixed number of slots in the component or a collection referenced by the component. A fixed number of slots limits the number of composites that the component can be a part of (Figure 4.1). A collection of backward pointers is unwieldy (Figure 4.2).

Figure 4.1 Limited number of backward pointers

Figure 4.2 A collection of backward pointers

Backward pointers must be updated as components are added, exchanged or discarded. To ensure that the backward pointers are kept consistent, the pointers must be updated in an atomic operation. For example, replacing the composite's old component in Figure 4.3 requires the following operations:

- 1) inform the old component that it is not part of the composite (Figure 4.4)
- 2) inform the new component that it is part of the composite (Figure 4.5)
- 3) assign the component to the composite (Figure 4.6)

Backward pointers have two drawbacks: they increase the size of the object, and they must be updated in an atomic operation.

Figure 4.3 Replacing Old component

Figure 4.4 Old no longer a component

Figure 4.5 New now a component

Figure 4.6 New assigned to composite

Even with backward pointers for determining which composites share a component, the clustering policy requires knowledge of which composite is used most frequently. This could be determined by keeping an access count for each composite. Alternatively, the user could be asked to specify with which composite a component should be clustered. This approach is used in ORION: the user specifies the composite by putting it at the top of the

component's list of backward pointers [Kim et al. 1989]. Note that this approach requires only one backward pointer per component - to point to the composite designated by the user.

A simpler clustering policy is to leave a shared component where it is, and to cluster an exclusive component with its composite. To determine whether or not a component is shared, it suffices to keep a count of the number of composites that reference the component. A count of one indicates that the component is an exclusive part of one composite. A count of greater than one indicates that it is shared.

Composite reference counts require only a fixed amount of space in each component. Like backward pointers, they can be updated as components are added, exchanged or discarded. Alternatively, the composite reference count could be determined when a component is about to be unloaded to disk, by traversing all the objects in memory. This approach is attractive if the secondary storage garbage collector calculates the number of objects referencing a persistent object when it is unloaded. For example, the Creepy garbage collector for secondary storage uses this technique [Nguyen 1989]. This garbage collector could be modified to calculate a component's composite reference count as well as its total reference count.

The overhead needed to support a policy that is optimal for shared components may degrade performance more than a policy that is not optimal. In this case, even simpler clustering policies may provide the best performance. For example, if most components are not shared, then clustering a component with its composite, regardless of whether the component is shared or exclusive, may prove most efficient. Conversely, if most components are shared, then leaving a component in its first disk location may be more efficient. In both of those policies, the mechanisms required to support clustering are minimal: it is only necessary to find components starting from the root of the composite.

In the examples above we have shown that the implementation of composite objects restricts which policies for clustering composites can be supported. The prototype uses a simple clustering policy: components are clustered with the composite if they become persistent at the same time as the composite. Nevertheless, our implementation of composites will support any of the policies described above that do not require backward pointers.

4.3 Part Variables

We have two purposes for composite references: to propagate persistence to volatile components, and to cluster composites on disk. In our prototype, composite references are uni-directional, from composite to component. This implementation serves our purposes because we can find all the components of a composite by starting at the root of the composite and following the composite references.

Composite references are distinct from weak references because they are held in *part variables*, whereas weak references are held in ordinary instance variables. For example, consider the class `Signal`; an instance of `Signal` represents the signal between two pins on a circuit board. A `Signal` instance has two instance variables, `pin1` and `pin2`, and one part variable, `path`, containing a collection of points that represent the path of the signal.

```
Object subclass: #Signal
  instanceVariableNames: 'pin1 pin2'
  partVariableNames: 'path'
  classVariableNames: ""
  poolDictionaries: ""
```

The subclassing protocols for `Class` are modified to support part variables. Part variables are processed in much the same way as ordinary instance variables: the list of part variable names is appended to the list of ordinary instance variable names for compiling. From the perspective of the compiler, there is no difference between an instance variable and a part variable; each variable accounts for one slot in a new instance. However, the position of each part variable slot is recorded in a `partPool` for the class. The position is used to distinguish the part variable from an ordinary instance variable.

The `instVarAt:` and `instVarAt:put:` protocols in `Object` access a variable according to the variable's position within the instance. We added the `isPartAt:` protocol to `Object` to test whether a variable at a given position is a part variable or an ordinary instance variable; `isPartAt:` refers to the receiver's class to determine whether the given position was recorded in the class' `partPool`. Using `isPartAt:` in conjunction with `instVarAt:` and `instVarAt: put:`, we can get and set all the part variables of an instance.

```
isPartAt: index  
    "Return true if the given index is the index  
    of a parts instance variable."  
  
    ^((self class) isPartVarAt: index)
```

In our `Signal` example above, the part variable holds a reference to a collection. This is a problem because propagating persistence through part variables will make the collection persistent, but not the objects held by the collection. We considered three approaches for supporting component-collections:

- 1) Add a "parts" bit to all collections. Setting the bit indicates that the collection is a collection of components.
- 2) Define a parallel class hierarchy of component-collections.
- 3) Propagate the `becomePersistent` operation through collections to their contents.

The first approach is inconsistent with good programming style. The second approach puts the onus on the user to ensure that a collection that is used to hold components must be instantiated from the correct class. The third makes an implicit assumption that all references from the collection should be composite references, and not weak references. It is unlikely that some objects in the collection will be components and others not, so we use the third approach in the prototype. In our example, the points defining the path of a signal composite are considered as components, even though the points are held in an instance of `OrderedCollection` (Figure 4.7).

Figure 4.7 Component collection

The unload operation described in Chapter Three has to be modified to support composites. The operation prepares a persistent object for unloading by replacing memory references to persistent objects with disk references, and by replacing references to volatile

objects with nil. This does not take into account the need to propagate persistence to volatile components. To solve this problem, all persistent composites that contain volatile components are sent the **partsBecomePersistent** message before an object is unloaded. This makes the components persistent, and ensures that a weak reference from a persistent object to a component will not be broken during unloading.

partsBecomePersistent

"Make the receiver's parts become persistent, and in turn propagate the operation to the parts of each part. This will continue until all the parts of the composite have become persistent."

```
self
  partsDo: [:part|
    (illegalClasses includes: part basicClass)
    ifFalse: [part becomePersistent]].
self
  partsDo: [:part|
    (illegalClasses includes: part basicClass)
    ifFalse: [part partsBecomePersistent]].
```

partsDo: aBlock

"Apply aBlock to each part of the enclosed object. Indexed instance variables are parts by default."

```
object class isBytes ifFalse: [
  1 to: object class instSize do: [:i|
    (object isPartAt: i) ifTrue: [
      aBlock value: (object instVarAt: i)]]].
(object class instSize) + 1 to: object class instSize + object basicSize do: [:i|
  aBlock value: (object instVarAt: i)]]
```

When all components have become persistent, a persistent object is prepared for unloading as before: memory references to persistent objects are replaced with disk references, and references to volatile objects are replaced with nil. Note that this approach does not rule out retaining weak references on disk. Weak references that are retained are

references to objects that the user has explicitly made persistent, or references to components of persistent composites.

Figure 4.8 Signal connecting microprocessor to nand gate

Consider the persistent circuit board design in Figure 4.8. The signal connecting the a15 pin of the microprocessor has just been connected to the o4 pin of a nand gate, and the engineer is about to unload the design to disk. The signal in memory is volatile, whereas the rest of the design, including the two semiconductors, is already persistent (Figure 4.9). Note the weak reference from selected in the signal object to a volatile object. The volatile object indicates whether or not the signal has been selected on the workstation's display. This object is not part of the signal composite; it is used only as long as the signal is being displayed on the screen. Before the circuit board is unloaded, the composite references are traversed to ensure that all components have become persistent. Thus, the signal and its component points become persistent, but the volatile selected object does not (Figure 4.10). When the circuit board is unloaded, the weak reference to this volatile object is deleted; only the components of the circuit board composite are unloaded to disk.

Figure 4.9 New signal is not persistent

Figure 4.10 Persistence propagated to signal and components, but not to selected

5 Transactions

Database systems use transactions to control access to shared, persistent data. In a traditional, commercial application such as payroll processing, database transactions access large amounts of uniformly structured data. Typically, each record is used for a very short period of time before it is released or updated in the database.

CAD, CASE, and OIS applications put different demands on transactions. Consider the CAD application described in Chapter One. In a typical session, an engineer opens a circuit board design, changes it, and then saves it to the object repository. A session may last for several hours. During this time, other engineers may work separately or together on the same design.

Thus, CAD transactions access large, complex objects which may be used for a prolonged period of time before being released or updated in the database. To make the best use of an engineer's time, CAD transactions must minimize the delay in loading objects from disk, and allow engineers to work concurrently without loss of effort due to conflicts. CAD applications, like CASE and OIS applications, require specialized transaction mechanisms to meet these demands [Skarra and Zdonik 1989].

Many new transaction models have been proposed for these applications. Chrysanthis and Ramamritham argue that these new models are merely a subset within the spectrum of possible models [Chrysanthis and Ramamritham 1990]. Rather than attempting to anticipate which model in this spectrum a programmer may require, we advocate that the programmer should have the flexibility to adapt a set of basic transaction mechanisms. Using this approach, the programmer is not constrained by a fixed set of mechanisms supplied by the system; the mechanisms can be changed or augmented to meet the specific demands of the application.

Transactions have two important properties: they are atomic, and their concurrent execution is serial equivalent. Crash recovery schemes are used to ensure that transactions are atomic; concurrency control methods are used to ensure that the execution of transactions is serial equivalent. In this chapter we briefly review crash recovery schemes, and then describe two optimistic concurrency control methods.

We incorporate one of these optimistic methods in our prototype. Instances of the class Transaction control shared access to persistent objects. Our transaction implementation has a significant advantage: the application programmer can select, change or augment the transaction mechanisms to suit the application. We demonstrate this advantage by subclassing Transaction to provide new transaction mechanisms.

5.1 Crash Recovery Schemes

Crash recovery schemes are used to ensure that transactions are atomic. Korth and Silberschatz describe three schemes for crash recovery: incremental logging with deferred updates, incremental logging with immediate updates, and shadow paging [Korth and Silberschatz 1986].

In incremental logging with deferred updates, the updates to persistent objects in memory are recorded in a log in stable store [Lampson et al. 1981]. The updates to the same persistent objects on disk are deferred until the transaction is committed. If the system crashes before the transaction commits, all updates in memory are lost and none are propagated to disk. If the system crashes as the transaction commits, the updates are propagated to disk from the log in stable store.

In incremental logging with immediate updates, the old and new values of updated persistent objects in memory are recorded in a log in stable store. The updates are written to disk immediately. If the system crashes before the transaction commits, all updates in memory are lost and the log is used to restore the objects on disk to their old values. If the system crashes as the transaction commits, the log is used to update the objects on disk to their new values.

In shadow paging, a shadow copy of each persistent object that is updated in memory is maintained on disk. When the transaction commits, the shadow copy becomes the current version of the object on disk, and the original is discarded. If the system crashes before the transaction commits, all shadow copies are discarded.

A crash recovery scheme has not been implemented in the prototype, but any of these three schemes could be used to support atomicity in transactions of persistent objects.

5.2 Optimistic Concurrency Control

Concurrency control methods that enforce serial equivalent transaction execution may be divided into two broad categories: timestamp ordering and locking. The timestamp ordering methods may be classified further according to whether they are pessimistic or optimistic. Optimistic methods are based on the assumption that conflicts between transactions are rare. Kung and Robinson argue that this assumption is reasonable if the number of shared objects in secondary storage is very large compared to the total number of objects involved in all concurrent transactions [Kung and Robinson 1981].

Using these optimistic methods, persistent objects are read from disk and updated in a local workspace. When the user decides to commit the updated objects to disk, the

transaction must be validated against previously committed transactions. If the transaction is valid, the updates are written to disk, and the transaction ends. If the transaction is invalid, it is rolled back; the updates in memory are discarded and the transaction is aborted.

Optimistic concurrency control methods have three advantages over locking and pessimistic timestamp ordering methods:

- 1) They are deadlock free.
- 2) They have no overhead during the read phase.
- 3) They do not force the user to wait for another user to relinquish control of an object.

These advantages are particularly attractive in interactive applications such as CAD, where a delay when committing a design to disk is preferable to a slow response when editing a design. In contrast, locking and pessimistic timestamp ordering methods impose delays throughout a transaction:

- 1) Locking methods can suffer deadlock. They must incur the overhead of preventing deadlocks, or detecting and removing deadlocks.
- 2) Locking and timestamp methods incur overhead when reading objects. In locking methods, a lock must be obtained as each object on disk is accessed; in pessimistic timestamp ordering methods, the timestamps are compared as each object on disk is accessed.

3) Locking methods force the user to wait until conflicting locks are relinquished.

Pessimistic timestamp ordering methods force the user to wait to prevent cascading aborts in dependent transactions.

Optimistic methods perform well as long as the optimistic assumption holds and transactions do not have to be rolled back. Bhargava has shown that optimistic methods perform better than locking methods in many situations [Bhargava 1982b]. However, Rahm notes that the performance degrades significantly if contention occurs frequently for "hot-spot" objects [Rahm 1987].

Kung and Robinson originally proposed optimistic concurrency control methods for centralized databases [Kung and Robinson 1981]; these methods have since been adapted for shared databases [Rahm 1987], and distributed databases [Ceri and Pelagetti 1984] [Bhargava 1982a]. Sheth and Liu have shown that optimistic timestamp ordering methods can be integrated with locking methods [Sheth and Liu 1986]. Rahm's optimistic method is well suited to our example CAD environment, and is used as the basis for our implementation.

5.2.1 Kung and Robinson's Method

In this method, each transaction goes through at least two, and possibly three, phases: read, validation, and write (Figure 5.1)⁶. Each transaction $T(i)$ is assigned a unique transaction number $t(i)$, from a global transaction counter. Transaction numbers act as timestamps; they are used to determine whether a serial equivalent schedule exists between a transaction and all previously committed transactions.

At the beginning of the read phase, the transaction number of the most recently committed transaction is recorded for future reference. During this phase, disk-based objects may be loaded into a local workspace. As the objects are read in, they are added to the transaction read set. Objects may be updated in the local workspace, but these updates are not reflected on disk. Updated objects are added to the transaction write set.

The user signals the beginning of the validation phase by committing the updates to disk. The transaction is examined to ensure that it is based on the updates of all previously committed transactions. Given a previously committed transaction $T(i)$, and validating transaction $T(j)$, where $t(i) < t(j)$, $T(j)$ is valid if there exists a serial equivalent schedule such that $T(i)$ precedes $T(j)$. A schedule exists if one of the three following conditions holds (Figure 5.2)⁷:

- 1) $T(i)$ finishes its write phase before $T(j)$ starts its read phase.
- 2) $T(i)$ finishes its write phase before $T(j)$ starts its write phase, and the write set of $T(i)$ does not intersect with the read set of $T(j)$.
- 3) $T(i)$ finishes read phase before $T(j)$ finishes read phase, and the write set of $T(i)$ does not intersect with the read set or the write set of $T(j)$.

⁶ Figure adapted from: H. Kung and J. Robinson, "On Optimistic Methods for Concurrency Control", *ACM Transactions on Database Systems*, Vol. 1, No. 2, 1981, p. 215.

⁷ Ibid, p. 219.

Figure 5.1 Optimistic transaction phases

Figure 5.2 Scheduling conditions

Figure 5.3 Unnecessary rollback

Under the first condition, $T(i)$ finishes before $T(j)$ starts, thus the transactions are serial. The second condition specifies that $T(i)$ does not update any object that $T(j)$ has read and will not overwrite any object that $T(j)$ updates; because $T(i)$ finishes before $T(j)$ writes, $T(j)$ cannot affect $T(i)$. The third condition specifies that $T(i)$ does not update any object involved in $T(j)$; $T(j)$ is too late to update any objects in the read set of $T(i)$ and $T(j)$ does not update any object that $T(i)$ updates.

During validation phase, the validating transaction is compared with all previously committed transactions to ensure that one of the three conditions holds. The comparisons are straightforward. Transactions that finished before the validating transaction started have a transaction number less than or equal to the number recorded at the beginning of the read phase. The rest of the previously committed transactions must be examined to ensure that their write sets do not intersect with the read set of the validating transaction.

If the validating transaction does not have a serial equivalent schedule with all previously committed transactions, it is invalid, and must be rolled back. If the transaction is valid, it proceeds to the write phase. In the write phase, the global transaction counter is incremented, and the transaction is given this new transaction number. The updated objects are then propagated from the local workspace to disk.

5.2.2 Rahm's Method

Kung and Robinson's method rolls back some transactions unnecessarily. Consider object *X* in Figure 5.3⁸. It is written to disk by transaction *T*(1), and subsequently read by transaction *T*(2). According to the previous method, *T*(2) should be rolled back, as *X* is in the read set of *T*(2) and the write set of *T*(1). Yet *X* was read after it was last written, so the transaction could be allowed to proceed.

Rahm's optimistic method appends to each object the transaction number of the last transaction that updated the object [Rahm 1987]. The read phase proceeds identically to Kung and Robinson's method. In the validation phase, the transaction number of each object in the read and write sets is compared with the transaction number of the same object on disk. If they are the same, the transaction has a current copy of the object. If all objects in the read and write sets are current, the transaction is valid. In the write phase, the transaction number is appended to all objects in the write set and these objects are written to disk.

This method has two advantages over Kung and Robinson's method. First, fewer transactions are rolled back, so concurrent transaction performance is improved. Second, there is no need to store the write sets of all committed transactions, and only one comparison per object in the read set and write set is required.

This method does impose one significant restriction: the most current transaction number for a persistent object must be available for the validation phase. This information could be retrieved from disk, but it would double the number of disk accesses required to read a persistent object. Alternatively, this information could be kept in a table for loaded

⁸ Figure adapted from: E. Rahm, "Design of Optimistic Methods for Concurrency Control in Database Sharing Systems", *Proceedings of the IEEE Conference on Distributed Systems*, 1987, p. 155.

objects. We use the latter approach in our implementation; the object controller records which objects have been loaded from the object repository, as well as their current transaction numbers.

5.3 Optimistic Transactions

In our prototype, transaction mechanisms are implemented in the class `Transaction`. `Transaction` instances manage access to shared, persistent objects. We use Rahm's optimistic concurrency control method to ensure serial equivalent execution of these transactions.

In Rahm's method, transactions are validated at a single database server. In our implementation, this function is performed by an instance of `ObjectController`. The object controller is shown as part of the object repository in the example CAD environment (Figure 1.1).

5.3.1 Transaction

The user must begin a transaction before loading persistent objects from disk. When the user commits the transaction, the transaction is validated by the object controller. If the transaction is valid, the object controller writes all the updated persistent objects to disk. The persistent objects are then flushed from the local workspace and the transaction ends. The user also has the option of aborting the transaction, thereby discarding all updates and flushing the persistent objects from the local workspace.

```
Object subclass: #Transaction
instanceVariableNames:
  'readSet writeSet timestamp '
partVariableNames: "
classVariableNames:
  'CurrentTransaction '
poolDictionaries: "
```

begin

"Start the transaction."

```
readSet := Set new.
writeSet := Set new.
(self class) currentTransaction: self.
^self.
```

commit

"Commit the transaction to persistent store.
Determine which objects in the readSet should be in the writeSet.
Ask the ObjectController to validate, and if valid,
to write the writeSet to disk. Otherwise abort the transaction.
The transaction ends."

```
self determineWriteSet.
```

```
(ObjectControl validateAndWrite: self)
ifTrue: [
  self releaseReadSet.
  (self class) endCurrentTransaction: self.
  self cleanUp.
  Transcript cr; show: 'Transaction committed']
ifFalse: [
  self abort].
^self
```

abort

"Abort the transaction."

```
self releaseReadSet.
(self class) endCurrentTransaction: self.
self cleanUp.
Transcript cr; show: 'Transaction aborted'.
^self
```

In contrast to conventional database systems which require the user to explicitly indicate which objects should be added to the read set and write set, a Transaction instance maintains these sets automatically. All objects that are read from disk and all volatile objects that becomePersistent during the course of the transaction are added to the transaction read set. When the user signals the end of the read phase by sending the commit message to the transaction, all objects in the read set are sent the partsBecomePersistent message, in accordance with the unload operation described in Chapter Four. The objects in the read set that have been changed are then added to the write set.

In our current prototype, when a persistent object is loaded from disk, a copy of the object is made and kept for future reference. Since the copy is made when the object is loaded, it will not contain references to any volatile objects. At commit time, the volatile references in each object in the read set are set to nil. The object is then compared to its copy to determine whether any persistent references have been changed. If the object has changed, the object is added to the write set of the transaction.

Comparing a persistent object to its copy is not an efficient means for detecting changes to objects. Our prototype implementation could be improved by adding a "dirty" bit to each persistent object. The bit would be set when the persistent object is updated. This would require changes to the compiler to provide support for detecting updates to persistent objects and for setting the dirty bit.

Since the write set is determined only when the user commits the transaction, all updates to persistent objects are buffered in the local workspace until that time. Buffering the updates until the end of the transaction limits the number of persistent objects that can be accessed during one transaction. Exceeding this limit causes a resource abort.

determineWriteSet

*Private: used by commit.

Make the volatile parts of all persistent composites becomePersistent.

Add to the writeSet all objects whose persistent references have changed."

readSet

```
do: [ :aPersistentObject |
    aPersistentObject partsBecomePersistent].
```

readSet

```
do: [ :aPersistentObject |
    aPersistentObject nilVolatileReferences.
    (aPersistentObject isChanged)
    ifTrue: [
        writeSet add: aPersistentObject]]
```

5.3.2 Object Controller

The object controller performs four tasks when the user commits a transaction. First, it validates the transaction. The validation phase requires that all objects in the read set have the same transaction number as the corresponding object on disk. To eliminate unnecessary disk accesses, the object controller maintains a table of the transaction timestamps for all objects loaded from disk. If the transaction is valid, the object controller performs the remaining three tasks: it updates the transaction timestamp from the global transaction counter, unloads the objects in the write set to disk, and updates the timestamps in the table of committed objects.

```
Object subclass: #ObjectController
instanceVariableNames:
'committedObjects currentTime '
partVariableNames: "
classVariableNames: "
poolDictionaries: "
```

validate: aReadSet postAction: aBlock

"Return true if aTransaction is valid, false otherwise.

aTransaction is invalid if the timestamp of a persistent object in its readSet is not equal to the timestamp of the same persistent object when it was last committed to disk. A new persistent object has not been committed to disk yet, so its timestamp is not found in the committedObjects table.

If aTransaction is valid, perform the given postAction."

| valid entry |

valid := true.

"<critical section begins>"

readSet

do: [:aPersistentObject |

entry := committedObjects

detect: [:committed |

(aPersistentObject isEqual: committed object)]

ifNone: [nil].

(entry notNil)

ifTrue: [

(aPersistentObject timestamp ~= entry timestamp)

ifTrue: [valid := false]]].

valid ifTrue: [aBlock value].

"<critical section ends>"

^valid

validateAndWrite: aTransaction

"Return true if aTransaction is valid, false otherwise.

If aTransaction is valid, write it to disk and broadcast it."

| valid |

valid := self validate: aTransaction readSet

postAction: [

self timestamp: aTransaction;

write: aTransaction writeSet;

updateCommitted: aTransaction writeSet].

valid ifTrue: [self broadcast: aTransaction writeSet].

^valid

timestamp: aTransaction

 "Obtain a timestamp for aTransaction.

 Update the timestamps of aTransaction's writeSet."

 [writeSet timestamp]

 writeSet := aTransaction writeSet.

 timestamp := (self time).

 aTransaction timestamp: timestamp.

writeSet

 do: [:aPersistentObject |

 aPersistentObject timestamp: timestamp].

write: writeSet

 "Unload all objects in the writeSet."

writeSet

 do: [:aPersistentObject |

 aPersistentObject determineDiskLocation];

 do: [:aPersistentObject |

 aPersistentObject unload].

updateCommitted: writeSet

 "Update committedObjects with writeSet."

 | entry |

writeSet

 do: [:aPersistentObject |

 entry := committedObjects

 detect: [:committed |

 (aPersistentObject isEqual: committed object)]

 ifNone: [nil].

 (entry notNil)

 ifTrue: [

 entry timestamp: aPersistentObject timestamp]

 ifFalse: [

 committedObjects add: ((CommittedObject new)

 initialize;

 object: aPersistentObject;

 timestamp: aPersistentObject timestamp;

 incrementTransactions;

 yourself)])

5.4 Transaction Subclasses

In our prototype, the programmer adapts basic transaction mechanisms to meet the specific demands of an application. Using this approach, the programmer is not limited to a fixed set of transaction mechanisms supplied by the system.

For example, consider the optimistic transaction mechanisms described in the previous section. These mechanisms are ideal for small transactions of a short duration, where the possibility of conflicts with other transactions is remote. However, they are inadequate in applications where transactions are long and where users in separate workspaces wish to work co-operatively. The CAD engineer in the midst of a long design session has no recourse when a shorter transaction with a conflicting write set commits, thereby rendering the engineer's transaction invalid. Two engineers working together on the same design cannot easily exchange their updates because the transaction mechanisms do not support the exchange of write sets. In our prototype, transaction mechanisms can be modified and augmented to address these inadequacies.

We present three examples of subclassing the Transaction class to create new transaction mechanisms. These mechanisms are not as complex as those required for the CAD problem above, but the principles used to create them are the same.

5.4.1 SaveTransaction

SaveTransaction is a subclass of Transaction. Instances of SaveTransaction provide the user with an additional transaction operation: **save**. The save method validates the current write set and, if valid, writes the objects to disk. Unlike commit, save does not

terminate the transaction; it leaves the persistent objects of the read set intact, and does not flush them from memory.

The save operation improves transaction performance when a user wants to write updated objects to disk, and then continue to use the objects in memory. Using the commit operation, the same user would have to start a new transaction and reload the objects after each commit.

```
Transaction subclass: #SaveTransaction
instanceVariableNames: "
partVariableNames: "
classVariableNames: "
poolDictionaries: "
```

save

```
"Save the transaction to persistent store.
Determine which objects in the readSet should be in the writeSet.
Ask the ObjectController to validate, and if valid,
to save the writeSet to disk; reset the writeSet and continue.
Otherwise abort the transaction."
```

```
self determineWriteSet.
```

```
(ObjectControl validateAndSave: self)
  ifTrue: [
    writeSet:= Set new.
    Transcript cr; show: 'Transaction saved']
  ifFalse: [
    self abort].
^self
```

5.4.2 LargeTransaction

LargeTransaction is a subclass of SaveTransaction. Instances of LargeTransaction have additional read and write sets, called readSubSet and writeSubSet. A LargeTransaction behaves like a SaveTransaction, adding loaded persistent objects to the

readSet until the user signals that loaded objects should be added to the readSubSet. When the transaction is saved successfully to disk, the transaction retains the readSet, but flushes the persistent objects in the readSubSet from memory. Persistent objects loaded after a save are added to the readSet, until the user signals again that loaded objects should be entered into the readSubSet.

LargeTransactions improve the performance of transactions that span large portions of the persistent object space. For example, consider a user updating several large, composite objects on disk. Loading all of the composites into the local workspace will cause a resource abort; each composite must be loaded, updated, written to disk and flushed from memory before the next composite is loaded.

The composites are accessed through a persistent dictionary. The user loads the dictionary into the readSet, and then loads a composite into the readSubSet, makes the necessary changes, and saves the composite to disk. After each save, the composite is purged from memory, but the dictionary is retained in memory. This transaction operation spares the user the effort of starting a new transaction from scratch for each composite that is updated.

```
SaveTransaction subclass: #LargeTransaction
instanceVariableNames:
  'readSubSet writeSubSet subSetBegin '
partVariableNames: "
classVariableNames: "
poolDictionaries: "
```

beginSubSet

```
"Add persistent objects into the readSubSet
from this point on in the transaction."
```

```
subSetBegin := true.
```

```
addRead: aPersistentObject
    "If the readSubSet has been entered,
    add aPersistentObject to the readSubSet.
    Otherwise, add it to the readSet.
    If it is already in either set, flag an error."

    (self isInReadSet: aPersistentObject)
        ifTrue: [
            self error: 'PersistentObject is already in read set']
        ifFalse: [
            (self subSetBegun)
                ifTrue: [
                    readSubSet add: aPersistentObject]
                ifFalse: [
                    readSet add: aPersistentObject]].
```

5.4.3 InformedTransaction

InformedTransaction is a subclass of SaveTransaction. Instances of InformedTransaction are informed by the object controller when other transactions successfully commit to disk. From this information, the informed transaction can perceive conflicts that will render itself invalid. When a conflict is detected, the informed transaction attempts to reconcile the conflict.

The methods used to reconcile the conflict are specific to the application, and supplied by the programmer. There are a number of ways in which a conflict can be dealt with. For example, a transaction can be rolled back and restarted, by releasing the objects in the write set and the invalid objects in the read set, and then reloading them. Another possibility is to provide the user with a means to propagate changes from objects in the local workspace to the newly committed objects. A conflict resolution browser that displays the differences between an object in the workspace and the newly committed object on disk could be provided to help the user in this task.

InformedTransactions could be used as the basis for co-operative work on a common group of objects. Consider two engineers editing the same design on separate workstations (Figure 5.4). The first engineer makes a change to the design and saves it to disk. The object controller validates the save, and broadcasts the write set to the transaction running in the second engineer's workstation (Figure 5.5). The second engineer then uses the methods provided in the CAD application to incorporate the changes into the transaction (Figure 5.6). This transaction operation for co-operative work is similar to that supported by notifiers and alerters in object-oriented database systems.

```
SaveTransaction subclass: #InformedTransaction
instanceVariableNames: "
partVariableNames: "
classVariableNames: "
poolDictionaries: "
```

```
committedObjects: aSet
    "aSet of objects have just been updated on disk. Find the
    intersection of aSet and the receiver's readSet. If there is
    an intersection, then the receiver must be reconciled with
    the intersection."
```

```
| intersection |
```

```
intersection := (self intersect: aSet).
```

```
(intersection ~= notEmpty)
    ifTrue: [
        self reconcile: intersection].
```

Figure 5.4 Design loaded into two workstations

Figure 5.5 Design saved, invalid on workstation #2

Figure 5.6 Design loaded into workstation #2

rollBackAndRestart: readSetSubset

"One way to deal with a read set intersection is to release the objects in the intersection (readSetSubset) as well as the writeSet and then reload them."

```
self release: readSetSubset
readSetSubset
do: [ :aPersistentObject |
    readSet remove: aPersistentObject].
```

```
self determineWriteSet.
self release: writeSet.
writeSet
do: [ :aPersistentObject |
    readSet remove: aPersistentObject].
```

```
readSetSubset
do: [ :aPersistentObject |
    aPersistentObject load].
writeSet
do: [ :aPersistentObject |
    aPersistentObject load].
writeSet := Set new.
```

Transcript cr; show: 'Transaction restarted'.

5.5 Integration with Secondary Storage Garbage Collection

The transactions described above assume that the objects on disk are left unchanged. However, secondary storage garbage collectors, such as Creepy, may reclaim dead objects on disk even when a transaction is in progress. Modifications are required to integrate transactions with the secondary storage garbage collector.

The garbage collector reclaims any object that cannot be reached from a root set of objects. For example, consider a persistent line and two persistent points on disk (Figure 3.6). The points are reachable from the root set only through the disk references in the line. A message is sent to the line's proxy in memory, causing it to be loaded into

memory. The line's disk reference to pt2 is replaced with a reference to a new point in memory (Figure 5.7). The disconnected point is no longer accessible from the root set, so it is reclaimed by the garbage collector (Figure 5.8).

Changes made to objects in memory should not be reflected on disk until the user commits the transaction. If the transaction is aborted or the system crashes, all changes made to persistent objects in the local workspace are discarded, and none are propagated to disk. Returning to the example, a system crash subsequent to the change to the line leaves the database in an inconsistent state: the disk-based garbage collector has removed the point, and the disk reference from the line on disk to the point is invalid (Figure 5.9).

Keeping a set of referenced objects prevents the garbage collector from reclaiming persistent objects before the transaction is committed. When a persistent object is loaded from disk, the persistent objects that it references are entered into the reference set. The reference set maintains a path from the root set to the objects in the set, so they are not reclaimed by the garbage collector (Figure 5.10). When the transaction commits, the reference set is discarded, and persistent objects that cannot be reached are left to the garbage collector.

We do not maintain an explicit set of referenced objects in our prototype because the object copy kept by the persistence encapsulator to detect changes to the enclosed object serves the same purpose.

Figure 5.7 aLine adds a new point in main memory

Figure 5.8 aPoint on disk is garbage collected

Figure 5.9 System crashes, aLine on disk is inconsistent

Figure 5.10 Reference Set prevents aPoint on disk from being garbage collected

6 Conclusion

We have described three aspects of an experimental prototype persistent object system that provides support for shared, persistent objects. Throughout this thesis, we have emphasized the importance of giving the application programmer control over how shared, persistent objects are implemented and used. Our prototype reflects this emphasis.

The prototype uses persistence encapsulators to add persistence operations to volatile objects. This approach is flexible, because it adds the operations on an instance basis. When a persistent object is resident in memory, the encapsulator intercepts all messages sent to the object, accepting messages that pertain to persistence operations, and forwarding the rest to the object. When the object is resident on disk, the encapsulator is left behind in memory to act as a proxy for the object on disk. Upon receipt of a message, the proxy causes the object on disk to be loaded into memory, and then forwards the message to the object.

We provide support for composite objects in the prototype. The subclassing protocols in `Class` are modified so that the user can define part variables. Part variables hold unidirectional composite references. Composite references are used for propagating persistence to volatile objects; this gives the user control over persistence without imposing a heavy burden, and does not propagate persistence to unnecessary objects. Composite references are also used for clustering persistent objects on disk, thereby improving system performance by reducing disk I/O.

Finally, the prototype uses transactions to control shared access to persistent objects on disk. Transaction mechanisms are implemented in the class `Transaction`. Rather than attempt to provide all the transaction mechanisms an application programmer could require for an application, we allow the programmer to adapt a set of basic mechanisms to meet the

application demands. We demonstrated this approach by subclassing the Transaction class to create new transaction mechanisms.

Our approach gives the prototype a distinct advantage over other systems. The application programmer is not constrained by a fixed set of transaction mechanisms supplied by the system; the mechanisms can be changed or augmented to meet the demands of the application.

6.1 Future Work

Our experimental prototype is intended as a testbed for further research into shared, persistent objects. There are several areas that could be explored.

In Chapter Four, we described clustering policies that can be supported by our implementation of composite objects. The performance of these policies could be compared by implementing them in the prototype and testing each one with composites of shared and exclusive components.

In Chapter Five, we noted that several new transaction models have been proposed for CAD, CASE, and OIS applications. Our prototype could be used to implement and study new transaction mechanisms.

A major extension of this thesis would be to provide support for distributed, persistent objects. Distributing persistent objects raises several problems. Persistent object identifiers must be extended to specify which repository the object is stored on. In addition, a distributed network of workstations and repositories may suffer node and link failures. Ideally, these failures should not impede processing in the rest of the network. In particular, an object stored at one node should be available even in the event of the failure

of that node, or failure of the links connecting that node to the rest of the network. Distributing replicas of an object throughout the network ensures that in the event of a node or link failure, a copy of an object can be found somewhere in the network. However, adding replicas to the network adds to the complexity of the concurrency control technique, as the replicas must be kept consistent with each other. An extension of our prototype could be used to investigate alternative approaches to these problems.

7 References

Andrews and Harris 1987

T. Andrews and C. Harris, "Combining Language and Database Advances in an Object-Oriented Development Environment", *Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference*, Orlando, Florida, October 1987, pp. 430-440. Published as ACM SIGPLAN Notices, Vol. 22, No. 12, December 1987.

Bhargava 1982a

B. Bhargava, "Resiliency Features of the Optimistic Concurrency Control Approach for Distributed Database Systems", *Proceedings of the Second IEEE Symposium on Reliability in Distributed Software and Database Systems*, Pittsburgh, Pennsylvania, July 1982, pp. 19-32.

Bhargava 1982b

B. Bhargava, "Performance Evaluation of the Optimistic Approach to Distributed Database Systems and its Comparison to Locking", *Proceedings of the Third IEEE International Conference on Distributed Computing Systems*, Miami, Florida, October 1982, pp. 508-517.

Bennett 1987

J. Bennett, "The Design and Implementation of Distributed Smalltalk", *Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference*, Orlando, Florida, October 1987, pp. 318-330. Published as ACM SIGPLAN Notices, Vol. 22, No. 12, December 1987.

Bernstein and Goodman 1981

P. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems", in *Distributed Systems, Vol II: Distributed Data Base Systems*, W. Chu (editor), Artech House, Inc., Massachusetts, 1986, pp. 121-158.

Bjornerstedt and Britts 1988

A. Bjornerstedt and S. Britts, "AVANCE: An Object Management System", *Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference*, San Diego, California, November 1988, pp. 206-221. Published as ACM SIGPLAN Notices, Vol. 23, No. 11, November 1988.

Blake and Low 1987

E. Blake and C. Low, "On Including Part Hierarchies in Object-Oriented Languages, with an Implementation in Smalltalk", *Proceedings of the European Conference on Object-Oriented Programming*, Paris, France, June 1987, pp 41-50. Published as Lecture Notes in Computer Science, Vol. 276, Springer Verlag, 1987.

Bloom and Zdonik 1987

T. Bloom and S. Zdonik, "Issues in the Design of Object-Oriented Database Programming Languages", *Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference*, Orlando, Florida, October 1987, pp. 441-451. Published as ACM SIGPLAN Notices, Vol. 22, No. 12, December 1987.

Borning 1981

A. Borning, "The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory", *ACM Transactions on Programming Languages and Systems*, Vol. 3, No. 4, October 1981, pp. 353-387.

Carey et al. 1986

M. Carey, D. DeWitt, D. Frank, G. Graefe, M. Muralikrishna, J. Richardson, and E. Shekita, "The Architecture of the EXODUS Extensible DBMS", *Proceedings of the International Workshop on Object-Oriented Database Systems*, Pacific Grove, California, September 1986, pp. 52-65. Published by IEEE Computer Society, Los Alamitos, California, 1986.

Carey et al. 1988

M. Carey, D. DeWitt, S. Vandenburg, "A Data Model and Query Language for EXODUS", *Proceedings of the International Conference on Management of Data, SIGMOD '88*, Chicago, Illinois, June 1988, pp. 413-423.

Caruso and Zdonik 1987

M. Caruso and S. Zdonik (coordinators), "Report on the Object-Oriented Database Workshop: Semantic Aspects", *Addendum to the Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference*, Orlando, Florida, October 1987, pp. 45-50. Published as ACM SIGPLAN Notices, Vol. 23, No. 5, May 1988.

Ceri and Pelagetti 1984

S. Ceri and G. Pelagetti, *Distributed Databases: Principles and Systems*, McGraw-Hill, New York, 1984.

Chrysanthis and Ramamritham 1990

P. Chrysanthis and K. Ramamritham, "ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior", *Proceedings of the 1990 ACM SIGMOD International Conference on the Management of Data*, Atlantic City, New Jersey, pp. 194-203. Published as ACM SIGMOD Record, Vol. 19, No. 2, June 1990.

Cockshot et al. 1984

W. Cockshot, M. Atkinson, and K. Chisholm, "Persistent Object Management System", *Software Practice and Experience*, Vol. 14, 1984, pp. 49-71.

Copeland and Maier 1984

G. Copeland and D. Maier, "Making Smalltalk a Database System", *Proceedings of the 1984 ACM SIGMOD International Conference on the Management of Data*, New York, New York, 1984, pp. 316-324. Published as ACM SIGMOD Record Vol. 14, No. 2, 1984.

Digitalk 1988

Digitalk Inc., *Smalltalk/V 286 Tutorial and Programming Handbook*, Los Angeles, California, 1988.

Dixon et al. 1989

G. Dixon, G. Parrington, S. Shrivastava and S. Wheeler, "The Treatment of Persistent Objects in Arjuna", *Proceedings of the European Conference on Object-Oriented Programming*, Nottingham, England, July 1989. Published as Lecture Notes in Computer Science, Springer Verlag, 1989.

Holtz 1989

N. Holtz, *Private Communication*, Ottawa, Canada, Oct. 1989.

Kaehler 1986

T. Kaehler, "Virtual Memory on a Narrow Machine for an Object-Oriented Language", *Proceedings of the International Workshop on Object-Oriented Database Systems*, Pacific Grove, California, September 1986, pp. 87-106. Published by IEEE Computer Society, Los Alamitos, California, 1986.

Kim et al. 1987a

W. Kim, N. Ballou, J. Bannerjee, H. Chou, J. Garza, and D. Woelk, "Features of the Orion Object-Oriented Database System", *Microelectronics and Computer Technology Corporation Technical Report ACA-ST-308-87*, Austin, Texas, September 1987.

Kim et al. 1987b

W. Kim, J. Bannerjee, H. Chou, J. Garza, D. Woelk, "Composite Object Support in an Object-Oriented Database System", *Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference*, Orlando, Florida, September 1987, pp. 118-125. Published as ACM SIGPLAN Notices, Vol. 22, No. 12, December 1987.

Kim et al. 1988

W. Kim, N. Ballou, J. Bannerjee, H. Chou, J. Garza, and D. Woelk, "Integrating an Object-Oriented Programming System with a Database System", *Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference*, San Diego, California, November 1988, pp. 142-152. Published as ACM SIGPLAN Notices, Vol. 23, No. 11, November 1988.

Kim et al. 1989

W. Kim, E. Bertino, and J. Garza, "Composite Objects Revisited", *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*, Portland, Oregon, pp. 337-347. Published as ACM SIGMOD Record, Vol. 18, No. 2, June 1989.

Korth and Silberschatz 1986

H. Korth and A. Silberschatz, *Database System Concepts*, McGraw-Hill, New York, 1986.

Kung and Robinson 1981

H. Kung and J. Robinson, "On Optimistic Methods for Concurrency Control", *ACM Transactions on Database Systems*, Vol. 1, No. 2, 1981, pp. 213-226.

Lampson et al. 1981

B. Lampson, "Atomic Transactions", in *Distributed Systems - Architecture and Implementation*, B. Lampson, M. Paul, and H. Siegart (editors), Springer-Verlag, New York, 1981, pp. 246-265.

Low 1988

C. Low, "A Shared, Persistent Object Store", *Proceedings of the European Conference on Object-Oriented Programming*, Oslo, Norway, 1988, pp 390-410. Published as Lecture Notes in Computer Science, Vol. 322, Springer Verlag, 1988.

Maier 1989

D. Maier, "Making Database Systems Fast Enough for CAD Applications", in *Object-Oriented Concepts, Databases, and Applications*, W. Kim, F. Lochovsky (editors), ACM Press, New York, New York, 1989, pp. 573-582.

Maier et al. 1986

D. Maier, J. Stein, A. Otis, and A. Purdy, "Development of an Object-Oriented DBMS", *Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference*, Portland, Oregon, September 1986, pp. 472-482. Published as ACM SIGPLAN Notices, Vol. 21, No. 11, November 1986.

Manola 1987

F. Manola, from discussion in, "Report on the Object-Oriented Database Workshop", S. Thate (chairman), *Addendum to the Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference*, Orlando, Florida, October 1987, pp. 73-87. Published as ACM SIGPLAN Notices, Vol. 23, No. 5, May 1988.

McCullough 1987

P. McCullough, "Transparent Forwarding: First Steps", *Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference*, Orlando, Florida, October 1987, pp. 331-341. Published as ACM SIGPLAN Notices, Vol. 22, No. 12, December 1987.

Merrow and Laursen 1987

T. Merrow and J. Laursen, "A Pragmatic System for Shared Persistent Objects", *Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference*, Orlando, Florida, October 1987, pp. 103-110. Published as ACM SIGPLAN Notices, Vol. 22, No. 12, December 1987.

Milinkovich 1988

S. Milinkovich, "An Object-Oriented Database for a Computer Aided Software Engineering Environment", *M.Sc. Thesis*, Carleton University Department of Systems and Computer Engineering, Ottawa, Canada, September 1988.

2

of/de

2

MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS
STANDARD REFERENCE MATERIAL 1010a
(ANSI and ISO TEST CHART No. 2)

Moss and Sinofsky 1988

J. Moss and S. Sinofsky, "Managing Persistent Data with Mneme: Designing a Reliable, Shared Object Interface", *Proceedings of the Second International Workshop on Object-Oriented Database Systems*, West Germany, September 1988, pp. 298-316. Published as Lecture Notes in Computer Science, Vol. 334, Springer Verlag, 1988.

Nierstrasz and Tsichritzis 1989

O. Nierstrasz and D. Tsichritzis, "Integrated Office Systems", in *Object-Oriented Concepts, Databases, and Applications*, W. Kim, F. Lochovsky (editors), ACM Press, New York, New York, 1989, pp. 199-215.

Nguyen 1989

T. Nguyen, "Creepy: An Incremental Secondary Storage Garbage Collector", M.C.S. Thesis, Carleton University School of Computer Science, Ottawa, Canada, December 1989.

O'Brien et al. 1986

P. O'Brien, B. Bullis, and C. Schaffert, "Persistent and Shared Objects in Trellis/Owl", *Proceedings of the International Workshop on Object-Oriented Database Systems*, Pacific Grove, California, September 1986, pp. 113-123. Published as ACM SIGPLAN Notices, Vol. 21, No. 11, November 1986.

Pascoe 1986

G. Pascoe, "Encapsulators: A New Software Paradigm in Smalltalk-80", *Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference*, Portland, Oregon, September 1986, pp. 341-346. Published as ACM SIGPLAN Notices, Vol. 21, No. 11, November 1986.

Rahm 1987

E. Rahm, "Design of Optimistic Methods for Concurrency Control in Database Sharing Systems", *Proceedings of the Seventh IEEE International Conference on Distributed Computing Systems*, Berlin, West Germany, September 1987, pp. 154-161.

Rowe 1986

L. Rowe, "A Shared Object Hierarchy", *Proceedings of the International Workshop on Object-Oriented Database Systems*, Pacific Grove, California, September 1986, pp. 160-170. Published by IEEE Computer Society, Los Alamitos, California, 1986.

Rumbaugh 1987

J. Rumbaugh, "Relations as Semantic Constructs in an Object-Oriented Language", *Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference*, Orlando, Florida, October 1987, pp. 466-481. Published as ACM SIGPLAN Notices, Vol. 22, No. 12, December 1987.

Rumbaugh 1988

J. Rumbaugh, "Controlling Propagation of Operations using Attributes on Relations", *Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference*, San Diego, California, November 1988, pp. 285-296. Published as ACM SIGPLAN Notices, Vol. 23, No. 11, November 1988.

Sheth and Liu 1986

A. Sheth and M. Liu, "Integrating Locking and Optimistic Concurrency Control in Distributed Database Systems", *Proceedings of the Sixth International IEEE Conference on Distributed Computing Systems*. Cambridge, Massachusetts, May 1986, pp. 89-99.

Skarra and Zdonik 1989

A. Skarra and S. Zdonik, "Concurrency Control and Object-Oriented Databases", in *Object-Oriented Concepts, Databases, and Applications*, W. Kim, F. Lochovsky (editors), ACM Press, New York, New York, 1989, pp. 395-421.

Skarra et al. 1986

A. Skarra, S. Zdonik, and S. Reiss, "An Object Server for an Object-Oriented Database System", *Proceedings of the International Workshop on Object-Oriented Database Systems*, Pacific Grove, California, September 1986, pp. 196-204. Published by IEEE Computer Society, Los Alamitos, California, 1986.

Smith and Zdonik 1987

K. Smith and S. Zdonik, "Intermedia: A Case Study of the Differences Between Relational and Object-Oriented Database Systems", *Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference*, Orlando, Florida, October 1987, pp. 452-465. Published as ACM SIGPLAN Notices, Vol. 22, No. 12, December 1987.

Stonebraker 1986

M. Stonebraker, "Object Management in POSTGRES Using Procedures", *Proceedings of the International Workshop on Object-Oriented Database Systems*, Pacific Grove, California, September 1986, pp. 66-72. Published by IEEE Computer Society, Los Alamitos, California, 1986.

Stonebraker and Rowe 1986

M. Stonebraker and L. Rowe, "The Design of POSTGRES", *Proceedings of the International Conference on Management of Data, SIGMOD '86*, Washington, D.C., May 1986, pp. 340-355.

Sutherland 1989

J. Sutherland, "An Introduction to Object-Oriented Database Management Systems: Concepts and Applications", *Tutorial No. 6 Notes from the Object-Oriented Programming Systems, Languages and Applications Conference*, New Orleans, Louisiana, October 1989.

Thatte 1986

S. Thatte, "Persistent Memory: A Storage Architecture for Object-Oriented Database Systems", *Proceedings of the International Workshop on Object-Oriented Database Systems*, Pacific Grove, California, September 1986, pp. 148-159. Published by IEEE Computer Society, Los Alamitos, California, 1986.

Thomas et al. 1987

D. Thomas, W. LaLonde and J. Duimovich, "Efficient Support for Object Mutation and Transparent Forwarding", *Carleton University School of Computer Science Technical Report*, Ottawa, Ontario, November 1987.

END

1 | 2 | 0 | 4 | 9 | 1

FIN