PCLOS: A Flexible Implementation of CLOS Persistence

Andreas Paepcke
Hewlett-Packard Laboratories
1501 Page Mill Rd.
Palo Alto, California 94304
paepcke@hplabs.hp.com

Abstract

We describe the design of a prototype which makes objects persistent. Our target language is the CommonLisp Object System (CLOS), although we pay attention to the eventual sharing of data with other languages. Our design is very flexible, in that it allows the simultaneous use of multiple, different databases. This is accomplished by defining a virtual database layer which consists of a core protocol that is expected to be implemented on all databases, and of protocol adapters which accommodate features offered by some databases, but not by others. This virtual database has been implemented for a simple, single-user, in-core data store, and for Iris, a multi-user, object-oriented database management system. We outline the advantages of the CLOS Metaclass Protocol for implementing object persistence or other low-level modifications to the CLOS implementation.

Keywords and phrases: Object persistence, CLOS.

1 Introduction

The attraction of making data structures persistent has been recognized for some time. PS-algol [ea82, ABC*83, AM86] has, for instance, attempted to introduce persistence into an Algol-like language. The spread of object-oriented programming has been followed by several more efforts in this direction. Coral3 [ML87] and GemStone [CM84, MSOP86] are examples of systems that provide object persistence for Smalltalk-80 [Gol84]. Other projects, like *Altaïr* O_2 [Ban87, BBDV87, BLRV87, LRV87, AH87], attempt to add objects and persistence to existing languages, like C, Lisp or Basic. Some efforts [BKKK87, GK87, KBC*87, Rei85] attempt to fuse data definition, manipulation and persistence into one system, using a single language. All of these systems use one particular database or object server to provide persistence.

We plan to experiment with object models in general and with object persistence in particular and have therefore decided to modify an object-oriented system so that its objects may reside on persistent storage. We do not, however, want to be limited to particular databases, and therefore need a design that is flexible enough to allow the use of significantly different databases and that does not exclude the use of multiple languages sharing data. A prototype of such a system has now been completed. It is called Persistent CommonLisp Object System (PCLOS) and this paper presents its design and our initial experiences with it.

2 Design Goals and Early Decisions

The following sections explain our design goals and some of our early design decisions.

2.1 Support for Multiple Databases

Functionality must usually be paid for by decreased performance, and database management systems differ in their position on this functionality versus performance spectrum. The ability to accommodate concurrency is, for instance, a piece of functionality which requires sophisticated mechanisms that impact performance. Systems which are particularly tolerant of hardware failure will also generally incur some run-time cost. We expect that applications will have varying needs which will determine how fast object storage or retrieval must be, and which pieces of functionality are indispensable. Application developers will therefore want to pick among various storage facilities offering different tradeoffs to handle object persistence. Such a choice might also be made on the grounds that data is to be shared with other parts of an organization which already use one particular database management system.

We therefore decided that our system is to support multiple data storage mechanisms, and that we want the ability to store objects using any of these mechanisms. The interface to object storage must be defined well enough that an adapter module may be written for any database or storage scheme that meets some minimal requirements.

2.2 Support for Multiple Languages

If a system is to benefit a wide range of users, it should not unnecessarily limit the choice of languages that can access it. Programming languages differ in their expression of data structure, and earlier experiments have taught us [BK86] that we should not expect data persistence to make these differences transparent. Even among object-oriented languages, immediate, perfectly matched sharing of data *structure* might not be possible, for example because some object models allow multiple inheritance, while others provide single inheritance. We believe, however, that data sharing may be useful, even if arrangements must be made to accommodate different capabilities of expressing the structure of data.

Another design goal is therefore the support of multiple programming languages sharing persistent objects. We would like the sharing of data *structure* to degrade gracefully, as producers and consumers of data are made progressively different. We expect stored data to be organized in such a way that objects in one object-oriented language may be mapped to analogous objects in another object-oriented language through the storage medium. For non-object-oriented languages we can still utilize the fact that the stored state of an object can be viewed as a logical unit. Our work does, however, focus on an object-oriented view of data, and we will use the term "objects" to refer to units of storage.

2.3 Integration of Persistence

Wherever possible we would like to introduce longevity as a transparent property of objects in an existing language. Programs are to run on transient and on persistent objects alike, and we want to avoid syntactic changes to languages. We have found that two conditions may necessitate a deviation from this guiding principal: when database concepts prove to be so useful that they should be introduced into the realm of programming, appropriate means must be found to reflect these concepts in the programming language. A second reason for compromising on complete transparency of persistence is the desire to bring performance to acceptable levels. Whenever we do introduce persistence-related operations, we try to incorporate them naturally into the programming language.

2.4 Our First Supported Language

Our initial target and implementation language is the CommonLisp Object System (CLOS) [ea87b]. This is an object model which is in the process of being standardized and is integrated into the CommonLisp language. The language is particularly well-suited because it offers the *metaclass* concept which is a well-defined protocol for extending its implementation. CLOS is defined in [ea87b], and we merely summarize the aspects relevant to making CLOS objects persistent: CLOS supports the concept of *classes* which define *slots* that hold pieces of state. In general, slots are instance-allocated which means that every instance of a class controls a private set of values for its slots. The value of a class-allocated slot, on the other hand, is shared by all instances of the class. The combined values of all the slots represent the state of an object. Classes may inherit from multiple parents. Inheritance means that instances of a subclass will have the union of all slots of the subclass and its ancestors as their state. A *method* is conceptually a set of programs, from which one is selected to run when a method is invoked. This selection is based on the types of the arguments passed to the method.

2.5 Our First Supported Databases

Since we want to avoid making hidden assumptions about the properties of one particular database in our implementation, we decided to support two very different storage mechanisms from the start.

Figure 1: Design Model

The first is a very simple in-core database serving one user at a time and allowing data to be written to a file on request. The second database is Iris, a multi-user, object-oriented database offering transaction management and sophisticated query processing [ea87a].

3 System Model

Figure 1 shows the current model of our system as seen by the implementor of object persistence for one language. In order to support multiple databases, we need to provide an interface between the language implementation at the top of the figure and the set of data storage mechanisms at the bottom. We call this conceptual interface a *virtual database.* Every physical database appears to the language implementation as one such virtual database.

The virtual database consists of a *core protocol* and multiple *protocol adapters.* All operations of the core protocol must be implemented for each physical database. It is therefore kept as simple as possible, while still providing the capability to store and retrieve data comprising the state of objects. A virtual database, of which only the core protocol is implemented, would therefore be sufficiently powerful to provide persistence 'without frills'. While the core protocol is sufficient to store and recover objects and associated administrative information from most databases, an unfortunate drawback of using the core protocol alone is the fact that it would prevent the use of any advanced features which some databases might offer. One protocol adapter is therefore added to the virtual database for each *concept* which is useful but which may not be supported by all physical databases. Examples of such concepts are transactions, support for the storage of particular language datatypes, special query capabilities or unusual data locks. We use the protocol adapter idea to help us organize the various database features and to define unified programmatic interfaces to them. This usage could conceivably be expanded towards complete formal specifications, but we have taken no steps in this direction.

The *implementation* of a protocol adapter is called a *protocol converter*. One such converter is provided for each of the physical databases. If the associated adapter's concept is directly supported by a converter's database, the converter is straight-forward. Otherwise, it may choose to either implement the concept, or to return a failure indication when used.

The virtual database guarantees that we may make use of very simple database managers, while not restricting the use of sophisticated features supported by advanced database management systems.

When implementing the model of Figure 1, we generally expect that we must make changes in the *implementation* of a language, in order to have object state accesses go through the virtual database to physical storage. The interface ensures, however, that we may subsequently add database management systems without having to change the language implementation further. In addition, the definition of the virtual database is a very concise specification of the capabilities that may be relied upon when a language layer is modified to use object persistence.

3.1 The Core Protocol

The core protocol uses as its basic data structure a table model, similar to relational databases. The following assumptions are made about tables:

- Tables have unlimited height and width.
- They are uniquely named.
- Rows of tables are uniquely identified by a row number.
- Tables, and rows within them, may be dynamically created and destroyed.
- Field values are not required to have type restrictions.

Operations are defined to perform these functions. The permissive typing for values of fields is required for object-oriented languages that do not restrict the type of values assigned to slots.

3.2 Protocol Adapters

No universally valid implementation rule can, of course, be stated for the realization of protocol adapters. But we outline two adapters in this section to illustrate how we use the scheme.

Some databases allow the use of built-in types, such as `integer` when declaring data items in a schema. The use of such built-in types is desireable because of efficiency reasons or to facilitate data access through other languages. PCLOS handles the mapping of language datatypes to datatypes understood by various databases through a protocol adapter. It is implemented through a class hierarchy of *type mappers*. Instances of the root of this hierarchy convert all language items to strings which are assumed to be a supported datatype on all databases. A subclass of this root type mapper is written for each database to be supported. Such a subclass will shadow the parent mapper's conversion for the types that are directly supported by the associated database. Each protocol converter for the type mapping adapter is thus an instance of an appropriate type mapper.

Transactions are also represented in the virtual database as a protocol adapter. When the message `begin-transaction` is sent to the implementation of the virtual database for Iris – which directly supports transactions – an appropriate command is passed to the database: the protocol converter is trivial in this case. When the same operation is invoked on the virtual database implementation of the single-user in-core database, the concurrency control aspect of the transaction concept is ignored. But all subsequent operations on the database are logged as they pass through the implementation of the virtual database. The protocol converter then does implement transaction rollback.

4 Implementation Strategy

Implementing the system model described above implies that two mappings must be provided: language objects must be mapped onto the virtual database, and the virtual database must be mapped onto the physical database management systems. The first of these two is currently done as shown in Figure 2. We reserve one table in each database for information about the classes of objects stored in the database. This table, called the MasterClassTable, can be used to check for class consistency, or to communicate a class structure to other users.

All instances of one class are then stored in one table, with rows representing instances, and columns holding values of instance variables. The concept of 'table' therefore models the concept of 'class' in its role as the keeper of a set of instances related by a common structure.

The mapping of the core protocol onto physical databases varies with the characteristics of the database. For object-oriented databases, this mapping is particularly easy, since a database type can be created for each virtual table. An instance of such a type models one row, in that its state contains all the information contained in the columns of a row. A set of database functions is defined for each type, where each function takes an instance of the type and returns one piece of the object's state. This is used to model the retrieval of one virtual field. Since functions are made updatable, Master Class Table:

Row ID	Class Name	Inheritance	Slot Names	...
0	“Ship”	NIL	“Captain Tonnage Flag”	...
1	“Book”	NIL	“Title Author Publ. Price”	...
⋮	⋮	⋮	⋮	⋮

Ship Instances:

Row ID	Slot 0	Slot 1	Slot 2
0	“Cook”	3	“US”
⋮	⋮	⋮	⋮

Book Instances:

Row ID	Slot 0	Slot 1	Slot 2	Slot3
0	“Oil Spills”	<corp-obj>	“Glib, Inc.”	32.50
1	“Barter Economy”	R. Blank	“Nostalgia Publ.”	<pig-obj>
⋮	⋮	⋮	⋮	⋮

Figure 2: Mapping Language Objects onto the Virtual Database

the same function may be used to retrieve and to set one virtual column of one virtual row.

If we combine the mapping of programming language objects onto the core protocol with the mapping of the core protocol onto an object-oriented physical database, we find the following, very natural relationship between language objects and their storage:

Language classes	⟶	Database types
Language instances	⟶	Database instances
Language slots	⟶	Database functions

The virtual database and the implementation strategy described here do not make sharing of data among multiple languages trivially easy. But several aspects do help us along towards this goal. The simplicity of the core protocol ensures that there is a clear way to describe the structure of the data as it is stored in the database. In the case of an object-oriented database the structures used to implement the core protocol will be more elegant than tables and rows, but we know that the core protocol represents an equivalent representation.

Figure 3: Modules of the PCLOS Implementation

In addition, the database access implementation of a language can use the MasterClassTable to recover data structure that might have been lost in the storage implementation.

Protocol adapters represent a further benefit of the virtual database abstraction in the context of multi-language data access because they force a separation of functionality description and implementation. The type mapping adapter, for example, does not solve the problem of translating an elaborate built-in type of one language to an equivalent type in another language, but it isolates the problem which is helpful.

5 Some Implementation Details

The following sections on selected implementation details begin with a description of the current architecture of the prototype which implements the design model. We then describe our attempts at handling performance issues and the advantages of CLOS in the context of modifications to the language implementation.

5.1 Current PCLOS Architecture

Figure 3 restates the conceptual system view of Figure 1 in terms of system modules of our prototype. The box at the top represents the CLOS implementation. Each protector and database interface (DB-IF) pair represents the implementation of the virtual database for one session on one of the databases.

```
(find-all iris-protector
          'ship
          '(and
              (> tonnage 2)
              (or
                (= captain ''Cook'')
                (= flag ''Kuwait''))))
```

Figure 4: A Sample Query

Figure 3 thus shows the modules active in a session where CLOS objects are located on either of two physical databases of kind **A**, or on one database of kind **B**.

The database interfaces above each DBMS in the figure are instances of database interface classes of which one is defined for each kind of DBMS. Each instance has the responsibility to translate core protocol operations into actions on the associated physical database, and to implement various protocol adapters.

Above each database interface, we place a *protector* object. Each of these protectors is an instance of a single system class, and it represents one database session to the CLOS implementation and directly to users. Protectors perform administrative functions, such as object caching, and they are responsible for presenting the unified virtual database interface to the language layer.

Finders in Figure 3 represent the query protocol adapter. Queries are expressed by users or programs in terms of CLOS classes, slots and objects. They are then converted by a finder object into queries on the virtual database, and are passed on through the protector of one of the sessions to the associated database interface object. The 'virtual query' is re-formulated there to be recognized and solved by the underlying DBMS. The same virtual query could be handed to more than one of the open sessions.

The query in Figure 4 is an example of our associative search capabilities at the core-protocol level. The system converts the form shown in the figure into an equivalent intermediate form that replaces the type and slot references with table and column specifications. The database interface module for the Iris database, for example, then converts this intermediate form to a query involving objects, types and functions. A "complex-query" protocol adapter supported only by the Iris database allows much more sophisticated queries that include multi-valued results and existential variables and may range over multiple classes.

5.2 Performance

Access concurrency is maximized, if no objects are cached into memory but reside on a database only. Every atomic slot manipulation then accesses the database without automatically caching the affected object into memory. Such a scheme is feasible for slots and objects that are accessed rarely, and it is the default in PCLOS. This mechanism is, however, too slow when object slots are accessed frequently. In addition to this mode of operation, PCLOS therefore provides a large degree of flexibility for determining locality of information. All but the last of the following facilities has been implemented:

- Individual and recursive object caching/snapshots.
- Slot-level caching/snapshots.
- Transient slots.
- Transient objects.
- Caching within the scope of a method (method auto-cache).

Objects may be brought into memory within a transaction, which is in effect a caching operation because it write-locks the database copy. If this operation is not done under the protection of a transaction, it represents a snapshot of the object and will not prevent the database copy from being modified. For convenience, these operations may be done recursively for all objects reachable through references starting with some root object.

Granularity of locality control is further increased by the ability to cache or snapshot individual slots, while leaving the rest of the object exclusively on the database.

Slots whose state is not important across sessions may be declared transient. Values for such slots are only kept in memory, and will potentially differ among users sharing the object. If none of the slots of an object need to persist, the object can be made transient. In this case the database does not hold a copy of the object.

We plan to facilitate object caching further in the context of method execution. This will include the option to have objects that are passed as parameters cached automatically, and to have them uncached when the method terminates. We also plan to provide a macro which causes specified objects to be cached while operations within its scope execute.

Caching, while necessary for reasons of performance, can destroy the important advantage of transaction rollback, if the rollback does not also restore the state of cached, that is in-memory objects. The PCLOS prototype therefore optionally takes a snapshot of cached objects when a transaction is begun. Any subsequent rollback then includes these objects.

5.3 The Use of CLOS

The addition of persistence to an object system implementation is generally intrusive, in that permanent modifications must be made at a low level. Here are some examples of requirements that call for low-level changes:

- Every slot (instance variable) access must be intercepted so that the appropriate slot manipulation is done on the database, if the proper data structures are not cached.
- It must be possible to inspect most low-level aspects of class hierarchies so that appropriate virtual database schemas may be produced.
- It is advantageous to add room for administrative information to the memory representation of instances.
- Access to the entire state representation of instances must be provided to support rollback for cached objects.

CLOS is exceptionally well suited to solving these problems for two reasons:

- All important data structures of the CLOS implementation are themselves objects. This includes, in particular, the representation of classes.
- The CLOS Metaclass Protocol is flexible enough to augment virtually any aspect of the CLOS implementation.

The Metaclass Protocol defines interfaces to class definition and re-definition, instance allocation, slot access mechanisms, the inheritance scheme and many other areas. Every user-defined class is conceptually an instance of some metaclass. Different metaclasses therefore implement sets of classes which are potentially radically different. All of these classes may coexist during one session. CLOS objects were made persistent without a single change to the standard language implementation. Everything was accomplished by writing a metaclass that inherits from the default metaclass and adds all persistence-related behavior. One advantage of this is that persistence capabilities may be added to a system by simply loading appropriate modules. The CLOS session is not disrupted, and modules whose classes are not of the PCLOS metaclass operate properly without change or recompilation. The remainder of this section outlines how our metaclass works.

PCLOS objects which are created during a Lisp session or which are retrieved from a database through associative search must somehow be represented in memory to provide a proper destination for messages sent to them and to preserve Lisp *eq*ness for objects. This is done through the use of so-called *husk* objects which are regular objects that do not, however, contain slot values unless they are cached or transient. In this latter case they are very similar to standard CLOS objects. The ability to create and manipulate husk objects is embodied in the `pclos-class` metaclass which inherits from the standard metaclass `class` and uses four mechanisms to accomplish its goals:

- Appropriate methods inherited from `class` are shadowed.
- Some slots are added to class objects to hold persistence-related information.
- The class precedence list of user classes is modified during the processing of `defclass` forms to include a persistence-related class called `persistent-class-parent-class`.
- The physical storage of instances is modified to make room for some administrative information needed on a per-instance basis.

Standard metaclass methods which must behave differently for persistent classes are, of course, all methods involved with slot accesses, most notably those that implement `slot-value`. In addition, all slot access optimization is suppressed by the `pclos-class` metaclass, because instances may be transient or persistent at run time. Different action must be taken to access slots in these cases and compile-time decisions are therefore not possible.

Other modifications of standard metaclass behavior involve the implementation of the new slot option `:transient`. This requires shadowing of methods involved during the processing of `defclass` forms to accept this option as legal and to perform appropriate bookkeeping.

The class from which all user classes inherit automatically through the mechanism described above introduces no slots to those classes but makes relevant methods available to their instances. These include, for example, methods to `cache` and `uncache` instances, to make them persistent or transient, or to obtain persistence-related information from them.

Information needed for each instance includes a pointer to the database interface object currently responsible for it and an indication of whether the instance is currently cached. This is kept in fields of an array that is normally used by CLOS to contain slot values only. Methods which index into this array simply use a fixed offset to index past this administrative information.

In addition to these mechanisms, the `pclos-class` metaclass features methods needed to cache class-allocated slots, to map slots onto virtual columns by using their position in the class definition and to take care of other necessary functions related to persistence.

6 Status and Plans

Persistence for CLOS objects has just been implemented and is now being tested. Figure 5 shows an example configuration of our system. Workstations are connected through a local area network. One or more Iris server processes provide database service, and some workstations use the in-core database.

Initial experiments confirm the obvious suspicion that object caching is important for performance and must be optimized. Test users have quickly expressed their desire to specify arbitrary groups of objects which should be cached into memory together whenever one of the objects is referenced. A similar mechanism has been proposed in [Row86].

Figure 5: Example System Configuration

Notification locks were another mechanism that was requested quickly, when programmers began thinking about their use of the system. These would allow reading and writing of an item in the database, but would cause 'interested parties' to be notified whenever the item was modified. PCLOS does not currently support such a facility.

We need to investigate the problems and opportunities arising from the simultaneous use of multiple databases in one session. Deadlocks are one potential problem to be dealt with. The inclusion of many databases in a query is one of the opportunities we plan to take advantage of. This means merely that the system will submit the same query to several databases or that we might specify some filtering operations on the data from the various scans.

It is too early to tell what system designers will do with the new ability to search associatively over the object space. Through use of our prototype we hope to gain some insight into searching needs of systems and applications programmers, and to expand PCLOS query capabilities accordingly.

We have found that the rollback support for cached objects is very useful. Since all state information of an object-oriented system resides in objects, transaction rollback may be used as a convenient *undo* facility.

The PCLOS experiment is proving to be very valuable in increasing our understanding of object persistence, and its usefulness in this respect is by no means exhausted. As we learn more, we will assess the suitability of our virtual database model and the usefulness of protocol adapters.

7 Acknowledgments

Brian Beach, Jim Kempf and Joe Mohan laid extensive groundwork for PCLOS with their DOOM project. Nancy Kendzierski has improved the system model and its implementation through her suggestions. Alan Snyder, Dennis Freeze and Craig Zarmer have helped to clarify many aspects of previous versions of this paper, and initial test users have cheerfully reported the painstakingly collected details of their system crashes to me – with a smile.

References

[ABC*83] M.P. Atkinson, P.J. Bailey, K.J. Chisholm, W.P. Cockshott, and R. Morrison. An approach to persistent programming. *The Computer Journal*, 26(4):360–365, 1983.

[AH87] Timothy Andrews and Craig Harris. Combining language and database advances in an object-oriented development environment. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications.*, Association of Computing Machinery, 1987.

[AM86] M.P. Atkinson and R. Morrison. Integrated persistent programming systems. In B.D. Shriver, editor, *Proceedings of the 19th Annual Hawaii Conference on System Sciences*, pages 842–854, , , 1986. Vol. IIA, Software.

[Ban87] François Bancilhon. *Object Oriented Multilanguage Systems: the Answer to Old and New Database Problems?* Technical Report, Altaïr, BP 105; 78153 Le Chesnay Cedex; France, October 1987.

[BBDV87] François Bancilhon, Véronique Benzaken, Claude Delobel, and Fernando Velez. *The O_2, V0 Object Manager Interface.* Technical Report Altaïr 11-87, Altaïr, BP 105; 78153 Le Chesnay Cedex; France, September 1987.

[BK86] Brian Beach and James Kempf. *DOOM: Permanent Objects for Common Lisp.* Technical Report STL-TM-86-09, HP Labs, September 1986.

[BKKK87] Jay Banerjee, Won Kim, Hyoung-Joo Kim, and Henry F. Korth. Semantics and implementation of schema evolution in object-oriented databases. In Umeshwar Dayal and Irv Traiger, editors, *Proceedings of the ACM Special Interest Group on Management of Data*, Association of Computing Machinery, 1987.

[BLRV87] Gilles Barbedette, Christophe Lécluse, Philippe Richard, and Fernando Velez. *The O_2 Programming Environment, Version V0.* Technical Report, Altaïr, BP 105; 78153 Le Chesnay Cedex; France, October 1987.

[CM84] G. Copeland and D. Maier. Making Smalltalk a database system. In *Proceedings of the ACM/SIGMOD International Conference on the Management of Data*, 1984.

[ea82] Malcom Atkinson et al. PS-Algol: an Algol with a persistent heap. *Sigplan Notices*, 24–30, July 1982.

[ea87a] D. Fishman et al. Iris: an object-oriented database management system. *ACM Transactions on Office Information Systems*, 5(1):48–69, April 1987.

[ea87b] Daniel G. Bobrow et al. *Common Lisp Object System Specification.* Technical Report 87-001, ANSI, September 1987.

[GK87] Jorge F. Garza and Won Kim. *Transaction Management in an Object-Oriented Database System.* Technical Report ACA-ST-292-87, MCC, September 1987.

[Gol84] Adele Goldberg. *Smalltalk-80: The Interactive Programming Environment.* Addison Wesley, 1984.

[KBC*87] Won Kim, Jay Banerjee, Hong-Tai Chou, Jorge F. Garza, and Darrell Woelk. Composite object support in an object-oriented database system. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications.*, Association of Computing Machinery, 1987.

[LRV87] Christopher Lécluse, Philippe Richard, and Fernando Velez. *O_2, an Object Oriented Data Model.* Technical Report Altaïr 10-87, Altaïr, BP 105; 78153 Le Chesnay Cedex; France, September 1987.

[ML87] Thomas Merrow and Jane Laursen. A pragmatic system for shared persistent objects. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications.*, Association of Computing Machinery, 1987.

[MSOP86] David Maier, Jacob Stein, Allen Otis, and Alan Purdy. Development of an object-oriented DBMS. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications.*, Association of Computing Machinery, 1986.

[Rei85] Stephen P. Reiss. *GARDEN: An Environment for Graphical Programming.* Brown University, October 1985. Reference and Programmers Manual.

[Row86] Lawrence A. Rowe. A shared object hierarchy. In Klaus Dittrich and Umeshwar Dayal, editors, *Proceedings of the International Workshop on Object-Oriented Database Systems*, Association of Computing Machinery, 1986.