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.

NL-339 (r. 88/04) c

Canada

The Design of an Object-Oriented Data Server for a Co-operative CAD Engineering Environment

by

Sergio A. Fiszman

A thesis submitted to the
School of Graduate Studies and Research
of the University of Ottawa
in partial fulfillment of the requirements for the degree of
Master of Applied Science in Electrical Engineering.

Ottawa-Carleton Institute for Electrical Engineering
University of Ottawa
Ottawa, Ontario

© Sergio A. Fiszman, Ottawa, Canada, I99I

National Library of Canada
Bibliothèque nationale du Canada

Canadian Theses Service
Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-75046-0

Canada

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

The undersigned hereby recommends to the

School of Graduate Studies and Research acceptance of the thesis

The Design of an Object-Oriented Data Server for a

Co-operative CAD Engineering Environment

submitted by Sergio A. Fiszman, P. Eng.,

in partial fulfillment of the requirements for the degree of

Master of Applied Science in Electrical Engineering.

Dr. Moshe Krieger

Thesis Supervisor

University of Ottawa

August 7, 1991

Abstract

A significant problem that users encounter when they must work co-operatively in a CAD engineering environment is the lack of support for data management and data control. In data management the main problems are the mismatch between data models used by users and the CAD engineering environment, the absence of mechanisms for tracking data evolution, and the lack of support for integrating data produced by multiple designers. In data control the main problems are the lack of suitable co-operative development activities, the absence of suitable data-ownership mechanisms, and the lack of control of changes-notification mechanisms.

These deficiencies in existing CAD engineering environments motivated the author to provide a solution by designing and prototyping a `data server`. The `data server` is an object-oriented datastore which is equipped with a services-layer. This services-layer supports both the data management and data control aspects in a co-operative CAD engineering environment. To implement these two functions and to narrow the data model semantics gap, the object-oriented model has been extended with a construct which captures not only the structural and behavioral parts of data, but also semantic relationships among data. The four services provided by the services-layer are the versioning service which addresses the tracking of data evolution, the configuration management service designed to support co-operative work, the data-ownership service which controls and safeguards the release of data during multi-stream development, and the changes-notification service that enhances the dynamic data control capability of the CAD engineering environment.

Acknowledgements

I am specially indebted to my thesis supervisor, Dr. Moshe Krieger, who provided encouragement and counseling at all times.

I am grateful to my colleague Dr. Yan-Zhen Qu, who provided helpful comments and criticism of this work.

Special thanks are due to Mr. Jose Duran, Mr. John Hogg, Ms. Jennifer Robinson, Ms. Catherine Weaver, and Mr. Ron Zwierzchowski who helped the author to make this thesis understandable.

I would like to thank my parents for all their support and encouragement throughout my studies.

Last, but not least, I would like to thank my wife, Viviana, and my two children Sebastian and Eytan, for their love, patience, help and understanding during the writing of this thesis.

Dedicated to my wife Viviana,

and to my children:

Sebastian and Eytan.

Contents

List of Figures viii
Chapter 1 Introduction 1
1.1 Background, Motivation, and Summary 1
1.2 Thesis Organization 6
Chapter 2 Object-Oriented Paradigm and Object-Oriented Databases 8
2.1 Introduction 8
2.2 The Object-Oriented Paradigm 8
2.2.1 Object and Object Identity 8
2.2.2 Class 9
2.2.3 Inheritance 10
2.2.4 Object-Oriented Programming in a NutShell 10
2.3 Object-Oriented Databases 11
Chapter 3 Requirements, Architecture, and Kernel Classes 15
3.1 Introduction 15
3.2 Requirements 15
3.2.1 Data Management 16
3.2.2 Data Control 17
3.3 Data Server Architecture 18
3.4 Kernel Constructs and Kernel Classes 22
Chapter 4 The Design of the Data Management Services 27
4.1 Introduction 27
4.2 The Versioning Model 28
4.2.1 Building Blocks 28
4.2.2 Operational Aspects 42
4.2.3 Schema Evolution 47
4.3 The Configuration Management Model 48
4.3.1 Associations 49
4.3.2 Configurations 52
4.3.3 Operational Aspects 55
Chapter 5 The Design of the Data Control Services 56
5.1 Introduction 56
5.2 The Data-Ownership Model 56
5.2.1 Building Blocks 56
5.2.2 Data-Ownership Mechanisms 59
5.2 3 Operational Aspects 65
5.3 The Changes-Notification Model 66
5.3.1 Building Blocks 66
5.3.2 Mechanisms 66
5.3.3 Operational Aspects 72
Chapter 6 Prototype and Test Suite 73
6.1 Introduction 73
6.2 Class Inheritance and Class Composition Hierarchies 74
6.3 Testing Strategy: The Test Suite 78
Chapter 7 Conclusions, Comparisons, and Future Work 83

7.1 Thesis Contributions....83
7.2 Comparisons....84
7.3 Future Work....85
Appendix A The Characteristics of the CAD VLSI Circuit Design Domain....87
A.1 Introduction....87
A.2 VLSI Design Philosophy....87
A.3 Characteristics of VLSI Data Entities....90
A.4 VLSI Design Interaction Characteristics....91
Appendix B A Survey of Commercial Object-Oriented Repositories....93
B.1 Introduction....93
B.2 Common Functionality....93
B.3 Non-Supported Functionality....94
B.4 Architecture....94
B.4.1 Objectivity/DB Architecture....95
B.4.2 Versant Architecture....95
B.4.3 ObjectStore Architecture....95
B.4.4 GemStone Architecture....96
Appendix C Description of the Kernel Classes....97
C.1 Kernel Classes....97
Appendix D The Data Server Prototype....100
D.1 Test Suite....100
Appendix E Source Code Used for the Data Server Prototype....115
Appendix F Glossary....154
References....165

List of Figures

Figure 1 An Abstract View of a CAD Engineering Environment 1
Figure 2 An Abstract View of the Data Server 4
Figure 3 Architecture of the Data Server 19
Figure 4 The Multi-Client Data Server Environment 21
Figure 5 The Kernel Class Hierarchy 23
Figure 6 Layered Hierarchy of Libraries, and Library Class Hierarchy 30
Figure 7 Library Structure, and Logical to Physical Mapping of the Life Cycle. 33
Figure 8 An FR Diagram for the Structure Part of a Library and its Components 34
Figure 9 The Class Hierarchy of the Version Class 35
Figure 10 Linear and Branching Versioning 37
Figure 11 Branching Characteristics: Derivation and Merging 38
Figure 12 Static and Dynamic Version Binding 41
Figure 13 Checking-in of N-levels of a Composite Configuration 44
Figure 14 An Abstract Representation of a Virtual Path 46
Figure 15 Two-Level Configuration: An Example of Nested Configuration 53
Figure 16 The Class Hierarchy for the Configuration Class 54
Figure 17 An ER Diagram for the Data-Ownership Model 58
Figure 18 Checking-in Data between Paths 59
Figure 19 A Two-Levels: Composite Scope 63
Figure 20 An ER Diagram for the Changes-Notification Model 68
Figure 21 The Transition States of a Monitored Object 71
Figure 22 The Prototype Inheritance Class Hierarchy 76
Figure 23 The Prototype Composition Class Hierarchy 77
Figure 24 The Parts of a Microcontroller Module 79
Figure 25 Parts of an Instance Version: Microcontroller.3 81
Figure 26 Abstractions, Representations, and Data Entities 89
Figure 27 The OO-DBMS Symbol Dictionaries 101
Figure 28 Libraries, Streams and Contexts 102
Figure 29 The Parts of a Microcontroller Class 104
Figure 30 The Contents of a Context 107
Figure 32 The Parts of an Instance Version: Microcontroller.2 109
Figure 33 The Closure for the instance version: Microcontroller.3 110
Figure 34 The Virtual Path of an Instance Version 113
Figure 35 An Association Between Instance Versions 114

Chapter 1

Introduction

1.1 Background, Motivation, and Summary

In recent years there has been a considerable development of computer-aided design (CAD) environments for a wide variety of engineering applications. An abstract view of a CAD environment is shown in Figure 1. A CAD engineering environment consists of a toolset (collection of tools) and a datastore.[1]

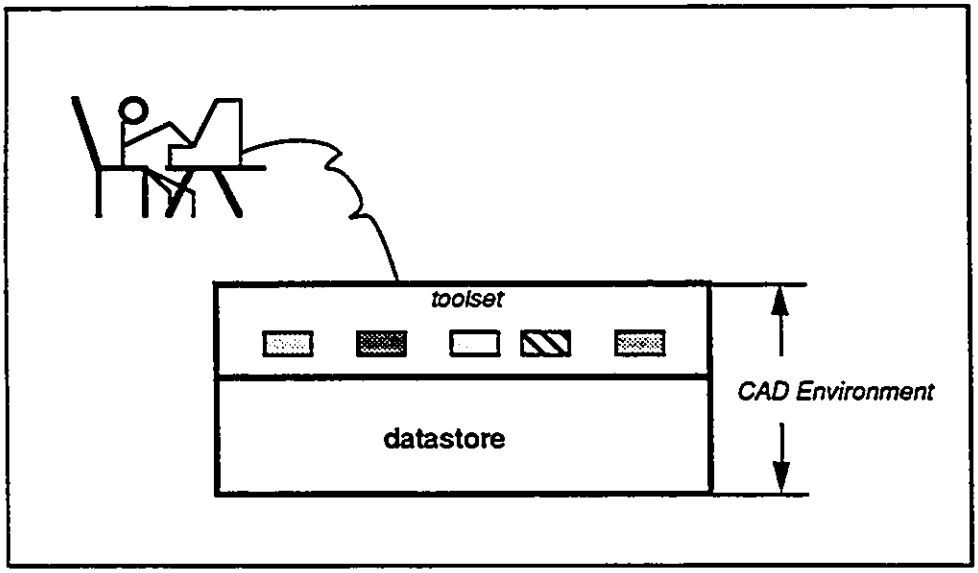


Figure 1 An Abstract View of a CAD Engineering Environment

The toolset is usually composed of various kinds of tools. For example, *synthesis tools* are used to create a design (based on specifications) which obeys engineering criteria; *analysis tools* are

[1] Data is considered design information or design-related information (for example, documentation), or both. Design information includes design requirements and specifications, design implementation, and execution and testing related information.

used to check the correctness of a design according to a set of rules, or by a simulation of the design; *data-management* tools are used to organize data according to specified relations among data.

The *datastore*, the other component of a CAD environment, stores data persistently.[1] It can also be considered as a place that holds multiple views of a design, which are processed by different tools quasi-sequentially.

The most significant deficiencies of current CAD environments are the lack of support for the data management and the data control aspects needed in co-operative CAD. These conclusions are based on the outline on CAD engineering environments given in Appendix A, and on the survey on object-oriented databases presented in Appendix B.

Data management refers to the ability to model, track, and relate complex data. There are three significant deficiencies related to this aspect of CAD environments. First, there is a semantic gap between the data used by the toolset, and the corresponding representation(s) in the datastore. This gap is significant when a CAD environment uses a conventional database management system (relational, hierarchical, or network) as the datastore. Conventional database management systems (DBMS) were not designed for CAD applications; they lack the modeling power and performance needed to support complex CAD data structures [Maier 89a, Zdonik 87]. More advanced (also called state-of-the-art) CAD environments use an object-oriented DBMS as the datastore. Object-oriented database management systems (OO-DBMS) support not only the services provided by conventional database management systems[2] (for example, relational), but also the object-oriented model [Stonebraker 90]. The object-oriented model [Wegner 89] provides a natural way to map real-world objects directly to computer representations [Thompson 91], and is suitable for the

---

[1] That is, the stored data exists beyond the life time of the tools that created it. Generally, the datastore has a set of primitives that is used by the CAD tools to store/retrieve data, and to search/query for specific data.
[2] The key services supported by conventional database management systems include concurrency control, a query facility, and persistent storage.

support of the extensibility and optionality aspects needed in CAD.[1] However, the object-oriented model does not automatically provide the semantic constructs needed to relate and associate complex data. A second important deficiency is the lack of support for history management of both data updates and design alternatives. While conventional DBMSs do not support this at all, many OO-DBMSs use versioning as a history management mechanism; however, versioning is not integrated with other data-management-related services (for example, configuration management[2]). The third important data management deficiency is the lack of support for the integration of parts of a design produced by multiple designers.

Data control in co-operative CAD refers to the ability to control access to data, provide notifications of changes to data, and control concurrent data access. Current CAD environments have three major shortcomings in this area. First, there is a lack of control over access to data by multiple designers. Conventional or OO-DBMSs, used by current CAD engineering environments do not automatically support the mechanisms needed for sharing of data in different stages of development (in progress, stable or released, and so on) by multiple designers. These CAD environments were designed either for a single user or, at most, a small number of users. In such an environment, a design evolves like a product on an assembly line. By contrast, in co-operative CAD engineering environments, the parts of a design are usually developed in parallel, and data in progress must be shared among multiple designers. Second, current CAD environments lack support for the ownership of data by multiple users. Thus, there is an absence of control mechanisms that safeguard the access to data. Third, current CAD environments do not notify affected parties when data updates occur, or when other specific events occur on data.

The challenge of solving these deficiencies motivated this thesis. The following chapters describe a solution which overcomes the problems listed above. A `data server` consisting of a datastore equiped by a services-layer was designed and prototyped. The services-layer provides

---

[1] Existing functionality can be extended via inheritance, polymorphism, and specialization mechanisms.

[2] Configuration management is the ability to relate versions of different types of data.

remedies for each of the datastore deficiencies. An abstract view of the `data server` is shown in Figure 2.

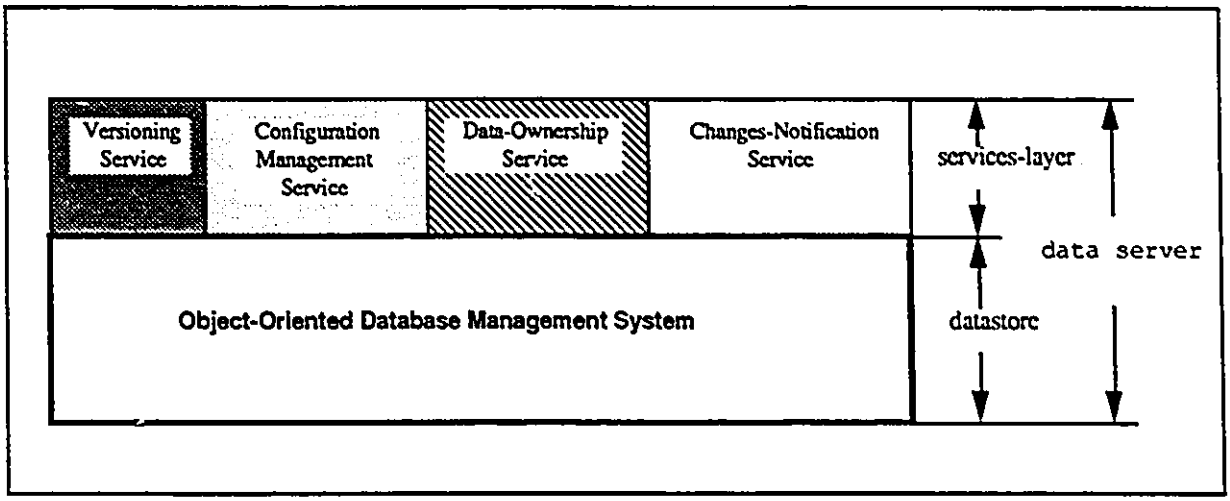


Figure 2 An Abstract View of the Data Server

The services-layer consists of four service-functional modules. These modules provide the following services:

- Versioning service. This allows multiple users to manage efficiently the evolution of their prototypes or design alternatives during the design life cycle, by means of versions.[1]

- Configuration management service. This provides multiple users with the ability to associate and relate versions of different types, and to represent complex data by means of configurations.[2]

[1] A version is a persistent snapshot of information.
[2] A configuration is a set of related versions.

- Data-Ownership service. This allows multiple users to control both access to information and the release of information.

- Changes-notification service. This service allows multiple users to be notified when a set of events occur on specific data.

An object-oriented data base management system (OO-DBMS) plays the role of the datastore and supports concurrency control, which is one of the key services of the data control aspect of co-operative CAD.

The two main contributions of this thesis correspond to the two key aspects of a co-operative CAD engineering environment: data management and data control.

First, to support the data management aspect, we provide a suitable semantic model. The model can be used in both end-user applications and the datastore, so the semantic gap between application and datastore models is removed. The data that tools manipulate can be saved in the datastore without changes to representations of semantics. This model is also used as the kernel model to integrate the `data server` services. It extends the object-oriented model with a link construct. A *link* is an object which is used to implement associations and relations among CAD objects. For example, this construct can be used to relate a version of a modular view of a microcontroller with a version of documented features. To track the evolution of complex data, a *versioning* service is developed which allows designers to manage efficiently the evolution of prototypes or alternative designs during the design life cycle. For example, a designer who is requested to give two different implementations of a VLSI half-adder to the "testing" department, can use the `data server` versioning service to create two versions, one for each VLSI half-adder implementation. To relate complex data, a *configuration management* service is developed. This service provides the ability to associate and relate versions of different types of data. For example, this service allows the inclusion within a container of other containers which represent different views of a chip design.

Second, to support the data control aspect of a co-operative CAD engineering environment a set of services are developed. To control the access and release of data, a *data-ownership* service is provided. It controls the release of data to the public domain to guarantee the "sanity" of the data. For example, this service can be used to stop the release of an untested version of a bypass cardiac unit to the public domain. To notify dependents of changes to data, a *changes-notification* service is developed. This service informs to the affected parties when a particular event happens to some data (for example, when a specific component of a VLSI chip is upgraded). To control concurrent access to data and support data integrity, the `data server` uses the transaction management and concurrency control services provided by the OO-DBMS.

1.2 Thesis Organization

This thesis is divided into four parts. The first part provides an overview of both the object-oriented model and object-oriented databases. The second part specifies the requirements for the `data server`, describes the `data server` architecture, and provides the `data server` kernel semantic model. The third part provides the design of the `data server` services. The last part presents the implementation of the `data server` prototype. An outline of each chapter of the thesis is given below.

Chapter 2 provides an overview of the object-oriented model and object-oriented databases.

Chapter 3 presents the list of `data server` requirements and their rationale. In addition, this chapter discusses the adopted multi-client `data server` architecture, and the Kernel constructs and classes.

Chapters 4 and 5 provide the semantic model and the design of the services presented as requirements in Chapter 3. Chapter 4 provides the design of the *data management* services and Chapter 5 presents the design of the *data control* services.

Chapter 6 describes the prototype implementation[1] of the `data server`.

Chapter 7 summarizes the thesis contributions and the associated benefits. This chapter also contrasts the concepts used in the design of the `data server` with other works in this area; in addition, it indicates the need for future work.

The thesis also includes six appendices which present supplementary material. Appendix A provides an overview of the VLSI design process which may help the reader to understand some of the problems faced by the design engineering community at large. Appendix B provides a survey of a number of commercial object-oriented databases (GemStone, Objectivity/DB, Versant, and ObjectStore). Appendix C provides a description of the structural and behavioral parts for all the kernel classes. Appendix D describes a test suite which was used to demonstrate and test the functionality of the `data server` prototype. Appendix E includes the OPAL and Smalltalk-80™ source code used to implement the `data server` prototype. Finally, Appendix F contains a glossary for this thesis.

---

[1] The prototype was coded using Smalltalk-80 [Goldberg 83] and OPAL. OPAL is the programing language supported by GemStone [Maier 89], which was the adopted OO-DBMS. The server was a Sun-4™ workstation, and the clients were Sparc-1™ workstations. The operating system used was Unix™. The software and hardware were available to us at Bell-Northern Research (BNR). UNIX is a registered trademark of AT&T. Sun-4 and Sparc-1 are registered trademarks of Sun Microsystems, Inc.

™OPAL is a registered trademark of Servio Corp., and Smalltalk-80 is a registered trademark of ParcPlace Systems, Inc.

Chapter 2

Object-Oriented Paradigm and Object-Oriented Databases

2.1 Introduction

This chapter provides an overview of both the object-oriented paradigm and object-oriented databases, to introduce the basic terminology used in this thesis. The organization of this chapter is as follows. First, the object-oriented paradigm is introduced. Second, a short overview of object-oriented databases is provided.

2.2 The Object-Oriented Paradigm

The object-oriented paradigm provides a natural way to map real world objects and their relationships directly to computer representations. This section describes the building blocks of the object-oriented paradigm: object and object identity, class, and inheritance.

The definitions and discussions provided in this section are based on the works of Thompson [Thompson 91], Wegner [Wegner 89], and Meyer [Meyer 88, Meyer 87].

2.2.1 Object and Object Identity

Objects are constructs that are used to represent abstract or concrete real-world things in the application domain being modeled. For example, an adder in a computer-aided design (CAD) application is a real-world thing. An object is an artifact that has a local state and an ability to manipulate its local state in response to external requests. The local *state* of an object is the set of values of its attributes (variously called instance variables, properties, data members, or slots). The external requests are called *messages*; and the program code that operates on the state to change it in response to messages is called a *method*. The collection of all messages defined for an object

constitutes its *abstract interface* or *type*. The collection of all methods for an object defines its *behavior*.

The principle of *data abstraction* states that the local state and the methods of an object are not visible to users of the object; they may only interact with the object by making requests to the object through messages. This principle promotes modularity and maintainability. Since the user of an object cannot make assumptions about the implementation and internal representations of the object, the underlying implementations can be changed without affecting users.

Each object is associated with a unique identifier called its *object identity*, regardless of its current state. The idea is that an object has an existence which is independent of its value. Identity is a stronger concept than simply a value describing an object; it can not be changed in the same way that other values describing the object can be changed.

2.2.2 Class

Class is a means of grouping objects that share the same attributes and behavior. A class is implemented by choosing a collection of attributes, or instance variables, in which to store the internal state of the instances, by writing a method for each message defining the abstract interface of the instances of the class. A class is, therefore, the implementation of the abstract interface or type of an object, and an object's structure and behavior are defined by its class. Members of the class are called *instance objects* or *instances*.

A class consists of a set of attributes. The domain of an attribute may be a class that, in turn, may have attributes with domains as classes. This nested structure of a class can give rise to a directed, possibly cyclic graph representing the composition relationship between a class and its attributes. In object-oriented systems, the composition relationship is equivalent to the semantic modeling concept of *aggregation*. The *class composition hierarchy* is orthogonal to the class inheritance

hierarchy discussed below. The composition hierarchy provides a way to model rich, complex structures without first flattening out the structure. Examples of aggregation relationships are *is-part* and *is-owned-by*.

2.2.3 Inheritance

Object-oriented systems allow the user to derive a new class from an existing class. The new class *inherits* all attributes and methods of the original class and may define additional attributes and methods and may redefine inherited methods. The new class, a *subclass*, specializes the original class. The original class is called the *superclass* of the derived class, and is its generalization. Inheritance realizes the semantic modeling relationship *is-a*. It reduces the need to specify redundant information and hence simplifies updating and modification. It is also used to create objects that are almost like other objects with a few incremental changes. The common use of *class libraries* is a good example of software reuse though the inheritance mechanism. Designers build upon basic objects provided by class libraries by specializing the classes in the libraries. When a new class can have only one immediate superclass, it is called *single inheritance*; when a class inherits from multiple classes, we have *multiple inheritance*. The tree resulting from the subclass-superclass relationships among classes, under single inheritance, is referred to as the *class hierarchy*; the graph resulting from the subclass-superclass relationships among classes, under multiple inheritance, is called the *inheritance hierarchy*.[1]

2.2.4 Object-Oriented Programming in a NutShell

In summary, object-oriented programming introduces systematic techniques for managing software components. *Objects* provide a high-level primitive notion of modularity for modeling

[1] However, the names given to both hierarchies are frequently referenced in the literature as class hierarchy.

applications directly. *Classes* facilitate the management of objects by treating them as values. *Inheritance* supports the management of classes by organizing them into hierarchies.

2.3 Object-Oriented Databases

This section provides an overview of object-oriented databases, based on the works of Breitbart [Brei 90], Thompson [Thompson 91], Kim [Kim 90, Kim 89a, Kim 89, Kim 88], Kort [Kort 83], Bernstein [Bern 81], Ullman [Ullman 80], and Gray [Gray 81, Gray 78].

An *object-oriented database* is a class of programming system with the capability of a conventional database management system (management of large amounts of persistent data, transaction-based concurrent access, data model, and query language), along with the features of the object-oriented paradigm.

The most fundamental function of an object-oriented database (OODB) is to provide a way for objects to persist beyond the scope of an individual programming execution. At the core of the OODB there is a database object-oriented model which is the mechanism for specifying the semantics of a database and the operations that can be performed on the data in the database. The representations of the persistent objects are captured through a database *schema* which is an organization of classes that describe the logical structure of the database. By comprehending the physical class definition alone, OODBs can materialize an object's[1] state (recreate it or restore it from secondary storage) and store that state when requested.

Another primary purpose of a database is to allow sharing of information. An OODB must regulate access to information by multiple concurrent users, each of whom is potentially unaware of the existence of the other users. The concurrency control mechanism of an OODB preserve the illusion that each user is executing alone in a dedicated system. Thus, it prevents data

---

[1] Within an OODB, an object has a unique object identifier (OID). Thus, the OID space is large enough to provide unique OIDs for all persistent objects over the OODB's lifetime.

modifications performed by one user from interfering with data retrievals and updates performed by another. A *transaction* is the fundamental building block used by the concurrency control mechanism. An OODB enforces the rule that all access to persistent data must take place within a transaction. Transactions serve to group process execution into units (also called operations) that are, from the point of view of other processes, *atomic*: that is, they either happen all at once or not at all. A transaction never partially completes: it either commits (completes successfully) or aborts (the effects of its execution are "*rolled back*", so it appears to other processes that these effects never occurred at all). The *transaction* is the unit of concurrency and the unit of recovery for database applications. As a unit of concurrency, the operations of several transactions can be interleaved so they will not interfere with each other. As a unit of recovery, a transaction either succeeds totally, or it has no effect on the object store.

In general, OODBs achieve concurrency control by adopting synchronization techniques that ensure the serializability of interleaved transactions. These techniques prevent or detect conflicts when they occur. Synchronization techniques may be categorized as pessimistic or optimistic-concurrency-control. *Pessimistic-concurrency-control* assumes that the concurrent transactions that require access to the same object will lead to conflict and hence makes one transaction wait for another to complete before it is started. Synchronization techniques based on *two-phase locking*[1] fall into this category. *Two-phase locking* (2PL) is a pessimistic-concurrency-control technique which synchronizes reads and writes by explicitly detecting and preventing conflicts between concurrent operations. Before reading a data item x, a transaction must "own" a *read-lock* on x. The ownership of locks is governed by two rules:

(1) different transactions cannot simultaneously own conflicting locks

(2) once a transaction surrenders ownership of a lock, it may never obtain additional locks.

---

[1]There are different implementations of 2PL, which can be found in [Bern 81].

The disadvantage of the 2PL approach is the possibility of deadlock. A *deadlock* arises when a set of transactions are waiting for each other to commit before they can proceed.[1] On the other hand, *optimistic-concurrency-control* takes the view that concurrent transactions that require access to the same object will not necessarily conflict. Transactions are allowed to proceed until a conflict is detected. The disadvantage of this approach is the overhead incurred in re-doing transactions once conflicts are detected. The basic idea of optimistic methods is the following: instead of suspending or rejecting conflicting operations as in 2PL or timestamp-ordering (T/O)[2], always execute a transaction to completion. However, write operations are performed only in a local workspace. Only if the validation test is passed at transaction commit time are the writes applied to the database. If the validation test fails, the temporary writes are ignored and the transaction is restarted.

It is worth mentioning that most OODBs support long transactions. A *long transaction* is a transaction which spans multiple short transactions and multiple application processes, and whose lifetime does not have any system-imposed upper bound (unlike short transactions). A long transaction can be committed in its entirely, or rolled back.[3]

---

[1] The basic characteristic of a deadlock is the existence of a set of transactions in which each transaction is waiting for a resource locked by another transaction that is directly or indirectly waiting for a resource locked by other transaction(s). This situation can be conveniently represented with a *wait-for* graph. A *wait-for* graph is a directed graph having transactions as nodes; an edge from transaction T1 to transaction T2 represents the fact that T1 is waiting for a resource locked by T2. The existence of a deadlock situation corresponds to the existence of a cycle in the *wait-for* graph. Two general techniques are available for deadlock resolution:

- Deadlock prevention: This is a cautious scheme in which a transaction is aborted and restarted when the system detects that deadlock may occur.
- Deadlock detection: transactions wait for each other in an uncontrolled manner and are aborted only if a deadlock actually occurs.

[2] *Timestamp ordering (T/O)* is a technique whereby a serialization order is selected a priori and a transaction execution is forced to obey this order. Each transaction has a starting time and each piece of data has the timestamp of the transaction that last read it and the transaction that last wrote it. These techniques allow a transaction to read or write a data object x only if x had last been written by an older transaction; otherwise it rejects the operation and restarts the transaction.

[3] Other types of transactions which can be supported by OODB's are: nested, distributed, and multi-threaded transactions. A *nested transaction* is a way of grouping operations/short transactions as an atomic step within a transaction. A *distributed transaction* is a way of grouping a set of operations/short transactions that access disjoint data stored in multiple sites in the computer network, in one atomic step. A *multi-threaded transaction* is a way of grouping operations into one atomic step from a single thread to multiple threads running on different workstations. A multi-threaded transaction supports cooperative design work by modeling each team

A query language is the other basic functionality supported by an OODB. A *query* is a declarative specification of a set of *objects* in the *database* that satisfy a set of conditions. Applications use queries to retrieve or manipulate information that satisfies some predicate; the targets and results of queries are sets of objects. In addition, to speed-up query processing, an OODB supports: caching, clustering[1] , and indexing[2] techniques.

In summary, OODBs support not only the functionality provided by conventional databases systems (relational, hierarchical or network) but also the ability to model complex objects naturally [Stonebraker 90]. In Appendix B a survey of current commercial object-oriented databases (usually called object-oriented repositories or object-oriented database management systems) is presented. This survey helped to uncover the functionality needed to support co-operative CAD engineering environments.

---

member's work as a thread, and the entire team's work as a transaction. Members of a team, being threads of a transaction, have shared access to each other's persistent data.

[1] Clustering refers to storing related objects close together on secondary storage. Clustering is a highly useful technique that database systems have used to minimize the I/O cost of retrieving a set of related objects.

[2] An index is a data structure that an OODB uses to expedite the evaluation of a query that retrieves a small subset of a large database. An index is maintained on an attribute (or combination of attributes) of a class; an index is logically a list of pairs *<key-value, list of object identifiers>* , where the *key-value* is a value in the indexed attribute(s), and each *identifier* in the list of *object identifiers* is the identifier of an object for which the indexed attribute holds the *key-value*. An index is usually organized for storage efficiency in a *B-tree* structure or its variants [Comer 79].

Chapter 3

Requirements, Architecture, and Kernel Classes

3.1 Introduction

The set of requirements presented in this chapter is organized according to the motivation of this thesis, i.e. to address the requirements related to the data management and data control aspects in a co-operative CAD engineering design environment. To implement the requirements, we adopted a top-down engineering design approach. First, we considered the architecture of the `data server` to determine the deployment of functional modules that will support the data management and data control functionality. Second, we defined the kernel classes that support both the functional architecture of the `data server`, and the data management and data control aspects of co-operative CAD.

The organization of this chapter is as follows: First, the requirements for the `data server` are presented. Second, the architecture of the `data server` is described. Third, the kernel constructs, which extend the object-oriented model, and which are used to develop the `data server` services, are presented.

3.2 Requirements

The analysis of the problems[1] that users experience when they must co-operate in a CAD engineering environment, and the CAD design work experience acquired at BNR over eight years, drove the analysis of the requirements for the `data server`.[2] These requirements specify *what* the `data server` will provide without describing *how* it will do it. These requirements, which

---

[1] These problems are described in Chapter 1, Appendix A, and Appendix B.

[2] There is little agreement in the general literature as to what forms a proper set of requirement for a co-operative engineering design environment.

specify the set of services that should be supported by the `data server`, are divided into two categories: data management and data control.

3.2.1 Data Management

The data management services should provide the ability to track data evolution, and to relate complex data, which are two essential mechanisms needed to support a suitable semantic model. A ***suitable semantic model*** is a set of well-defined constructs used to describe the semantics of the target to be modeled (e.g. applications, design activities, complex interrelations between data, and so on). All the `data server` services are one aspect of the system functionality that realize the modeling power of the semantic model. The data management services are further divided into two related services: versioning and configuration management.

Versioning Service: The versioning service should provide the ability to track and to manage the history of instance versions. An *instance version* is a persistent snapshot of an instance and an *instance* refers to an object created from a class.[1] This service should support:

- Instance versioning (a mechanism which tracks the evolution of instance versions during a design life cycle). This mechanism should support two types of versioning: branching versioning and linear versioning. *Branching versioning* is the ability of a parent version to have multiple successors. *Linear versioning* is the ability of a parent version to have only one successor.

- Data sharing (the support of common mechanisms to exchange and share instance versions across libraries).[2]

- Ad-hoc queries (the ability to efficiently retrieve data from a library based on associative conditions), and navigation (the ability to search for a specific instance version).

---

[1] The methods and structure of an instance are determined by its class.

[2] A *library* is a datastore unit.

- Class-schema-hierarchy versioning (a mechanism which tracks the evolution of the class schema hierarchy during a design life cycle). A *class schema hierarchy* is an organization of class definitions that describe the logical structure of an object-oriented library, and which are related by means of inheritance relationships.

Configuration Management Service: It should provide the ability to manage and relate complex data. This service should support:

- Nested configurations: A *nested configuration* is a nested set of atomic configurations. An *atomic configuration* is a set of instance versions associated by relationships.
- Tracking the evolution of configurations during the design cycle.
- Relationships and associations between instance versions, and between configurations.
- Operations on the transitive closure of a relationship.

3.2.2 Data Control

The data control requirements are concerned with the ability to control the release of data, the notification of changes to data, and the control of concurrent access to data. *Release of data* is checking-in (writing) data to a public domain. The public domain can be accessed by the client community at large. The services specified by these requirements are given below.

Data-Ownership Service: It should provide the ability to control the access rights that designers have over information. Thus, this service should support common mechanisms to control and to co-ordinate the release of information to the public domain.

Changes Notification Service: It should provide the ability to notify affected parties that an event occurred on a monitored instance version. This service should, in addition, support the mechanisms that allow clients to approve the notified changes.

Transaction Service: It should provide the ability to safeguard the data integrity of the datastore. This service should support:

- A Transaction Model. The `data server` should support the persistent storage and retrieval of data, and ensure that all access to persistent data take place within a transaction. A transaction should be the unit of concurrency and the unit of recovery for the `data server`. As a unit of concurrency, the operations of several transactions can be interleaved so they will not interfere with each other. As a unit of recovery, a transaction either succeeds totally or it has no effect on the object store.

- Data Integrity. The `data server` should support both referential integrity, and constraint integrity. Referential integrity guards against the existence of dangling references. Constraint integrity provides a means of ensuring that changes made to a datastore by authorized users do not result in loss of data consistency.

3.3 Data Server Architecture

The architecture describes the deployment of the `data server`'s service-functional modules and the deployment of libraries. An abstract view of the layout of the service-functional modules and libraries is shown in Figure 3.

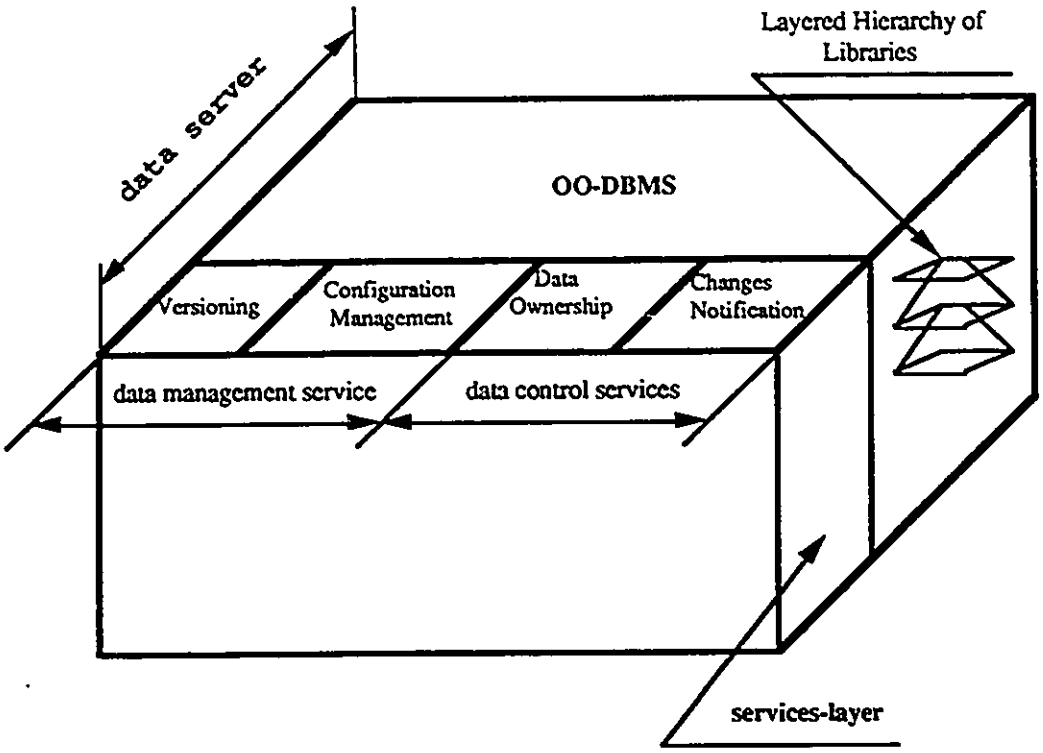


Figure 3 Architecture of the `Data Server`

The three building blocks of the architecture are:

Services-Layer: This layer supports the services specified in the requirements for the `data server`. Each service outlined in the set of requirements is supported by a service-functional module. The service-functional modules are:

- Versioning module: This service-functional module supports the requirements for the versioning service.
- Configuration Management module: This service-functional module supports the requirements for the configuration management service.
- Data-ownership module: This service-functional module supports the requirements for the data-ownership service.

- Changes-notification module: This service-functional module supports the requirements for the changes-notification service.

Object-Oriented DBMS: At the core of the OO-DBMS there is a database object-oriented model which is the mechanism for specifying the semantics of a database and the operations that can be performed on the data in the datastore. The object-oriented model assigns semantics to the data stored in the database by specifying the structure and behavior of the data. The operational component of the database object-oriented model consists of a general-purpose collection of primitives that support the query and modification of a database. The key role of the OO-DBMS is to control the database. This control involves the four aspects of semantic integrity:

- making sure the database is an accurate model of its application environment
- security (authorization)
- concurrency (handling multiple simultaneous users)
- recovery (restoring the database in the event of a failure of some type)

Layered Hierarchy of Libraries: A three-layered hierarchy of libraries allows clients[1] to share and store data. A *library* is a datastore unit. The top layer contains public libraries. Public libraries contain stable data (data which can neither be updated nor deleted). The middle layer contains semi-public (or project) libraries. A semi-public library contains data in progress, which can be shared among authorized team members. The lower layer contains private libraries. A private library can be accessed only by its owners.

Clients and the `data server` use a LAN-Communications layer to communicate over the LAN. Clients interact with the `data server` through a well-defined programmatic interface. This

[1] Users and applications which interact with the `data server`.

interface consists of a set of primitives that clients use to access both the `data server` services and libraries. The deployment of clients and the `data server` over the LAN is shown in Figure 4.

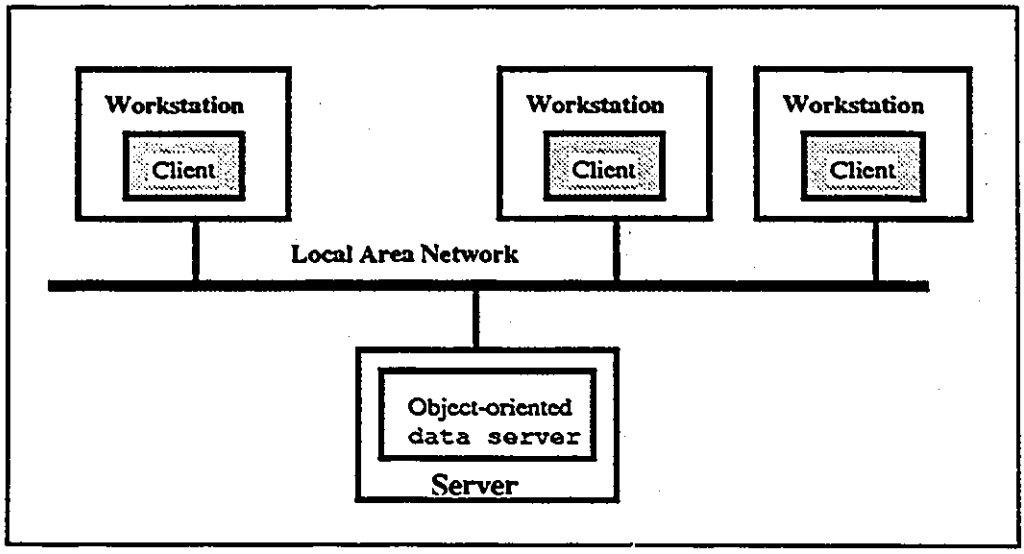


Figure 4 The Multi-Client `Data Server` Environment

The organization of clients and `data server` in the CAD environment is a multi-client `data server` topology. The advantages of a multi-client `data server` organization are as follows. First, it requires simpler data management and data control that other types of organizations (for example, multi-client multi-server). Second, this organization is frequently used in the industry, where many engineering workstations are connected to a server over a LAN; thus the `data server` functionality can be easily supported and upgraded.

On the other hand, it is acknowledged that the multi-client `data server` topology may create a bottleneck,[1] and may also introduce undesirable vulnerability.[2]

---

[1] All requests are processed through the data server.

[2] When the data server fails, the clients must wait until the problem is fixed.

3.4 Kernel Constructs and Kernel Classes

One of the requirements that must be supported in a CAD environment is the provision of a suitable model to capture the semantics of complex data. The model developed in this thesis extends the object-oriented model, and satisfies these requirements. We use this extended object-oriented model to implement the `data server` services. The two constructs used in this model are object and link. An *object* is a construct which represents the behavior and structure of real world data, while a *link* is a construct which represents associations between objects, and has attributes and behavior. The attributes of the link construct are used to indicate its cardinality, to specify the origin and destination objects associated by the link, and to indicate what distinguishes the link. The behavioral part of the link construct allows creation of a link, access to the link attributes, and the update of the link attributes.

In the object-oriented paradigm, objects with similar semantics are organized into classes. Each class has a type definition, which defines attributes and behavior for the objects that the class groups. To implement both the link and the objects used by the data server services, we have categorized them into a set of Kernel classes as shown in Figure 5.

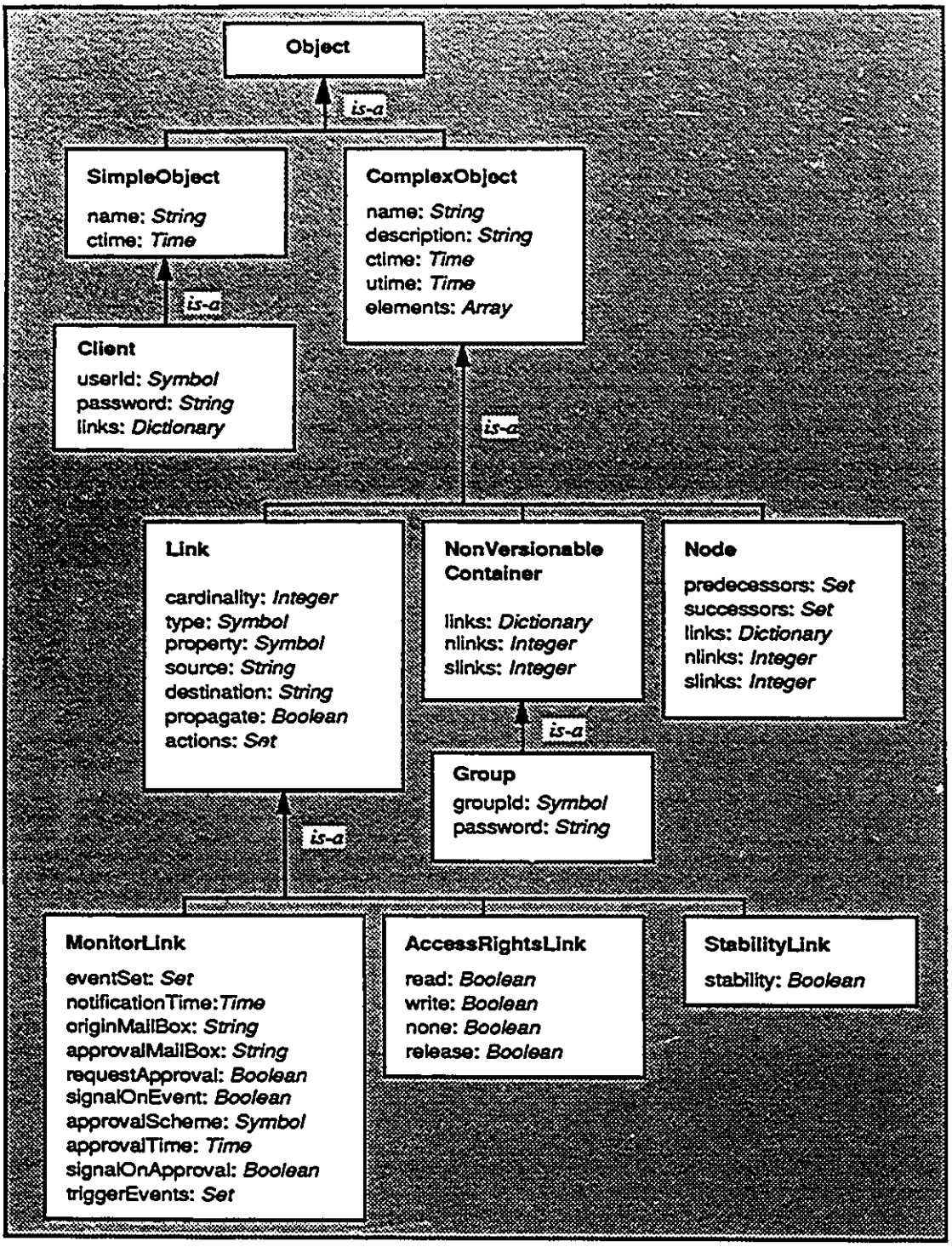


Figure 5 The Kernel Class Hierarchy

If an object contains other objects as its components, it is considered *complex*; otherwise it is considered *simple*. In this work, a complex object can be either a Link, a NonVersionableContainer, or a Node. A *Link* is a complex object which is used to represent associations and relationships between objects. A *NonVersionableContainer* is a complex object which is used to represent a non-versioned object. A *Node* is a complex object which is used to represent a versioned object. The shown Kernel class hierarchy has been developed after prototyping the `data server`.[1] It should not be a surprise that we have chosen the prototype direction to gain experience with the semantic model that we are proposing. After all, we were not only basing the suitable semantic model for co-operative CAD on the object-oriented model, but we were also following the object-oriented methodology (which promotes prototyping) as a way of understanding and realizing semantic concepts rapidly. The descriptions of the structure and behavior parts for two kernel classes (Link and Node) are given below. A detailed description of the structure and the behavior components of the remaining kernel classes is given in Appendix C.

Link

A *Link* is an object which is used to associate two objects. The orientation of this association is from the source object to the destination object. The attributes (structure) of a Link are:

cardinality — specifies the number of instances of both the source object and the destination object that can be associated through the link. The format of cardinality is source/destination, and the allowable cardinality numbers are: *1/1*, *1/n*, *n/1*, and *n/m*

type — specifies whether a reverse link must be set automatically between the destination object and the source object

property — determines the type of operations which can be performed on the destination object (e.g. *Uses*, *Stability*, *Monitor*, and so on)

---

[1] The data server prototype is described in Chapter 6.

source — specifies the origin object name

destination — specifies the destination object name

propagate — specifies whether a set of actions performed on the source object should be propagated to the destination object

actions — specifies the set of actions to be propagated to the destination object (for example, a lock performed on the source object can be propagated to the destination object)

The behavioral component of the Link class supports a basic set operations. A client can create a link, can add a link to a pair of objects, can drop a link from a pair of objects, can move a link from one pair of objects to another pair of objects, and can copy a link from one pair of objects to another pair of objects. As with other classes, all the attributes defined in the Link class can be read and can be updated by a client. Furthermore, the *property* attribute of a link determines not only the purpose of the link, but also, in combination with the class definition, determines the set of operations which can be invoked on the destination object.

Node

A *Node* is a versioned object which can contain objects. The attributes of a Node are:
predecessors — specifies a set of parent nodes (this is used for versioning purposes)

successors — specifies a set of child nodes (this is used for versioning purposes)

links — specifies a dictionary of instances of class Link

nlinks — specifies the number of instances of class Link leading from this node

slinks — specifies the number of instances of class Link leading to this node

The behavior part of a Node supports a basic set of operations. A client can create a node, can query the instance of the class Node, can read, can display and update the attributes of a node, and can add and delete links from a node. Since nodes are versioned objects, a node can become a successor of a set of nodes, or a node can become a predecessor of a set of nodes. As with other Kernel classes, a set of navigation and query operations are supported. A client can perform a search for a node which contains specific values, and can query for a set of nodes which satisfy specific conditions.

Chapter 4

The Design of the Data Management Services

4.1 Introduction

This chapter will concentrate on the design of the data management services. The data management services consist of the versioning service and the configuration management service. The *versioning* service allows multiple users to manage and track through versions (persistent snapshots of information) the inevitable changes that happen to a design[1] during its life-cycle.[2] These changes happen mainly due to the iterative and exploratory nature of the engineering design process. The building blocks of the versioning service are nodes, links, libraries and workspaces, versions, and composite-configurations. The *configuration management* service allows the creation of associations and relationships between objects, and also allows the tracking of the evolution of configurations during the life-cycle. The configuration management service uses the services provided by the versioning service to version configurations. The building blocks of the configuration management service are objects, links, associations, relationships, and configurations.

This chapter is organized in two parts. The first part discusses the design of the versioning service. The second part discusses the design of the configuration management service. The design of the versioning service describes the building blocks and the operational aspects of the versioning model, and the schema evolution scheme. The design of the configuration management service describes the building blocks and the operational aspects of the configuration management model.

---

[1] A design is "anything" that can be modeled and engineered.

[2] The design-life-cycle approach, in co-operative CAD, is closer to the Spiral [Boehm 88], Fountain, and Prototype [Sellers 90] models, where the design is created through an incremental, iterative approach, in which the products of the design gently unfold over time. The design approach in co-operative CAD no longer follows the Waterfall [Roy 70] model.

4.2 The Versioning Model

The description of the versioning model consists of three parts. The first part describes its building blocks. The second part presents its operational aspects. The third part discusses the schema evolution related to versioning.

4.2.1 Building Blocks

The building blocks of the versioning model, which support the versioning requirements outlined in Chapter 3, are:

- Kernel classes (specifically the Node and Link classes described in Chapter 3)
- libraries and workspaces (data is distributed and organized into libraries and workspaces)
- versions (persistent snapshots of data)
- composite-configurations (sets of objects related by associations)

The Kernel classes have been described in Chapter 3. A detailed description of the other building blocks is given below.

4.2.1.1 Libraries and Workspaces

In this thesis, we consider a *library* as a datastore unit, and a *workspace* as a private file system. The key differences between a library and a workspace are:

- First, clients manage their workspaces (for example, UNIX file systems, the Smalltalk-80 environment, and so on), while the `data server` manages the libraries.

- Second, a client is responsible for the integrity of his workspace. In contrast, the `data server` safeguards the integrity of the libraries by means of the OO-DBMS concurrency control, data-ownership and changes-notification (data control) mechanisms.[1]

- Third, a client has (write) access to his workspace data (for example UNIX files) at all times. In contrast, a client is subject to contention for accessing data in a library, through the `data server`.

- Fourth, a client is the clear owner of his workspace. On the other hand, multiple users or groups can own a library.

The deployment of libraries and workspaces, as well as the implementation of the class hierarchy of the libraries are illustrated in Figure 6.

[1]The data-ownership and changes-notification mechanisms are described in detail in Chapter 5.

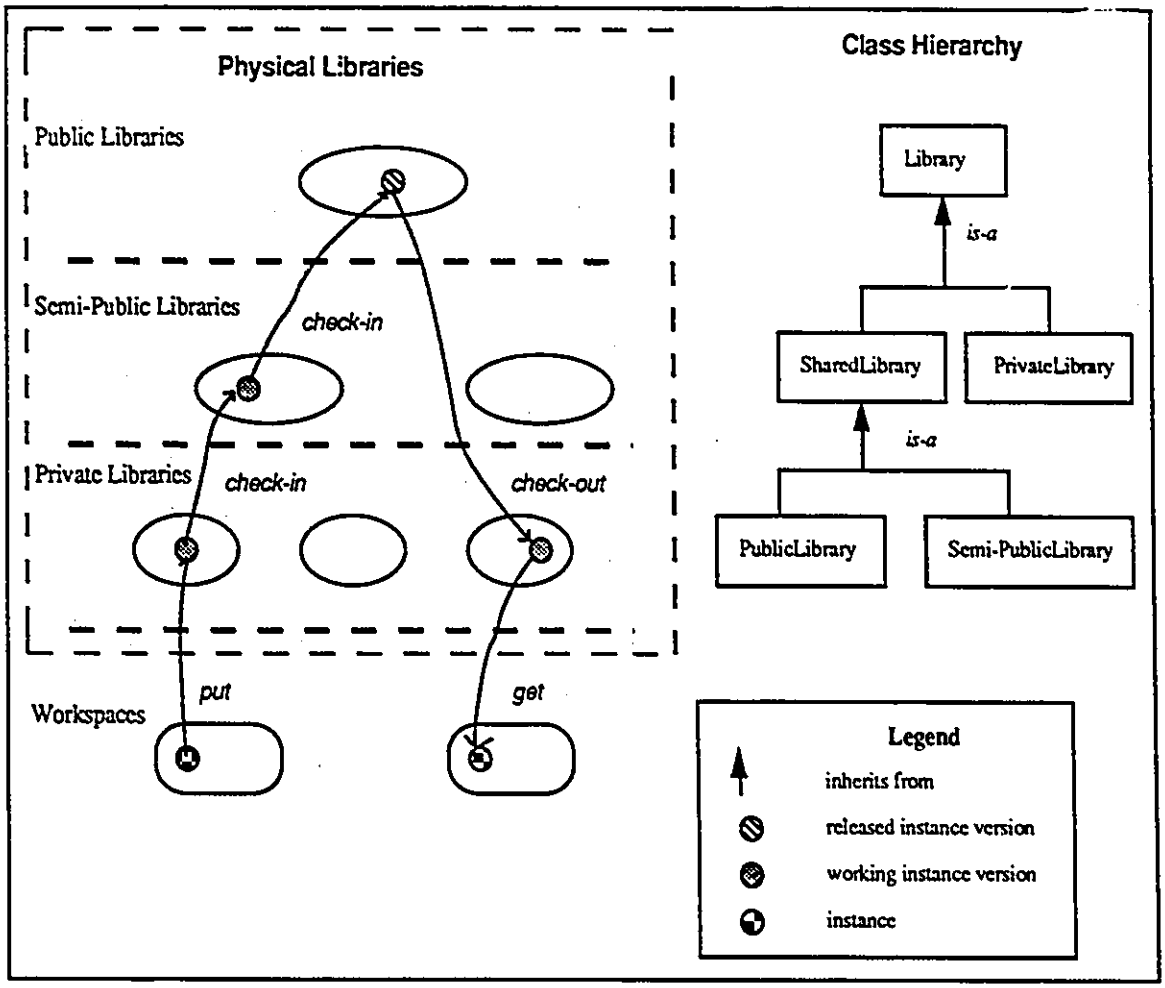


Figure 6 Layered Hierarchy of Libraries, and Library Class Hierarchy

The libraries are organized into a layered hierarchy. The top layer contains public libraries, the middle layer contains semi-public (or project) libraries, and the lower layer contains private libraries. The semantics associated with each of these libraries is:

- ***Public*** **Library:**

  - Data stored in a public library is by nature stable (has been released), and thus it can neither be deleted nor changed.

- Public libraries belong to specific engineering design environments (e.g. CAD, CAM, CASE, and so on).

- Instance versions[1] stored in this library are called *released* instance versions.

• ***Semi-public*** **Library:**

- Data stored in a semi-public library is by nature dynamic, since it is considered to be work in progress,[2] and can be deleted by authorized clients.

- A semi-public library is owned by a group or groups of clients.

- Instance versions placed in a semi-public library are called *working* instance versions.

• ***Private*** **Library:**

- Data stored in a private library is by nature dynamic (since it is considered to be work in progress[3]) and can be deleted only by its owner.

- A private library is owned by one client only. The owner of a private library is the clear owner of the data stored in that library.

- Instance versions placed in this layer are also called *working* instance versions.

The common characteristics of these three types of libraries are as follows: First, the access rights that a group has to a library are determined by an "Access-Rights" association between the group and the library.[4] Second, data contained in one library can be related to data contained in other libraries.

---

[1] Persistent snapshots (versions) of instances.

[2] The instance versions stored in a semi-public library may be either partially complete, or they may not have been simulated, or they may not have been tested, or all of the preceding.

[3] The instance versions stored in a semi-public library may be either partially complete, or they may not have been simulated, or they may not have been tested, or all of the preceding.

[4] This type of association is described in Chapter 5.

The layered hierarchy of libraries is associated with check-in, put, check-out, and get mechanisms which support the migration of data across libraries and workspaces. As shown in Figure 6, clients can read information either by checking-out or by getting data, and can write information either by checking-in or by putting data. A *check-out* operation installs a copy of an instance version from a library into a lower-level-library. A *get* operation reads an instance version from a library into a workspace.[1] A *check-in* operation installs a copy of an instance version from a library into a higher-level-library. A *put* operation writes an instance version from a workspace into a library.[2]

The implementation of the class hierarchy of the three library types is shown in Figure 6. The PublicLibrary and Semi-PublicLibrary classes specialize the behavior inherited from the SharedLibrary abstract class. This specialization is required due to the different semantics of these two types of libraries. The PrivateLibrary and SharedLibrary classes inherit the class definition from the Library class. The behavior of the Library class provides the methods needed to manipulate the structure of instances of the Library class (called libraries). The structure part of the Library class provides the organization of metadata required to track the evolution of data within the design-cycle. An abstract view of this structure is illustrated in Figure 7.

---

[1] The get operation also converts an instance version into an instance.

[2] The put operation also converts an instance into an instance version.

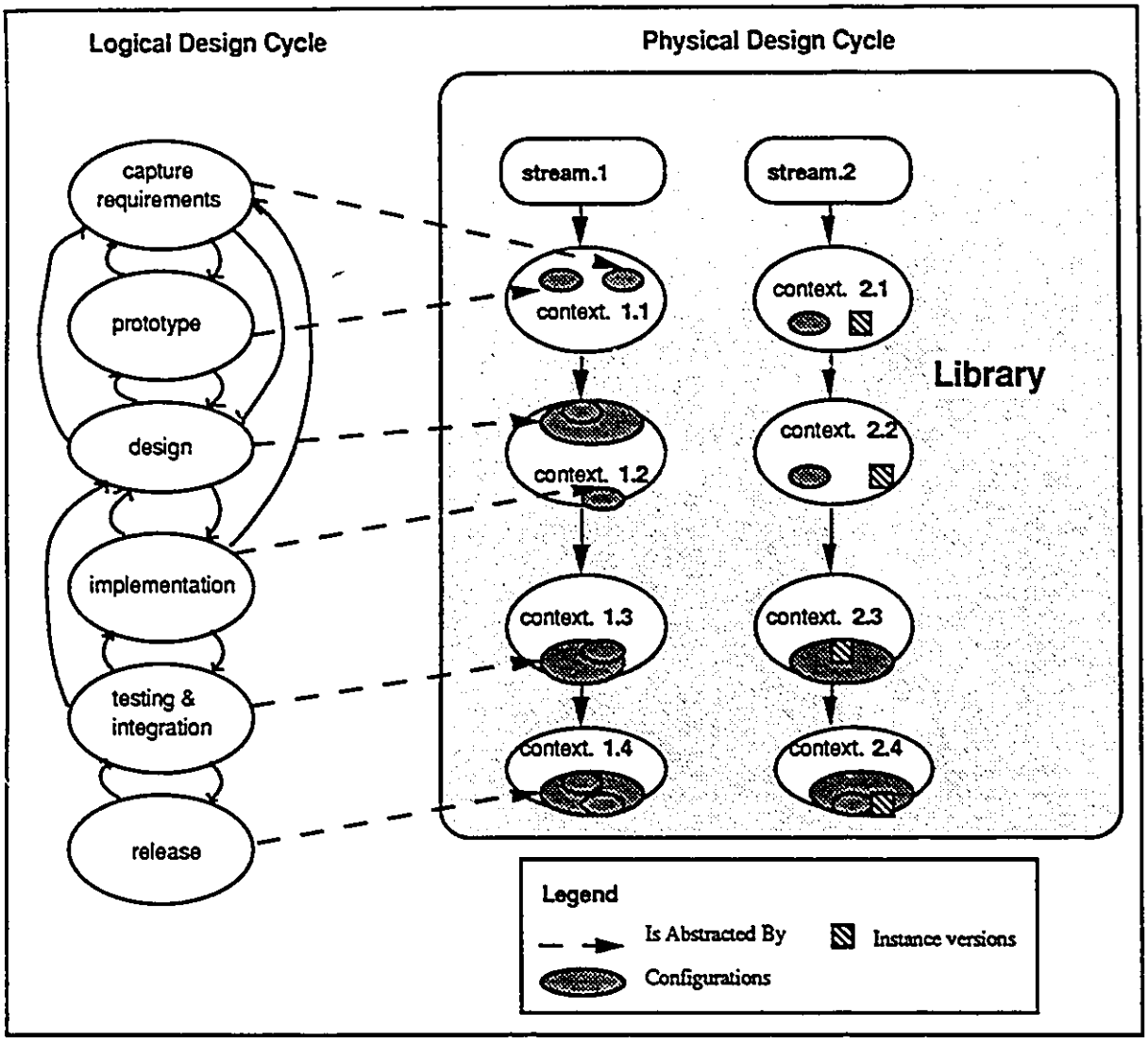


Figure 7 Library Structure, and Logical to Physical Mapping of the Life Cycle.

The structural part of the library stores the representations of data during the design life cycle. The library structure consists of a set of streams. Data generated in different stages of the logical design life cycle can be stored in a stream. Thus, a *stream* is a physical representation of a logical design life cycle. A library contains a set of streams; this provides the ability to support parallel design cycles. For example, the implementations of similar designs, or experiments with different technologies, and so on, can be captured within streams. A stream contains a set of contexts; and a

context is a physical representation of a set of logical design activities within the design life cycle. A context can contain either instance versions, or versions of configurations, or both. A *configuration* is a versionable container of instance versions, which have associations among them.[1] The structure part of the different components of a library and the relationships among them are shown in an Entity-Relationship (ER) diagram [Chen 76], illustrated in Figure 8.

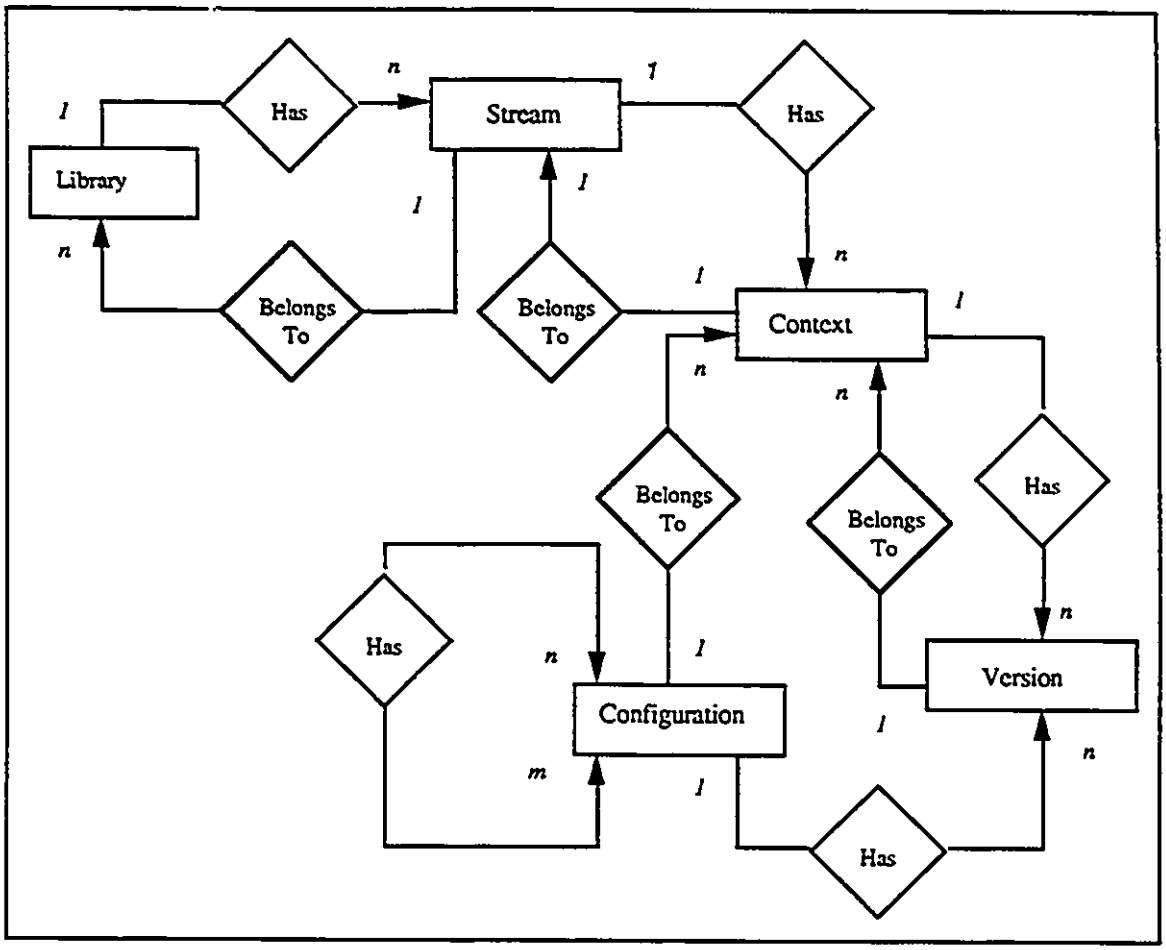


Figure 8 An ER Diagram for the Structure Part of a Library and its Components

[1] The class hierarchy used to implement a configuration is discussed in detail in Chapter 5.

The behavioral part associated with the Library class, provides the following capabilities. A client can create, add, delete, and query libraries; can add, delete, and query streams. Furthermore, a client can query and navigate library data and library metadata.

4.2.1.2 Versions

A *version* is a persistent snapshot of data. An *instance version* is a version of an instance of a class. A library-stream-context triplet has a set of instance versions and configurations. A context has a hash table of all the classes which have instance versions, and the root of a number of instance versions of a class can be obtained by hashing the class in the hash table. The class hierarchy for the Version class is shown in Figure 9.

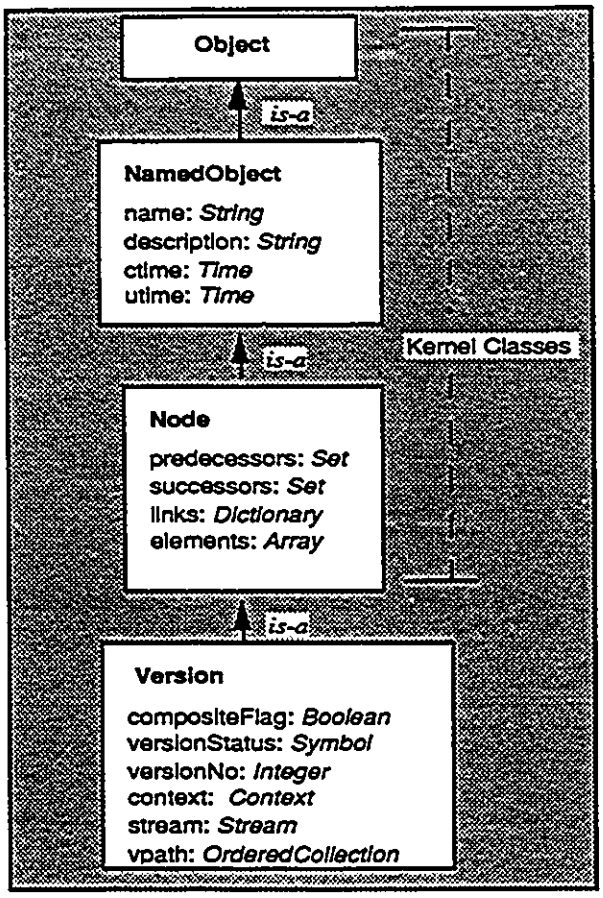


Figure 9 The Class Hierarchy of the Version Class

The class Version is a subclass of the class Node. An instance of a class Version has an attribute called "elements". This attribute contains a persistent snapshot of an instance of a class. An instance version of a class has a name which is the concatenation of the name of the class and an integer number (e.g. Microcontroller.4 is the fourth instance version for a Microcontroller class). The `data server` assigns new instance version names whenever an object is checked-in/put, or checked-out. Thus, in each library, instance versions are assigned monotonically by increasing integers according to the order of their creation. To gain access to an instance version, a client has to request a specific path (called the current path) through the `data server`. A *path* is a sequence of names composed of a library name, a stream name and a context name.[1] An instance of a class Version inherits from Node two attributes: predecessors and successors. The attribute predecessor is a set which contains predecessor associations to instance versions and the attribute successors is a set which contains successor associations to instance versions.[2] By means of both the predecessor and successor instance variables, the `data server` can support both linear and branching versioning schemes. These versioning schemes are shown in Figure 10.

---

1 A path remains in effect until it is explicitly changed by a client.

2 The successor and predecessor associations are described in detail in section 4.3.1.2. Both associations form a relationship.

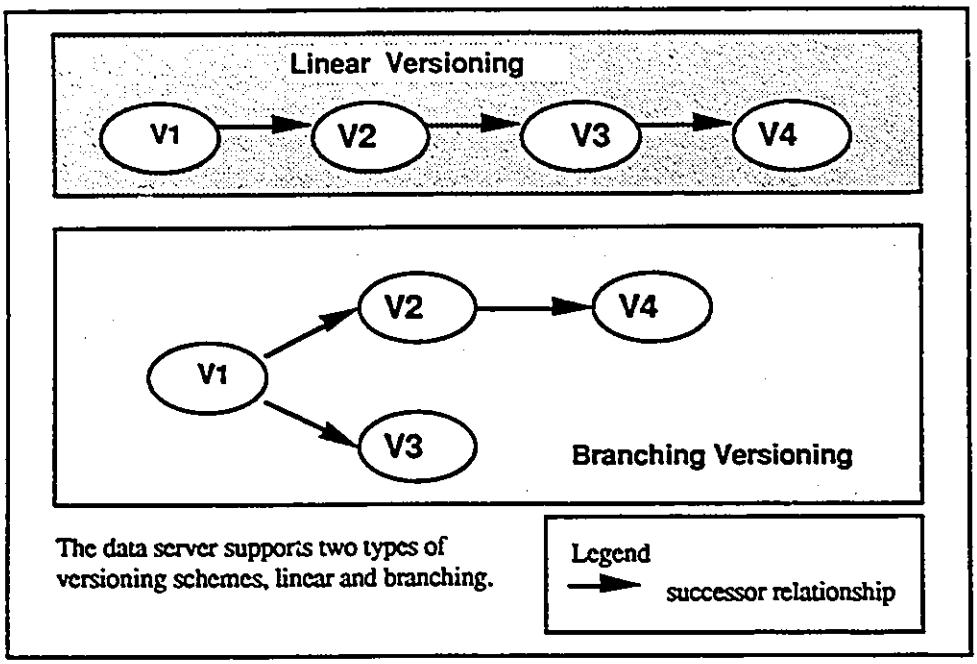


Figure 10 Linear and Branching Versioning.

In a *linear versioning* scheme, a version can have only one successor. In the example shown in the top part of Figure 10, V1 is the root of a linear version-derivation hierarchy.

V2 was created after V1, V3 was created after V2, and V4 was created after V3. In the bottom part of Figure 10 is illustrated a branching versioning scheme. In a *branching versioning* scheme, every version can have multiple successors (e.g. the root V1 has two successors, V2 and V3). The branching versioning scheme allows clients to experiment with alternative implementations of a design. This branching scheme leads to a version-derivation hierarchy which captures the evolution of designs. Although not shown in the figure, the `data server` adds a predecessor association for each successor to its parent. This allows easier navigation through the version-derivation-hierarchy. It is worth mentioning two characteristics of branching versioning. First, a successor can be considered to be derived from a parent version, and as such it can keep the same associations that the parent has with respect to other instance versions. Second, branching

provides the ability to indicate that an instance version may be the result of a manual merging of other instance versions.[1] Both branching characteristics are illustrated in Figure 11.

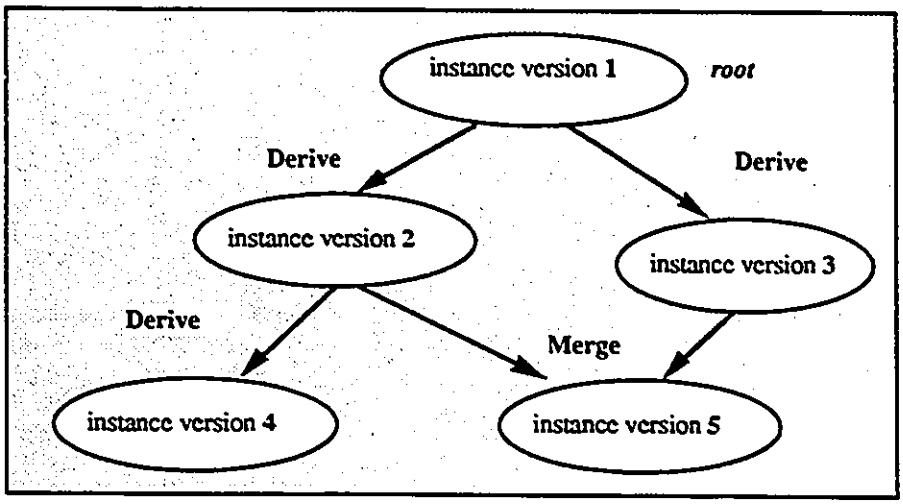


Figure 11 Branching Characteristics: Derivation and Merging

The behavior part of instance versions depends on their location in the library hierarchy. The basic set of operations that are allowed on working and released instance versions are:

- *Behavior of a Working Instance Version:*

  - can be deleted by the client who created it

  - can be created by a client or derived[2] from an existing working instance version

  - can be referenced by other instance versions[3]

  - can be frozen by its client (that is, it may not be deleted)

  - can become the parent of other instance versions

---

1 Automatic merging of versions is still an unsolved topic.

2 Through a check-out operation.

3 The data-ownership mechanism determines who can reference a version.

- can be checked-in and checked-out

• *Behavior of a Released Instance Version:*

- can be deleted only by a public library administrator[1]

- can be created either as a new version, or by a checking-in a working version

- can become the parent of other versions

- can be referenced by other versions

- associations with other instance versions can be automatically copied when a released instance version is derived

- can be checked-out

4.2.1.3 Composite-Configurations

It is frequently necessary to design an object by composing it from other objects. A *configuration* is a set of instance versions related by associations. A *composite-configuration* is a versionable container of either instance versions or configurations, or both. The contained elements are associated by an aggregation relationship.[2] Since instance variables, in the object-oriented model, can store objects, the instance variables are considered a kind of aggregation (*is-a-part)* relationship. Thus, objects can be referenced or bound through instance variables by two types of symbolic[3] bindings: static binding and dynamic binding. *Static Binding* is a configuration scheme by which the exact instance versions names that make up the composite-configuration are specified. *Dynamic Binding* is a configuration scheme which dynamically creates a composite-

---

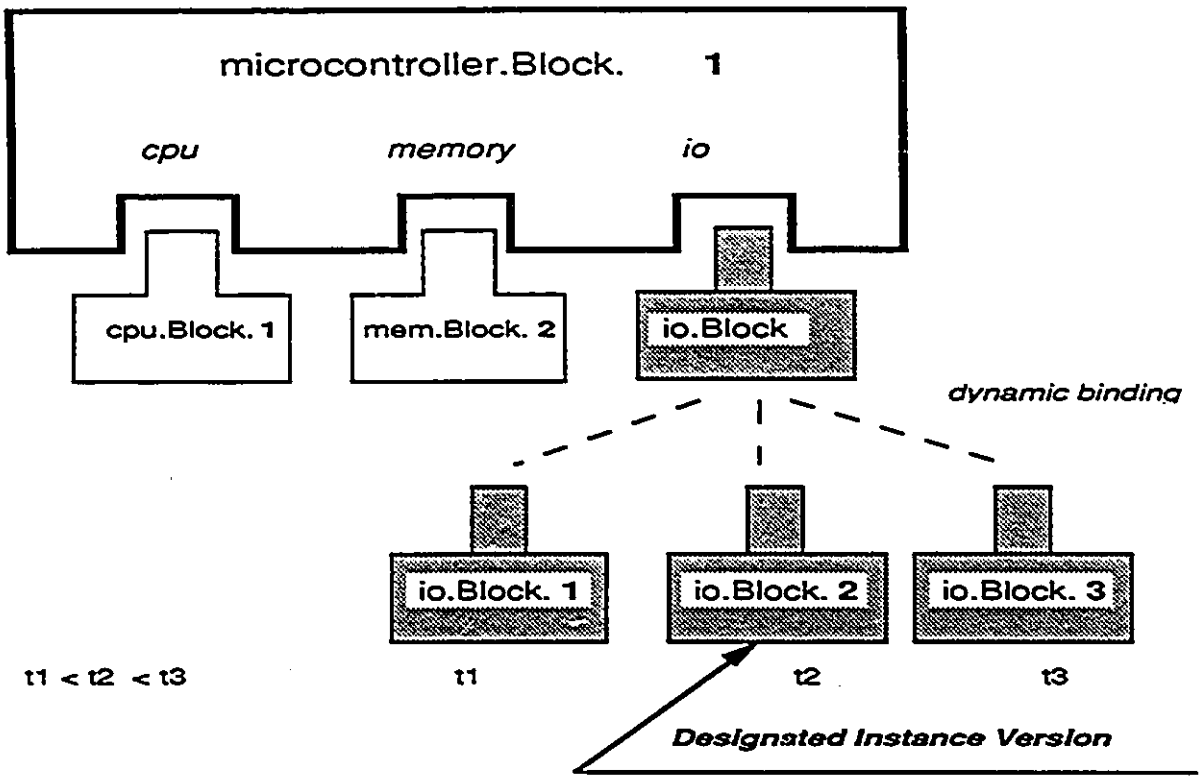
[1] This should be done only after a reasonable period of time, and after data is safely backed up.

[2] A composite configuration is, in general, a directed acyclic graph (DAG).

[3] Symbolic means that objects are referenced by a name, and at some point in time, the name is de-referenced to access the referenced object.

configuration either with the latest time-stamped instance version, or with a designated instance version name. A client can use the dynamic binding scheme when he requires that changes made to a part be reflected immediately throughout the composite configuration. A *designated instance version* is a name which specifies a particular instance version to be used in lieu of the latest instance version.[1] For a given *path*, the designated instance version can be set by a client through the `data server` interface. An example of the usage of these two bindings is illustrated in Figure 12.

[1] The latest instance version is used as the default instance version in a dynamic binding scheme.



The instance version **microcontroller.Block.1** is statically bound to instance versions: cpu.Block. **1** and mem.Block. **2**, and dynamically bound to the instance version **Io.Block** .

As shown, **Io.Block** has three instance versions, and io.Block. **2** has been defined as the designated instance version.

When the microcontroller instance version, which is the root of a composite configuration, is checked-in/checked-out, the reference to io.Block is resolved at run time to io.Block. **2**.

If io.Block. **2** is not defined as the designated instance version, then the latest instance version, io.Block. **3**, will be used to resolve (de-reference) the dynamic binding.

The latter is the default behavior.

Figure 12 Static and Dynamic Version Binding

An instance version of the class Microcontroller.Block, called microcontroller.Block.1 is composed of three instance versions: cpu.Block.1, mem.Block.2, and **io.Block** (called a *generic instance version*). The three instance versions are stored in the instance variables of the instance version microcontroller.Block.1.[1] Only the generic instance version **io.Block** does not have any explicit version identifier. Therefore, microcontroller.Block.1 could be bound at run time to any of the io.Block instance versions (such as io.Block.1, io.Block.2, or io.Block.3). However, when a designated instance version is set by a client,[2] then that is the one used to resolve (de-reference) a dynamic binding. The instance version io.Block.2 is shown as being set as the designated instance version. Thus, the instance version microcontroller.Block.1 will be bound to the instance version io.Block.2 at run time.

4.2.2 Operational Aspects

The operational aspects are the aspects related to the behavior part of instance versions. The behavior associated with the check-in and check-out mechanisms depends on a set of options that a client can use. These options are:

Search-Order-Path:

A *search-order-path* is a sequence of path names that determine the order in which components, referenced by name via a composite-configuration, are either read or written across libraries. The default search order for components is bottom-up according to the library hierarchy. The search starts where the root of the composite configuration is stored. That is, the components are searched for first in private libraries, next in semi-public libraries, and last in public libraries. This default ordering can be changed by a client, if he explicitly provides a specific order of paths.

---

[1] The class definition for the Microcontroller.Block has three instance variables: cpu, memory and io. The term Block is used here, to indicate that it is a block view of the microcontroller design.
[2] A designated instance version is valid for a given path.

Transitive Closure:

The *transitive closure* of a composite-configuration is a set of all contained instance versions. When the transitive closure for a composite configuration is checked-in (or checked-out), the instance version names of the related objects are converted to new static references that have meaning in the destination library. All the dynamic references are converted to static references, and then the transitive closure for the specified configuration is written (or read).

N-levels:

The *N-levels* option checks-in (check-out) n levels of a composite configuration from one library to another library. In addition, the checking-in and checking-out of n levels of a composite configuration integrates parts stored in different libraries. For example, assume that there exists a composite configuration microcontroller.1 with three components (cpu.1, memory.1, and io.3) in a private library, as shown in Figure 13. In the public library there is a complete composite-configuration microcontroller.1, and a composite-configuration io.3. Then, at time **t1**, the instance version microcontroller.1, and its parts: cpu.1, memory.1, and io.3, are checked-in from the private library into the public library. This check-in creates a new instance version microcontroller.2 which will use the composite-configurations: cpu.1, memory.1, and io.3 stored in the public library.

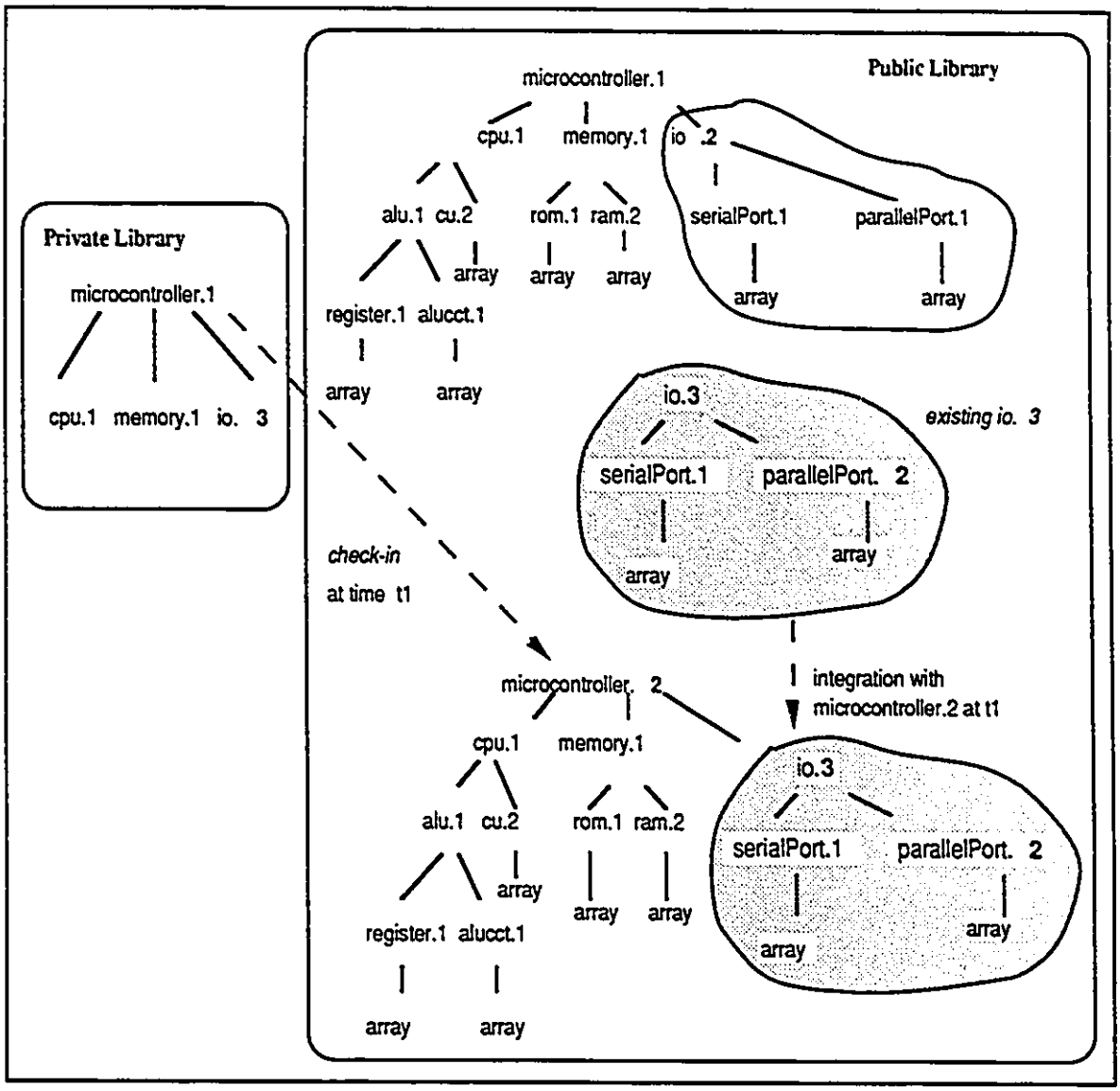


Figure 13 Checking-in of N-levels of a Composite Configuration

Data Integrity:

This option forces the checking of the followin integrities:

- *Referential integrity*: All referenced versions must exist.
- *Circular references*: A version can not have as a part its same version.

- *Constraints on parts*: The types of parts of a version must agree with the specified types in the class definition.

Parent and Virtual Path:

When a client checks-out or checks-in an instance version x, there are two options :

- either specify the parent for x in the destination library, or
- rely on the `data server` (default mechanism) to determine the parent for x.

Figure 14 illustrates an abstract view of the mechanism used by the `data server` to determine the parent of a version. This default mechanism relies on a *virtual path,* which is an ordered collection of paths. Each instance version has an attribute which stores a virtual path. As shown, the parent for the instance version v2 in a public library can be determined by navigating a virtual path. For example, the paths to be navigated to find the parent of v2 are d, c and b.

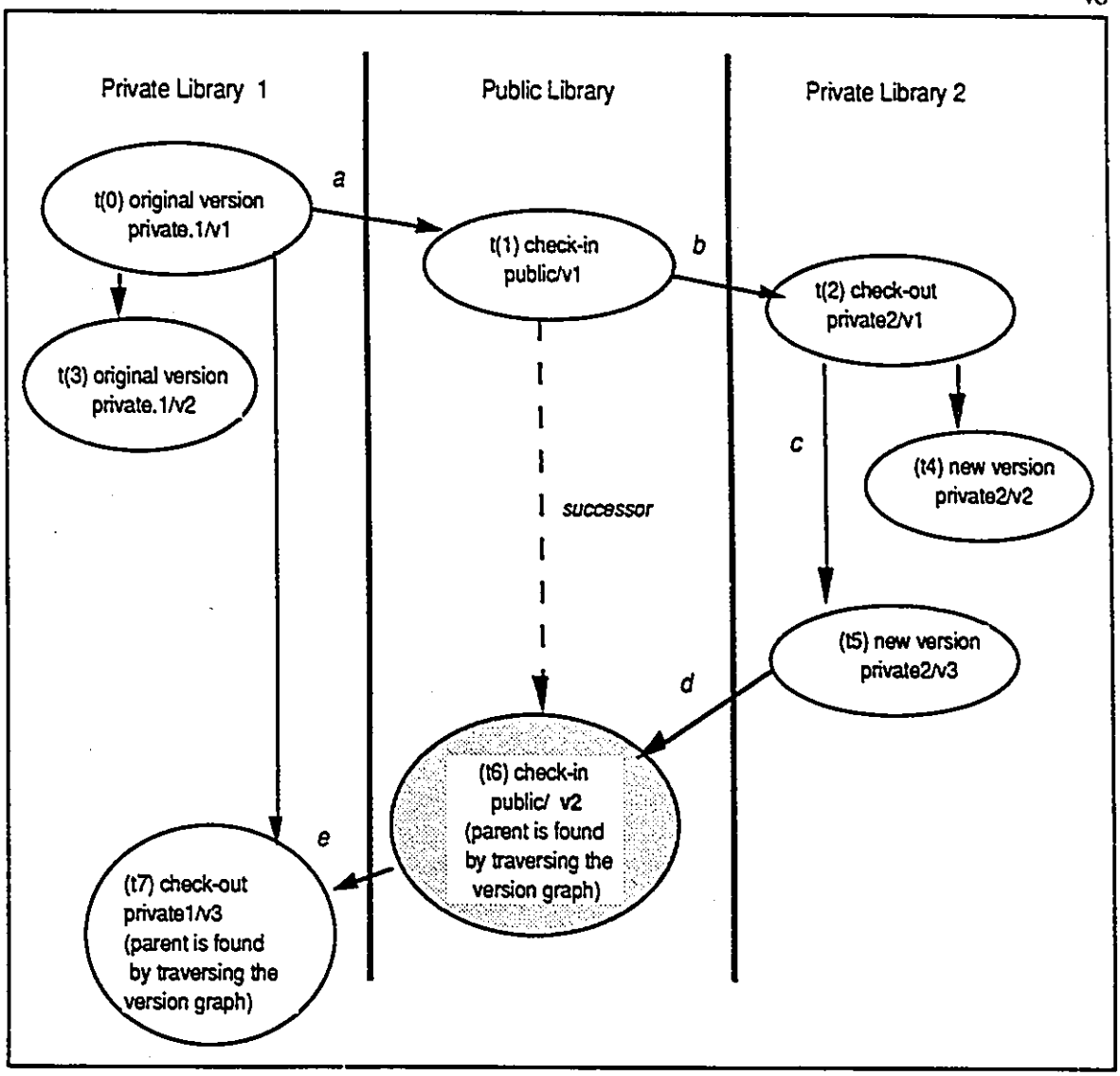


Figure 14 An Abstract Representation of a Virtual Path

Next, we describe the operational aspects related to the get and put operations. A client has the ability to integrate an instance with references to existing instance versions in his workspace, and then *put* the configured instance into a private library.[1] The *get* operation is the converse of the put operation. By means of a get operation, a client can obtain an instance for a specific instance version (stored in a private library) into his workspace.

[1] A client must commit the transaction to commit the objects.

4.2.3 Schema Evolution

Schema evolution is the other aspect of the versioning model which is related to the management of changes performed on the datastore schema. These changes require:

- creation of a class
- deletion of a class
- alteration of inheritance relationship between classes
- addition of instance variables and methods
- deletion of instance variables and methods

Existing object-oriented database systems support only a few changes to the schema, and lack support for management of schema evolution. In this work, to simplify the schema-evolution problem, we have grouped possible changes[1] to a schema into three categories:

- changes to the class schema[2]
- changes to the class hierarchy made by changing the relationships between classes
- addition or deletion of classes in the class hierarchy

Any of the above changes forces the creation of a new version for the class-schema-hierarchy. A *class-schema-hierarchy* is an organization of classes related by the inheritance relationship. A version of a class-schema-hierarchy is a persistent snapshot of the class-schema-hierarchy. A class-schema-hierarchy version has an identifier which consists of a name followed by an integer. This integer number is monotonically incremented when the checking-in of changes performed to

---

[1] These changes affect existing instance versions. Thus, "old" instance versions of a class may have to co-exist with instance versions created under a new version of a class-schema-hierarchy.
[2] The class schema of a class provides the class definition.

the schema succeeds. The constraints associated with the management of class-schema-hierarchy versions are as follows: First, relationships between instance versions that belong to different class-schema-hierarchy versions are not allowed. Second, if working instance versions are checked-in to a library, then either the class-schema-hierarchy version must exist in the library, or the checking-in of the new class-schema-hierarchy must precede the checking-in of the instance versions.

We have adopted a class-schema-hierarchy versioning scheme instead of a class-schema versioning scheme, since the former versioning scheme has the following advantages:

- A version of a class-schema-hierarchy represents a complete and meaningful working schema. Thus, there is no need to manage multiple versions of class-schemas (class definitions).

- A context can be associated with only one version of a class-schema-hierarchy instead of with multiple class-schema versions. Thus, the management of schema evolution is simpler.

4.3 The Configuration Management Model

This section describes the basic components used in the design of the configuration management model and the related operational aspects. The building blocks of the configuration management model, based on the data management requirements defined in Chapter 3, are:

- Object and Link, kernel classes

- associations (linkage of two objects)

- relationships (bidirectional associations)

- configurations (set of related instance versions)

The Object and Link classes have been described in Chapter 3. A detailed description of the other building blocks is given below.

4.3.1 Associations

An association links two objects: a source object and a destination object. Conceptually speaking, an association provides a linkage between two objects for a specific purpose. An association is represented by means of an instance of class Link, which implements the link construct. To provide a pre-defined set of associations, we have reviewed the types of associations most commonly used in CAD (including CASE and CAM) and published in related literature, specifically the works of Katz [Katz 90] and McLeod [McLeod 90]. The set of predefined associations is classified into unidirectional associations and bidirectional associations (called relationships).

4.3.1.1 Unidirectional Associations

A unidirectional association links in unidirectional way a source object with a destination object. A set of predefined unidirectional associations is given below in alphabetical order.

Access-Rights:

The *Access-Rights* association determines the access rights (read, write, release) that the source object has with respect to the destination object. This association is one of the building blocks of the *data-ownership* model described in Chapter 5.

DependsOn:

The *DependsOn* association specifies a source object as a dependent of a destination object. Thus, this association provides a mechanism by which updates performed on the destination

object trigger a set of actions on the dependent object (according to the methods implemented in the dependent, which are invoked when the destination object is changed). For example, this association can be used when a client is working "simultaneously" with two views of a design, and he requires that changes produced in one view be reflected immediately in the other one.

Existence:

The *Existence* association asserts that the destination object exist only by virtue of its being a component of the source object. This association can be used, for example, to delete unused components of a design when a main user object is deleted.

IsAssociatedWith:

The *IsAssociatedWith* association specifies that the source and destination objects are associated or related. For example, this association can be used as a way of grouping objects that are logically related.

Stability:

The *Stability* association specifies that the destination object can not be deleted. For example, this association can be used as a predecessor relation between a version and its parent, to avoid the deletion of the parent version.

4.3.1.2 Relationships

A *relationship* is a bidirectional association between two objects, and it can be represented as a pair of mutually inverse links: a forward link and a back link. The orientation of the *forward link* is from the source object to the destination object; the orientation of the *back link* is the inverse of the forward link. In contrast with the Relational model, the relationships supported by the extended

object-oriented model capture not only the structure, but also the behavioral part which is not supported in the Relational model. The set of predefined relationships is given below.

Has/Belongs:

The *Has/Belongs* relationship consists of two associations: *Has* and *Belongs*. The *Has* association works for the source object to indicate that it has the destination object as a component. The *Belongs* association is the corresponding back link. Usually this relationship is defined on objects in order to navigate back and forth through related objects easily in both directions. Another reason for having these two associations as a pair is to aid in deleting the *Has* association when the destination object is deleted (e.g. the *Belongs* association is navigated, and then the *Has* association is deleted). The *Has/Belongs* relationship can be used to integrate components of a composite object.

Monitor/MonitoredBy:

The *Monitor/MonitoredBy* relationship consists of two associations: *Monitor* and *MonitoredBy*. The *Monitor* association indicates that the destination object is monitored for a set of events by the source object. Thus, this association provides the mechanisms by which the occurrence of events (that match the set of specified events) causes the monitored object to send notification messages to the source objects. The notification message can trigger a set of actions on the source object. The *MonitoredBy* association is a back link which is used to determine the source object to be notified for the occurrence of an event on the monitored object. The *Monitor/MonitoredBy* relationship is used in the *changes-notification* model described in Chapter 5.

Owns/IsOwnedBy:

The *Owns/IsOwnedBy* relationship consists of two associations: the *Owns* and the *IsOwnedBy* association. The *Owns* association specifies that the source object is the owner of the

destination object. The *IsOwnedBy* is the corresponding back link. This relationship is one of the building blocks of the *data-ownership* model described in Chapter 5.

Predecessor/Successor:

The *Predecessor/Successor* relationship consists of two associations: *Predecessor and Successor*. The *Predecessor* association indicates that the destination object is the predecessor of the source object. The *Successor* is the corresponding back link. The *Predecessor/Successor* relationship is used in the versioning model for tracking the evolution of data.

Uses/UsedBy:

The *Uses/UsedBy* relationship consists of two associations: *Uses* and *UsedBy*. The *Uses* association indicates that the source object uses the destination object. The *Used-By* association is a back link. For example, this relationship can be used to indicate an inclusion of a number of documents within a main document.

Besides the above relationships, three other relationships are automatically supported by the object-oriented model; these are: the *inheritance* (is-a), *aggregation* (is-a-part), and *specialization* relationships.

It is worth mentioning that the Link Class is used to represent the (unidirectional) associations and relationships. The attribute called *property* is used to store the type of association (e.g. *Existence, Has,* and so on).

4.3.2 Configurations

Configurations are building blocks of the configuration management service. A *configuration* is a set of instance versions related either by associations, by relationships, or both. A configuration

can also be nested, i.e. a configuration can contain other configurations. For example, Figure 15 shows a composite-configuration of a microcontroller block view and a configuration microcontroller state view. Both configurations are nested within a configuration.

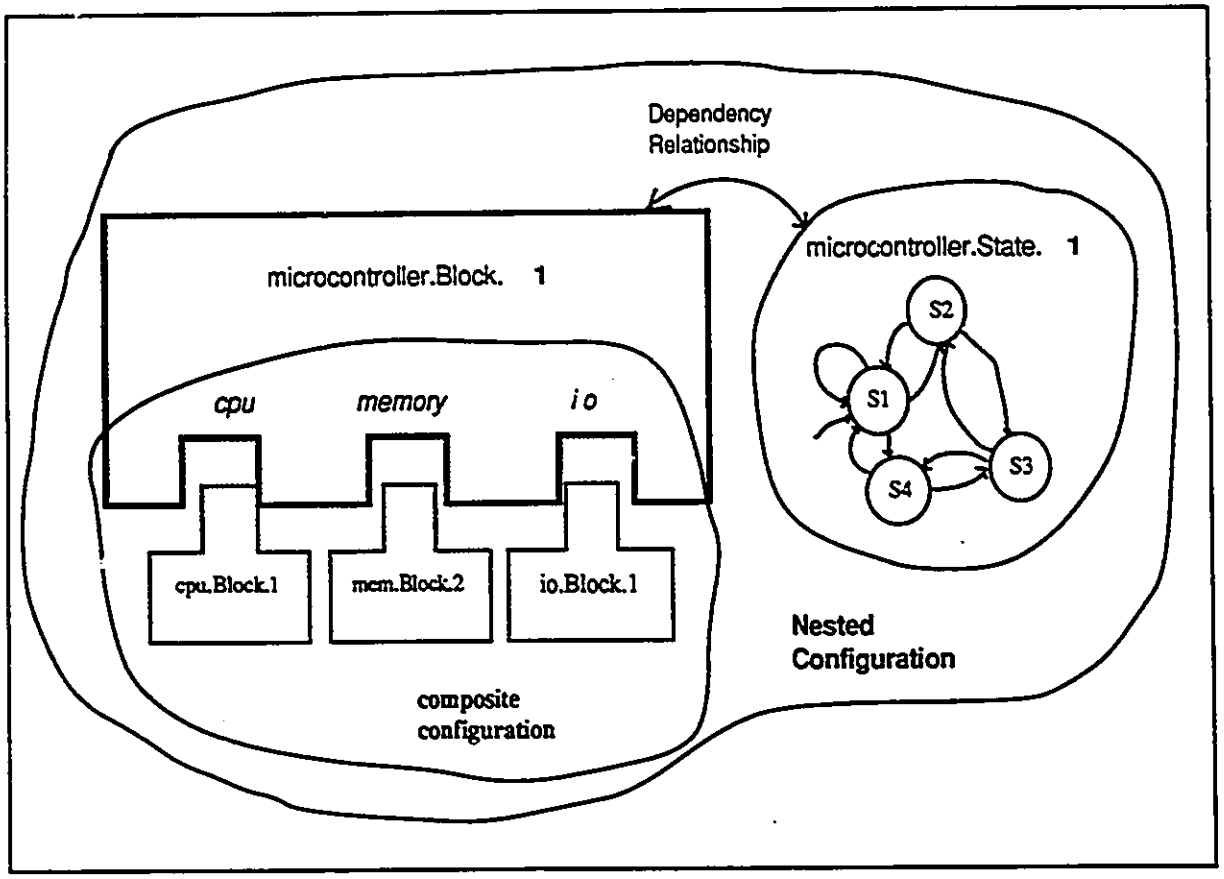


Figure 15 Two-Level Configuration: An Example of Nested Configuration

The support for nested configuration provides the ability to encapsulate related objects closely together, as this is the usual case when a client needs to represent a design by means of multiple views. Furthermore, it provides the ability to simplify the management of and navigation among related objects, since the whole configuration container can be considered as a unit of design work. We have implemented a configuration by means of a Configuration class. The class hierarchy of the class Configuration is shown in Figure 16.

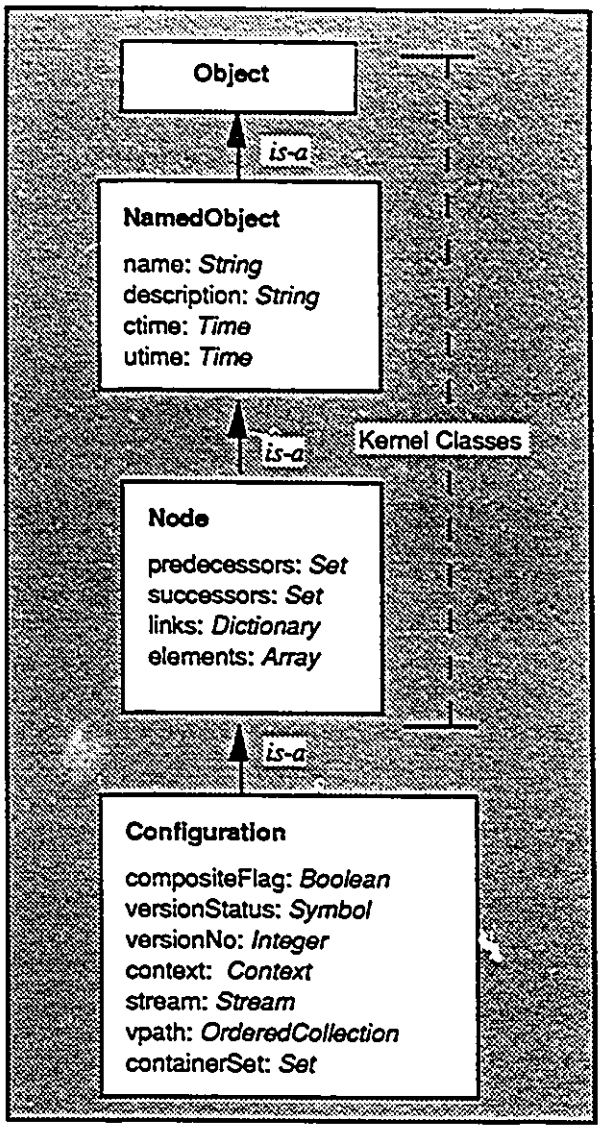


Figure 16 The Class Hierarchy for the Configuration Class

The Configuration class is a subclass of the class Node. Thus, it can have instance versions as elements, can contain links, and can contain nested configurations. Nested configurations are stored in the instance variable called "containerSet". The instance variable *containerSet* is of type Set.[1] An instance of the class Configuration can contain nodes, links, and containers.

[1]The class Set represents an unordered collection of elements that are not duplicated [Goldberg 83].

4.3.3 Operational Aspects

The operational aspects of the configuration management service are related to the basic operations supported by the behavioral part of this service. The basic set of operations supported by this service is as follows: An object can be either added to a configuration or deleted from a configuration. When an object is deleted from a configuration, all the associations and relationships to the object to be deleted are deleted. An object can be also moved across configurations; as a result associations or relationships contained in a configuration can expand across configurations. A configuration can be checked-in to a library and checked-out from a library. Thus, either all the components or n levels of a configuration can be checked-in and checked-out as an atomic unit. If a configuration is locked, then all its components are locked. As with other kernel classes, an instance of a Configuration class can be created and deleted. If a configuration is deleted, then all its members and the corresponding associations are deleted. Since a Configuration is a subclass of Node, it can be also versioned.

Chapter 5

The Design of the Data Control Services

5.1 Introduction

The purpose of this chapter is twofold. First, it provides the design for the data-ownership service. Second, it discusses the design of the changes-notification service. The *data-ownership* service provides the ability to control who is responsible for the release of data, and who can have access to the data. The *changes-notification* service provides the ability to notify affected parties that a specific event occurred on an object being monitored. *Data-ownership* is a traditional data control; it can be considered as a static type of control, since there is up-front knowledge about the access rights that groups/clients have with respect to data. On the other hand, the *changes-notification* can be considered as a dynamic type of control, since it is only after the occurrence of an event that the triggering of actions on the notified objects can be controlled by means of notification messages.

The first part of this chapter describes the data-ownership model, while the second part presents the changes-notification model.

5.2 The Data-Ownership Model

The data-ownership model consists of a set of building blocks and a set of mechanisms. These are described below.

5.2.1 Building Blocks

In addition to the Kernel classes (Node, Link, NonVersionableContainer, described in Chapter 3) the data-ownership model is based on the following building blocks:

Scope

A *scope* is a non-versioned container, which contains configuration names, instance version names, generic instance version names,[1] or some or all of the preceding. Scopes are associated with groups (set of clients) via access-rights associations.

Prime

A *prime* is a privileged client who is responsible for the release (checking-in) of data to a public library, and is the owner of the data referenced by a scope. A prime performs the following tasks:

- determining the contents of a scope
- specifying which objects within a scope can be used by any other object
- specifying who is allowed to read, to write, and release data referenced in a scope
- negotiating a release policy[2] with team members

Group

A *group* is a non-versioned container of clients. A group can contain other groups. Members of a group share the access rights of the *access-rights* association defined between a group and a scope. The same group can have different access rights on a number of scopes.

Access-Rights Association

The *Access-Rights Association* determines the access rights that a group has with respect to the components of a scope. A group can read (check-out) the components of a scope, can write (update) the component of a scope, can release (check-in) to a public library the components of

---

[1] To de-reference generic instance names, the rules specified for dynamic version binding are used.

[2] A release policy is a set of rules that govern the release to a public library of objects referenced in a scope.

a scope, and can be restricted from doing any of the preceding. The relationships among the building blocks used in the data-ownership model are illustrated via an ER diagram in Figure 17.

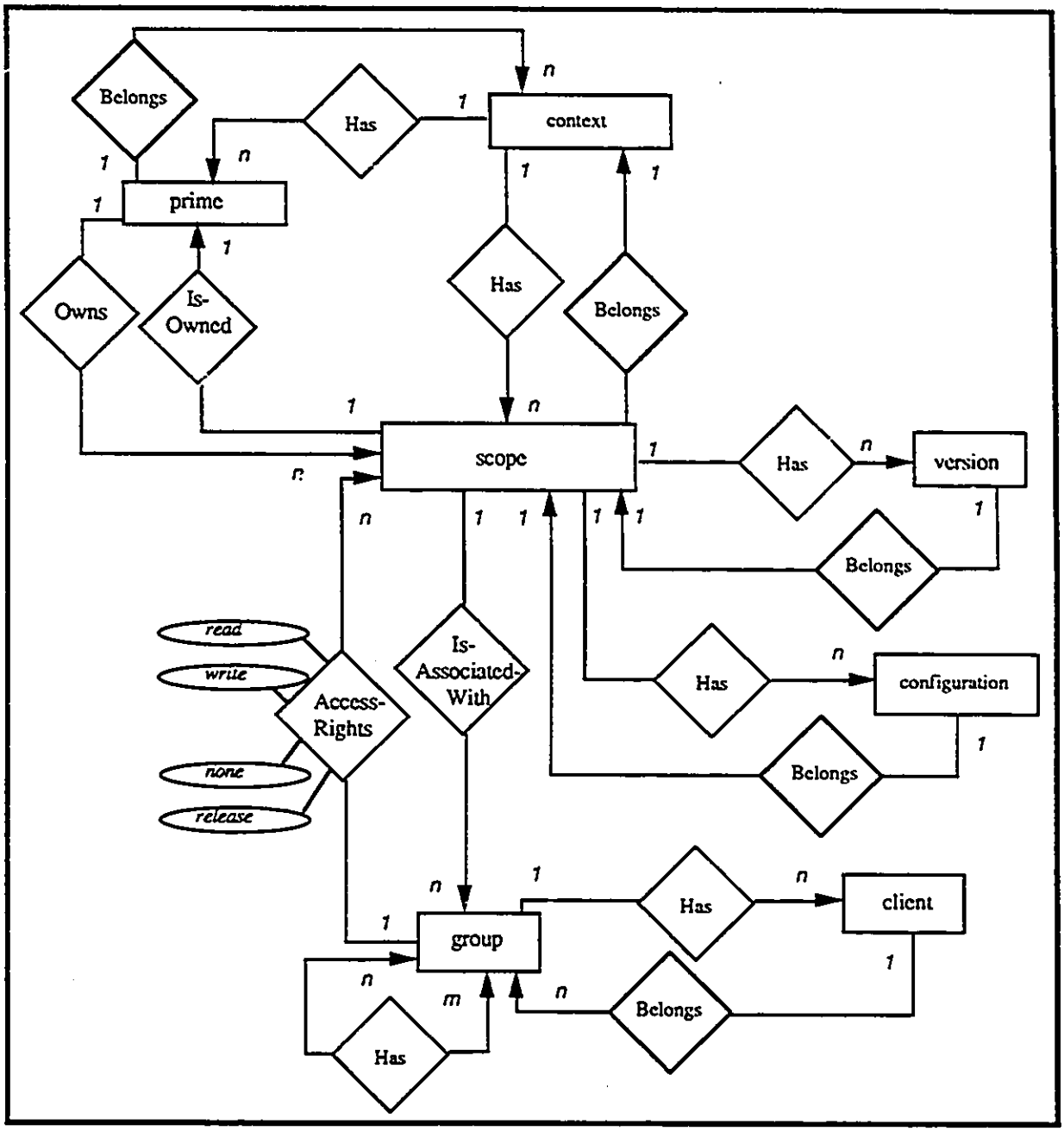


Figure 17 An ER Diagram for the Data-Ownership Model

5.2.2 Data-Ownership Mechanisms

We will describe how the data-ownership mechanism works by a walking through two examples written in pseudo-code, which are related to Figure 18.

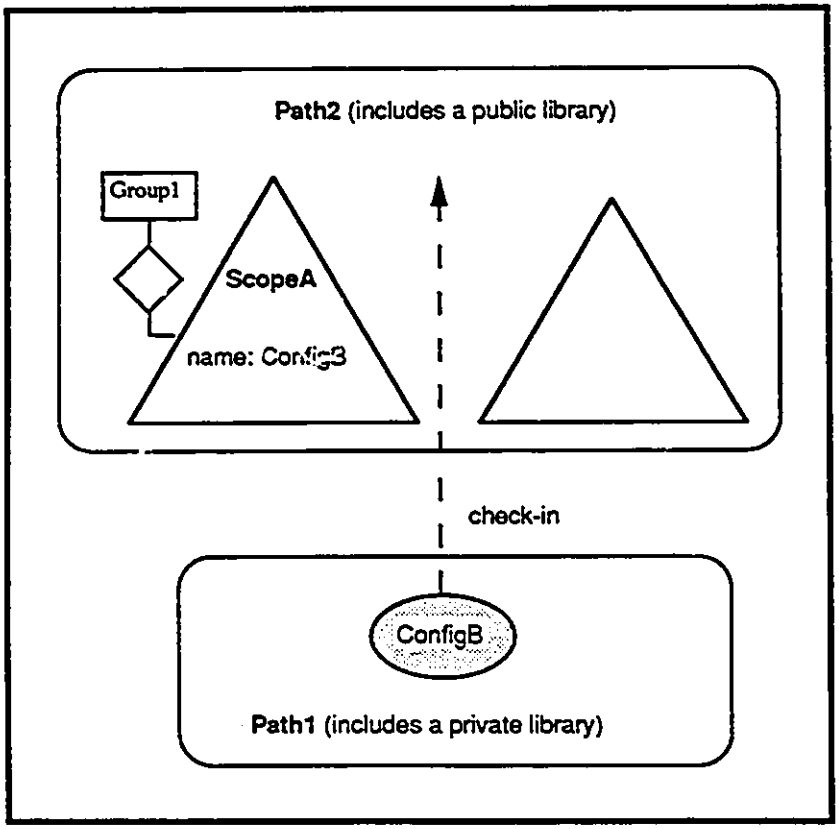


Figure 18 Checking-in Data between Paths

Example I

The below example explains how a prime of a scope could check-in a configuration from one library to a higher-level library, and the types of integrity checks performed.

Suppose that a designer called X is the prime of a scope **ScopeA** in a path **path2**. He wants to release a configuration **ConfigB** which belongs to scope **ScopeA**, from **path1** to **path2**.

The path called **path2** includes a public library. Then, X sends the following message to the `data server` interface:

> aDataServer checkIn: **ConfigB** inScope: **ScopeA** from: **path1** to: **path2**

This message will trigger the following sequence of actions:

1 The existence of **path1** is checked. If it exists then continue, else an appropriate error message is returned.

2 The existence of **ConfigB** in **path1** is checked. If it exists then continue, else an appropriate error message is returned.

3 The existence of **path2** is checked. If it exists then continue, else an appropriate error message is returned.

4 The name **X** is searched for in the dictionary of primes for **path2**. If found then continue, else... *(see Example II*, where we consider the case in which **X** is not the prime).

5 All the scopes owned by **X** in **path2** are searched, until the given scope **ScopeA** is found, or the end of the set is reached. If **ScopeA** is found then continue, else.an appropriate error message is returned.

6 The existence of the name **ConfigB** in **ScopeA** is checked. If found then continue, else an appropriate error message is returned.

7 **ConfigB** from **path1** is checked-in to **path2**. If the transaction commits then "succeed" is returned, else an appropriate error message is returned.

Example II

The example below explains how a member of a group which has access rights over a scope could check-in a configuration from one library to a higher-level library, and the types of integrity checks performed.

Suppose that **X** is not the prime of a scope called **ScopeA**, however he belongs to a group **Group1**, which has release access rights with respect to the **ScopeA**. **X** would like to release a configuration called **ConfigB** which is contained in **ScopeA**, from a path identified as **path1** to a path identified as **path2**. Then, **X** executes the same command as in *Example I*. That message will trigger the following sequence of actions:

1 The existence of **path1** is checked. If it exists then continue, else an appropriate error message is returned.

2 The existence of **ConfigB** in **path1** is checked. If it exists then continue, else an appropriate error message is returned.

3 The existence of **path2** is checked. If it exists then continue, else an appropriate error message is returned.

4 The name **X** is searched for in the dictionary of primes for **path2**. If found then continue as in Example I, else...

(here is where we made a reference to this example in *Example I*. We will now show the sequence of actions for the "else" part.)

...else

5 All the scopes which belong to **path2** are searched, until the given scope **ScopeA** is found or the end of the set is reached. If **ScopeA** is found then continue, else an appropriate error message is returned.

6 The name **X** is searched for within the set of groups which are associated with **ScopeA**, and that have write access rights. If **X** is found then continue, else an appropriate error message is returned.

7 **ConfigB** from **path1** is checked-in to **path2.** If the transaction commits then "succeed" is returned, else an appropriate error message is returned.

It is worth noting that the activity of releasing data to a public library is based on a release policy which both the prime and group members have agreed on. A *release policy* is a set of rules that govern the release to a public library of those components referenced by a scope. This release policy is similar to a "business" contract. Thus, both parties (prime and groups) must comply with it. We distinguish two basic release policies.[1] The semantics of these release policies are described below:

- *Prime-Approval-Policy*: Group members can release data referenced by a scope only if they have the explicit approval of the prime for the scope.

- *Non-Prime-Approval-Policy*: Group members are authorized to release data referenced by a scope without having to request the approval of the prime for the scope. However, the prime is notified about what is released, and if parallel versions of data are created, he is responsible for the eventual merging of them.

[1] The following policies represent a minimum set. Other policies can be added if required.

5.2.2.1 Data-Ownership Granularity

As stated before, a scope is a non-versioned container, which contains configuration names, instance version names, generic instance version names,[1] or some or all of the preceding. The names contained in a scope have two characteristics: they are unique, and they can belong to a number of scopes owned by different clients (or primes). For example, Figure 19 illustrates a composite-configuration with parts owned by two clients.

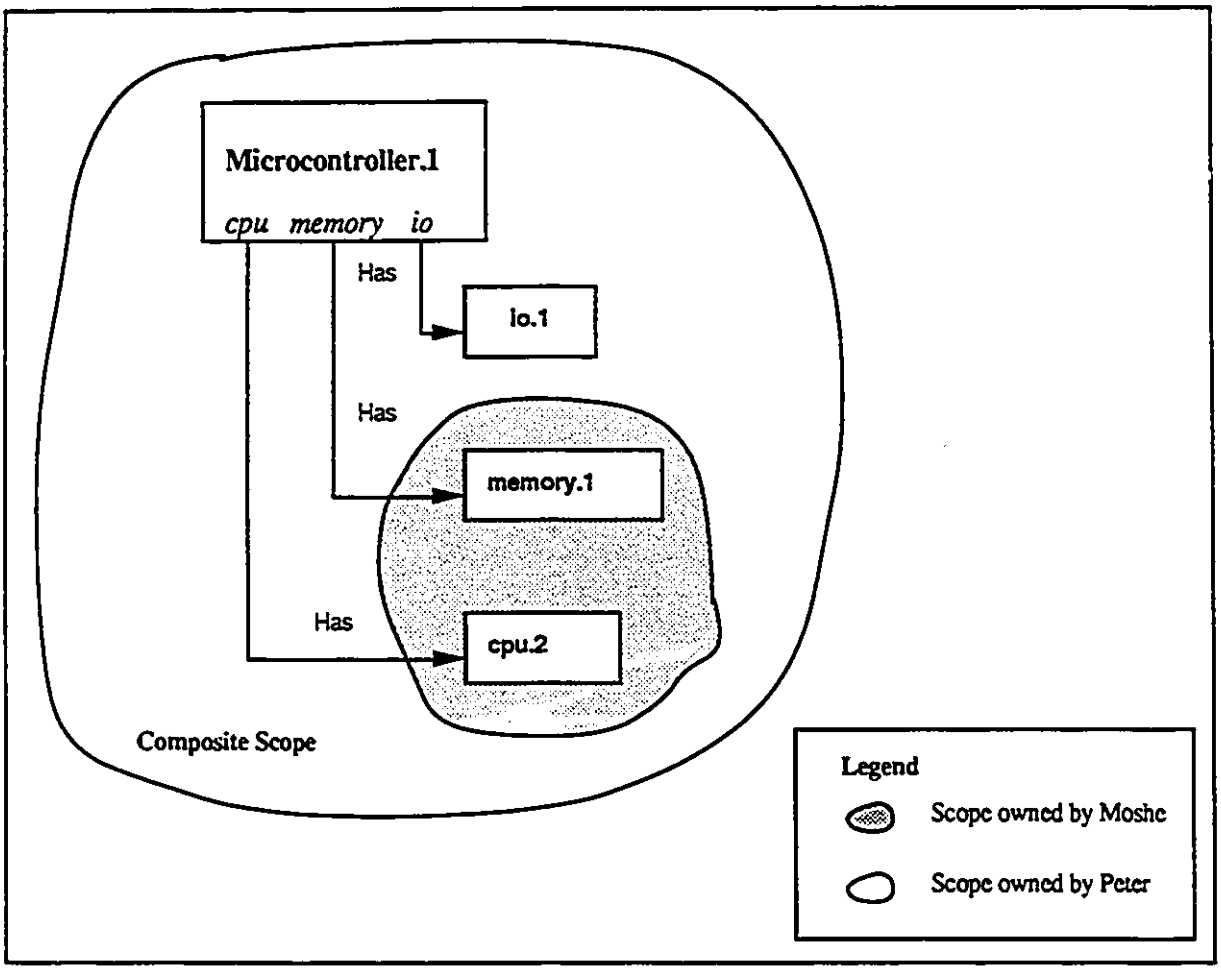


Figure 19 A Two-Levels: Composite Scope

[1] To de-reference generic instance names, the rules specified for dynamic version binding are used.

Peter owns a composite scope which contains the name *Microcontroller.1* which identifies a composite-configuration. The microcontroller *Microcontroller.1* contains the slots where the memory, io, and cpu parts are plugged in. The microcontroller components cpu.2 and memory.1 belong to a scope whose prime is Moshe.

As a rule, the prime of a composite scope can perform changes, and release to the public library the components whose names belong to the owned scope. The changed components are up-versioned when they are released. Thus, in the example, Peter can upgrade any of the components of his scope and release them to the public library. On the other hand, Moshe can change and release to the public library only the components (memory.1 and cpu.2) which are referenced in his scope. This example illustrates the correspondence between the data-ownership hierarchy and the composite-configuration hierarchy. In the example, Peter is considered a first level scope-owner, while Moshe is considered a second level scope-owner. Moshe can change and release only two components (memory.1 and cpu.2) of the composite configuration, while Peter can perform any of the following actions:

- check-out the parts io.1, memory.1, and cpu.2 from the public library into his private library
- create an instance of the Microcontroller class
- change and configure the parts in his private workspace, and put the configuration into his private library
- release (check-in) to the public library the new version of the Microcontroller configuration (including the two components memory.1 and cpu.2)

It is worth noting that the owner of a scope can release either all the components referenced in the scope, or n levels of the scope.

5.2 3 Operational Aspects

The operational aspects are related to the behavioral part of this service. The basic set of operations supported are as follows. A client who defines a class within a path, via the `data server`, becomes the owner of that class. A scope for that class is automatically created for the path. Thus, new instance versions of that class are considered part of the original scope, within the path. A client can also register a scope by a prime-registration operation. In the prime-registration operation a client specifies the following values for a specific path:

- *Prime name*: is the designer's name
- *Password*: is a secret token only known by the scope prime
- *Scope*: defines the scope name
- *Components:* defines the components of the scope
- *Groups*: specifies the set of clients with access-rights over a scope
- *Access Right Association*: specifies the access rights of a group over a scope, through an Access-Rights association
- *Designated Instance Version:* specifies a particular instance version to be used in lieu of the latest instance version[1]

As with other services, the operations that are related to data-ownership can be classified into five categories: create, delete, add, copy, and move; in addition, the data-ownership service supports operations which are specific to the service (e.g. an operation which delegates prime-ownership of a scope). Thus, a prime can create a scope in a path (as discussed previously), can delete a scope from a path, can add a scope to a path, can copy a scope from one path to another path, can move a

---

[1] This parameter is optional, and is used for dynamic binding of instance versions.

scope from one path to another path, and can delegate the prime ownership of a scope to another client. A client can also perform operations on *Access-Rights* associations between groups and the owned scope. Furthermore, a client can perform operations to the components of a scope. For example, a component can be added to a scope, deleted from a scope, and moved from one scope to another. There are also operations related to groups. For example, a prime can add members to a group, and delete members from a group.

As with other `data server` services, the data-ownership service provides the capability to formulate basic queries, e.g. who is the prime of a specific scope, which groups are associated with a specific scope, what are the access rights that a specific group has to a specific scope within a path, and so on.

5.3 The Changes-Notification Model

As stated previously, the *changes-notification* service provides the ability to notify affected parties that a specific event occurred on an object being monitored. The *changes-notification* model consists of a set of building blocks and a set of mechanisms. Both of them are described below.

5.3.1 Building Blocks

The building blocks of the changes-notification model are the Kernel classes: Links, NonVersionableContainers, Nodes, and SimpleObjects defined in Chapter 3.

5.3.2 Mechanisms

The set of mechanisms (*monitor, notification and approval)* used in the *changes-notification* model are discussed below.

Monitor

A monitor is the mechanism of using the relationship called Monitor/MonitoredBy (implemented by means of the MonitorLink and Link classes) to monitor and to provide notification of the occurrence of events on an object. An *event* is an asynchronuous action that can be performed on an object (e.g. an update, a lock, and so on). The *Monitor/MonitoredBy* relationship relates two objects a source object and a monitored object. The *source object* is the object that is notified of the occurrence of an event (as consequence of the notification, a number of actions can be triggered on the source object). The *monitored object* is the object which is being monitored for the occurrence of an event. The *Monitor* part of the relationship monitors the events, and the *MonitoredBy* part of the relationship is used to determine who should be notified when an event occurs. The *Monitor/MonitoredBy* relationship and related objects are illustrated, in terms of an ER diagram, in Figure 20.

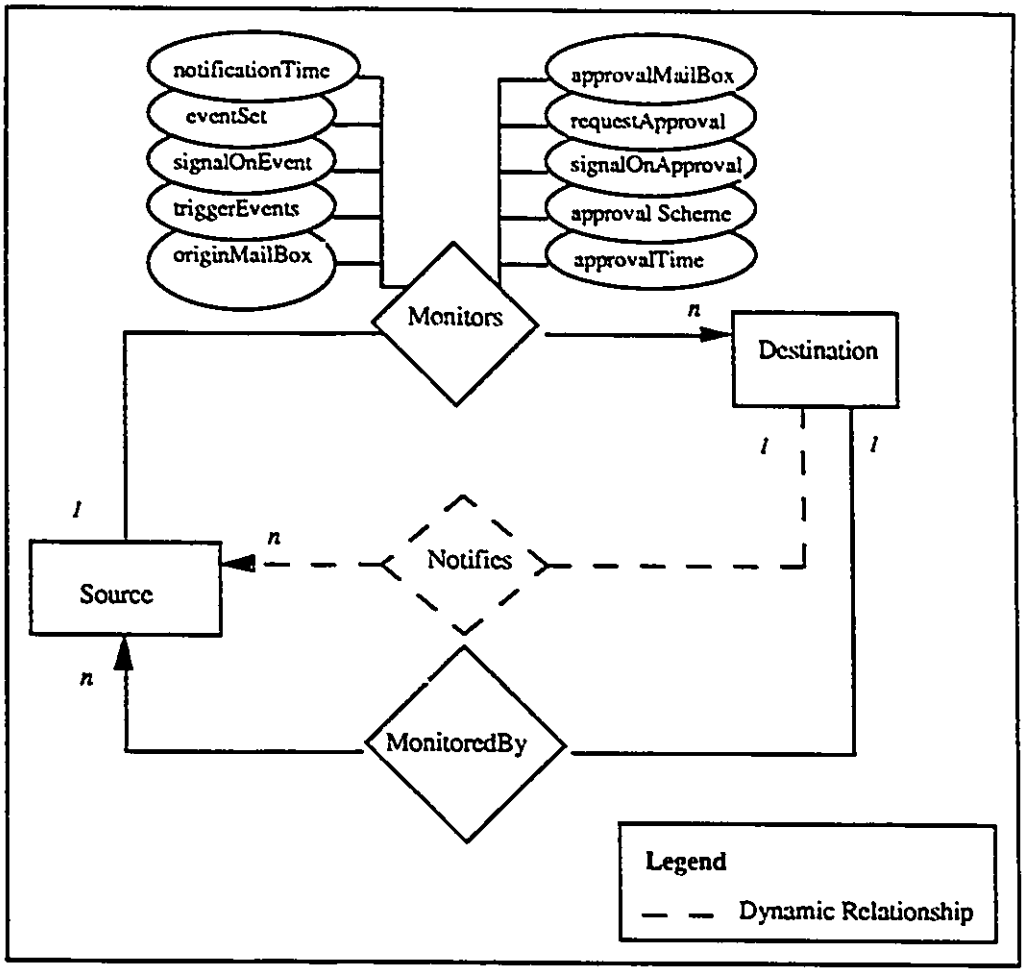


Figure 20 An ER Diagram for the Changes-Notification Model

The description of the attributes of the *monitor* association is as follows:

- *notificationTime:* specifies the time when a changes-notification is sent
- *eventSet:* specifies a set of events to be monitored with respect to an object
- *signalOnEvent:* specifies whether an event has matched a member of the *eventSet* attribute.
- *triggerEvents:* specifies a set of actions to be triggered on the source object upon the receipt of a notification message.

- *originMailBox:* specifies the place where a notification message is sent
- *approvalMailBox:* specifies the name of the mailbox where an approval message is sent
- *requestApproval:* specifies whether an approval is required for a change
- *signalOnApproval:* specifies whether an approval is granted
- *approvalScheme:* specifies an approval policy (e.g. whether the approval for a change must come from a number of clients)
- *approvalTime:* specifies the time when a change is approved

Since, a notification can be considered as an association which has temporal existence (shown as a *Notifies* association in Figure 20), the ER diagram has been extended with a dynamic association (shown as dashed lines). A *dynamic association* is an association which has short existence, and its existence depends on the temporal occurrence of events on an object.

Notification

Notification is a mechanism which provides the capability to send a notification message to objects when a specific event occurs on a monitored object. We distinguish two types of notification schemes:

- *Immediate-Notification:* A client is "immediately" notified of the occurrence of a specific event on a specific object. In reality, there is a time delay from the time that an event happens on a monitored object, until the time that the client is notified. The main reason for this delay is to only send a notification when the event is, in fact, committed to a semi-public or public library. Otherwise, a source object could be notified of an event which is afterwards aborted. Immediate notification is achieved by sending a message

by means of electronic mail (for example, by UNIX mail) to a mailbox. If the source object is an active object (also called an *actor*) it will be interrupted. The source object will then save its context in a stack, execute the interrupt,[1] invoke the actions referenced in the *triggerEvents* attribute, and resume the work from the point where it was interrupted.

- *Delayed-Notification:* A client is not notified immediately of the occurrence of an event, but rather when a client gains access to the monitored object. This is a deferred notification scheme, which is achieved by a flag-based message scheme. In a flag-based message scheme, it is only when a client gains access to the monitored object where an event happened, that the client is notified that information has been stored in a mailbox (i.e. *originMailBox).*

The following example illustrates how the changes-notification mechanism, previously described, can be implemented.

Example

Assume that there is a monitor association set between a client (source object) and an instance version (monitored object) *Iv*. As shown in Figure 21, an instance version can be in one of two states: active or passive. An instance version is in a *passive* state when it is not referenced by any monitor association. An instance version is in *active* state when it is referenced by a set of monitor associations.

[1] The actions to be executed are stored in the *triggerEvents* attribute of the monitor link.

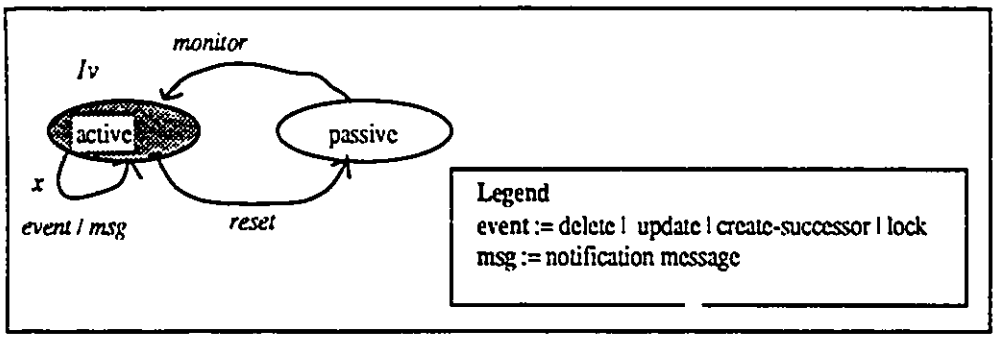


Figure 21 The Transition States of a Monitored Object

When an event *x* occurs on the instance version *Iv*, a set which contains instance versions in the active state is searched for *Iv*. If *Iv* exists in the (active) set and the event *x* matches an event specified in the *eventSet* attribute of the monitor association, then a notification message (shown as *msg*) is sent to the mailbox of the source object. The value of the attribute *signalOnEvent* is set to *true* before sending a notification message, and is set to *false* after the notification message is successfully sent. The events that occur on the monitored object, and that match any of those specified in the *eventSet* attribute, are queued in a *ready-event* queue during the time that the value of the *signalOnEvent* attribute is hold true. Then, after the corresponding notification message is successfully sent to the source object the value of the *signalOnEvent* attribute is set to *false*, and the next notification message related to the first event on the *ready-event* queue is sent to the corresponding source object. It is worth noting that a notification message can either trigger actions on the source object, or can inform of the occurrence of the event *x* on *Iv*, or both of the preceding. An instance version can be reset to a passive state by a deletion of the monitor associations that lead to the instance version.

Approval

Approval is a mechanism that provides the capability to request the notified parties to approve a change. A *change* is an event that modifies either the status of an object or data stored in an

object. Since the *approval* mechanism is a blocking scheme, a change does not take effect on the monitored object until the change is approved by the source object. Thus, no further events on the monitored object are allowed until the pending approval is granted. It is clear that when an approval is granted, a change takes effect on the monitored object if and only if:

$$approvalTime > notificationTime$$

5.3.3 Operational Aspects

The basic set of operations related to the changes-notification service are as follows. A client can define a *Monitor/MonitoredBy* relationship between a source object and a monitored object, and a client can define, delete or update the values for the attributes of the *Monitor* association. As with other types of associations, a *Monitor* association can be copied from a pair of objects to another pair, can be moved from a pair of objects to another pair, can be deleted from, and can be added to a pair of objects. As with other services, the *changes-notification* service also supports query operations, e.g. a client can query the attributes of a *Monitor* association, can query all objects that require to be informed of the occurrence of a specific event, can query all objects which are being monitored, and so on.

Chapter 6

Prototype and Test Suite

6.1 Introduction

Prototyping is an unavoidable activity within the design life-cycle, and is used to experiment with designed functionality, test several design alternatives, and so on. Through prototyping, we gained design experience which was reflected later in both the design of the kernel classes (described in Chapter 3) and the design of the services which support the data management and data control aspects of a co-operative CAD engineering environment (described in Chapter 4 and Chapter 5). Thus, after building the prototype we achieved a better classification and generalization of the kernel classes and we also gained a better understanding of the behavioral part of the `data server`.

In this thesis, both the Smalltalk-80 programming language and the OPAL programming language were used for the implementation of the prototype since they both provide a friendly development environment [Diederich 87] and support the object-oriented model. These two factors, combined with the previous object-oriented programming work experience acquired at Bell-Northern Research helped us implement the `data server` prototype in a short period of time.

The organization of this chapter is as follows. First, the prototyped class hierarchy, and the composition class hierarchy are presented. These two hierarchies complement each other, and provide suitable information about the prototype. It is worth noting that one can obtain very complete details by reading the Smalltalk-80 and OPAL source code files given in Appendix E. Next, the role of each of the prototyped classes is described. Second, a test suite which tests versioning, configuration management and data-ownership functionality is described. The test suite is based on the design of a microcontroller.

6.2 Class Inheritance and Class Composition Hierarchies

To understand the prototype implementation, one must consider two class hierarchies together: the composition class hierarchy and the inheritance class hierarchy. The (single inheritance) class hierarchy provides the organization of classes as related by inheritance relationships; the composition class hierarchy provides the organization of classes as related by aggregation relationships.

The class hierarchy used for the prototype is shown in Figure 22. The basic role of each class is as follows:

- *Object:* The Object is the root class of the class hierarchy.[1]
- *NamedObject:* The class NamedObject is used to name and describe its subclasses.
- *Node:* Instances of class Node are used as containers which can be versioned (via predecessor/successor relationships) and can be associated with other objects.
- *Library:* An instance of class Library is a datastore unit. A library contains a dictionary of instances of class Stream.
- *SharedLibrary:* Instances of this class represent either public or semi-public libraries.
- *PrivateLibrary:* Instances of this class represent private libraries. Only its owner can gain access to a private library.
- *Stream:* This class is a physical representation of a design cycle. A stream contains an array of instances of class Context.

---

[1] Every object created in the system can respond to messages defined in class Object.

- *Context:* A context is a collection of composite-configurations which occur in a specific phase of the design cycle. These composite configurations are implemented as instance versions (which are instances of class VersionedObject).

- *VersionedObject:* An instance of class VersionedObject implements either an instance version or a version of a composite configuration.

- *VersionManager:* The class VersionManager is a holder of instances of the class VersionedObject.

- *User:* Instances of this class represent clients of the `data server`. The class User provides the ability to register users and to query them.

- *Group:* A group consists of either multiple users (instances of class User), or groups, or both. The class Group provides the ability to register groups of users in the `data server`.

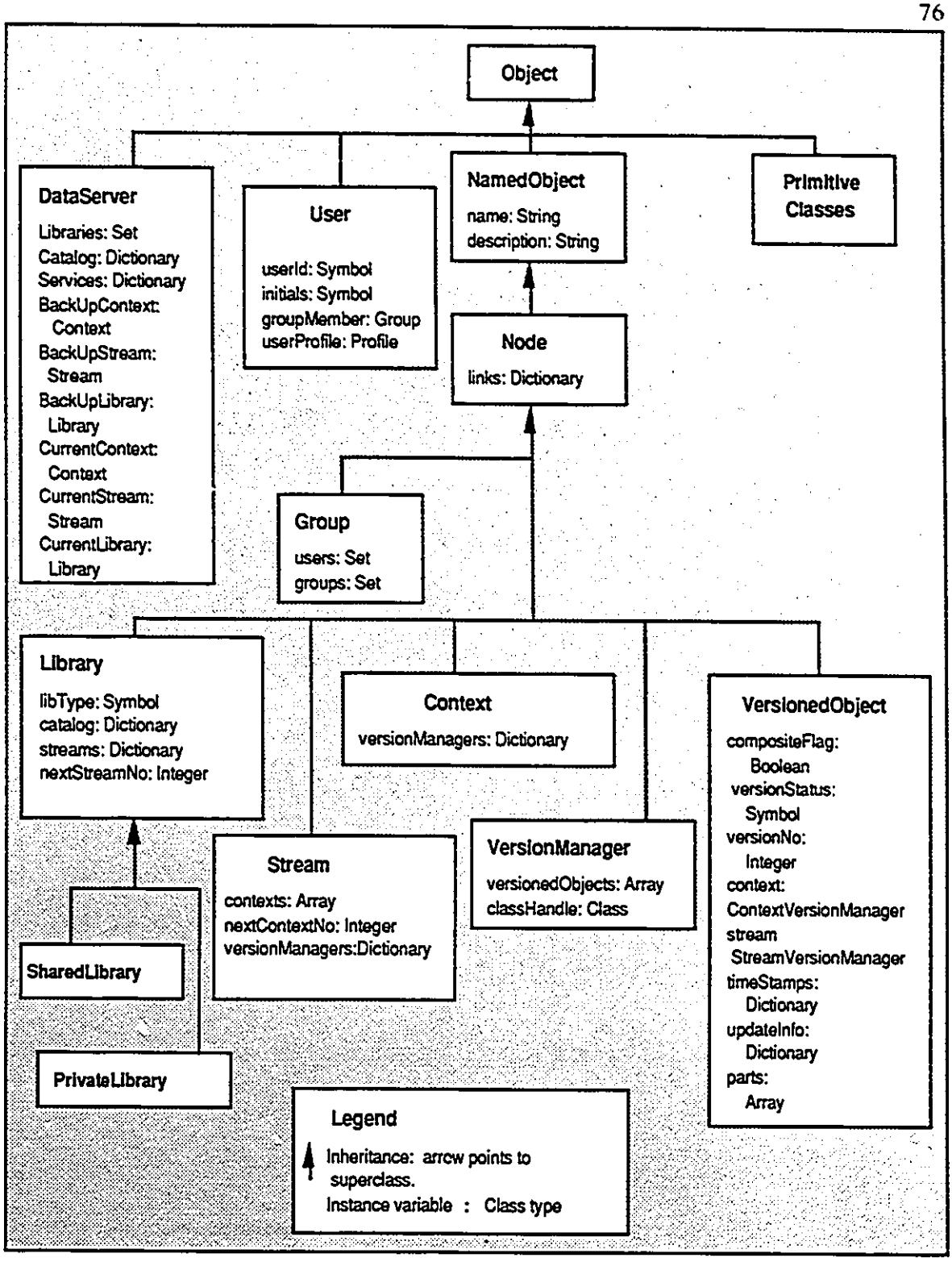


Figure 22 The Prototype Inheritance Class Hierarchy

As stated previously, the class hierarchy shows the inheritance relationship between classes (the arrows always point to a superclass). For example the class VersionedObject inherits structure and behavior from the superclass Node. The data type of the instance variables is also indicated in the Figure 22. For example, the instance variable called *links* (in the class Node) is of type Dictionary. The types Set, Dictionary, Array, Symbol, Integer and String are OPAL and Smalltalk-80 classes, that have been used in the prototype. They are clustered under the name of *Primitive Classes*. Besides the inheritance relationship between classes, Figure 23 illustrates the corresponding composition class hierarchy.

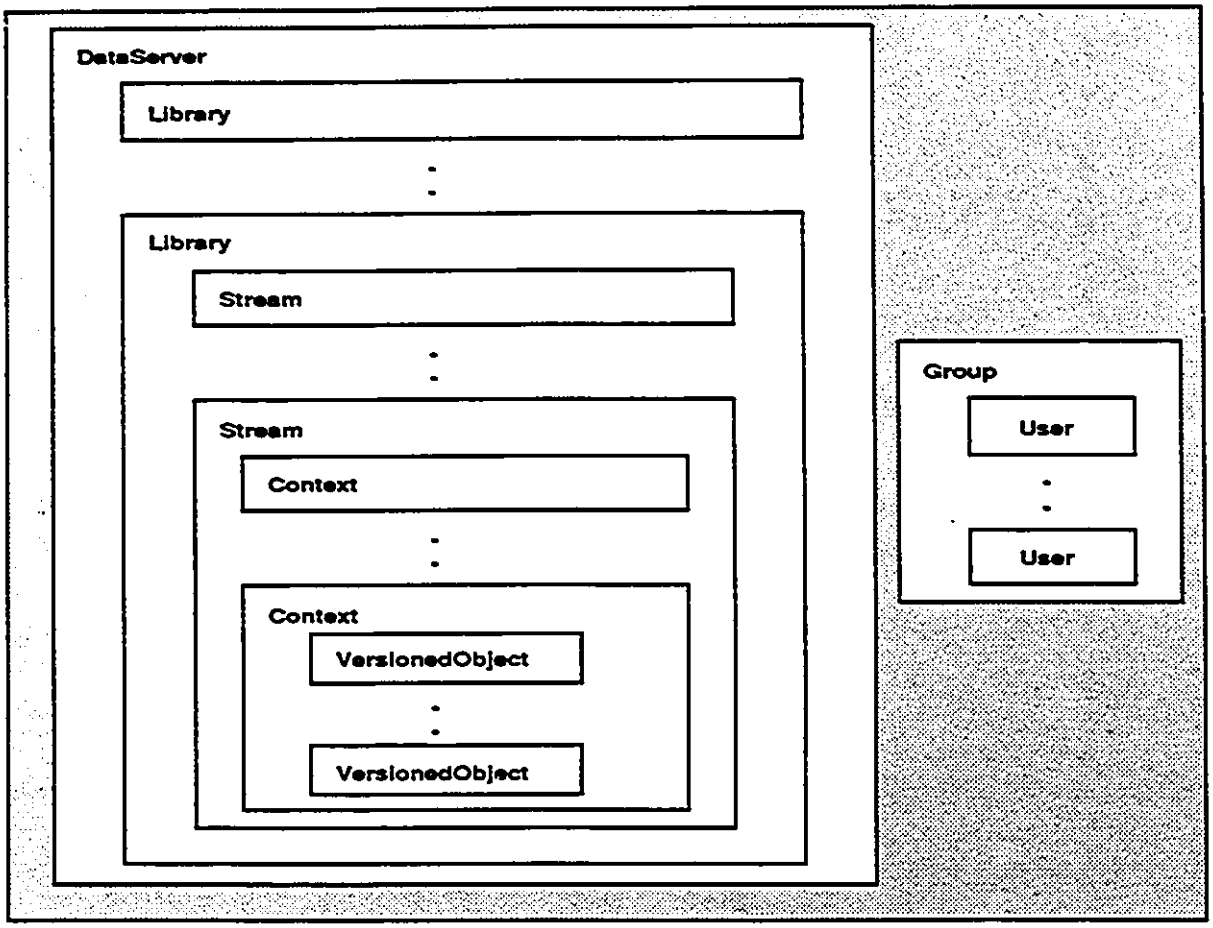


Figure 23 The Prototype Composition Class Hierarchy

Instances of the class VersionedObject are contained within an instance of class Context. Instances of classes Context are contained within an instance of class Stream. Instances of class Stream are contained within an instance of class Library, through the instance variable called *streams*, whose type is a Set (as shown in Figure 22). Thus, each member of the slot streams, of type Set, is an instance of class Stream. Instances of class "Library" are contained and tracked by means of the DataServer class variable named Libraries. The class variable named *Libraries* is also of type Set. Each member of this set is either an instance of the class SharedLibrary or an instance of the class PrivateLibrary.

6.3 Testing Strategy: The Test Suite

Testing is a phase of the software life cycle [Davis 90] that includes three distinct stages:

- *Unit testing*. Is the stage of the software life cycle immediately following coding, during which software modules are checked by their own designers to see if the modules meet design specifications.

- *Integration testing:* Is the stage of the software life cycle immediately following unit testing during which previously unit-tested software modules are integrated to see if they function as a team.

- *System testing*. Is the stage of the software life cycle immediately following integration testing, during which the fully integrated software is checked to see if it meets all requirements stated in the software requirements specification.

An important task of testing is to design a suitable test suite that exercises the above testing stages. Thus, to test the prototype of the `data server` we built a test suite. It is worth noting that when the prototype was built we considered not only the functionality aspects but also the testing aspects,

i.e. to support unit testing the behavioral part of the classes provides in addition to the specific functionality a set of operations that support the testing of the class functionality.

The test suite exercises unit, integration, and system testing by performing a walk-through of the design of a microcontroller module. The components of the microcontroller module are shown in Figure 24.

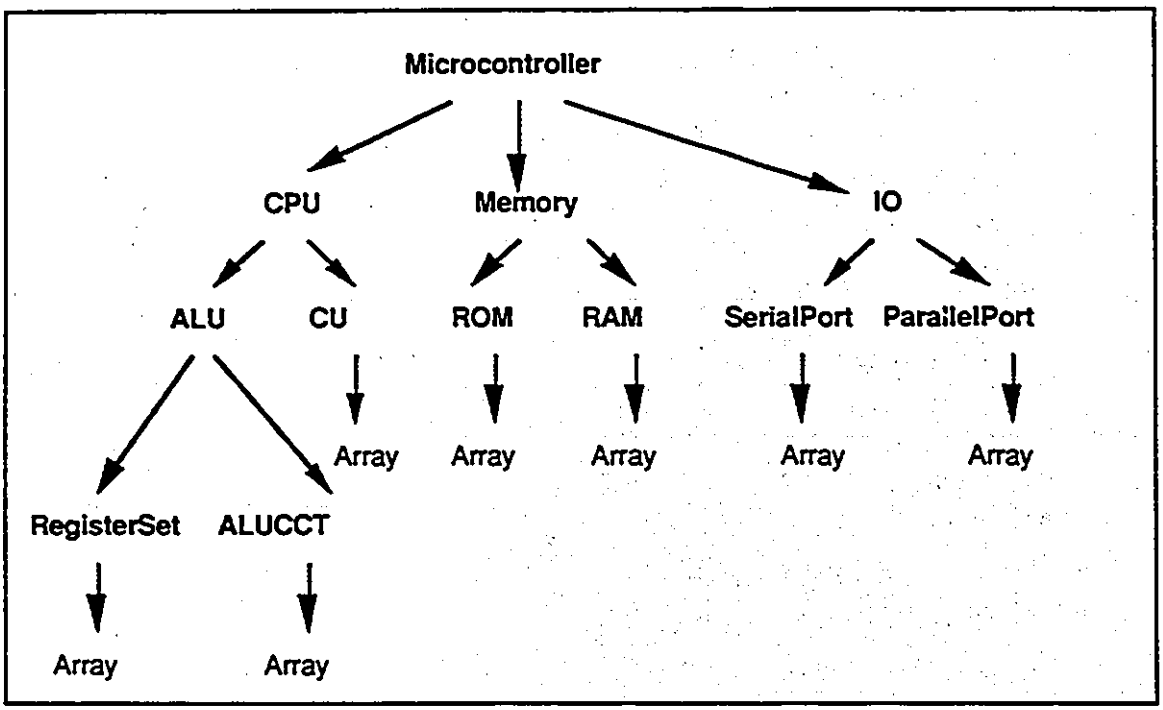


Figure 24 The Parts of a Microcontroller Module

The three parts of the microcontroller module are as follows:

Central Processor Unit (CPU), which contains:

- Arithmetic Logic Unit (ALU), which contains a *Register Set* (RegisterSet) and an *Arithmetic Logic Unit Circuitry* (ALUCCT)

- Control Unit (CU)

Memory Unit, which consists of:

- Read Only Memory (ROM).
- Read Access Memory (RAM).

Input-Output Ports (IO), which contains:

- Serial Port.
- Parallel Port.

The microcontroller module is represented by a composite-configuration. The leaves of the composite-configuration are instances of class Array[1], which contains design information.

An example of one of the queries used in the test suite is given below. A complete description of the test suite can be found in appendix D.

Example

Assume that an instance version of a microcontroller called *Microcontroller_3* already exists in a public library. A client has set a path, and needs to find all the parts for that instance version. Then the client formulates the following query to the data server:

aDataServer getClosureFor: 'Microcontroller_3'.

The meaning of this query is as follows: When the message *getClosureFor:* is sent to an instance of a DataServer class (with a parameter which represents the name of the instance

[1] An instance of class array may contain files of design related information.

version *Microcontroller.3* of type String), a dictionary is returned to the sender. A real snapshot of the computer screen, showing this dictionary is illustrated in Figure 25.

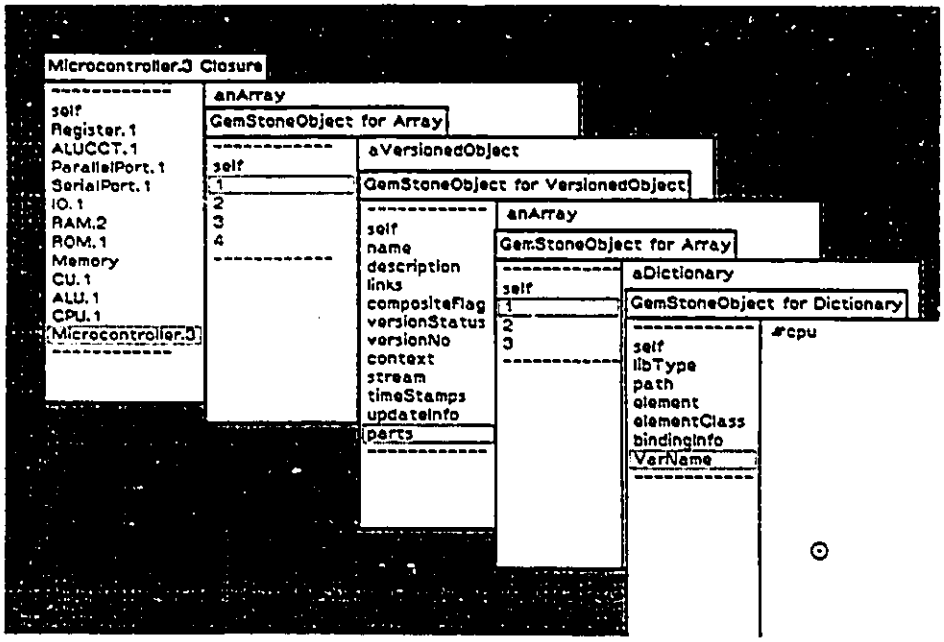


Figure 25 Parts of an Instance Version: Microcontroller.3

The dictionary contains associations, each of which contains a key and a value. The *key* is a name of an instance version, e.g. Microcontroller.3, and the *value* is an array which contains four elements. These elements contain metadata information (e.g. the first element of the array is the instance version which corresponds to the Microcontroller.3 key, the second element is the version number of the instance version, the third element is the version number which will be given once it is stored in a target context, and the fourth element is the class to which the instance version refers). Also, the figure illustrates the instance variables of the instance version contained in the first element of the array (shown as *aVersionedObject)* . The high-lighted instance variable *parts* is an array of dictionaries. This array contains three elements, since the Microcontroller instance version contains three parts: *CPU.1, Memory* (which is dynamically bound), and *IO.1*. For each part, there is a dictionary (on bottom right part of the

Figure 25). This dictionary contains six associations. The key and value for each of these associations are as follows:

- *libType:* is a library type (e.g. public, semi-public, private)
- *path:* is a virtual path
- *element:* is the element name (e.g. CPU.1)
- *elementClass:* is the class of the element (e.g. CPU)
- *bindingInfo:* indicates whether the binding is either static or dynamic
- *varName*: is the instance variable name of the host (e.g., Microcontroller.3) which is bound to a part (e.g, #cpu[1])

This example describes how a client can query all the parts of an instance version (e.g. a Microcontroller.3). In addition, the example illustrates the physical implementation of the binding of an instance version to its parts (e.g. Microcontroller.3 and its static binding with both the IO.1 and CPU.1 parts, and the dynamic binding with the Memory part).

[1] The "#" symbol indicates that the *cpu* is stored as a symbol.

Chapter 7

Conclusions, Comparisons, and Future Work

7.1 Thesis Contributions

This thesis proposes a solution to significant problems encountered in CAD engineering environments. It does this by introducing a services-layer on top of an OODBMS. More concretely, three contributions have been made by this work, as follows:

Extending the object-oriented model:

To support the ability to model complex structures, to represent the semantics of existing data, and to relate complex data, we extended the object-oriented model with a link construct. It is the third element of a newly defined triad: *structure-behavior-link*. The structure defines the attributes of objects, the behavior specifies the messages that the object understands and how it will behave, and the link construct associates objects. By using the extended object-oriented model, one can represent complex data relationships and assign semantics to data.

Investigating and clarifying requirements in a co-operative CAD engineering environment:

A complete set of requirements was presented for a `data server` that supports a co-operative CAD engineering environment. This set of requirements which covers the two aspects of data integration (data management and data control) was used to focus the design of the `data server` to provide the services of these two aspects.

Designing and prototyping a `data server` services-layer:

The `data server` versioning and configuration management services support the data management aspect of a co-operative engineering design environment, and the data-ownership

and changes-notification services support the data control aspect. Thus, the system integration of a services-layer which provides these services with an object-oriented database management system, fully supports data integration in a co-operative CAD engineering design environment. Finally, we have demonstrated, by means of a `data server` prototype, that the `data server` can be implemented easily and rapidly, using and extending the object-oriented model.

7.2 Comparisons

The main differences between this thesis, and other works [Katz 90, Kim 89, Katz 87b, Katz 87, Katz 86, Chou 86, Dittrich 85] including commercial object-oriented databases (ObjectStore, Objectivity/DB, Versant)™ [OS 90, OE 90 , VE 90] are as follows:

First, we have extended the object-oriented model with the link construct, which has structure and behavior. The link construct models multiple semantic associations between instance versions (version of instances). On the other hand, current object-oriented databases (for example, Versant, Objectivity/DB and ObjectStore) do not support these multiple semantic associations. They support only associations of any cardinality between instance versions (for example, 1 to n, n to 1, and n to m). Other related work, like Katz [Katz 90], uses the relational model, while Kim [Kim 89] and Chou [Chou 86] use the object-oriented model without extensions.

Second, this thesis has presented a complete set of requirements for a `data server` that must support a suitable semantic model and rich data integration in a co-operative engineering design environment. Katz [Katz 90], Chou [Chou 86], and Dittrich [Dittrich 85] consider only a small sub-set of the `data server` requirements.

™ObjectStore is a registered trademark of Object Design, Inc; Objectivity/DB is a registered trademark of Objectivity, Inc.; Versant is a registered trademark of Versant Technology, Inc.

Third, the data server supports versioning of configuration constructs. Other work considers only composite object constructs, equivalence and aggregation constructs, as introduced by Katz [Katz 90, Katz 87], Banerjee [Banerjee 86], and Chou [Chou 86], and molecular objects, as described by Batory [Batory 85]. The data server configuration construct can represent any of the above. It can contain nested configurations, and any type of associations/relations between contained instance versions.

Fourth, the data server supports a data-ownership service to safeguard the release of data to the public domain. None of the current object-oriented databases support this service.

Fifth, the data server supports a changes-notification service to inform a user or a process when a particular event happens to an object. None of the current object-oriented databases support this service.

Last, the data server integrates versioning, configuration management, data-ownership and changes-notification services to support a co-operative engineering design environment, adopting and extending the object-oriented model. To the best of our knowledge, other academic works focus their efforts on only a sub-set of these services (e.g. versioning and configuration management, or concurrency control). Most of this related work is based on the relational model (i.e. Katz [Katz 90]).

7.3 Future Work

The work in this thesis assumes a co-operative CAD engineering environments over a LAN (local area network). More research is needed on how to deploy the services-layer to support co-operative CAD engineering environments over a WAN (wide area network). This is a more difficult problem for several reasons:

- First, medium to large size companies (for example, telecommunication companies) have their designers located at different geographic sites (across provinces, countries, and even continents), and design information must be shared among those sites. Thus, the multi-client `data server` architecture must be extended, and design information must be efficiently distributed, controlled, and managed.

- Second, these companies store data in heterogeneous repositories (for example relational databases, object-oriented databases, and so on), and designers must access the data transparently. Thus, a seamless integration of heterogeneous repositories is required.

- Third, users require access to data stored in heterogeneous platforms transparently over a WAN. Thus, efficient data translation and data manipulation mechanisms must be considered.

- Finally, users may need to reference data by name on several sites. Thus, suitable naming rules must be supported to avoid name conflicts.

Appendix A

The Characteristics of the CAD VLSI Circuit Design Domain

A.1 Introduction

This appendix describes the CAD VLSI circuit engineering domain as an example of a typical CAD engineering environment. With this objective in mind, we will briefly review the current VLSI design process, focusing on the semantic modeling, data integration, and co-operative teamwork design issues. A clear understanding of the VLSI design process will help in understanding some of the problems faced by the design engineering community at large.

A.2 VLSI Design Philosophy

The design of a Very Large Scale Integrated (VLSI) circuit follows a design life cycle similar to that of other engineering design domains. The *design life cycle* tracks the phases through which a design evolves. These phases are: problem and requirements analysis, requirements definition and specification, design specification, implementation, integration, testing and documentation.

VLSI CAD design environments are currently used and developed to help designers to accomplish the design life cycle phases successfully. Typically, a CAD design environment combines design CAD tools, project management aids, and a design library of VLSI components (stored in a database).

CAD tools can be classified into three broad categories:

- *Synthesis* tools. They assist a designer in creating the object being designed. For example, circuit module generators are used to map a functional description of a subsystem (in terms of register transfer description or input/output behavior specified as Boolean equations) into mask geometries suitable for fabrication [Thom 83, Sout 83].

- *Analysis* tools. They assist a designer in checking design description for correctness. For example, simulators are used at all levels of the design to ensure that the intended characterization at any level has been realized.

- *Information management* tools. They assist a designer in organizing the design data within a database. For example, a schema editor is used to redefine the definition of a component in

the design library; navigators are used to navigate through the data; and browsers are used to view parts of the design data.

The VLSI circuit design process typically begins with a descriptive specification of the design, consisting primarily of dataflow and timing graphs, which together describe the data-transformation and timing behavior of the desired hardware. Less detailed structural (i.e. schematic) and physical specifications are given, describing static properties of the target circuit.

In the following stages of the design process, VLSI circuits are described in several *abstractions*. An *abstraction* captures the description of a VLSI design in one of its three dimensions: behavioral, structural and physical. In addition, an abstraction may consists of one or more design *representations* (which will be defined later). VLSI design *abstractions* are grouped into:

- *Behavioral* abstractions: These typically describe the design in terms of event-response pairs; as such, they are representative of the behavior and interactions of pieces of the design. They may include one or more design *representations* such as: Boolean expressions, Finite State Machines (FSM), programs written in Hardware Description Languages (HDLs), and so on.

- *Structural* abstractions: These typically describe the design in terms of interconnection of various types of modules (data entities). They may include one or more design *representations* such as: logic circuit schematics, transistor networks, Register Transfer Level (RTL) , circuit modules, and so on.

- *Physical* abstractions: These typically describe the design in terms of placement and geometry. They may include one or more design *representations* such as: super-macro cells, mask geometry, and so on.

A *representation* is a hierarchy of data entities (see Figure 26), which are appropriate for different characterizations of the design. Thus, it represents the VLSI design from many different viewpoints. For example, geometric representations are used for mask-making and geometric design rule-checking; transistor, logic, and functional representations are used for electrical rule-checking, simulation and timing verification at various levels of the VLSI circuit design. A *data entity* is design information or design-related information.

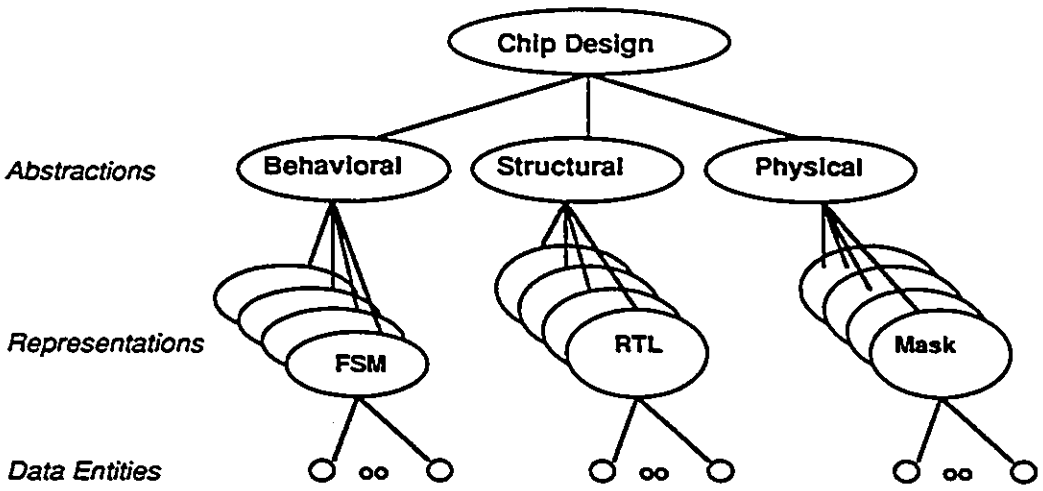


Figure 26 Abstractions, Representations, and Data Entities

In addition, several *relationships* might be specified among the components of a representation, or between representations (e.g. among specific time intervals and data operations in the timing and dataflow graphs). Various *constraints* can be attached to a representation; for example, the duration of a time interval can be limited, a schematic wire can be specified to be a bidirectional bus connection, and the area of a physical bounding box can be limited. Several *translations* might be used to map representations; for example, the characterization of the *behavioral* abstraction in a hardware description language (HDL) is translated through successive design steps to the *mask geometry* representation. These translations are normally categorized into:

- *Vertical* translations: They map the characterization of any representation of one abstraction to the corresponding representation of some other abstraction. For example, an algorithmic description can be mapped onto a set of interconnected Register Transfer Logic (RTL) blocks, each block being mapped onto a physical super-macro cell with input/output ports located on the cell boundary.

- *Horizontal* translations: They transform one level of description into the next lower level in the hierarchy of a representation. For example, a RTL block adder can be translated to a logic circuit with gates, each gate translated to a set of interconnected MOS transistors, each transistor in turn mapped on to a set of electrically interconnected geometric shapes on different layers (for example, metal, poly-silicon, and diffusion).

There are a large number of other operations associated with multiple representations of different abstractions of a VLSI circuit design, for example, simulation, rule-checking, layout, and so on. Depending on the representation types, the above operations may differ.

Since a description of an object being designed exists simultaneously in several representations and several abstractions, keeping the design description consistent within and across representations is a critical challenge. These design representations are usually large and complex (nested) hierarchies of data entities. The main characteristics of these data entities are described below.

A.3 Characteristics of VLSI Data Entities

The characteristics of VLSI data entities are:

- They may be composite. For example, an Adder may be composed of two Half-Adders .
- They may relate to other entities. For example, the layout representation of an ALU may be associated by an equivalence relationship with its ALU transistor representation.
- They may be constrained. For example, a schematic wire can be specified to be a bidirectional bus connection.
- They may be typed. For example, a data entity could be considered to be of type Cell.
- They may have a well-defined interface. For example, a Flip-Flop may have an interface which describes its abstract behavior, usage information, and associated performance (speed, power, area).
- They may hide their implementation details from their external interface (called data encapsulation).

The characteristics of these VLSI data entities, combined with the type of operations that VLSI designers may want to do on them, suggest that the object-oriented paradigm [Wegner 89] could be used to model the VLSI data entities. In fact, this is not a new idea; in recent years, many university and industry projects have used the object-oriented paradigm as a natural modeling mechanism. Some references to CAD VLSI circuit design environments based on the object-oriented paradigm can be found in [Kropf 89, Zimmermann 89, Chung 88, Demers 87, Wolf 86, Afsarmanesh 85]. We will briefly discuss some of those references:

- The work done by Kropf [Kropf 89], presents an approach for the representation of design data based on the object-oriented paradigm. It is shown that the *structural* and *behavioral* representations of design data can be modeled using the object-oriented model. In their approach, a VLSI circuit consists of modules. Modules may consist of other modules, with any arbitrary depth of hierarchy. The interconnection of several modules produces the structure of the design. The behavior is modeled either by a combinational function representation, or a complex description with internal states.

- In the work done by Demers [Demers 87], the *circuit cell* is considered the principal object (used as the heart of the majority of the design representations). The circuit cell hides its implementation details (encapsulation), which can be manipulated by a well-defined set of operations, that is, the *cell-object's methods*. A hierarchy of interconnected cells forms the final top-level cell: the VLSI circuit.

- In the work done by Zimmerman [Zimmerman 89] and in the EDIF standard proposal, the *cell* entity is considered an object with a well-defined interface and functions based on the design specification.

In summary, the rapidly emerging number of VLSI design environments, combined with the iterative and exploratory nature of the VLSI design process, and the characteristics of VLSI data entities, give us the correct idea that the VLSI design process is a very complex and challenging endeavor. That is why VLSI circuits are normally designed by skilled teams of designers. The characteristics of the interactions between teams of designers and VLSI design entities will be discussed next.

A.4 VLSI Design Interaction Characteristics

Teams of skilled designers (called artists in [Katz 85]) are assigned different tasks in the design process; for example, some are responsible for abstracting either the behavioral, or structural, or physical representations of a VLSI chip design. These designers interact with VLSI CAD environments that are characterized by a high degree of heterogeneity in both software and hardware. Normally, designers follow a hierarchical design methodology, applying generalization, specialization, and aggregration approaches, to produce the VLSI design within time and budget constraints.

A reasonable model of how a designer interacts with the data is as follows. He first checks-out (reads) a portion from the public file server copy of the design into a private workspace. Over time, the design portion is modified by the designer with the tools available to him at the

workstation. When he is satisfied that his changes are complete, he checks-in (writes) the design portion back onto the file server. However, before it can be incorporated into the public copy of the design, it must pass through a consistent battery of tests, such as simulation, design rule checks, and so on.

Since the designer may make improvements to an existing implementation, it is important that these improvements not overwrite an existing public version. Once a design portion is checked-out, it is returned to the public repository as a new version. When the design activity is complete, a complete chip description is assembled from a particular configuration of versions of its parts, and is released to manufacturing.

The above model is suitable only for a single designer. However, when multiple designers must work co-operatively in a CAD engineering environment, the above model must be re-engineered to support co-operative work (i.e. efficient and common mechanisms to manage and to control data are required).

Appendix B

A Survey of Commercial Object-Oriented Repositories

B.1 Introduction

This section provides a survey on object-oriented repositories to uncover those areas where further research is needed to support a co-operative CAD engineering environment. This survey, which was conducted by us at Bell Northern Research, provides the following information:

- the common functionality, which shows consensus about some features among vendors
- functionality not currently supported , which shows what has not yet been addressed
- the architecture, which provides a short description of both the micro-architecture and macro-architecture aspects

Only a short list of all the surveyed repositories is presented. This short list is representative of the current object-oriented database technology. The short list of repositories is as follows:

- Objectivity/DB, from Objectivity Inc. [OV 90]
- Versant, from Versant Technology Inc. [VE 90]
- ObjectStore, from Object Design Inc. [OE 90]
- GemStone,™ from Servio Inc. [GemStone 90, Maier 89]

B.2 Common Functionality

This section presents the common features supported by each of the repositories on the short list. A feature supported by at least two repositories is considered as common functionality. The common features are given below.

- object model

™ GemStone is a registered trademark of Servio, Corp.

- short and long transactions[1]; two-phase-locking; two-phase-commit protocols
- data distribution over a local area network
- associations between objects of 1-n, n-1, and n-m cardinality[2]
- query language[3]
- iterator constructs and dynamic arrays
- C and C++ language interfaces[4]

B.3 Non-Supported Functionality

This section presents the features not currently supported by each of the repositories on the short list. The features not supported are:

- schema versioning and schema evolution
- versioning of logical and physical containers, and nested containers
- associations which contain structure and behavior
- data-ownership service, and changes-notification service
- crash-recovery mechanisms
- non-stop incremental backups

B.4 Architecture

The architecture consists of two parts: the micro-architecture and the macro-architecture. The *micro-architecture* is related to the deployment of processes; the *macro-architecture* is related to the distribution of data.

---

[1] GemStone release 1.5 does not support long transactions.
[2] Not supported by GemStone release 1.5.
[3] Objectivity/DB (version 1.1) does not support queries.
[4] GemStone and Versant OODBs provide a Smalltak-80 interface.

This section will provide a brief overview of both the micro-architecture and macro-architecture for each of the repositories on the short list .

B.4.1 Objectivity/DB Architecture

- The micro-architecture is a peer-to-peer process architecture. Thus, any workstation can behave either as a server, as a client, or both.
  - The management of locks is centralized.[1]
  - NFS is used as the networking protocol.
  - A 64-bit address space and handles[2] are used to reference objects.
- The macro-architecture supports a federated database. A federated database consists of multiple databases. A database consists of a set of containers, and a container consists of multiple objects.

B.4.2 Versant Architecture

- The micro-architecture is a multi-client multi-server process architecture.
  - The management of locks is distributed.[3]
  - Remote Procedure Control (RPC) is used as the networking protocol.
  - A 64-bit address space and pointers are used to send messages to persistent objects.
- The macro-architecture supports a distributed repository, which consists of multiple group and private databases.

B.4.3 ObjectStore Architecture

- The micro-architecture is a multi-client multi-server process architecture.
  - The management of locks is distributed.[4]

---

[1] A container is used as the unit of locking granularity, clustering, and transmission over the LAN.
[2] A handle is a proxy used to send messages to persistent objects.
[3] An object is used as the unit of locking granularity and the smallest unit of transmission over the LAN.
[4] A page is used as the unit of locking granularity and the smallest unit of transmission over the LAN.

- TCP/IP is used as the networking protocol.

- A 32-bit address is used for virtual memory address space. It is a virtual memory faulting architecture (the logical object identifier is 89 bits long).

• The macro-architecture supports a distributed repository, which consists of a hierarchy of workspaces.

B.4.4 GemStone Architecture

• The micro-architecture is a multi-client server process architecture.

- The management of locks is centralized.[1]

- Uses TCP/IP as the networking protocol.

- A 32-bit address space is used. Proxies are used to send messages to persistent objects.

• The macro-architecture supports a centralized repository.

[1]Optimistic concurrency control is the default mechanism; however, pessimistic concurrency control can be used by explicitly locking a set of objects. An object is used as the unit of locking granularity and the smallest unit of transmission over the LAN.

Appendix C

Description of the Kernel Classes

C.1 Kernel Classes

This appendix describes the structural and behavioral parts of the kernel classes, excluding the description for the classes Node and Link (given in Chapter 3). The description of the Kernel classes is given below.

SimpleObject

A *SimpleObject* is an object which represents unstructured data. The attributes of a SimpleObject are described below:

name: specifies the name of the object

ctime: specifies the creation time of the object

Client

A *Client* is a simple object which represents a `data server` client (either a user or a CAD tool). The attributes of a Client are described below:

userId: specifies a user identifier

password: specifies the password used to be able to access the services provided by the `data server`

links: specifies a dictionary of links which are used to represent associations between objects.

ComplexObject

A *ComplexObject* represents an object which can contain other objects. The attributes of a ComplexObject are described below:

name: specifies the name of the object

description: specifies the purpose of an object

ctime: specifies the creation time of the object

utime: specifies the latest time the object was updated

elements : holds the contents of an object.

MonitorLink

A *MonitorLink* is a link which is used to monitor a destination object for the occurrence of a set of pre-defined events. The attributes of a MonitorLink[1] are described below:

- *eventSet:* specifies a set of events to be monitored on a object
- *notificationTime:* specifies the time when a changes-notification is sent
- *originMailBox:* specifies the place where a notification message is sent
- *approvalMailBox:* specifies the name of the mailbox where an approval message is sent
- *requestApproval:* specifies whether an approval is required for a change
- *signalOnEvent:* specifies whether an event has matched a member of the *eventSet* attribute.
- *approvalScheme:* specifies an approval policy (e.g., whether the approval for a change must come from a number of clients)
- *approvalTime:* specifies the time when a change is approved
- *signalOnApproval:* specifies whether an approval is granted
- *triggerEvents:* specifies a set of actions to be triggered on the source object upon the receipt of a notification message.

AccessRightsLink

The *AccessRightsLink* is a link which is used to assign to the source object access rights with respect to a destination object. The attributes of a AccessRightsLink[2] are described below:

---

[1] The MonitorLink is a key building block of the *changes-notification* service.

[2] The AccessRights Link is a key building block of the *data-ownership* service.

read: specifies that the source object can have read access to the destination object

write: specifies that the source object can have write access to the destination object

none: specifies that the source object can have neither update access, nor have read access to a destination object

release: specifies a set of rules (or contract) that both the source object and destination object have agreed on

StabilityLink

The *StabilityLink* is a link which specifies that a destination object cannot be deleted. Thus, the attribute *stability* specifies whether the destination object can be deleted.

NonVersionableContainer

A *NonVersionableContainer* is an non-versioned object which can contain a number of objects. The attributes of a NonVersionableContainer are described below:

links: is a dictionary of instances of class Link

nlinks: specifies the number of instances of class Link leading from this object

slinks: specifies the number of instances of class Link leading to this object

Group

A *Group* can contain clients (instances of a class Client), and an instance of class Group can also be nested. The attributes of the class Group are given below:

groupId: specifies the group indentifier

password: specifies the password used by a group to access the `data server`

It is worth mentioning that the behavioral part of all the kernel classes provide the capability to read and update their structural part. Thus, for each kernel class a client can create a number of instances, as well as read and update the attributes of the instances.

Appendix D

The Data Server Prototype

D.1 Test Suite

In this thesis, the test *suite* is a set of tests used to check the prototype functionality. A walk-through the test suite also demonstrates the `data server` functionality. The test suite consists of two parts:

- An administration part: This part describes the creation of libraries, streams and contexts (steps 1 to 5 of the test suite).[1]
- A design part: This part presents the design of a microcontroller module (steps 6 and up of the test suite).

The test suite is written in the OPAL object-oriented language, comments are enclosed within quotes, and OPAL statements are terminated with a period. Where required, information related to the action performed and a snapshot taken of the computer screen are included. A walk-through of the test suite is as follows.

"Start of test suite "

"1 Add a dictionary to the list of symbol dictionaries managed by the OO-DBMS[2]"
DataServer new addDictionaryToSymbolList: #ThesisData."[3] see Figure 27"

---

[1] The administration tasks are usually the responsibility of the `data server` administrator.

[2] The SymbolDictionary contains the associations which define the names of all the objects that clients might need.

[3] When the message *new* is sent to the Data Server class, the following steps happen. First, an instance of the DataServer class is created. Second, the message *addDictionaryToSymbolList:* with an argument called "ThesisData" is sent to that instance. The `data server` instance implements this message by inserting the "ThesisData" dictionary in the SymbolDictionaries managed by the OO-DBMS.

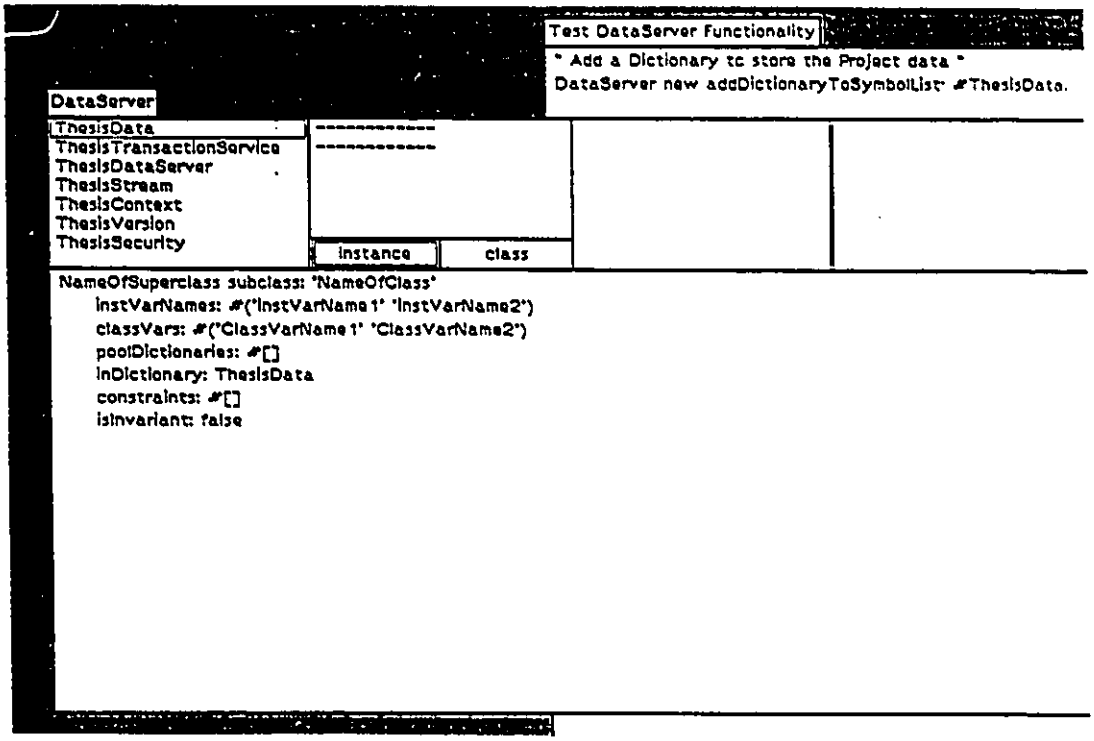


Figure 27 The OO-DBMS Symbol Dictionaries

"DataServer initDataServer.[1]"

"2 Add libraries to the list of libraries accessed through the `data server` *"*

DataServer new addLibrary: (SharedLibrary name: 'Microcontroller Public' type: #Public description: 'Released Design of a Microcontroller')."[2]"
DataServer new addLibrary: (SharedLibrary name: 'Microcontroller SemiPublic' type: #SemiPublic
description: 'Design of a Microcontroller').
DataServer new addLibrary: (PrivateLibrary name: 'Microcontroller Private' type: #Private description: 'Private Design of a Microcontroller'). "[3]"

---

[1] When the class method *initDataServer* is sent to the class DataServer, it clears all the DataServer class variables. (This message is available only to a `data server` administrator).

[2] The result of the execution of the expression within parenthesis is an instance of the class SharedLibrary. When the message *addLibrary* is sent to an instance of the Data Server class with the instance of the class SharedLibrary as an argument, the argument is added to the (class variable) Libraries which belongs to the DataServer class.

[3] After the execution of this statement, three libraries are added to the class variable Libraries, which belongs to the DataServer class.

"3 Query a specific library which can be accessed through the `data server`*"*

DataServer new libraryNamed: 'Microcontroller Private' type: #Private.
DataServer new libraries."[1]"

"4 Add an instance of a Stream class to a specific library"

(DataServer new libraryNamed: 'Microcontroller Public' type: #Public) addStream: 'Release 1.0'.
(DataServer new libraryNamed: 'Microcontroller SemiPublic' type: #SemiPublic) addStream: 'Release 1.0'.
(DataServer new libraryNamed: 'Microcontroller Private' type: #Private) addStream: 'Release 1.0'. "[2] "

"5 Query the `data server` *libraries and set the current path"*

DataServer new libraries. "[3] See Figure 28"

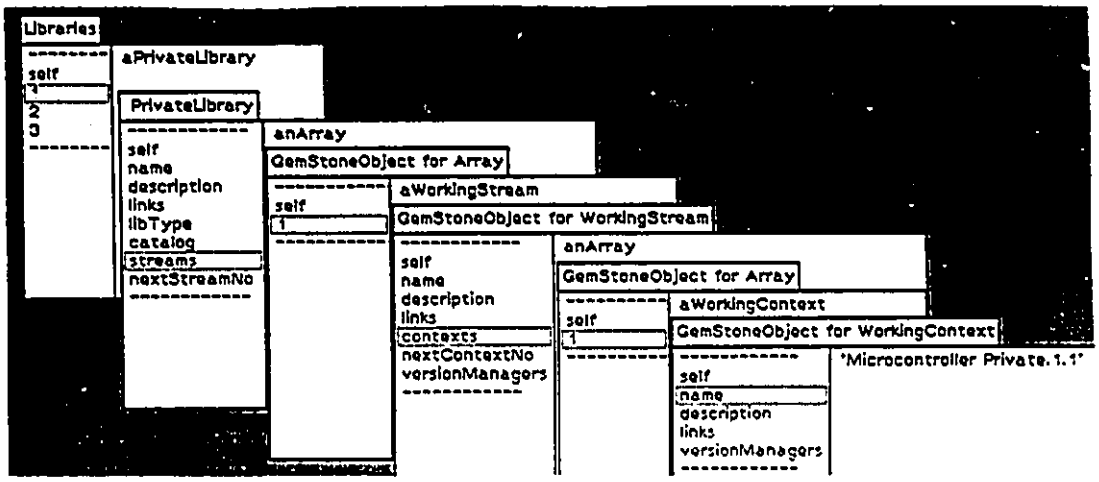


Figure 28 Libraries, Streams and Contexts

---

[1] When the message *libraries* is sent to an instance of the DataServer class, the class variable Libraries, which is an instance of a class Set, is returned to the message sender.

[2] When the expression within parenthesis is executed, an instance of a PrivateLibrary class, named "Microcontroller Private", is created. Next, the message *addStream:* with a string argument "Release 1.0" is sent to this instance. As a consequence, an instance of a class Stream is added to the attribute called "streams" which belongs to the private library called "Microcontroller Private" with a reason described as "Release 1.0". In addition, the variable nextStreamNo is incremented, and an instance of a Context class is added to the attribute "contexts" which belongs to the created instance of class Stream.

[3] When the message *libraries* is sent to an instance of the Data Server class, the query returns a set of libraries which can be accessed through the `data server`. Figure 28 is shows the contents of an instance of a PrivateLibrary class.

```
DataServer new setCurrents: 'Microcontroller Private' type: #Private
        streamNamed: 'Microcontroller Private.1'
        contextNamed: 'Microcontroller Private.1.1'."[1]"
```

"6 Query the current path setting"

```
DataServer new currentPath."[2]"
```

"7 Register in the DataServer those classes which will be used as the microcontroller parts"

```
| aDataServer |
aDataServer := DataServer new.
aDataServer defineClass: 'Microcontroller' asSubClassOf: Object withParts: #('cpu'
'memory' 'io') inDictionary: ThesisData. "[3]"
aDataServer defineClass: 'CPU' asSubClassOf: Object withParts: #('alu' 'controlUnit')
inDictionary: ThesisData.
aDataServer defineClass: 'ALU' asSubClassOf: Object withParts: #('registers' 'alucct')
inDictionary: ThesisData.
aDataServer defineClass: 'CU' asSubClassOf: Object withParts: #('cu') inDictionary:
ThesisData.
aDataServer defineClass: 'Memory' asSubClassOf: Object withParts: #('rom' 'ram')
inDictionary: ThesisData.
aDataServer defineClass: 'IO' asSubClassOf: Object withParts: #('serial' 'parallel')
inDictionary: ThesisData.
aDataServer defineClass: 'Register' asSubClassOf: Object withParts: #('register')
inDictionary: ThesisData.
aDataServer defineClass: 'ALUCCT' asSubClassOf: Object withParts: #('alucct')
inDictionary: ThesisData.
aDataServer defineClass: 'ROM' asSubClassOf: Object withParts: #('rom') inDictionary:
ThesisData.
aDataServer defineClass: 'RAM' asSubClassOf: Object withParts: #('ram') inDictionary:
ThesisData.
aDataServer defineClass: 'SerialPort' asSubClassOf: Object withParts: #('serial')
inDictionary: ThesisData.
aDataServer defineClass: 'ParallelPort' asSubClassOf: Object withParts: #('parallel')
inDictionary: ThesisData.
```

---

[1] When the message *setCurrents:streamNamed:contextNamed:* is sent to an instance of a DataServer class, the passed arguments are assigned to the class variables CurrentContext, CurrentStream, and CurrentLibrary, defined in the DataServer class.

[2] When the message *currentPath* is sent to an instance of the DataServer class, a string with the following format: " current library name/current stream name/current context name" is returned.

[3] When the message *defineClass:asSubClassOf:withParts:inDictionary:* is sent to an instance of a DataServer class, the following actions happen:

- At run time, a class is added to the OO-DBMS schema with the name passed as a parameter (e.g., Microcontroller). Microcontroller becomes a subclass of the passed parameter Object. In addition, the class Microcontroller will have as instance variables the names specified by the parameter: #('cpu' 'memory' 'io').
- The instance methods which can access and update the instance variables: cpu, memory and io, are generated automatically at run time.
- The metadata information required to track the evolution of the class Microcontroller is created

DataServer new defineClass: 'Document' asSubClassOf: Object withParts: #('title' 'abstract' 'glossary' 'symbolTable' 'tables' 'figures' 'introduction' 'body' 'conclusions' 'recommendations' 'references') inDictionary: ThesisData.
"After the execution of the above steps, a snapshot of the DataServer Browser, which shows the microcontroller parts, is given in Figure 29."

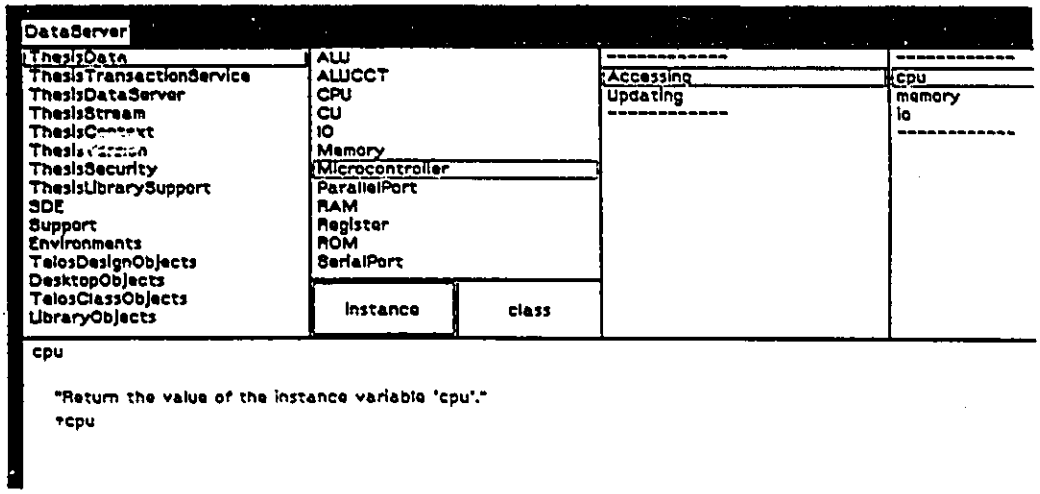


Figure 29 The Parts of a Microcontroller Class

"8 Query the current path, and perform some cleaning"

DataServer new currentPath.
"DataServer new currentContext resetAllVersionedObjects.
DataServer new currentStream resetAllVersionedObjects[1]"

"9 Create instances for the defined classes and store them into a library"

| anALU anALUCCT aCPU aCU anIO aMemory aMicrocontroller aParallelPort aRam aRegister aROM aSerialPort aDataServer |
"temporary variables"
aDataServer := DataServer new.
"For class Register create two instance versions"
aRegister := Register new.
aRegister register: Array new."[16]"
aDataServer put: aRegister why: 'Fast Register' asLeafOf: nil."[2]"

---

[1] These messages are available only to an administrator of the `data server`.

[2] When the message *put:why:asLeafOf:* is sent to a `data server` instance, it causes the following actions to be executed:

- First, the passed instance aRegister is converted to an instance of class VersionedObject by gathering appropriate information from the class of the aRegister instance. Specifically, the information hold in the instance aRegister is stored in the attribute parts of an instance of the class VersionedObject .

```
        aRegister := Register new.
        aRegister register: Array new.
        aDataServer put: aRegister why: 'Slow Register' asLeafOf: nil.
"For class ALUCCT, create two instance versions"
        anALUCCT := ALUCCT new.
        anALUCCT alucct: Array new.
        aDataServer put: anALUCCT why: 'Fast Circuitry' asLeafOf: nil.
        anALUCCT := ALUCCT new.
        anALUCCT alucct: Array new.
        aDataServer put: anALUCCT why: 'Slow Circuitry' asLeafOf: nil.
"For class RAM, create two instance versions"
        aRAM := RAM new.
        aRAM ram: Array new.
        aDataServer put: aRAM why: 'Fast RAM' asLeafOf: nil.
        aRAM := RAM new.
        aRAM ram: Array new.
        aDataServer put: aRAM why: 'Slow RAM' asLeafOf: nil.
"For class ROM, create two instance versions"
        aROM := ROM new.
        aROM rom: Array new.
        aDataServer put: aROM why: 'Fast ROM' asLeafOf: nil.
        aROM := ROM new.
        aROM rom: Array new.
        aDataServer put: aROM why: 'Slow ROM' asLeafOf: nil.
"For class SerialPort, create two instance versions"
        aSerialPort := SerialPort new.
        aSerialPort serial: Array new.
        aDataServer put: aSerialPort why: 'Fast SerialPort' asLeafOf: nil.
        aSerialPort := SerialPort new.
        aSerialPort serial: Array new.
        aDataServer put: aSerialPort why: 'Slow SerialPort' asLeafOf: nil.
"For class ParallelPort, create two instance versions"
        aParallelPort := ParallelPort new.
        aParallelPort parallel: Array new.
        aDataServer put: aParallelPort why: 'Fast ParallelPort' asLeafOf: nil.
        aParallelPort := ParallelPort new.
        aParallelPort parallel: Array new.
        aDataServer put: aParallelPort why: 'Slow ParallelPort' asLeafOf: nil.
"For class Memory, create three instance versions"
        aMemory := Memory new.
        aMemory rom: (VersionedName new: 'ROM.1').
        aMemory ram: (VersionedName new: 'RAM.1').
        aDataServer put: aMemory why: 'Fast Memory' asLeafOf: nil.
        aMemory := Memory new.
        aMemory rom: (VersionedName new: 'ROM.2').
        aMemory ram: (VersionedName new: 'RAM.2').
        aDataServer put: aMemory why: 'Slow Memory' asLeafOf: nil.
```

- Second, the instance of VersionedObject is added to the attribute versionedObjects of the context within the current path.

```
"From an existing instance version, derive another instance version"
        aMemory := DataServer new getInstanceFrom: 'Memory.1'. "Static Binding[1]"
        aMemory ram: (VersionedName new: 'RAM.2'). "Redefine the RAM"
        aDataServer put: aMemory why: 'Medium Speed Memory' asLeafOfName:
        'Memory.1'."[2]"
"For class ALU, create two instance versions"
        anAlu := ALU new.
        anAlu registers: (VersionedName new: 'Register.1').
        anAlu alucct: (VersionedName new: 'ALUCCT.1').
        aDataServer put: anAlu why: 'Fast ALU' asLeafOf: nil.
        anAlu := ALU new.
        anAlu registers: (VersionedName new: 'Register.2').
        anAlu alucct: (VersionedName new: 'ALUCCT.2').
        aDataServer put: anAlu why: 'Slow ALU' asLeafOf: nil.
"For class CU, create two instance versions"
        aCU := CU new.
        aCU cu: Array new.
        aDataServer put: aCU why: 'Fast CU' asLeafOf: nil.
        aCU := CU new.
        aCU cu: Array new.
        aDataServer put: aCU why: 'Slow CU' asLeafOf: nil.
"For class CPU, create three instance versions"
        aCPU := CPU new.
        aCPU alu: (VersionedName new: 'ALU.1').
        aCPU controlUnit: (VersionedName new: 'CU.1').
        aDataServer put: aCPU why: 'Fast CPU' asLeafOf: nil.
        aCPU := CPU new.
        aCPU alu: (VersionedName new: 'ALU.2').
        aCPU controlUnit: (VersionedName new: 'CU.2').
        aDataServer put: aCPU why: 'Slow CPU' asLeafOf: nil.
        aCPU := DataServer new getInstanceFrom: 'CPU.1'.
        aCPU alu: (VersionedName new: 'ALU.2').
        aDataServer put: aCPU why: 'CPU with Slow ALU and Fast CU' asLeafOfName:
        'CPU.1'.
"For class IO, create one instance version"
        anIO := IO new.
        anIO serial: (VersionedName new: 'SerialPort.1').
        anIO parallel: (VersionedName new: 'ParallelPort.1').
        aDataServer put: anIO why: 'Fast ALU' asLeafOf: nil.
"For class Microcontroller, create three instance versions"
        aMicrocontroller := Microcontroller new.
        aMicrocontroller cpu: (VersionedName new: 'CPU.1').
```

---

[1] When the *getInstanceFrom:* message is sent to an instance of a DataServer class, another message is sent to the current Context to search for the instance version, Memory.1. With the information available from the class of the instance version Memory.1 and the information stored in the instance version Memory.1, an instance of class Memory is created in the client workspace.

[2] When an instance version name is passed instead of nil (e.g., "Memory.1"), the instance aMemory is converted to an instance version and is linked to its parent instance version Memory.1 by means of a Successor/Predecessor relationship. When the last parameter is nil, linear versioning is performed instead.

```
        aMicrocontroller memory: (VersionedName new: 'Memory.1').
        aMicrocontroller io: (VersionedName new: 'IO.1').
        aDataServer put: aMicrocontroller why: 'Fast Microcontroller' asLeafOf: nil. "(1)"
        aMicrocontroller := Microcontroller new.
        aMicrocontroller cpu: (VersionedName new: 'CPU.2').
        aMicrocontroller memory: (VersionedName new: 'Memory.2').
        aMicrocontroller io: (VersionedName new: 'IO.1').
        aDataServer put: aMicrocontroller why: 'Slow Microcontroller' asLeafOf: nil.
        aMicrocontroller := Microcontroller new.
        aMicrocontroller cpu: (VersionedName new: 'CPU.1').
        aMicrocontroller memory: (VersionedName new: 'Memory')."[1]"
        aMicrocontroller io: (VersionedName new: 'IO.1').
        aDataServer put: aMicrocontroller why: 'Micro with Dynamic Binding Memory'
        asLeafOf: nil.
    "Commit the transaction "
        DataServer new commit. "To commit the transaction [2]"
"10 Inspect the current context data"
    DataServer new currentContext versionTrackers. "[3] see Figure 30"
```

[1] When the *memory* message is sent to an instance of a Microcontroller class (e.g., aMicrocontroller) the attribute memory is instantiated to a generic Memory part, instead of being instantiated to an specific instance version of Memory. Thus, when this instance is latter *put* to a private library, the memory attribute of the microcontroller instance version will be bound either to the latest or to a designated default instance version.

[2] When the *commit* message is sent to an instance of the DataServer class, the transaction for the current library is ended.

[3] When the message *versionTrackers* is sent to an instance of the Context class, a dictionary is returned to the sender. The dictionary has the class names as keys, and the holders of the corresponding instance versions as values.

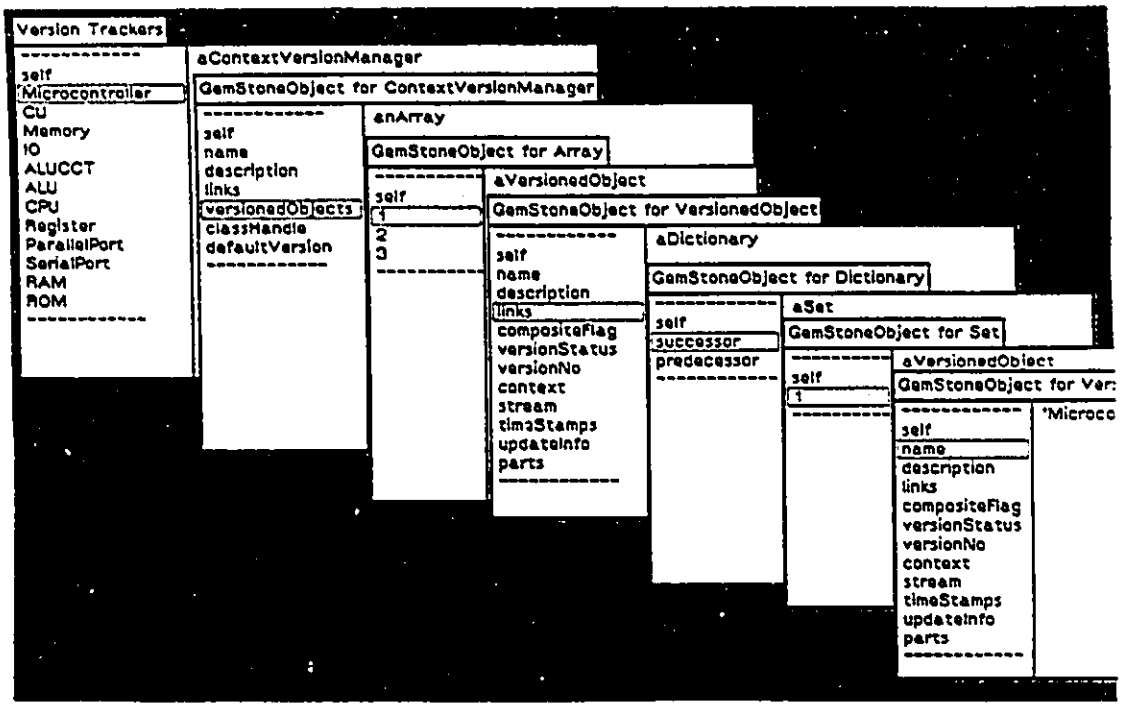


Figure 30 The Contents of a Context

"11 Get a versioned object"

"• Dynamic Binding"
DataServer new getVersionedObjectCopy: 'Memory'.
"• Static Binding"

DataServer new getVersionedObject: 'Microcontroller.1'.

"12 Print only the information related to a specific instance version"

"• Microcontroller.1"
Transcript cr; show: ((GemStone objectNamed: 'DataServer') gsnew gsgetVersionedObject: 'Microcontroller.1') gsprint asLocalObject;cr.

"• Memory.3"
Transcript cr; show: ((GemStone objectNamed: 'DataServer') gsnew gsgetVersionedObject: 'Memory.3') gsprint asLocalObject;cr.

"13 Query and print the transitive closure for an instance version."

"• Query the parts of Microcontroller.1"

(DataServer new getVersionedObject: 'Microcontroller.1') elements.

"• Print the transitive closure for the instance version Microcontroller.1."

Transcript cr; show: ((GemStone objectNamed: 'DataServer') gsnew gsprintVersion: 'Microcontroller.1' onStream: ((GemStone objectNamed: 'WriteStream') gson:

(SpecialGemStoneObjects at: #OString) gsnew)) asLocalObject: cr.

"• Print[1] the transitive closure for the instance version Microcontroller.2, which has a dynamic binding to the memory part. See Figure 32"

Transcript cr; show: ((GemStone objectNamed: 'DataServer') gsnew
gsprintVersion: 'Microcontroller.2' onStream:
((GemStone objectNamed: 'WriteStream') gson:
(SpecialGemStoneObjects at: #OString) gsnew)) asLocalObject; cr.

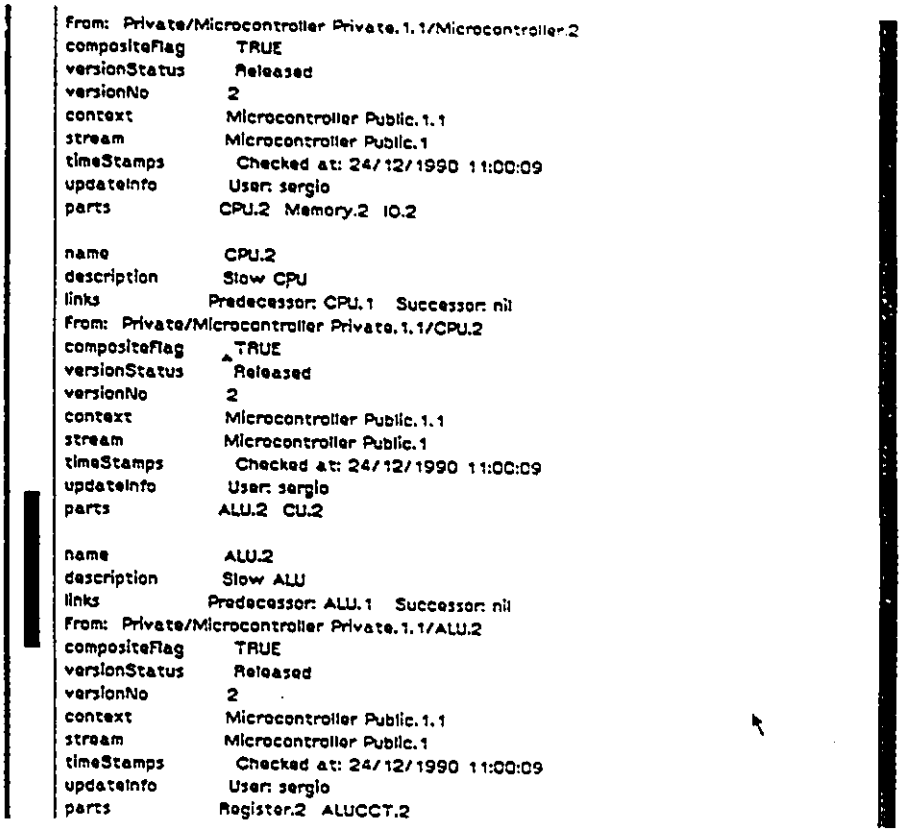


Figure 32 The Parts of an Instance Version: Microcontroller.2

"• Print the transitive closure for the instance version Microcontroller.3"

Transcript cr; show: ((GemStone objectNamed: 'DataServer') gsnew
gsprintVersion: 'Microcontroller.3' onStream:
((GemStone objectNamed: 'WriteStream') gson:
(SpecialGemStoneObjects at: #OString) gsnew)) asLocalObject; cr.

"14 Set a default (designated) instance version for the Memory part"

[1] The components of the instance version Microcontroller.2 are printed to the Smalltalk-80 Transcript.

DataServer new defaultVersionFor: 'Memory.2'."[1]"

"15 Query the closure[2] of an instance version. See Figure 33 [3]"

DataServer new getClosureFor: 'Microcontroller.3 '

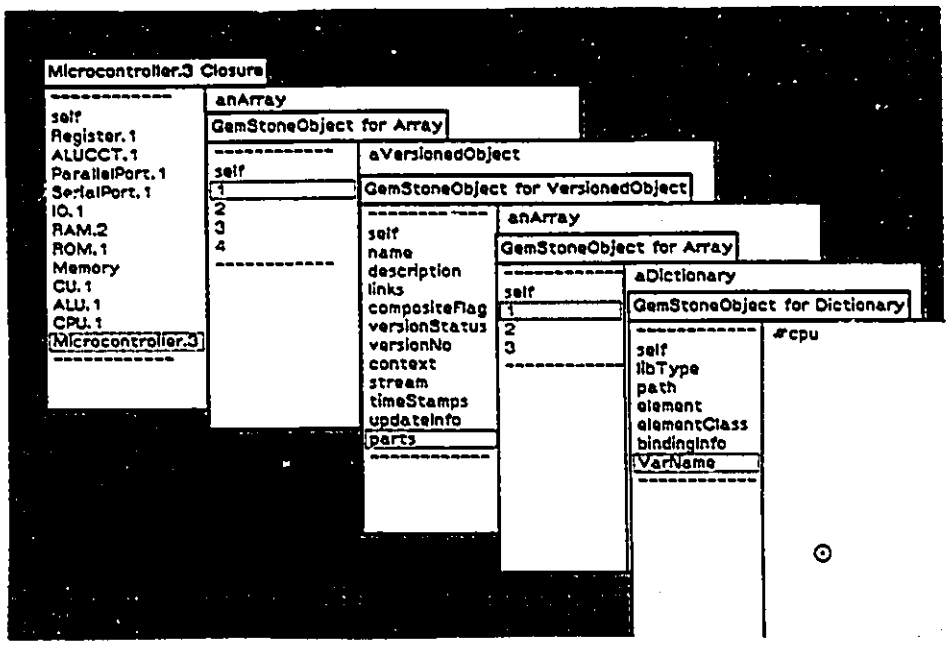


Figure 33 The Closure for the instance version: Microcontroller.3

---

[1]When the message *defaultVersionFor:* is sent to an instance of the DataServer class, a designated instance version is assigned to the current Context. The designated instance version is stored in the instance variable named "defaultVersion" of the instance of the Context class.

[2]When the message *getClosureFor:* is sent to an instance of the DataServer class a dictionary is returned. This dictionary contains the instance version name as keys, e.g. "Microcontroller.3", and an array, which contains four elements, as the corresponding value. The first element of the array is the corresponding instance version, the second element is the version number, the third element is the version number which will be given once it is stored in a target context, and the fourth element is the class to which the instance version refers, e.g. "Microcontroller".

[3] An instance version (shown as aVersionedObject on Figure 33) contains an instance variable named "parts". The "parts" attribute is an array of dictionaries. Each element of the array corresponds to an instance variable of the instance version (e.g. cpu, memory, and io). For each instance variable there is a dictionary. This dictionary provides the following information:

- *libType*:: The library type, i.e. private, semi-public or public.
- *path*: The path where a part is stored.
- *element*:. The instance variable which stores either a primitive object (e.g an instance of the Array class) or a reference to another instance version (i.e. by means of an instance of class VersionedName).
- *bindingInfo*:. provides binding information (i.e. dynamic binding or static binding).
- *VarName*: The instance variable name, e.g. cpu (shown as #cpu).

"16 Check-in the instance versions Microcontroller.1, Microcontroller.2, and Microcontroller.3 to a public library from the current path"

```
DataServer new
        checkIn: 'Microcontroller.1'
        fromLibrary: 'Microcontroller Private'  withType: #Private
        fromStream: 'Microcontroller Private.1' fromContext: 'Microcontroller Private.1.1'
        toLibrary: 'Microcontroller Public' withType: #Public
        toStream: 'Microcontroller Public.1' toContext: 'Microcontroller Public.1.1'
        asLeafOf: nil withChecking: true.
DataServer new
        checkIn: 'Microcontroller.2'
        fromLibrary: 'Microcontroller Private'  withType: #Private
        fromStream: 'Microcontroller Private.1' fromContext: 'Microcontroller Private.1.1'
        toLibrary: 'Microcontroller Public' withType: #Public
        toStream: 'Microcontroller Public.1' toContext: 'Microcontroller Public.1.1'
        asLeafOf: nil withChecking: true.
DataServer new
        checkIn: 'Microcontroller.3'
        fromLibrary: 'Microcontroller Private'  withType: #Private
        fromStream: 'Microcontroller Private.1' fromContext: 'Microcontroller Private.1.1'
        toLibrary: 'Microcontroller Public' withType: #Public
        toStream: 'Microcontroller Public.1' toContext: 'Microcontroller Public.1.1'
        asLeafOf: nil withChecking: true. "3"
```

"17 Set the current path to the public library"

```
        | aDataServer |
        aDataServer:= DataServer new.
        aDataServer setCurrents: 'Microcontroller Public' type: #Public
                streamNamed: 'Microcontroller Public.1'
                contextNamed: 'Microcontroller Public.1.1'.
        "Check the current path"
        DataServer new currentPath.
```

"18 Query instance versions stored in the public library"

```
DataServer new getVersionedObject: 'Microcontroller.3'. "Static Binding"
DataServer new getVersionedObject: 'Memory'. "Dynamic Binding"
"DataServer new getVersionedObject: 'ROM.2'."
```

"19 Print the information for the promoted version. Use GSI."

```
Transcript cr; show: ( (GemStone objectNamed: 'DataServer') gsnew
        gsprintVersion: 'Microcontroller.2'
        onStream: (GemStone objectNamed: 'WriteStream') gson:
        (SpecialGemStoneObjects at: #OString) gsnew) ) asLocalObject; cr.
```

"20 Check out instance versions from a Public library to a Semi-Public library[1]"

```
DataServer new
```

---

[1] When the message *checkOut....* is sent to an instance of the Dataserver class, a copy of the instance version Microcontroller.1 from the Public library is installed into the SemiPublic library named "Microcontroller SemiPublic".

```
        checkOut: 'Microcontroller.1'
        fromLibrary: 'Microcontroller Public'  withType: #Public
        fromStream: 'Microcontroller Public.1' fromContext: 'Microcontroller Public.1.1'
        toLibrary: 'Microcontroller SemiPublic' withType: #SemiPublic
        toStream: 'Microcontroller SemiPublic.1' toContext: 'Microcontroller
        SemiPublic.1.1' asLeafOf: nil.
    DataServer new
        checkOut: 'Microcontroller.3'
        fromLibrary: 'Microcontroller Public'  withType: #Public
        fromStream: 'Microcontroller Public.1' fromContext: 'Microcontroller Public.1.1'
        toLibrary: 'Microcontroller SemiPublic' withType: #SemiPublic
        toStream: 'Microcontroller SemiPublic.1' toContext: 'Microcontroller
        SemiPublic.1.1' asLeafOf: nil.
    "Check-out instance versions from a Public to a Private library"
    DataServer new
        checkOut: 'Microcontroller.3'
        fromLibrary: 'Microcontroller Public'  withType: #Public
        fromStream: 'Microcontroller Public.1' fromContext: 'Microcontroller Public.1.1'
        toLibrary: 'Microcontroller Private' withType: #Private
        toStream: 'Microcontroller Private.1' toContext: 'Microcontroller Private.1.1'
        asLeafOf: nil.
"21 Set the path to the private library and query the last microcontroller instance version"
    DataServer new setCurrents: 'Microcontroller Private' type: #Private
        streamNamed: 'Microcontroller Private.1'
        contextNamed: 'Microcontroller Private.1.1'.
    Transcript cr; show: ( (GemStone objectNamed: 'DataServer') gsnew
        gsprintVersion: 'Microcontroller'
        onStream: ((GemStone objectNamed: 'WriteStream') gson:
        (SpecialGemStoneObjects at: #OString) gsnew) ) asLocalObject; cr.
"22 Get an instance version from a semi-public library "
        | aDataServer |
        aDataServer:= DataServer new.
        aDataServer setCurrents: 'Microcontroller SemiPublic' type: #SemiPublic
            streamNamed: 'Microcontroller SemiPublic.1'
            contextNamed: 'Microcontroller SemiPublic.1.1'.
        "DataServer new currentContext resetAllVersionedObjects.
        DataServer new currentStream resetAllVersionedObjects"
        DataServer new getVersionedObject: 'Microcontroller.1'. "Static Binding"
        DataServer new getVersionedObject: 'Memory'. "Dynamic Binding"
"23 Get an instance for the instance version Memory.1"
    DataServer new getInstanceFrom: 'Memory.1'. "Static Binding"
"24 Class Ownership: add an owner of a class"
    "• Make the user named sergio as the owner of class Microcontroller"
    DataServer new addClass: Microcontroller owner: #sergio.
    "• Change the class ownership"
```

```
DataServer new newOwner: #moshe forClass: Microcontroller.

"• Make the user named sergio the owner of all the classes stored in the ThesisData
dictionary"
| aDataServer |
aDataServer := DataServer new.

DataServer new addOwner: aDataServer myId forClassesIn: #ThesisData.

"25 Query which classes are owned by sergio"

DataServer new classesOwnedBy: DataServer new myId.

"26 Query the owner of class Microcontroller"

(DataServer new ownerForClass: Microcontroller) = DataServer new myId."[1]"

"27 Compare the values of two instance versions"

DataServer new setCurrents: 'Microcontroller Private' type: #Private
        streamNamed: 'Microcontroller Private.1'
        contextNamed: 'Microcontroller Private.1.1'.
VersionedObject printDeltaBetween: 'Memory.3' and: 'Memory.1'.
VersionedObject existDeltaBetween: 'Memory.3' and: 'Memory.1'.
VersionedObject printDeltaBetween: 'Memory.1' and: 'Memory.1'.
VersionedObject existDeltaBetween: 'Memory.1' and: 'Memory.1'.

"28 One level check-in of an instance from the user workspace to a public library."

| aMicrocontroller |
aMicrocontroller := Microcontroller new.
aMicrocontroller cpu: (VersionedName new: 'CPU.1').
aMicrocontroller memory: (VersionedName new: 'Memory.1').
aMicrocontroller io: (VersionedName new: 'IO.1').
DataServer new OneLevelCheckIn: aMicrocontroller why: 'Fast Microcontroller'
asLeafOfName: 'Microcontroller.2' toLibrary: 'Microcontroller Public' withType: #Public
toStream: 'Microcontroller Public.1' toContext: 'Microcontroller Public.1.1' withChecking:
true.

"29 Query the virtual path of an instance version"

DataServer new setCurrents: 'Microcontroller Public' type: #Public
streamNamed: 'Microcontroller Public.1'
contextNamed: 'Microcontroller Public.1.1'.

"• Query the virtual path of an instance version"

(DataServer new getVersionedObject: 'Microcontroller.1') virtualPath.

"• Print the virtual path of an instance version"

(DataServer new getVersionedObject: 'Microcontroller.1') printVirtualPath."See Figure
34).
```

---

[1] When the message *myId* is sent to an instance of the DataServer class, a client identifier is returned.

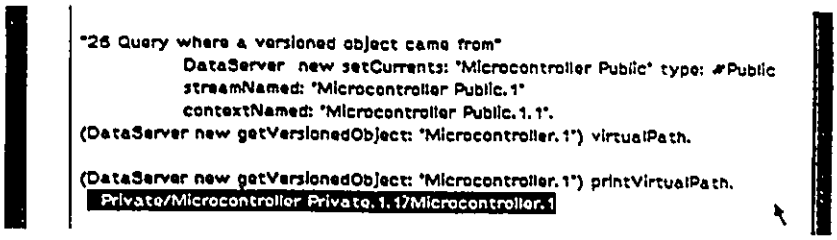
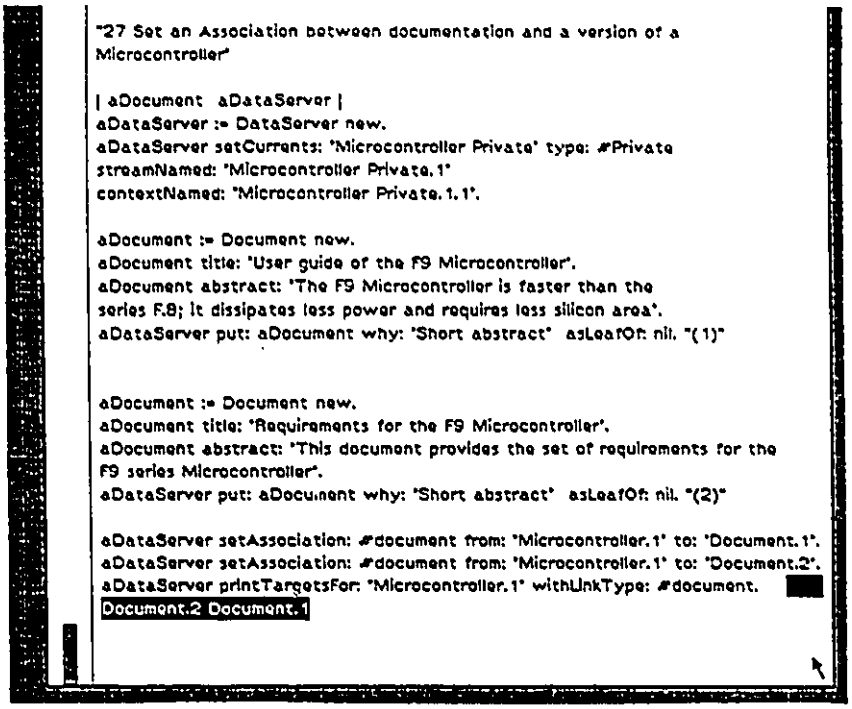


Figure 34 The Virtual Path of an Instance Version

"30. Set an association between a documentation instance version and an instance version of a Microcontroller"

```
| aDocument aDataServer |
aDataServer := DataServer new.
aDataServer setCurrents: 'Microcontroller Private' type: #Private
        streamNamed: 'Microcontroller Private.1'
        contextNamed: 'Microcontroller Private.1.1'.
aDocument := Document new.
aDocument title: 'User guide of the F9 Microcontroller'.
aDocument abstract: 'The F9 Microcontroller is faster than the series F.8; it dissipates less
power and requires less silicon area'.
aDataServer put: aDocument why: 'Short abstract' asLeafOf: nil.
aDocument := Document new.
aDocument title: 'Requirements for the Microcontroller F9'.
aDocument abstract: 'This document provides the set of requirements for the series F9 of a
Microcontroller'.
aDataServer put: aDocument why: 'Short abstract' asLeafOf: nil.
aDataServer setAssociation: #document from: 'Microcontroller.1' to: 'Document.1'.
aDataServer setAssociation: #document from: 'Microcontroller.1' to: 'Document.2'.
aDataServer printTargetsFor: 'Microcontroller.1' withLinkType: #document."See Figure
35).
```



```
"27 Set an Association between documentation and a version of a
Microcontroller"

| aDocument  aDataServer |
aDataServer := DataServer new.
aDataServer setCurrents: 'Microcontroller Private' type: #Private
streamNamed: 'Microcontroller Private.1'
contextNamed: 'Microcontroller Private.1.1'.

aDocument := Document new.
aDocument title: 'User guide of the F9 Microcontroller'.
aDocument abstract: 'The F9 Microcontroller is faster than the
series F.8; it dissipates less power and requires less silicon area'.
aDataServer put: aDocument why: 'Short abstract'  asLeafOf: nil. "(1)"


aDocument := Document new.
aDocument title: 'Requirements for the F9 Microcontroller'.
aDocument abstract: 'This document provides the set of requirements for the
F9 series Microcontroller'.
aDataServer put: aDocument why: 'Short abstract'  asLeafOf: nil. "(2)"

aDataServer setAssociation: #document from: 'Microcontroller.1' to: 'Document.1'.
aDataServer setAssociation: #document from: 'Microcontroller.1' to: 'Document.2'.
aDataServer printTargetsFor: 'Microcontroller.1' withLinkType: #document.
Document.2 Document.1
```

Figure 35 An Association Between Instance Versions

"End of test suite "

Appendix E

Source Code Used for the Data Server Prototype

The source code of the classes used in the implementation of the prototype is given below in alphabetical order.

```
! Class 'DataServer'
Object subclass: 'DataServer'
  instVarNames: #()
  classVars: #( 'Libraries' 'Catalog' 'Services' 'BackupContext' 'BackupStream' 'BackupLibrary' 'CurrentContext'
'CurrentStream' 'CurrentLibrary')
  poolDictionaries: #[]
  inDictionary: ThesisDataServer
  constraints: #[]
  isInvariant: false
! Instance Category 'Printing'
category: 'Printing'
method: DataServer
printVersion: aString onStream: aStream
        "Recursivelly, in order, prints the closure of a given version"
        | aDataServer       aVO  wichElement |

        aString isNil
                ifFalse:
                [
                aDataServer := DataServer new.
                aVO := DataServer new getVersionedObject: aString.
                aVO isNil
                        ifFalse:
                        [
                        aStream nextPutAll: aVO print.
                        aStream nextPut: Character lf.
                        aVO parts
                                do:
                                [:i |
                                (((i at: #bindingInfo) = #Static) or: [(i at: #bindingInfo) = #Dynamic])
                                        ifTrue:
                                        [ wichElement := (i at: #element) asString.
                                        self printVersion: wichElement onStream: aStream]
                                ] ].

        ^aStream contents
%
method: DataServer
printTargetsFor: aVersionName withLinkType: aLinkType
        | aVO |
        aVO := self getVersionedObject: aVersionName.
        aVO isNil ifTrue: [^' '].
        ^aVO printTargetsFor: aLinkType.
%
! Instance Category 'Get'
category: 'Get'
method: DataServer
getInstanceFrom: aVersionName
        | aVO aClass anInstance msg |
        aVO := self getVersionedObject: aVersionName.
        aVO isNil
                ifTrue: [^ nil].
        aClass := aVO itsClass.
```

```
        anInstance := aClass new.
        anInstance isNil
                ifTrue: [^ nil ].
        aVO parts
                do:
                [:each |
                        msg := ((each at: #VarName) asString + ':') asSymbol.
                        anInstance perform: msg with: (each at: #element).
                ].
        ^anInstance
%
! Instance Category 'Put'
category: 'Put'
method: DataServer
put: anInstance why: aString asLeafOf: aParentVersionedObject
        ^ self currentContext put: anInstance reason: aString asLeafOf: aParentVersionedObject.
%
method: DataServer
put: anInstance reason: aString asLeafOf: aParentVersionedObject
        ^ self currentContext put: anInstance reason: aString asLeafOf: aParentVersionedObject.
%
method: DataServer
put: anInstance why: aString asLeafOfName: aName
        | aVO |
        aName isNil
                ifTrue: [aVO := nil]
                ifFalse: [aVO := self getVersionedObject: aName].
        ^ self currentContext put: anInstance reason: aString asLeafOf: aVO.
%
! Instance Category 'Accessing'
category: 'Accessing'
method: DataServer
Libraries
  "Return the value of the instance variable 'Libraries'."
  ^Libraries
%
method: DataServer
Catalog
  "Return the value of the instance variable 'Catalog'."
  ^Catalog
%
method: DataServer
Services
  "Return the value of the instance variable 'Services'."
  ^Services
%
method: DataServer
CurrentContext
  "Return the value of the instance variable 'CurrentContext'."
  ^CurrentContext
%
method: DataServer
CurrentStream
  "Return the value of the instance variable 'CurrentStream'."
  ^CurrentStream
%
method: DataServer
CurrentLibrary
  "Return the value of the instance variable 'CurrentLibrary'."
  ^CurrentLibrary
%
method: DataServer
BackupContext
  "Return the value of the instance variable 'BackupContext'."
```

```
  ^BackupContext
%
method: DataServer
BackupLibrary
  "Return the value of the instance variable 'BackupLibrary'."
  ^BackupLibrary
%
! Instance Category 'Updating'
category: 'Updating'
method: DataServer
Catalog: newValue
  "Modify the value of the instance variable 'Catalog'."
  Catalog:= newValue
%
method: DataServer
addEntry: anAssociation
  Catalog add: anAssociation.
%
method: DataServer
addService: anAssociation
  Services add: anAssociation
%
method: DataServer
addLibrary: newLibrary
  Libraries add: newLibrary
%
method: DataServer
currentLibrary: aName type: aType
  "Modify the value of the class variable 'CurrentLibrary'."
  CurrentLibrary:= self libraryNamed: aName type: aType.
%
method: DataServer
currentStream: aName
  CurrentStream:= self streamNamed: aName
%
method: DataServer
currentContext: aName
  CurrentContext:= self contextNamed: aName
%
method: DataServer
defaultVersionFor: aName

        | aVM aString |
        aString := VersionedObject string: aName until: '.'.
        aVM := self currentContext getVM: aString.
        aVM defaul:Version: aName.
%
method: DataServer
setCurrents: aLibName2 type: aSymbol2
streamNamed: aStreamName2
contextNamed: aContextName2

        currentLibrary := self libraryNamed: aLibName2 type: aSymbol2.
        currentStream := self streamNamed: aStreamName2.
        currentContext := self contextNamed: aContextName2.
%
method: DataServer
setBackups
        BackupLibrary := CurrentLibrary.
        BackupStream := CurrentStream.
        BackupContext := CurrentContext.
%
method: DataServer
restoreCurrents
```

```
        currentLibrary := BackupLibrary.
        currentStream := BackupStream.
        currentContext := BackupContext.
%
! Instance Category 'Quering'
category: 'Quering'
method: DataServer
libraryNamed: aName type: aType

        "Returns a library named aName with type aType"

        ^self Libraries
                detect:
                        [:each| (each name = aName) and: [aType = each libType ]]
                ifNone: []
%
method: DataServer
streamNamed: aName
        | aLibrary |
        aLibrary := DataServer new currentLibrary.
        aLibrary isNil
                ifFalse:
                [^ aLibrary streams
                        detect:
                                [:each| (each name = aName) ]
                        ifNone: []
                ].
        ^nil.
%
method: DataServer
contextNamed: aName
        | aStream |
        aStream := DataServer new currentStream.
        aStream isNil
                ifFalse:
                [^ aStream contexts
                        detect:
                                [:each| (each name = aName) ]
                        ifNone: []
                ].
        ^nil.
%
method: DataServer
anyCurrentIsNil
((currentContext isNil or: [currentStream isNil]) or: [currentLibrary isNil])
         ifTrue: [^true]
         ifFalse: [^false]
%
method: DataServer
getVersionedObject: aVersionName
        ^(self currentContext getVersionedObject: aVersionName).
%
method: DataServer
getVersionedObjectCopy: aVersionName
        ^(self currentContext getVersionedObject: aVersionName) copy.
%
method: DataServer
getClassesDefinedIn: aDictionary
        "DataServer new getClassesDefinedIn: #ThesisData"
        ^ ((AllUsers userWithId: self myID) resolveSymbol: aDictionary) value
                selectAssociations:
                        [:each |
                                each key ~= aDictionary]
```

```
%
method: DataServer
currentPath
        | lib |
        "Returns the: 'LibraryType/LibraryName.Stream.Context'"
        lib := self currentLibrary.
        ^("lib name +'.'+" lib libType asString +'/'+
                "self currentStream name +'/'+"
                self currentContext name)
%
! Instance Category 'Promote a Version (Check In)'
category: 'Promote a Version (Check In)'
method: DataServer
OneLevelCheckIn: anInstance why: aString asLeafOfName: aName
toLibrary: aLibName2 withType: aSymbol2
toStream: aStreamName2 toContext: aContextName2
withChecking: aBoolean
        | aVO id toStream |
        self setBackups.
        self setCurrents: aLibName2 type: aSymbol2
      streamNamed: aStreamName2
      contextNamed: aContextName2.
        aName isNil
                ifTrue: [aVO := nil]
                ifFalse: [aVO := self getVersionedObject: aName].

        self currentStream addVersionManagerforClass: anInstance class description: ''.
        id := self currentContext put: anInstance reason: aString asLeafOf: aVO.
        self restoreCurrents.
        ^id
%
method: DataServer
checkIn: aVersionName
fromLibrary: aLibName withType: aSymbol
fromStream: aStreamName fromContext: aContextName
toLibrary: aLibName2 withType: aSymbol2
toStream: aStreamName2 toContext: aContextName2
asLeafOf: aVersionNameLeaf
withChecking: aBoolean
        "This method requires that the DataServer has its
        current pointers set to aLibName aStreamName and aContextName"
        "if the boolean is set to true then some semantic checking
         will be done before commiting the version"
        | fromLib toLib fromStream toStream fromContext toContext aTable table2
          aVO aClass aLeaf oldName aPath |
        self setBackups.
        fromLib := self libraryNamed: aLibName type: aSymbol.
        toLib :=  self libraryNamed: aLibName2 type: aSymbol2.
        fromStream := fromLib streamNamed: aStreamName.
   toStream := toLib streamNamed: aStreamName2.
        fromContext := fromStream contextNamed: aContextName.
        toContext := toStream contextNamed: aContextName2.
                self setCurrents: aLibName  type: aSymbol
                        streamNamed: aStreamName
                        contextNamed: aContextName . "Sets the currents to the source"
                aPath := self currentPath.
        (fromLib isNil or: [ toLib isNil or: [ fromStream isNil or: [toStream isNil
         or: [ fromContext isNil or: [ toContext isNil ]]]]])
                        ifTrue: [  self restoreCurrents. ^#failed].
        ">>>>>>>>>> Make the correction table <<<<<<<<<<"
        aTable := fromStream makeTableFrom: fromContext
        toStream: toStream
        toContext: toContext
        withVersion: aVersionName
```

```
        withTable: Dictionary new.
        ">>>>>>>>> Correct the parts of each version to be checked in
                                                and check them in to the target <<<<<<<<<<"
        self setCurrents: aLibName2 type: aSymbol2
                        streamNamed: aStreamName2
                        contextNamed: aContextName2. "Sets the currents to the target"
        self anyCurrentIsNil
                ifTrue: [  self restoreCurrents. ^#failed ].
        table2 := WorkingContext correctPartsFor: aVersionName accordingTranslation: aTable.
        table2
                doValues:
                [:each |
                        aVO := each at: 1.
                        oldName := aVO name.
                        aVO resetLinks. "Update the links"
                        aVO addSourcePath: aPath.
                        aVO name: (VersionedObject onlyName: oldName). "Change the VO name"
                        aClass := each at: 4.
                        toStream addVersionManagerforClass: aClass description: ".
                        aVersionName = oldName "The parenth is only valid for the given version"
                                ifTrue:
                                [ aVersionNameLeaf isNil
                                        ifFalse: [  aLeaf := self getVersionedObject: aVersionNameLeaf ]
                                        ifTrue:  [ aLeaf := nil ].
                                        toContext addVersionedObject: aVO reason: aVO description asLeafOf:
aLeaf ]
                                ifFalse: [toContext addVersionedObject: aVO reason: aVO description asLeafOf: nil ] ].
        self restoreCurrents.
        ^#succeeded.
%
! Instance Category 'Check Out a version'
category: 'Check Out a version'
method: DataServer
checkOut: aVersionName
fromLibrary: aLibName withType: aSymbol
fromStream: aStreamName fromContext: aContextName
toLibrary: aLibName2 withType: aSymbol2
toStream: aStreamName2 toContext: aContextName2
        ^self checkIn: aVersionName
                fromLibrary: aLibName withType: aSymbol
                fromStream: aStreamName fromContext: aContextName
                toLibrary: aLibName2 withType: aSymbol2
                toStream: aStreamName2 toContext: aContextName2
                withChecking: false.
%
method: DataServer
checkOut: aVersionName
fromLibrary: aLibName withType: aSymbol
fromStream: aStreamName fromContext: aContextName
toLibrary: aLibName2 withType: aSymbol2
toStream: aStreamName2 toContext: aContextName2
asLeafOf: aLeafName

        ^self checkIn: aVersionName
                fromLibrary: aLibName withType: aSymbol
                fromStream: aStreamName fromContext: aContextName
                toLibrary: aLibName2 withType: aSymbol2
                toStream: aStreamName2 toContext: aContextName2
                asLeafOf: aLeafName
                withChecking: false.
%
! Instance Category 'Class Ownership'
category: 'Class Ownership'
method: DataServer
```

```
addClass: aClass owner: aSymbol
        ^SecurityData addClass: aClass owner: aSymbol
%
method: DataServer
ownerForClass: aClass
        ^ SecurityData ownerForClass: aClass
%
method: DataServer
classesOwnedBy: aUserId
        ^SecurityData classesOwnedBy: aUserId
%
method: DataServer
addOwner: userId forClassesIn: aDictionary
        | aSymbolDict err |
        aSymbolDict := self getClassesDefinedIn: aDictionary.
        err := String new.
        aSymbolDict values
                do: [:each |
                                (SecurityData existClass: each)
                                        ifTrue: [ err := err + ' '+ each name] ].
        err isEmpty
        ifTrue:
                [ aSymbolDict values
                        do: [:each |
                                        self addClass: each owner: userId ].
                ^#done]
        ifFalse: [^('Classe(s): ' + err +  ' are owned')]
%
method: DataServer
newOwner: aUserId forClass: aClass
        SecurityData newOwner: aUserId forClass: aClass
%
! Instance Category 'Defining a Class'
category: 'Defining a Class'
method: DataServer
defineClass: aString asSubClassOf: aClass withParts: anArray inDictionary: aDict

        self currentStream
                defineClass: aString asSubClassOf: aClass withParts: anArray inDictionary: aDict
%
! Instance Category 'User information'
category: 'User information'
method: DataServer
myID
        ^System myUserProfile userID.
%
! Instance Category 'Closure'
category: 'Closure'
method: DataServer
getClosureFor: aVersionName
        ^self currentStream
        makeTableFrom: self currentContext
        toStream: self currentStream
        toContext: self currentContext
        withVersion: aVersionName
        withTable: Dictionary new.
%
! Instance Category 'Integrity Checking'
category: 'Integrity Checking'
! Instance Category 'Browser'
category: 'Browser'
method: DataServer
addDictionaryToSymbolList: aDictionary
        | aNewDict |
```

```
        aNewDict := System myUSerProfile createDictionary: aDictionary.
        System myUserProfile insertDictionary: aNewDict at: 1.
%
! Instance Category 'Commit - Abort'
category: 'Commit - Abort'
method: DataServer
commit
        ^TransactionManager new commit.
%
method: DataServer
abort
        ^TransactionManager new abort.
%
! Instance Category 'Unused'
category: 'Unused'
method: DataServer
checkIn: aVersionName
fromLibrary: aLibName withType: aSymbol
fromStream: aStreamName fromContext: aContextName
toLibrary: aLibName2 withType: aSymbol2
toStream: aStreamName2 toContext: aContextName2
withChecking: aBoolean
        "This method requires that the DataServer has its
        current pointers set to aLibName aStreamName and aContextName"
        "if the boolean is set to true then some semantic checking
         will be done before commiting the version"
        | fromLib toLib fromStream toStream fromContext toContext aTable table2
          aVO aClass |
        self setBackups.
        fromLib := self libraryNamed: aLibName type: aSymbol.
        toLib :=  self libraryNamed: aLibName2 type: aSymbol2.
        fromStream := fromLib streamNamed: aStreamName.
    toStream := toLib streamNamed: aStreamName2.
        fromContext := fromStream contextNamed: aContextName.
        toContext := toStream contextNamed: aContextName2.
        (fromLib isNil or: [ toLib isNil or: [ fromStream isNil or: [toStream isNil
         or: [ fromContext isNil or: [ toContext isNil ]]]]])
                        ifTrue: [  self restoreCurrents. ^#failed].
        ">>>>>>>>> Make the correction table <<<<<<<<<<"
        aTable := fromStream makeTableFrom: fromContext
        toStream: toStream
        toContext: toContext
        withVersion: aVersionName
        withTable: Dictionary new.
        ">>>>>>>>> Correct the parts of each version to be checked in
                                                and check them in to the target <<<<<<<<<<"
        self setCurrents: aLibName2 type: aSymbol2
                        streamNamed: aStreamName2
                        contextNamed: aContextName2. "Sets the currents to the target"
        self anyCurrentIsNil
                ifTrue: [  self restoreCurrents. ^#failed ].
        table2 := WorkingContext correctPartsFor: aVersionName accordingTranslation: aTable.
        table2
                doValues:
                [:each |
                        aVO := each at: 1.
                        aVO name: (VersionedObject onlyName: aVO name). "Change the VO name"
                        aVO resetLinks. "Update the links"
                        aClass := each at: 4.
                        toStream addVersionManagerforClass: aClass description: ''.
                        toContext addVersionedObject: aVO reason: aVO description asLeafOf: nil ].
        self restoreCurrents.
        ^#succeeded.
%
```

```
! Instance Category 'Associations (Links)'
category: 'Associations (Links)'
method: DataServer
setAssociation: aType from: aVersionedName to: aVersionedName2
        | aVO aVo2 |
        aVO := self getVersionedObject: aVersionedName.
        aVO isNil ifTrue: [^#failed].
        aVO2 := self getVersionedObject: aVersionedName2.
        aVO2 isNil ifTrue: [^#failed].
        ^ (aVO userDefinedAssociation: aType toTarget: aVO2)
%
method: DataServer
targetsFor: aLinkType
%
method: DataServer
targetsFor: aVersionName withLink: aType

        | aVO |
        aVO := self getVersionedObject: aVersionName.
        aVO isNil ifTrue: [^ nil].
        ^ aVO targetsFor: aType
%
! Class Category 'Initializing the Data Server'
category: 'Initializing the Data Server'
classmethod: DataServer
initDataServer
        Libraries := Set new.
        Catalog := Dictionary new.
        Services := Dictionary new.
        CurrentLibrary := nil.
        CurrentStream := nil.
        CurrentContext := nil.
        BackupLibrary := nil.
        BackupStream := nil.
        BackupContext := nil.
        SecurityData reset.
%
! Class Category 'creating'
category: 'creating'
classmethod: DataServer
new
        ^super new
%
category: 'initializing'
method: VersionedObject
init
        | aDataServer |
        aDataServer := DataServer new.
        compositeFlag := false.
        versionStatus := #Temporary.
        versionNo := 0.
        context := aDataServer currentContext.
        timeStamps := nil.
        updateInfo := nil.
        parts := Array new.
        stream := aDataServer currentStream.
%
! Class 'Library'
SecurityEntity subclass: 'Library'
  instVarNames: #( 'libType' 'catalog' 'streams'
          'nextStreamNo')
  classVars: #( 'CurrentLibrary')
  poolDictionaries: #[]
  inDictionary: ThesisLibrarySupport
```

```
 constraints: #[]
 isInvariant: false
%
! Instance Category 'initialization'
category: 'initialization'
method: Library
initialize
        catalog := Dictionary new.
        streams := Array new.
        nextStreamNo := 1.
%
! Instance Category 'accessing'
category: 'accessing'
method: Library
streams
        ^streams
%
method: Library
catalog
        ^catalog
%
method: Library
nextStreamNo
        ^nextStreamNo
%
method: Library
libType
        ^libType
%
! Instance Category 'updating'
category: 'updating'
method: Library
libType: aSymbol
        libType := aSymbol
%
method: Library
removeStream: aStream
        | aSet |
        aSet := self streams detect: [:i | i name = aStream name] ifNone: [ ].
        aSet isNil
                ifFalse: [self streams remove: aSet ifAbsent: [] ].
%
method: Library
incNextStreamNo
        nextStreamNo := nextStreamNo + 1.
%
method: Library
resetNextStreamNo
        nextStreamNo := 1.
%
method: Library
getStreamName
        ^(self name +'.'+ nextStreamNo asString)
%
method: Library
addStream: aReason
        | aWorkingStream |
        "(Library name: 'Micro' type: #Public description: 'Design of the 68000 micro') addStream: 'Release1.0'"
        aWorkingStream := (WorkingStream new name: self getStreamName) description: aReason.
        self streams add: aWorkingStream.
        self incNextStreamNo.
        aWorkingStream newContext: aReason.
%
! Instance Category 'quering'
```

```
category: 'quering'
method: Library
getStreamAt: aNumber

                    " ((Library name: 'Micro.1' type: #Public description: 'Test Library') addStream: 'Stream test')
getStreamAt: 1"

                    (self streams size >= aNumber and: [ aNumber > 0])
                            ifTrue: [^ self streams at: aNumber].
                    ^ nil.
%
method: Library
streamNamed: aName
        ^ self streams
                    detect:
                            [:each| (each name = aName) ]
                     ifNone: []
%
! Class Category 'creating'
category: 'creating'
classmethod: Library
name: libName type: aSymbol description: aString
        ^((super new initialize name: libName) libType: aSymbol) description: aString
%
! Class 'LibraryVersionManager'
VersionManager subclass: 'LibraryVersionManager'
  instVarNames: #()
  classVars: #()
  poolDictionaries: #[]
  inDictionary: Thesis
  constraints: #[]
  isInvariant: false
%
! Class 'Link'
NamedObject subclass: 'Link'
   instVarNames: #( 'type' 'source' 'destination'
            'property' 'cardinality')
  classVars: #( '')
  poolDictionaries: #[]
   inDictionary: ThesisLibrarySupport
  constraints: #[]
  isInvariant: false
%
! Instance Category 'testing'
category: 'testing'
method: Link
testForRemoval
        self removeSuccessor.
        self removePredecessor.
        ^self.
%
! Instance Category 'quering'
category: 'quering'
method: Link
hasSuccessors
        "Return true if there are successors; false otherwise"
        (self links includesKey: #successor)
                ifTrue: [^true]
                ifFalse: [^false]
%
! Instance Category 'Printing'
category: 'Printing'
method: Link
printPredecessor
```

```
        self predecessor isNil
                ifTrue: [^'nil'].
        ^ self predecessor name
%
method: Link
printSuccessor
        | aStream |
        self hasSuccessors
                ifFalse: [^'nil'].
        aStream := WriteStream on: (String new).
        self successor
                do:
                [:each |
                        aStream nextPutAll: (each name + ' ')].
        aStream nextPut: Character lf.
        ^aStream contents.
%
method: Link
printTargetsFor: aLinkType

        | theTargets aString |
        theTargets := self targetsFor: aLinkType.
        aString := String new.
        theTargets do:
                [ :i |
                        aString := aString + ' ' + i name].
        ^aString.
%
! Instance Category 'Define Associations'
category: 'Define Associations'
method: Link
targetsFor: aSymbol
        ^self links at: aSymbol ifAbsent: [nil]
%
! Instance Category 'Checking'
category: 'Checking'
method: Link
isReservedAssociation: aSymbol
        ^(((aSymbol = #from) or: [aSymbol = #predecessor])
         or: [ aSymbol =  #successor])
%
! Instance Category 'accessing'
category: 'accessing'
method: Link
predecessor

        (self links includesKey: #predecessor)
                ifTrue: [^self links at: #predecessor]
%
method: Link
successor

        (self links includesKey: #successor)
                ifTrue: [^self links at: #successor]
%
! Instance Category 'updating'
category: 'updating'
method: Link
removeSuccessor: aNode
        | anAssociation aMember |
        self hasSuccessors
                ifTrue:
                [ aMember :=  self successors detect: [:i | i name = aNode name] ifNone: [ ].
                aMember isNil
```

```
                    ifFalse: [self successors remove: aMember ifAbsent: []].
                self successors isEmpty
                    ifTrue: [ self removeSuccessor] ].
%
! Class Category 'testing'
category: 'testing'
classmethod: Link
addSuccessor: aVid1 andPredecessor: aVid2
        | aNode |
        "Node addSuccessor: 'Su.1' andPredecessor: 'Pre.3'"
        aNode:= Node new initialize.
        aNode name: 'Vivi'.
        aNode description: 'Testing'.
        aNode addPredecessor: aVid2.
        aNode addSuccessor: aVid1.
        ^aNode.
%
classmethod: Link
testForRemoval
        "Node testForRemoval"
        | aNode |
        aNode := self addSuccessor: 'Su.1' andPredecessor: 'Pre.2'.
        aNode removeSuccessor.
        aNode removePredecessor.
        ^aNode.
%
! Class Category 'new'
category: 'new'
classmethod: Link
new
        ^super new initialize.
%
! Class 'Microcontroller'
Object subclass: 'Microcontroller'
  instVarNames: #( 'cpu' 'memory' 'io')
  classVars: #()
  poolDictionaries: #[]
  inDictionary: ThesisData
  constraints: #[]
  isInvariant: false
%
! Instance Category 'Updating'
category: 'Updating'
method: Microcontroller
cpu: newValue
   "Modify the value of the instance variable 'cpu'."
   cpu:= newValue
%
method: Microcontroller
memory: newValue
   "Modify the value of the instance variable 'memory'."
   memory:= newValue
%
method: Microcontroller
io: newValue
   "Modify the value of the instance variable 'io'."
   io:= newValue
%
! Instance Category 'Accessing'
category: 'Accessing'
method: Microcontroller
cpu
   "Return the value of the instance variable 'cpu'."
   ^cpu
```

```
%
method: Microcontroller
memory

  "Return the value of the instance variable 'memory'."
  ^memory

%
method: Microcontroller
io
   "Return the value of the instance variable 'io'."
   ^io
%!
! Class 'NamedObject'
Object subclass: 'NamedObject'
  instVarNames: #( 'name' 'description')
  classVars: #( '')
  poolDictionaries: #[]
  inDictionary: ThesisLibrarySupport
  constraints: #[]
  isInvariant: false
%
! Instance Category 'updating'
category: 'updating'
method: NamedObject
name: aName
        name:= aName
%
method: NamedObject
description: aString
        description:= aString
%
! Instance Category 'accessing'
category: 'accessing'
method: NamedObject
name
        ^name
%
method: NamedObject
description
        ^description
%
! Instance Category 'initializing'
category: 'initializing'
method: NamedObject
initialize
        name:= String new.
        description:= String new.
%
! Class Category 'initialization'
category: 'initialization'
classmethod: NamedObject
new
        ^super new initialize
%
! Class Category 'testing'
category: 'testing'
classmethod: NamedObject
createAnInstance
        | aNamedObject |
        "NamedObject createAnInstance"
        aNamedObject:= self new.
        aNamedObject  name: 'Sergio'.
```

```
        aNamedObject description: 'This is a test'.
        ^aNamedObject.
%
! Class 'Node'
NamedObject subclass: 'Node'
 instVarNames: #( 'links')
 classVars: #( '')
 poolDictionaries: #[]
 inDictionary: ThesisLibrarySupport
 constraints: #[]
 isInvariant: false
%
! Instance Category 'quering'
category: 'quering'
method: Node
hasSuccessors
        "Return true if there are successors; false otherwise"
        (self links includesKey: #successor)
                ifTrue: [^true]
                ifFalse: [^false]
%
method: Node
isRoot
        "Returns true if the predecessor is nil"

        (links includesKey: #predecessor)
                ifTrue:
           [(links at: #predecessor) = nil
                 ifTrue: [^ true]
                 ifFalse: [^ false] ].
%
! Instance Category 'Printing'
category: 'Printing'
method: Node
printPredecessor
        self predecessor isNil
                ifTrue: [^'nil'].
        ^ self predecessor name
%
method: Node
printSuccessor
        | aStream |
        self hasSuccessors
                ifFalse: [^'nil'].
        aStream := WriteStream on: (String new).
        self successor
                do:
                [:each |
                        aStream nextPutAll: (each name + ' ')].
        aStream nextPut: Character lf.
        ^aStream contents.
%
method: Node
printTargetsFor: aLinkType

        | theTargets aString |
        theTargets := self targetsFor: aLinkType.
        aString := String new.
        theTargets do:
                [ :i |
                        aString := aString + ' ' + i name].
        ^aString.
%
! Instance Category 'Adding'
```

```
category: 'Adding'
method: Node
addPredecessor: aVersionId
        links add: (Association newWithKey: #predecessor value: aVersionId)
%
method: Node
addSuccessor: aVersionedObject
        | isThere |
        self hasSuccessors
                ifFalse:
                [       links add: (Association newWithKey: #successor value:
                                                (Set new add: aVersionedObject)). "There could be many
successors"
                ]
                ifTrue:
                [ "Check if the given VersionedObject instance is already a succesor"
                isThere := (links at: #successor)  detect: [:i | i name = aVersionedObject name] ifNone: [nil].
                isThere isNil
                        ifTrue: "Add the VersionedObject instance to the successor list"
                        [ (links at: #successor) add: aVersionedObject] ].
%
! Instance Category 'Define Associations'
category: 'Define Associations'
method: Node
defineAssociation: aSymbol toTarget: anObject
        | isThere aBol |
        (self links includesKey: aSymbol)
        ifTrue: [aBol := true]
        ifFalse: [aBol := false].
        (aBol = true)
                ifFalse:
                [       links add: (Association newWithKey: aSymbol value:
                                                (Set new add: anObject)). "There could be only unique targets"
        ]
                ifTrue:
                [ (links at: aSymbol) add: anObject ].
%
method: Node
defineAssociation: aSymbol withOrderedTargets: anObject
        | isThere aBol |
        (self links includesKey: aSymbol)
        ifTrue: [aBol := true]
        ifFalse: [aBol := false].
        (aBol = true)
                ifFalse:
                [       links add: (Association newWithKey: aSymbol value:
                                                (OrderedCollection new add: anObject)). "There could be equal
targets" ]
                ifTrue:
                [ (links at: aSymbol) add: anObject ].
%
method: Node
userDefinedAssociation: aSymbol toTarget: anObject
        | isThere aBol |
        (self isReservedAssociation: aSymbol)
                ifTrue: [^ #failed].
        (self links includesKey: aSymbol)
        ifTrue: [aBol := true]
        ifFalse: [aBol := false].
        (aBol = true)
                ifFalse:
                [       links add: (Association newWithKey: aSymbol value:
                                                (Set new add: anObject)). "There could be only unique targets"
        ]
```

```
                ifTrue:
                [ (links at: aSymbol) add: anObject ].
        ^#succeeded.
%
method: Node
targetsFor: aSymbol
        ^self links at: aSymbol ifAbsent: [nil]
%
! Instance Category 'Checking'
!
category: 'Checking'
method: Node
isReservedAssociation: aSymbol

        ^(((aSymbol = #from) or: [aSymbol = #predecessor])
         or: [ aSymbol = #successor])
%
! Instance Category 'accessing'
category: 'accessing'
method: Node
links
        ^links
%
method: Node
predecessor

        (self links includesKey: #predecessor)
                ifTrue: [^self links at: #predecessor]
%
method: Node
successor

        (self links includesKey: #successor)
                ifTrue: [^self links at: #successor]
%
! Instance Category 'initializing'
category: 'initializing'
method: Node
initialize

        links:= Dictionary new.
%
method: Node
resetLinks
        | aValue |
        aValue := links at: #from ifAbsent: [].
        links := Dictionary new.
        aValue isNil
                ifFalse: [links:= Dictionary new.
                                links at: #from put: aValue].
%
! Instance Category 'testing'
category: 'testing'
method: Node
testForRemoval
        self removeSuccessor.
        self removePredecessor.
        ^self.
%
! Instance Category 'updating'
category: 'updating'
method: Node
removePredecessor
        | anAssociation |
```

```
        (links includesKey: #predecessor)
                ifTrue: [ anAssociation := links associationAt: #predecessor.
                                links removeAssociation: anAssociation ifAbsent: [ ]].
        ^self
%
method: Node
removeSuccessor
        | anAssociation |
        self hasSuccessors
                ifTrue: [ anAssociation := links associationAt: #successor.
                                links removeAssociation: anAssociation ifAbsent: [ ]].
        ^self
%
method: Node
removeSuccessor: aNode
        | anAssociation aMember |
        self hasSuccessors
                ifTrue:
                [ aMember := self successors detect: [:i | i name = aNode name] ifNone: [ ].
                aMember isNil
                        ifFalse: [self successors remove: aMember ifAbsent: [ ]].
                self successors isEmpty
                        ifTrue: [ self removeSuccessor] ].

%
! Class Category 'new'
category: 'new'
classmethod: Node
new
        ^super new initialize.
%
! Class Category 'testing'
category: 'testing'
classmethod: Node
addSuccessor: aVid1 andPredecessor: aVid2
        | aNode |
        "Node addSuccessor: 'Su.1' andPredecessor: 'Pre.3'"
        aNode:= Node new initialize.
        aNode name: 'Vivi'.
        aNode description: 'Testing'.
        aNode addPredecessor: aVid2.
        aNode addSuccessor: aVid1.
        ^aNode.
%
classmethod: Node
testForRemoval
        "Node testForRemoval"
        | aNode |
        aNode := self addSuccessor: 'Su.1' andPredecessor: 'Pre.2'.
        aNode removeSuccessor.
        aNode removePredecessor.
        ^aNode.
%
category: 'Promote a Version (Check In)'
method: DataServer
OneLevelCheckIn: anInstance why: aString asLeafOfName: aName
toLibrary: aLibName2 withType: aSymbol2
toStream: aStreamName2 toContext: aContextName2
withChecking: aBoolean
        | aVO id toStream |
        self setBackups.
        self setCurrents: aLibName2 type: aSymbol2
      streamNamed: aStreamName2
      contextNamed: aContextName2.
```

```
        aName isNil
                ifTrue: [aVO := nil]
                ifFalse: [aVO := self getVersionedObject: aName].

        self currentStream addVersionManagerforClass: anInstance class description: ''.
        id := self currentContext put: anInstance reason: aString asLeafOf: aVO.
        self restoreCurrents.
        ^id
%
! Class 'PrivateLibrary'
Library subclass: 'PrivateLibrary'
 instVarNames: #()
 classVars: #()
 poolDictionaries: #[]
 inDictionary: ThesisLibrarySupport
 constraints: #[]
 isInvariant: false
%
! Class Category 'testing'
category: 'testing'
classmethod: PrivateLibrary
createSharedLibrary: aName withType: aSymbol why: aString
        "SharedLibrary createSharedLibrary: 'Project1' withType: #Public why: 'For testing'"
        ^ self name: aName libType: aSymbol description: aString
%
! Class Category 'creating'
category: 'creating'
classmethod: PrivateLibrary
name: aName libType: aSymbol description: aString
        ^((super new initialize name: aName) libType: aSymbol) description: aString
%
! Class 'SecurityData'
SecurityObject subclass: 'SecurityData'
 instVarNames: #()
 classVars: #( 'classOwners')
 poolDictionaries: #[]
 inDictionary: ThesisSecurity
 constraints: #[]
 isInvariant: false
%
! Class Category 'Administration'
category: 'Administration'
classmethod: SecurityData
newOwner: aUserId forClass: aClass
        classOwners at: aClass put: aUserId.
%
! Class Category 'adding'
category: 'adding'
classmethod: SecurityData
addClass: aClass owner: aSymbol
        (self existClass: aClass)
                ifTrue: [^ (aClass name + ' has owner: ' +
                                (self ownerForClass: aClass) asString)].
        classOwners at: aClass put: aSymbol.
        ^#done
%
! Class Category 'Accessing'
category: 'Accessing'
classmethod: SecurityData
classOwners
        ^classOwners
%
! Class Category 'creating'
category: 'creating'
```

```
classmethod: SecurityData
new
        ^super new
%
! Class Category 'Quering'
category: 'Quering'
classmethod: SecurityData
ownerForClass: aClass
        ^classOwners at: aClass ifAbsent: [#none]
%
classmethod: SecurityData
existClass: aClass
        | a |
        a := classOwners at: aClass ifAbsent: [nil].
        a notNil
                ifTrue: [ ^ true]
                ifFalse: [ ^ false].
%
classmethod: SecurityData
classesOwnedBy: aUserId
        | aDict aString |
        aString := String new.
        aDict := self classOwners
                 selectAssociations:
                        [:each | each value = aUserId ].
        aDict keys
                do:
                [:each |
                        aString := aString + ' ' + each name].
        ^aString.
%
! Class Category 'initializing'
category: 'initializing'
classmethod: SecurityData
reset
        classOwners := Dictionary new.
%
! Class 'SecurityEntity'
SecurityObject subclass: 'SecurityEntity'
  instVarNames: #()
  classVars: #()
  poolDictionaries: #[]
  inDictionary: ThesisSecurity
  constraints: #[]
  isInvariant: false
%
! Class Category 'creating'
category: 'creating'
classmethod: SecurityEntity
new
        ^super new.
%
! Class 'SecurityObject'
Node subclass: 'SecurityObject'
  instVarNames: #()
  classVars: #()
  poolDictionaries: #[]
  inDictionary: ThesisSecurity
  constraints: #[]
  isInvariant: false
%
! Class Category 'creating'
category: 'creating'
classmethod: SecurityObject
```

```
new
        ^super new.
%
! Class 'SharedLibrary'
Library subclass: 'SharedLibrary'
  instVarNames: #()
  classVars: #()
  poolDictionaries: #[]
  inDictionary: ThesisLibrarySupport
  constraints: #[]
  isInvariant: false
%
! Class Category 'testing'
category: 'testing'
classmethod: SharedLibrary
createSharedLibrary: aName withType: aSymbol why: aString
        "SharedLibrary createSharedLibrary: 'Project1' withType: #Public why: 'For testing'"
        ^self name: aName libType: aSymbol description: aString
%
! Class Category 'creating'
category: 'creating'
classmethod: SharedLibrary
name: aName libType: aSymbol description: aString
        ^((super new initialize name: aName) libType: aSymbol) description: aString
%
! Class 'Stream'
SecurityEntity subclass: 'Stream'
  instVarNames: #( 'contexts' 'versionManagers')
  classVars: #()
  poolDictionaries: #[]
  inDictionary: Thesis
  constraints: #[]
  isInvariant: false
%
! Class 'StreamVersionManager'
VersionManager subclass: 'StreamVersionManager'
  instVarNames: #( 'nextVersionNo')
  classVars: #()
  poolDictionaries: #[]
  inDictionary: ThesisVersion
  constraints: #[]
  isInvariant: false
%
! Instance Category 'accessing'
category: 'accessing'
method: StreamVersionManager
nextVersionNo
        ^nextVersionNo
%
! Instance Category 'updating'
category: 'updating'
method: StreamVersionManager
incNextVersionNo
        nextVersionNo := nextVersionNo + 1
%
method: StreamVersionManager
resetVersionNo
        nextVersionNo := 1
%
! Instance Category 'initializing'
category: 'initializing'
method: StreamVersionManager
initialize
```

```
            self resetVersionNo.
            versionedObjects := Array new.
            classHandle := nil.
%
! Class 'TransactionManager'
Object subclass: 'TransactionManager'
  instVarNames: #( 'sessions' 'locks')
  classVars: #()
  poolDictionaries: #[]
  inDictionary: ThesisTransactionService
  constraints: #[]
  isInvariant: false
%
! Instance Category 'Commiting -Aborting'
category: 'Commiting -Aborting'
method: TransactionManager
commit
            " returns true if succedes; otherwise returns false "
            ^System commitTransaction
%
method: TransactionManager
abort
            " returns true if succedes; otherwise returns false "
            ^System abortTransaction
%
! Instance Category 'Initializing'
category: 'Initializing'
method: TransactionManager
initialize
            sessions := Dictionary new.
            locks := Dictionary new.
%
! Class Category 'Creating'
category: 'Creating'
classmethod: TransactionManager
new
            ^super new initialize
%
! Class 'VersionedName'
Set subclass: 'VersionedName'
  instVarNames: #()
  classVars: #()
  poolDictionaries: #[]
  inDictionary: ThesisVersion
  constraints: Object
  isInvariant: false
%
! Instance Category 'translating'
category: 'translating'
method: VersionedName
asString
            | aString |
            aString := String new.
            self do:
                  [:i | aString := aString + i ].
            ^aString.
%
! Class Category 'Testing'
category: 'Testing'
classmethod: VersionedName
isA: aName

            "VersionedName isA: (VersionedName new: 'Micro.1').
             VersionedName isA: (VersionedName new: 'Micro').
```

```
        VersionedName isA:  Array new "
        (aName isMemberOf: VersionedName)
                ifTrue:
                [ "Then it is either a dynamic or static reference"
                 ((aName asString findString: '.' startingAt: 1) > 0)
                        ifTrue:  [^ #Static]
                        ifFalse:  [^ #Dynamic ] ].
        ^ #Instance

%
! Class Category 'Creating'
category: 'Creating'
classmethod: VersionedName
new: aString

        "VersionedName new: 'Fred.1'"
        ^super new add: aString.
%
! Class 'VersionedObject'
SecurityData subclass: 'VersionedObject'
  instVarNames: #( 'compositeFlag' 'versionStatus' 'versionNo'
            'context' 'stream' 'timeStamps' 'updateInfo'
            'parts')
 classVars: #()
 poolDictionaries: #[]
 inDictionary: ThesisVersion
 constraints: #[]
 isInvariant: false
%
! Instance Category 'Accessing'
category: 'Accessing'
method: VersionedObject
compositeFlag
  "Return the value of the instance variable 'compositeFlag'."
  ^compositeFlag
%
method: VersionedObject
versionStatus
  "Return the value of the instance variable 'versionStatus'."
  ^versionStatus
%
method: VersionedObject
versionNo
  "Return the value of the instance variable 'versionNo'."
  ^versionNo
%
method: VersionedObject
object
  "Return the value of the instance variable 'object'."
  ^object
%
method: VersionedObject
updateInfo
  "Return the value of the instance variable 'updateInfo'."
  ^updateInfo
%
method: VersionedObject
parts
  "Return the value of the instance variable 'parts'."
  ^parts
%
method: VersionedObject
context
        ^context
```

```
%
method: VersionedObject
stream
        ^stream
%
! Instance Category 'Private'
category: 'Private'
method: VersionedObject
varNames
        | ar |
        ar := Array new.
        self parts
                do:
                [:i |
                        ar add: (i at: #VarName)].
        ^ ar
%
method: VersionedObject
elementsBindingInfo
        | ar |
        ar := Array new.
        self parts
                do:
                [:i |
                        ar add: (i at: #bindingInfo)].
        ^ ar
%
method: VersionedObject
elementsClass
        | ar |
        ar := Array new.
        self parts
                do:
                [:i |
                        ar add: (i at: #elementClass)].
        ^ ar
%
method: VersionedObject
elements
        | ar |
        ar := Array new.
        self parts
                do:
                [:i |
                        ar add: (i at: #element)].
        ^ ar
%
method: VersionedObject
paths
        | ar |
        ar := Array new.
        self parts
                do:
                [:i |
                        ar add: (i at: #path)].
        ^ ar
%
method: VersionedObject
libTypes
        | ar |
        ar := Array new.
        self parts
                do:
                [:i |
```

```
                        ar add: (i at: #libTypes)].
        ^ ar
%
method: VersionedObject
addInfo: anInstance withVar: aVarName
        | aDict |
        aDict := Dictionary new.
        aDict at: #VarName put: aVarName.
        (anInstance isMemberOf: VersionedName)
                ifTrue:
                [ "Then it is either a dynamic or static reference"
                 ((anInstance asString findString: '.' startingAt: 1) > 0)
                        ifTrue:  [aDict at: #bindingInfo put: #Static]
                        ifFalse:  [aDict at: #bindingInfo put: #Dynamic].
                 self compositeFlag: true.
                ]
                ifFalse:
                [ aDict at: #bindingInfo put: #Instance].
        aDict   at: #elementClass put: anInstance class.
        aDict   at: #element put: anInstance.
        aDict   at: #path put: 'Do not worry for now'.
        aDict   at: #libType put: 'To be determined later'.
        self parts add: aDict.
%
! Instance Category 'Printing'
category: 'Printing'
method: VersionedObject
print
        | aStream anArray |
        anArray := self class allInstVarNames.
        aStream := WriteStream on: (String new).
        anArray do:
                [:i |
                        (i ~= nil)
                        ifTrue: [
                                aStream nextPutAll: (i asString width: 20).
                                ((i = #context)  or: [i = #stream])
                                        ifTrue: [aStream nextPutAll: (self  perform: i) name asString].
                                i = #links
                                        ifTrue:
                                        [aStream nextPutAll: ('Predecessor: ' + self printPredecessor).
                                         aStream  nextPutAll: '  '.
                                         aStream  nextPutAll: ('Successor: ' + self printSuccessor).
                                         aStream nextPut: Character lf.
                                         aStream nextPutAll: ('From: ' + self printVirtualPath)].
                                i = #parts
                                        ifTrue:
                                        [ self elements
                                                do: [:each| (each isKindOf: VersionedName)
                                                                ifTrue: [aStream nextPutAll:
(each asString + '  ') ]
                                                                ifFalse: [ aStream nextPutAll:
('an Instance of ' + each class name asString) ]
                                                                ]
                                        ].
                                i = #updateInfo
                                        ifTrue:
                                        [aStream nextPutAll: ('User: ' + self user asString) ].
                                i = #timeStamps
                                        ifTrue:
                                        [ aStream nextPutAll: ('Checked at: ' + self checkedTime ) ].
                                (((((((i ~= #links) and: [i ~= #context]) and: [i ~= #updateInfo]) and: [i ~= #parts])
                                        and: [i ~= #stream]) and: [i ~= nil]) and: [i ~= #timeStamps])
                                                ifTrue: [aStream nextPutAll: (self  perform: i) asString.].
```

```
                                        aStream nextPut: Character lf ]
                ].

        ^ aStream contents
%
! Instance Category 'initializing'
category: 'initializing'
method: VersionedObject
initParts: aNumber
        | aDict |
        aDict := Dictionary new.
        aDict at: #VarName put: nil;
                at: #bindingInfo put: nil;
                at: #elementClass put: nil;
                at: #element put: nil;
                at: #path put: nil;
                at: #libType put: nil.
        1 to: aNumber
                do:
                [ :i |
                        self parts at: i put: aDict copy].
        ^self
%
method: VersionedObject
init
        | aDataServer |
         aDataServer := DataServer new.
         compositeFlag := false.
         versionStatus := #Temporary.
         versionNo := 0.
         context := aDataServer currentContext.
         timeStamps := Dictionary new.
         updateInfo := Dictionary new.
         parts := Array new.
         stream := aDataServer currentStream.
%
! Instance Category 'TimeStamping'
category: 'TimeStamping'
method: VersionedObject
timeStamps
  "Return the value of the instance variable 'timeStamps'."
  ^timeStamps
%
method: VersionedObject
checkedTime: aTime
        self timeStamps at: #checked put: aTime.
%
method: VersionedObject
effectiveTime
        ^ self timeStamps at: #effective
%
method: VersionedObject
effectiveTime: aTime
        self timeStamps at: #effective put: aTime.
%
method: VersionedObject
checkedTime
        ^self timeStamps at: #checked
%
! Instance Category 'Quering'
category: 'Quering'
method: VersionedObject
getVMClassHandleFor: aName
        ^(self getVM: aName) classHandle
```

```
%
method: VersionedObject
getVM: aName
        ^ self context getVM: aName
%
method: VersionedObject
itsClass
        | aName |
        aName := VersionedObject onlyName: self name.
        ^self getVMClassHandleFor: aName.
%
! Instance Category 'Updating'
category: 'Updating'
method: VersionedObject
compositeFlag: newValue
  "Modify the value of the instance variable 'compositeFlag'."
  compositeFlag:= newValue
%
method: VersionedObject
versionStatus: newValue
  "Modify the value of the instance variable 'versionStatus'."
  versionStatus:= newValue
%
method: VersionedObject
versionNo: newValue
  "Modify the value of the instance variable 'versionNo'."
  versionNo:= newValue
%
method: VersionedObject
object: newValue
  "Modify the value of the instance variable 'object'."
  object:= newValue
%
method: VersionedObject
timeStamps: newValue
  "Modify the value of the instance variable 'timeStamps'."
  timeStamps:= newValue
%
method: VersionedObject
updateInfo: newValue
  "Modify the value of the instance variable 'updateInfo'."
  updateInfo:= newValue
%
method: VersionedObject
parts: newValue
  "Modify the value of the instance variable 'parts'."
  parts:= newValue
%
method: VersionedObject
context: aContext
        context := aContext.
%
method: VersionedObject
stream: aStream
        stream := aStream.
%
method: VersionedObject
statusForLibType: aSymbol
  "Modify the value of the instance variable 'versionStatus'."
        aSymbol = #Private
                ifTrue: [self versionStatus: #Temporary]
                ifFalse: [ aSymbol = #SemiPublic
                                ifTrue: [self versionStatus: #Working]
                                ifFalse: [self versionStatus: #Released] ].
```

```
%
! Instance Category 'UpdateInfo'
category: 'UpdateInfo'
method: VersionedObject
user: aUserId
        self updateInfo at: #user put: aUserId.
%
method: VersionedObject
user
        ^self updateInfo at: #user
%
! Instance Category 'VirtualPath'
category: 'VirtualPath'
method: VersionedObject
virtualPath
        "Returns an array. Each element of the array contains a source location
         for the versionedObject in this format:
         'Library Type/Library Name.Stream Number.Context Number/Version Id'
        "
        ^ self links at: #from ifAbsent: [nil]
%
method: VersionedObject
addSourcePath: aPathName
         self defineAssociation: #from withOrderedTargets: (aPathName + '/' + self name).
%
method: VersionedObject
printVirtualPath
        "Prints the elements of an OrderedCollection. Each element of the array contains a source location
         for the versionedObject in this format:
         'Library Type/Library Name.Stream Number.Context Number/Version Id'
        "
        | anArray aString |
        aString := String new.
        anArray := self links at: #from ifAbsent: [nil].
        anArray isNil
        ifFalse:
                [anArray do: [:i | aString := aString + ' '+ i] ].
        ^ aString.
%
! Class Category 'Delta'
category: 'Delta'
classmethod: VersionedObject
deltaBetweenTwoVersionedObjects: aVO1 and: aVO2
        "Returns true if there is a delta between the
        two version values. Note: aVersion and aVersion2 are
        instances of the VersionedObject class"

        | anArray  aDataServer elements1  elements2 i exist |
        aDataServer := DataServer new.
        (aVO1 isNil or: [aVO2 isNil ])
        ifTrue:
                [^ nil].
        elements1 := aVO1 elements.
        elements2 := aVO2 elements.
        i := 0.
        elements1
                do:
           [:each | i := i +1.
                        (each isKindOf: VersionedName)
                        ifTrue:
                         [ (each asString = (elements2 at: i) asString)
                          ifFalse: [^ true]
                         ]
```

```
                        ifFalse:
                        [ (each     = (elements2 at: i) )
                          ifFalse: [^ true].
                        ]
                ].
        ^ false.
%
classmethod: VersionedObject
existDeltaBetween: aVersion and: aVersion2
        "Returns true if there is a delta between the
        two version values"

        | anArray aVO1 aVO2 aDataServer elements1  elements2 i exist |
        aDataServer := DataServer new.
        aVO1 := aDataServer  getVersionedObject: aVersion.
        aVO2 := aDataServer  getVersionedObject: aVersion2.
        (aVO1 isNil or: [aVO2 isNil ])
        ifTrue:
                [^ nil].
        elements1 := aVO1 elements.
        elements2 := aVO2 elements.
        i := 0.
        elements1
                do:
           [:each | i := i +1.
                        (each isKindOf: VersionedName)
                        ifTrue:
                         [ (each asString = (elements2 at: i) asString)
                          ifFalse: [^ true]
                         ]
                        ifFalse:
                        [ (each     = (elements2 at: i) )
                          ifFalse: [^ true].
                        ]
                ].
        ^ false.
%
classmethod: VersionedObject
printDeltaBetween: aVersion and: aVersion2
        "Returns true if there is a delta between the
        two version values"

        | aStream anArray aVO1 aVO2 aDataServer elements1  elements2 i |
        aDataServer := DataServer new.
        aStream := WriteStream on: (String new).
        aVO1 := aDataServer  getVersionedObject: aVersion.
        aVO2 := aDataServer  getVersionedObject: aVersion2.
        (aVO1 isNil or: [aVO2 isNil ])
                ifTrue: [^ aStream contents ].
        elements1 := aVO1 elements.
        elements2 := aVO2 elements.
        i := 0.
        elements1
                do:
           [:each | i := i +1.
                        (each isKindOf: VersionedName)
                        ifTrue:
                         [ (each asString = (elements2 at: i) asString)
                          ifFalse:
                                [ aStream nextPutAll: '> ' + each asString.
                                  aStream nextPut: Character lf.
                                  aStream nextPutAll: '< ' + (elements2 at: i) asString.
                                ].
                         ]
```

```
                        ifFalse: [ (each      = (elements2 at: i) )
                                        ifFalse:
                                                [ aStream nextPutAll: '> ' + ('an Instance of ' + each class
name asString).
                                                  aStream nextPut: Character lf.
                                                  aStream nextPutAll: '> ' + ('an Instance of ' + (elements2
at: i) class name asString).
                                                ].
                                        ].
                aStream nextPut: Character lf ].
        ^ aStream contents
%
! Class Category 'translating'
category: 'translating'
classmethod: VersionedObject
asVersionedObject: anInstance reason: aString
        "Returns a Versioned Object instance.
         1. Reads the information of an instance, be getting its class
         2. Initializes a Versioned Object
         3. Assign values to the Versioned Object instance variables"
        | aClass aName aVersionedObject  instanceParts |
        anInstance isNil ifTrue: [^ nil].
        aClass := anInstance class.
        aName := aClass name.
        aVersionedObject := self name: aName reason: aString.
        instanceParts := aClass allInstVarNames.
        instanceParts  isEmpty
                ifFalse:
                [ aVersionedObject compositeFlag: true.
                  instanceParts do:
                        [:each |
                         aVersionedObject addInfo: (anInstance perform: each) withVar: each.
                        ].
                ].
        aVersionedObject description: aString.
        ^aVersionedObject
%
classmethod: VersionedObject
string: aString until: aString2
        "VersionedObject string: 'Micro.12' until: '.'.
        VersionedObject string: '.12' until: '.'.
        VersionedObject string: 'a.12' until: '.'"
        | index  a |
        (aString isKindOf: String)
                ifFalse: [^nil].
        (aString2 isKindOf: String)
                ifFalse: [^nil].
        a := String new.
        index := (aString findString: aString2 startingAt: 1).
        index > 1
                ifTrue:
                [ index = 2
                        ifTrue: [ a := a + aString at: 1]
                        ifFalse:
                                [
                                1 to: (index -1)
                                        do: [:i | a := a + (aString at: i)].
                                ].

                 ^ a.
                ]
                ifFalse: [^nil]
%
classmethod: VersionedObject
```

```
onlyName: aName
        "VersionedObject onlyName: 'Microprocessor.2' .
        VersionedObject onlyName: 'Microprocessor' "
        | aName2 |
        aName2 := self string: aName until: '.'.
        aName2 isNil ifTrue: [^ aName].
        ^aName2
%
classmethod: VersionedObject
newName: oldName withNo: aNumber
        "VersionedObject newName: 'Microprocessor.2' withNo: 1.
        VersionedObject newName: 'Microprocessor' withNo: 1"
        | aName |
        aName := self string: oldName until: '.'.
        aName isNil ifTrue: [^ oldName].
        ^(aName +'.'+ aNumber asString)
%
! Class Category 'creating'
category: 'creating'
classmethod: VersionedObject
name: aName type: aType reason: aString

        "VersionedObject name: 'Micro' type: '58000' reason: 'Improve overal performance'"
        ^((super new initialize) name: (aName + '.' + aType)).
%
classmethod: VersionedObject
name: aName reason: aString
        "VersionedObject name: 'Micro' type: '68000' reason: 'Improve overal performance'"
        ^((super new initialize init) name: aName).
%
! Class Category 'DeepCopy'
category: 'DeepCopy'
classmethod: VersionedObject
newInstance: anInstance forClass: aClass
        | anArray newInstance msg newParts atElement aDict k insParts |
        newInstance := aClass name: anInstance name reason: anInstance description.
        anArray := anInstance class allInstVarNames.
        anArray
        do:
                [:i |
                        (i ~= #links)
                        ifTrue:
                         [ i = #parts
                                        ifTrue:
                                         [k := 0.
                                                newParts := newInstance parts.
                                                insParts := anInstance parts.
                                                insParts isNil ifFalse:
                                                [
                                                insParts
                                                 do:
                                                 [:eachDict| k := k + 1.
                                                        aDict := Dictionary new.
                                                        atElement := eachDict at: #element.
                                                        (atElement isKindOf: VersionedName)
                                                        ifTrue: [aDict at: #element put:
                                                                        (VersionedName new: atElement
asString)]

                                                        ifFalse: [aDict at: #element put: atElement].
                                                        aDict at: #libType put: (String new add: (eachDict at:
#libType) ).

                                                        aDict at: #path put: (String new add: (eachDict at: #path) ).
                                                        aDict at: #elementClass put:
                                                                (eachDict at: #elementClass).
```

```
                                                    aDict at: #bindingInfo put:
                                                            (Symbol new add: (eachDict at: #bindingInfo) ).

                                                    aDict at: #varName put:
                                                                    (Symbol new add: (eachDict at:
#varName) ).
                                                    newParts at: k put: aDict.
                                            ] ].
                                        ]
                                        ifFalse:
                                         [msg := (i asString +':') asSymbol.
                                          newInstance perform: msg with: (anInstance perform: i)] ].
                ].
        ^newInstance.
%
! Class 'VersionManager'
SecurityData subclass: 'VersionManager'
  instVarNames: #( 'versionedObjects' 'classHandle')
  classVars: #()
  poolDictionaries: #[]
  inDictionary: ThesisVersion
  constraints: #[]
  isInvariant: false
%
! Instance Category 'updating'
category: 'updating'
method: VersionManager
classHandle: aClass
        classHandle := aClass
%
! Instance Category 'initializing'
category: 'initializing'
method: VersionManager
initialize
        versionedObjects := Array new.
%
! Instance Category 'accessing'
category: 'accessing'
method: VersionManager
versionedObjects
        ^versionedObjects
%
method: VersionManager
classHandle
        ^classHandle
%
! Class Category 'creating'
category: 'creating'
classmethod: VersionManager
name: aName description: aString
        "VersionManager name: 'Sergio' description: 'For Testing'"
        ^(super new initialize name: aName) description: aString
%
! Class 'WorkingContext'
SecurityEntity subclass: 'WorkingContext'
  instVarNames: #( 'versionManagers')
  classVars: #( 'CurrentContext')
  poolDictionaries: #[]
  inDictionary: ThesisContext
  constraints: #[]
  isInvariant: false
%
! Instance Category 'Printing'
category: 'Printing'
```

```
method: WorkingContext
printTCFor: aVersionName
        " "
%
! Instance Category 'accessing'
category: 'accessing'
method: WorkingContext
currentContext
        ^CurrentContext
%
method: WorkingContext
versionManagers
        ^versionManagers
%
method: WorkingContext
versionTrackers
        ^self versionManagers
%
! Instance Category 'updating'
category: 'updating'
method: WorkingContext
currentContext: aPathName
        CurrentContext := aPathName
%
! Instance Category 'quering'
category: 'quering'
method: WorkingContext
existVM: aName
        ^ versionManagers includesKey: aName asSymbol.
%
method: WorkingContext
getVM: aName
        (self existVM: aName)
                ifFalse: [^nil]
                ifTrue: [ ^self versionManagers at: aName asSymbol]
%
method: WorkingContext
getDynamicVersionedObject
        " "
%
method: WorkingContext
getVersionedObject: aVersionName
        | aVM aName  isStatic aSymbol |
        isStatic := true.
        aName := VersionedObject string: aVersionName until: '.'.
        aName isNil
                ifTrue:
                [aName := aVersionName.
                 isStatic := false].
        aVM := self getVM: aName.
        aVM  isNil
                ifTrue: [^nil]
                ifFalse:
                [ isStatic
                        ifTrue: [^aVM getVersionedObjectwithVersionId: aVersionName]
                        ifFalse: [^ aVM getDynamicVersionedObject] ].

%
! Instance Category 'initializing'
category: 'initializing'
method: WorkingContext
initialize
        versionManagers := Dictionary new.
%
```

```
method: WorkingContext
resetAllVersionedObjects
        versionManagers
                doValues:
                [:each |
                        each resetVersionedObjects  ].

%
! Instance Category 'defining a class'
category: 'defining a class'
method: WorkingContext
defineClass: aString asSubClassOf: aClass withParts: anArray
withGlobalParts: aGlobalArray
poolDictionaries: aPool
inDictionary: aDictionary
constraints: anArray2
isInvariant: aBoolean
aClass subclass: aString
   instVarNames: anArray
   classVars: aGlobalArray
    poolDictionaries: aPool
    inDictionary: aDictionary
   constraints: anArray2
   isInvariant: aBoolean
%
method: WorkingContext
defineClass: aString asSubClassOf: aClass withParts: anArray inDictionary: aDict
        " This instance method defines a class and adds a
          VersionManager for the new class in the receiver (SF)"
        "(WorkingContext  name: 'MyContext' description: 'Yes')
        defineClass: 'Try2' asSubClassOf: Object withParts: #('a' 'b' 'c')
        inDictionary: ThesisStream"

        | newClass |
        newClass := aClass subclass: aString
                instVarNames: anArray
                classVars: #()
                poolDictionaries: #[]
                inDictionary: aDict
                constraints: #[]
                isInvariant: false.
        newClass compileAccessingMethodsFor: newClass instVarNames.
        ^self addVersionManagerforClass: newClass description: ''.
%
! Instance Category 'adding'
category: 'adding'
method: WorkingContext
addVersionManager: aName description: aString
        " (WorkingContext name: 'Micro'  description: 'MicroProcessor Project') initialize
                addVersionManager: '68000.Spec' description: 'Motorola 68000 Specs'"

        (self existVM: aName)
                ifFalse:
                [ self versionManagers at: aName asSymbol put:
                        (ContextVersionManager name: aName description: aString)
                ].
        ^self versionManagers at: aName asSymbol


%
method: WorkingContext
addVersionManagerforClass: aClass description: aString
        " (WorkingContext name: 'Micro'  description: 'MicroProcessor Project') initialize
                addVersionManagerforClass: Array description: 'Motorola 68000 Specs'"
```

```
        | aName aSymbol |
        aName := aClass name.
        aSymbol :=  aName asSymbol.
        (self existVM: aSymbol)
                ifFalse:
                [ self versionManagers at: aSymbol put:
                        ((ContextVersionManager name: aName description: aString) classHandle: aClass)
                ].
        ^self versionManagers at: aSymbol
%
method: WorkingContext
addVersionedObject: aVersionedObject reason: aString asLeafOf: aParentVersionedObject

        | aName aVM |
        aName := aVersionedObject name.
        aVM := self addVersionManager: aName description: aString.
        ^ aVM addVersionedObject: aVersionedObject asLeafOf: aParentVersionedObject.
%
! Instance Category 'Checking In'
category: 'Checking In'
method: WorkingContext
checkIn: anInstance reason: aString asLeafOf: aParentVersionedObject
        | aVersionedObject |
        aVersionedObject := VersionedObject asVersionedObject: anInstance reason: aString.
        self addVersionedObject: aVersionedObject reason: aString asLeafOf: aParentVersionedObject.
%
method: WorkingContext
put: anInstance reason: aString asLeafOf: aParentVersionedObject
        | aVersionedObject |
        aVersionedObject := VersionedObject asVersionedObject: anInstance reason: aString.
        ^ self addVersionedObject: aVersionedObject reason: aString asLeafOf: aParentVersionedObject.
%
! Class Category 'creating'
category: 'creating'
classmethod: WorkingContext
name: aName description: aString
        ^((super new initialize) name: aName) description: aString.
%
! Class Category 'testing'
category: 'testing'
classmethod: WorkingContext
addSomeVersionedObjects
        "WorkingContext addSomeVersionedObjects"
        | wC |
        wC := self name: 'Test Context' description: 'For testing'.
        wC addVersionedObject:
                (VersionedObject name: 'Micro'
                        type: 'Layout' reason: 'No reason') reason: 'Less area' asLeafOf: nil;
                addVersionedObject:
                        (VersionedObject name: 'Micro'
                                type: 'Layout' reason: 'No reason') reason: 'More Power' asLeafOf: nil;
addVersionedObject:
                        (VersionedObject name: 'Micro'
                                type: 'Layout' reason: 'No reason') reason: 'More Current' asLeafOf: nil
                addVersionedObject:
                        (VersionedObject name: 'Micro'
                                type: 'Functionality' reason: 'No reason') reason: 'More Power' asLeafOf: nil;
                addVersionedObject:
                        (VersionedObject name: 'Chip'
                                type: 'Functionality' reason: 'No reason') reason: 'More Power' asLeafOf: nil.
        ^wC
%
! Class Category 'make checkOut translation'
category: 'make checkOut translation'
```

```
classmethod: WorkingContext
correctPartsFor: aVersionName accordingTranslation: aTable
        "1. Correct the parts in the translation table.
         2. Add the versioned Objects to the target context"
        | table2 aVO  oldName oldEntry nextNum  parenth anArray  aKey theDelta |
        aTable isNil ifTrue: [^ nil].
        table2 := Dictionary new.
        aTable doAssociations:
        [:eachAssoc |
          anArray := Array new.
          eachAssoc value do: [:k | anArray add: k].
          aKey := String new add: eachAssoc key.
          table2 add: (Association newWithKey: aKey
                                                value: anArray) ].
        table2 doValues:
                [:i |
                        i at: 1 put:
                         (VersionedObject newInstance: (i at: 1)
                                                                forClass: VersionedObject) ].
        table2
                doValues:
                [:eachEntry|
                        aVO := eachEntry at: 1. "a Versioned Object"
                        aVO parts
                                do:
                                [:eachPart |
                                        ((eachPart at: #bindingInfo)  = #Instance)
                                                ifFalse:
                                                [
                                                oldName := (eachPart at: #element) copy asString.
                                                oldEntry := aTable at: oldName ifAbsent: [].
                                                oldEntry isNil
                                                        ifFalse:
                                                        [ eachPart at: #element put: (VersionedName new:
                                                         ( VersionedObject newName: oldName
                                                                                withNo: (oldEntry at: 3) ) ) ]
].
                                ]
                ].
        ^table2
%
! Class 'WorkingStream'
SecurityEntity subclass: 'WorkingStream'
  instVarNames: #( 'contexts' 'nextContextNo' 'versionManagers')
 classVars: #( 'CurrentStream')
 poolDictionaries: #[]
 inDictionary: ThesisStream
 constraints: #[]
 isInvariant: false
%
! Instance Category 'defining a class'
category: 'defining a class'
method: WorkingStream
defineClass: aString asSubClassOf: aClass withParts: anArray inDictionary: aDict
        " This instance method defines a class and adds a
          VersionManager for the new class in the receiver (SF)"
        "(WorkingContext  name: 'MyContext' description: 'Yes')
        defineClass: 'Try2' asSubClassOf: Object withParts: #('a' 'b' 'c')
        inDictionary: ThesisStream"

        | newClass |
        newClass := aClass subclass: aString
                instVarNames: anArray
                classVars: #()
```

```
                    poolDictionaries: #[]
                    inDictionary: aDict
                    constraints: #[]
                    isInvariant: false.
          newClass compileAccessingMethodsFor: newClass instVarNames.
          ^self addVersionManagerforClass: newClass description: ''.
%
! Instance Category 'updating'
category: 'updating'
method: WorkingStream
incNextContextNo
          nextContextNo := nextContextNo + 1.
%
method: WorkingStream
resetNextContextNo
          nextContextNo := 1.
%
method: WorkingStream
getContextName
          ^ (self name + '.' + nextContextNo asString).
%
method: WorkingStream
incNextVersionNoForManager: aName
          | aVM |
          aVM := self getVM: aName.
          aVM incNextVersionNo.
%
method: WorkingStream
newContext: aDescription
          | aContext |
          '' (((Library name: 'Micro' type: #Public description: 'Micro') addStream: 'Test Stream') streams at: 1) newContext:
'Test'''
          aContext := WorkingContext name: self getContextName description: aDescription.
          self contexts add: aContext.
          self incNextContextNo.
%
! Instance Category 'quering'
category: 'quering'
method: WorkingStream
getContextAt: aNumber
                    '' ((WorkingStream name: 'Micro.1' description: 'Test Stream') newContext: 'Context test') getContextAt:
1''
                    (self contexts size >= aNumber and: [ aNumber > 0])
                              ifTrue: [^ self contexts at: aNumber].
                    ^nil.
%
method: WorkingStream
existVM: aName
          ^ versionManagers includesKey: aName asSymbol.
%
method: WorkingStream
getVM: aName
          (self existVM: aName)
          ifTrue: [^(self versionManagers at: aName asSymbol)]
          ifFalse: [^nil].
%
method: WorkingStream
getNextNoForVersion: aVersion
          | aVM aName |
          ''DataServer new currentStream getNextNoForVersion: 'Microprocessor'''
          aName := VersionedObject string: aVersion until: '.'.
          aName isNil
                    ifTrue: [ aName := aVersion].
          aVM := self getVM: aName.
```

```
        aVM isNil ifTrue: [^nil].
        ^aVM nextVersionNo
%
method: WorkingStream
contextNamed: aName
        ^ self contexts
                detect:
                        [:each| (each name = aName) ]
                 ifNone: []
%
! Instance Category 'initializing'
category: 'initializing'
method: WorkingStream
initialize
        contexts := Array new.
        sel:' resetNextContextNo.
        versionManagers := Dictionary new.
%
method: WorkingStream
resetAllVersionedObjects
        versionManagers
                doValues:
                [:each |
                        each resetVersionNo ].
%
! Instance Category 'accessing'
category: 'accessing'
method: WorkingStream
contexts
        ^contexts
%
method: WorkingStream
versionManagers
        ^versionManagers
%
! Instance Category 'adding'
category: 'adding'
method: WorkingStream
addVersionManagerforClass: aClass description: aString
        " (WorkingContext name: 'Micro'  description: 'MicroProcessor Project') initialize
                addVersionManagerforClass: Array description: 'Motorola 68000 Specs'"
        | aName aSymbol |
        aName := aClass name.
        aSymbol := aName asSymbol.
        (self existVM: aSymbol)
                ifFalse:
                [ self versionManagers at: aSymbol put:
                        ((StreamVersionManager name: aName description: aString) classHandle: aClass)
                ].
        DataServer new currentContext addVersionManagerforClass: aClass description: aString.
        ^self versionManagers at: aSymbol
%
method: WorkingStream
addVersionedObject: aVersionedObject reason: aString asLeafOf: aParentVersionedObject
                "Works with the current context "
                "self addVersionManager: aVersionedObject name.
                ^DataServer new currentContext
                        addVersionedObject: aVersionedObject
                        reason: aString asLeafOf: aParentVersionedObject."
%
! Instance Category 'Translating for CheckOut'
category: 'Translating for CheckOut'
method: WorkingStream
makeTableFrom: aContext
```

```
toStream: aStream2
toContext: aContext2
withVersion: aString
withTable: aTable
        "aTable is a dictionary. Its key is the version name; e.g. Microprocessor.1. The
        value is a four element array. The array first element is the versionedObject,
        the second element is its version no, the third element contains the versionNo
        which will be given once it is stored in the target context, and the fourth
        element is the VM class type."
        | aDataServer aVO wichElement aRR no2 |
        aString isNil
                ifFalse:
                [
                aVO := aContext getVersionedObject: aString.
                aVO notNil
                ifTrue: [
                aRR := Array new: 4.
                aRR at: 1 put: aVO copy. "A copy of the versioned Object"
                aRR at: 2 put: aVO versionNo. "A copy of the versioned Object current version no"
                no2 := aStream2 getNextNoForVersion: aString.
                no2 isNil ifTrue: [ no2 := 1].
                aRR at: 3 put: no2. "The version no which the current version will have in the target"
                aRR at: 4 put: (aVO getVMClassHandleFor: (VersionedObject onlyName: aVO name)). "VM classHandle"
                aTable at: aString put: aRR.
                aVO parts
                        do:
                        [:i |
                        (((i at: #bindingInfo) = #Static) or: [(i at: #bindingInfo) = #Dynamic])
                                ifTrue:  [ wichElement := (i at: #element) asString]
                                ifFalse: [ wichElement := (i at: #element) class name asString].
                                self makeTableFrom: aContext
                                toStream: aStream2
                                toContext: aContext2
                                withVersion: wichElement
                                withTable: aTable
                        ] ] ].
        ^aTable
%
! Class Category 'creating'
category: 'creating'
classmethod: WorkingStream
name: aName description: aString
        ^(super new initialize name: aName) description: aString
%
```

Appendix F

Glossary

abstraction.

- Within the VLSI context, it captures the description of a VLSI design in one of its three dimensions: behavioral, structural and physical.

abort.

- Undoing a *transaction* before it is completed.

behavior.

- A collection of all the *methods* defined in a *class* description.

branching versioning.

- Refers to the ability of a parent to have multiple successors.

check-in.

- The installation of a copy of an *instance version* from a *library* into an upper level *library*.

check-out.

- The installation of a copy of an *instance version* from a *library* into a lower level *library*.

class.

- A description of one or more similar *objects*. These *objects* share the same *structure* and *behavior*.

  class hierarchy.

  - A directed tree of *classes*.

  class inheritance.

  - The means by which a *class* inherits all the attributes and methods of its *superclasses*.[1]

client.

- Here, a user or an application which requests services from the `data server`. A wider definition considers a client as any object that uses the resources of another object.

---

[1] This means that any variable that is defined higher in the *class hierarchy* will also appear in *instances* of this *class*.

clustering.

• The capability to store related *objects* physically close together on secondary storage.

commit.

• A normal termination of a *transaction*. When a *transaction* commits, its changes to persistent secondary memory are made permanent and visible.

composite object.

• An aggregation of a root object and its parts (which can themselves be composite objects)[1].

composite configuration.

• A set of *instance versions* associated by a composition *relationship*

configuration.

• A set of *instance versions* associated by *relationships*[2]

concurrency control.

• A mechanism that regulates access to *objects* and prevents executing inconsistent *transactions* in a *database*. That is, it guarantees that the concurrent execution of a set of *transactions* does not result in an inconsistent *database* state.

context.

• A collection of meaningful *configurations* which occur in a specific phase of the *design cycle*.

data.

• Design information, or design-related information (for example, documentation), or both. Design information includes design requirements and specifications, design implementation, execution, and testing-related information.

data control.

• The ability to control the access of *data*, send notifications of changes to *data*, and control concurrent access to *data*.

---

[1] The interconnected objects are instantiated together.

[2] The *relationships* may have different semantics. A configuration consist of one or more configurations, it is a recursive concept. A configuration is called simple when it contains data associated by a single kind of semantic relationship. A configuration is called complex when it is composed of nested configurations.

data integration.

- The ability to manage *data*, and control the access to *data* among *clients* via common mechanisms.

data management

- The ability to model, track, and relate complex *data*.

data model.

- A set of logical constructs used to describe real-world things (such as data, data relationships, data semantics, and data constraints).

data-ownership mechanism.

- A *mechanism* which controls who is responsible for the release of *data*, who can use what, and how the (owned) *data* can be used by others.

data server.

- An *object-oriented database management system* which is equipped with a *services*-layer.

data server class.

- A *class* which is available to *clients* through the `data server` interface.

database.

- A *persistent* repository for *data*.

database management system.

- A generalized collection of integrated mechanisms and tools to support the definition, manipulation, and control of databases.

designated instance version.

- It specifies a particular *instance version* to be used in lieu of the latest *instance version* .

design.

- An aggregation of *design representations*, treated as a coherent unit by designers.

design cycle.

- A set of discrete phases (activities) occurring during the development and use of a product (synonymous with *life cycle*).

design object.

• See *design.*

design representation.

• A view or facet of a design.

distributed database.

• A collection of possibly independent or *federated* database systems. Each system has some set of facilities for exchanging data and services with other members. The glue that ties the system together is the *transaction management* system. It coordinates a sequence of interactions with different nodes, providing data *consistency* and *atomicity.*

dynamic binding.

• A configuration scheme which allows dynamic creation of a *composite configuration* either with the latest timestamped *instance version*, or with a *designated instance version.*

encapsulation.

• A packaging of *data* and *methods* into a single programmatic structure. This provides information hiding.

gemstone proxy.

• An *instance of* the "GemStoneObject" *class.*[1]

get.

• An operation which converts an *instance version* into an *instance.*[2]

index.

• A data structure that a *database management system* uses to expedite the evaluation of a *query* that retrieves a small subset of a large *database.*

initial value.

• A value for an *instance variable* that is computed and installed in the *instance* at *object* creation time.

---

[1] From the Smalltalk-80 workspace, a client can send messages to persistent objects via a gemstone proxy.

[2] An instance is placed in either the user workspace or a collection which has as a handle a Smalltalk-80 class variable.

instance.

• Refers to an *object* created from a *class*.

instance of.

• Describes the relation between an *object* and its *class*. The *methods* and *structure* of an instance are determined by its *class*.

instance variable.

• A variable for which local storage is available in instances. This contrasts with *class* variables, which have storage only in the class (sometimes called slots).

instance version.

• A persistent snapshot of an *instance*.

instance versioning.

• A mechanism that allows tracking of the evolution of *instance versions* during a design *life cycle*.

instantiate.

• An operation which makes a new *instance of* a *class*.

library.

• A datastore unit.

link.

• A complex *object* which is used to model associations and relationships between *objects*.

long transaction.

• A *transaction* that spans multiple *short transactions*. Here, long transactions are *transactions* that result from the interaction of *clients* with a *private library*.

mechanism.

• Here, a structure whereby *objects* work together to provide a required behavior.[1]

---

[1] A mechanism represents a design decision about how collections of *objects* cooperate.

message.

- The specification of an operation to be performed on an object. Similar to a procedure call, except that the operation to be performed is named indirectly through a selector whose interpretation is determined by the class of the object, rather than through a procedure name with a single interpretation.

metadata.

- Information and constructs which allow management of *data* and *schema*.

method.

- A function that implements the response when a message is received by an object.

nested transaction.

- Consists of either a group of primitive operations (such as reads, writes, inputs, outputs) or a group of nested transactions (it is a recursive concept). Nested transactions are used to enhance both performance and modularity. They also provide a finer-grained recovery, allowing better control over *transaction* execution, and simplifying the programming of reliable transaction systems.

object.

- Anything that can be modeled. Here, the primitive element of the *object-oriented paradigm*. Objects combine the attributes of procedures and *data*. Objects store data in variables, and respond to *messages* by carrying out *methods*.

object-oriented design.

- A paradigm where the pieces of the design are *objects* which are grouped into *classes* for specification purposes. In addition to traditional dependencies between data elements, an *inheritance* relation between *classes* is used to express *specializations* and *generalizations* of the concepts represented by the *classes*.[1]

optimistic concurrency control.

- A *concurrency control* mechanism that takes the view that concurrent *transactions* requiring access to the same object may not conflict. *Transactions* are allowed to proceed until a conflict is detected.

---

[1] We have adapted Wegner's [Wegner 87] definition for object-oriented languages to object-oriented design.

paradigm.

• An archetype, exemplar, pattern, or an idea according to which all things of a certain type are made.

path.

• Here, it specifies a sequence of names, which include: a *library* name, a *stream* name, and a *context* name.

persistence.

• Existence of *objects* beyond the life time of the processes that created them.

pessimistic concurrency control.

• A type of *concurrency control* which assumes that concurrent *transactions* requiring to access to the same object will lead to conflict, and hence makes one *transaction* wait for another to complete before it is started.

polymorphism.

• The capability of different *classes* of *objects* to respond differently to exactly the same *messages*.[1]

private data.

• *Data* visible and accessible only to its owner.

private library.

• A *library* which contains *private data*. If a designer requests the creation of a private library, he owns its contents.

protocol.

• A standardized set of *messages* for implementing something. Two *classes* which implement the same set of messages are said to follow the same protocol.

public library.

• A *library* in which designers store stable *instance versions* or stable *configurations* for public use.

---

[1] Method overloading is a kind of polymorphism.

put.

• An operation that converts an *instance* into an *instance version.*

query.

• A declarative specification of a set of *objects* in the *database* that satisfy a set of conditions.

readset.

• The collection of objects read from the *database* by a *transaction.*

relationship.

• A bidirectional association between two *objects.* A relationship is modeled by a pair of mutually inverse *links.*

released version.

• An *instance version* stored in a *public library.*

rollback.

• Actions of the *database management system* restoring a *database* state that existed in the past.[1]

scheduler.

• A portion of the *database system* that arbitrates conflicting requests. A scheduler handles deadlocks and *non-serializability.*

schema.

• A set of *class* definitions (types) that describe the logical structure of an object-oriented *database.*

schema evolution.

• The ability to make a wide variety of changes to the *database schema,* dynamically.

schema version.

• A version of a schema.

schema versioning.

• A mechanism that allows tracking of the history of the *classes* and *inheritance relationships* among *classes.*

---

[1] Rollback is possible only if a past state has been recorded and stored within the repository. The three basic alternative implementations of rollback use snapshots, checkpoints, or a system log.

• **scope.**

• A non-versioned container of objects.

semi-public library.

• A *library* that holds *data* accessible only by only one team/client, or by more authorized teams/clients.

serializability.

• Means that if the operations of two atomic *transactions* are interleaved, the result is the same as if one *transaction* ran to completion before the other started.

server.

• A provider of specialized *services* to *clients*.

service.

• Here, a unit of functionality that the `data server` provides to *clients* to support both *data management* and *data control* in a co-operative CAD engineering environment.

session.

• Here, the duration of a *client* interaction with the `data server`.

short transaction.

• See *transaction*.

slot.

• See *instance variable*.

Smalltalk-80.

• The name of one of the most popular object-oriented programming languages.

specialization.

• The process of modifying a generic thing for a specific use.

static binding.

• A configuration scheme by which the exact *instance versions* that make up the *composite configuration* are specified.

stream.

• A physical representation of a *design cycle*. A stream contains one or more *contexts*.

subclass.

A *class* that is lower in the inheritance *class hierarchy* than a given *class*.

superclass.

A *class* that is higher in the inheritance *class hierarchy* than a given *class*.

structure.

• Describes the *instance variables* of a *class* in the *object-oriented paradigm*.

transaction.

• In a *database*, all access to *persistent data* must take place within a transaction. Transactions serve to group process execution into units (also called operations) that are, from the point of view of other processes, *atomic*; that is, they either happen all at once or not at all. A transaction never completes partially; it either commits (completes successfully) or aborts (the effects of its execution are "*rolled back*", so it appears to other processes as if these effects never occurred at all). In other words, a transaction is normally viewed as both the unit of concurrency[1] and the unit of recovery[2] for *database* applications.

transitive closure.

• A set of *objects* that are transitively related by the same *relationship*.

total ordering.

• Refers to the ability of a parent to have only one successor (synonymous to linear versioning).

two-phase commit.

• A *transaction* atomicity control protocol.[3]

two-phase locking.

• A protocol that ensures *serializability*. This protocol requires that each *transaction* issue lock and unlock requests in two phases: a growing phase,[4] and a shrinking phase.[5]

---

1 As a unit of concurrency, the operations of several transactions can be interleaved so they will not interfere with each other.

2 As a unit of recovery, a transaction either succeeds totally or it has no effect on the object store.

3 During phase one, all hosts are queried as to whether they can commit a transaction. All hosts agreeing to commit are equally prepared either to commit or to abort the *transaction*. All disagreeing hosts send abort messages. Using these votes, a decision to commit or to abort the transaction is reached. There are different versions of the protocol depending on how this decision is made. In phase two, the decision is communicated to all other hosts, which uniformly abort or commit the transaction.

4 A transaction may obtain locks, but must not release any lock.

version.

• A *persistent* snapshot of *data* .

version derivation graph.

• A capture of the evolution of *instance versions*. *Instance versions* in a version derivation graph are related by means of successor/predecessor *relationships*. The root of the version derivation graph is the first instance version created by a client. Successors of an instance version are considered as being derived.

virtual path.

• A history of *check-in(s)* and *check-out(s)* of a *version*. It is a sequence of *paths*.

working version.

• An *instance version* placed in either *semi-public* or a *private library*.

writeset.

• The collection of objects updated by a *transaction*.

---

[5] A transaction may release lock, but may not obtain any new locks.

References

[Afsarmanesh 85] Afsarmanesh, H., McLeod, D.; Knapp, D., and Parker, A. "An Approach to Engineering Design Databases With Applications to VLSI/CAD". International Symposium on New Directions in Computing. pp 177-186. Univ. of Southern California, Los Angeles. CA, USA. 1985.

[Banerjee 86] Banerjee, J., and Kim W.. "Supporting VLSI Geometry Operations in a Database System". Proc of IEEE Int. Conf. on Computer Design (ICCD' 86). pp 409-414. 1986.

[Batory 85] Batory D., and Kim W. "Modeling Concepts for VLSI CAD Objects". ACM Trans. Database Syst. pp 322-346. September 1986.

[Bern 81] Bernstein, P.A. and N. Goodman. "Concurrency Control in Distributed Database Systems". ACM Computing Surveys, 13:2.pp 185-221. June 1981.

[Brei 90] Breitbart, Y., Silberschatz, A., and Thompson, G. "Reliable Management in Multidatabase System". 1990.

[Boehm 88] Boehm, B. "A Spiral Model of Software and Development and Enhancement". IEEE Computers, pp 61-72. May 1988.

[Chen 76] Chen, Peter. "The Entity-Relationship Model-Towards an Unified View of Data". ACM Transactions on Database Systems. Vol 1. pp 9-36. March 1976.

[Chou 86] Chou, Hong Tai, and Kim W. "A Unifying Framework for Version in a CAD Environment". Proc. Int Conf. on Very Large Data Bases. pp 336-344. Kyoto, Japan. August 1986.

[Comer 79] Comer, D. "The Ubiquitous B-Tree". Computing Surveys, Vol. 11 No 2. June 1979.

[Demers 87] Demers, L. et al. "Cheshire: An Object-Oriented Integration of VLSI CAD Tools". 24th ACM/IEEE Design Automation Conference Proceedings. pp 750 -756. 1987.

[Diederich 87] Diederich, J. and Milton J. "Experimental Prototyping in Smalltalk". IEEE Software. pp 50-64. May 1987.

[Dittrich 85] Dittrich, K., and Lorie, R. "Version Support for Engineering Database Systems". IBM Research Report: RJ4769. IBM San Jose Research Laboratory. July 1985.

[GemStone 90] Servio Corp. "The OPAL Programming Guide". Servio Corp. Oregon. Release 1.5. 1990.

[Goldberg 83] Goldberg A., and D. Robson. Smalltalk-80: The Language and its Implementation. Addison-Wesley. 1983.

[Gray 78] Gray, J.N. "Notes on Data Base Operating Systems". IBM Research Laboratory. California. 1978.

[Gray 81] Gray, J.N. ."The Recovery Manager of a Data Management System".ACM Computing Surveys. Vol. 13, no. 2. pp 223-242. June 1981.

[Katz 85] Katz, R. Surveys in Computer Science: Information Management for Engineering Design.Springer-Verlag. Berlin. 1985.

[Katz 86] Katz, R. "Design Version Management".IEEE Design & Test.pp 13-22.University of California, Berkeley. February 1987.

[Katz 87] Katz, R., et al. "A Version Server for Computer Aided Design Data". Proc. ACM/IEEE 23rd Design Automation Conf. pp 27-33. 1986.

[Katz 87b] Katz, R., and Chang, E. "Managing Change in Computer-Aided Design Databases". Proc. of the Int. Conf. on Very Large Databases. pp 455-462. September 1987.

[Katz 90] Katz, R. "Towards a Unified Framework for Version Modeling in Engineering Databases". ACM Computing Surveys. Vol 22, No. 4, pp 375-408. December 1990.

[Kim 88] Kim, W., and Korth, H., and Bancilhon, F. "On Long-Duration CAD Transactions". ACM, pp 408-431. 1988.

[Kim 89] Kim, W. et al. "Features of the ORION Object Oriented Database System". Object-Oriented Concepts, Databases, and Applications. ACM Press. NY. Addison-Wesley. 1989.

[Kim 89a] Kim, W. "A Model of Queries for Object-Oriented Databases". Proc. of the Fifteenth Int. Conf. on Very Large Data Bases. pp 423-432. Amsterdam. 1989.

[Kim 90] Kim, W. "Architectural Issues in Object-Oriented Databases". JOOP. pp 29-38. March 1990.

[Kort 83] Korth, H.F. "Locking Primitives in Database System".Journal of the ACM. Vol 30 No1. pp 55-79.January 1983.

[Kropf 89] Kropf, T., and Feldbush, F. "DISK – An Object-Oriented Design Tool for the Synthesis and Evaluation of Digital Circuits". Proceedings of VLSI and Computer Peripherals. pp 28-30. 1989.

[McLeod 90] McLeod, Dennis, and Cárdenas, Alfonso. Research Foundations in Object-Oriented and Semantic Database Systems. Prentice Hall Series in Data and Knowledge Base Systems. New Jersey, USA. 1990.

[Maier 89] Maier, D. "The GemStone Data Management System". Object-Oriented Concepts, Databases, and Applications. ACM Press.NY.Addison-Wesley.1989.

[Maier 89a] Maier, D. "Making Databases Systems Fast Enough for CAD Applications". Object-Oriented Concepts, Databases, and Applications.ACM Press.NY.Addison-Wesley.pp 573-582.1989.

[Meyer 87] Meyer, B. "Reusability: The Case for Object-Oriented Design".IEEE Software. pp. 50-63. March 1987.

[Meyer 88] Meyer, B. Object-Oriented Software Construction.Prentice Hall. UK. 1988.

[OE 90] ObjectStore Product Profile. Object Design, Inc. MS, USA. 1990.

[OV 90] Objectivity/DB Product Profile.Objectivity, Inc. CA, USA. 1990.

[Roy 70] Royce, W. "Managing the Development of Large Software Systems: Concepts and Techniques".Proceedings, WESCON, August 1970.

[Sellers 90] Sellers, B., and Edwards, M. "The Object-Oriented Systems Life Cycle". Communications of the ACM. Vol 33, No 9. September 1990.

[Stonebraker 90] Stonebraker, M., et al. "Third-Generation Database System Manifesto". Sigmod Record, Vol. 19, No. 3. ACM press. September 1990.

[Thom 83] Thomas, D.E., et al. "Automatic Data Path Synthesis". IEEE Computer Magazine,.Vol 16, No.12, December 1983.

[Thompson 91] Thompson, C., Wells, D., Joseph J., and Thatte S. "Object-Oriented Databases: Design and Implementation". Proc. of the IEEE, Vol. 79. January 1991.

[Ullman 80] Ullman, Jeffrey D. Principles of Database Systems. Computer Science Press. Standford University. 1980.

[VE 90] Versant Object Technology. "The Total Performance OO-DBMS". CA, USA. 1990.

[Wegner 89] Wegner, P. "Concepts and Paradigms of Object-Oriented Programming". Expansion of Oct 4 OOPSLA-89 Keynote Talk. Brown University. October 1989.

[Zdonik 87] Zdonik, S., and Smith, K. "Intermedia: A Case Study of the Differences Between Relational and Object-Oriented Database Systems". OOPSLA'87 Proceedings. pp 462-465. October 1987.

[Zimmermann 89] Zimmermann, G., and Siepmann, E. "An Object-Oriented Data Model for the VLSI Design System PLAYOUT".ACM/IEEE 26th Design Automation Conference. pp 814-817. 1989.