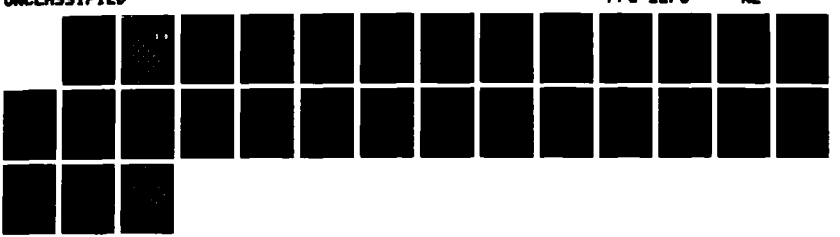
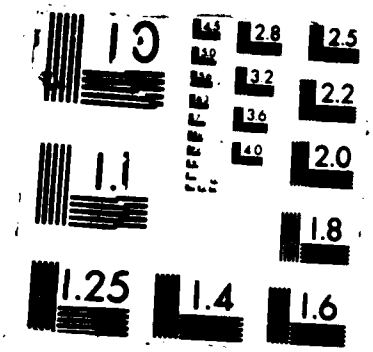
AD-A182 903 TOWARD A PERSISTENT OBJECT BASE(U) CARNEGIE-MELLON UNIV 1/1
PITTSBURGH PA SOFTWARE ENGINEERING INST J R NESTOR
JUL 86 CMU/SEI-86-TM-8 ESD-TR-86-215 F19628-85-C-0003
UNCLASSIFIED F/G 12/5 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0

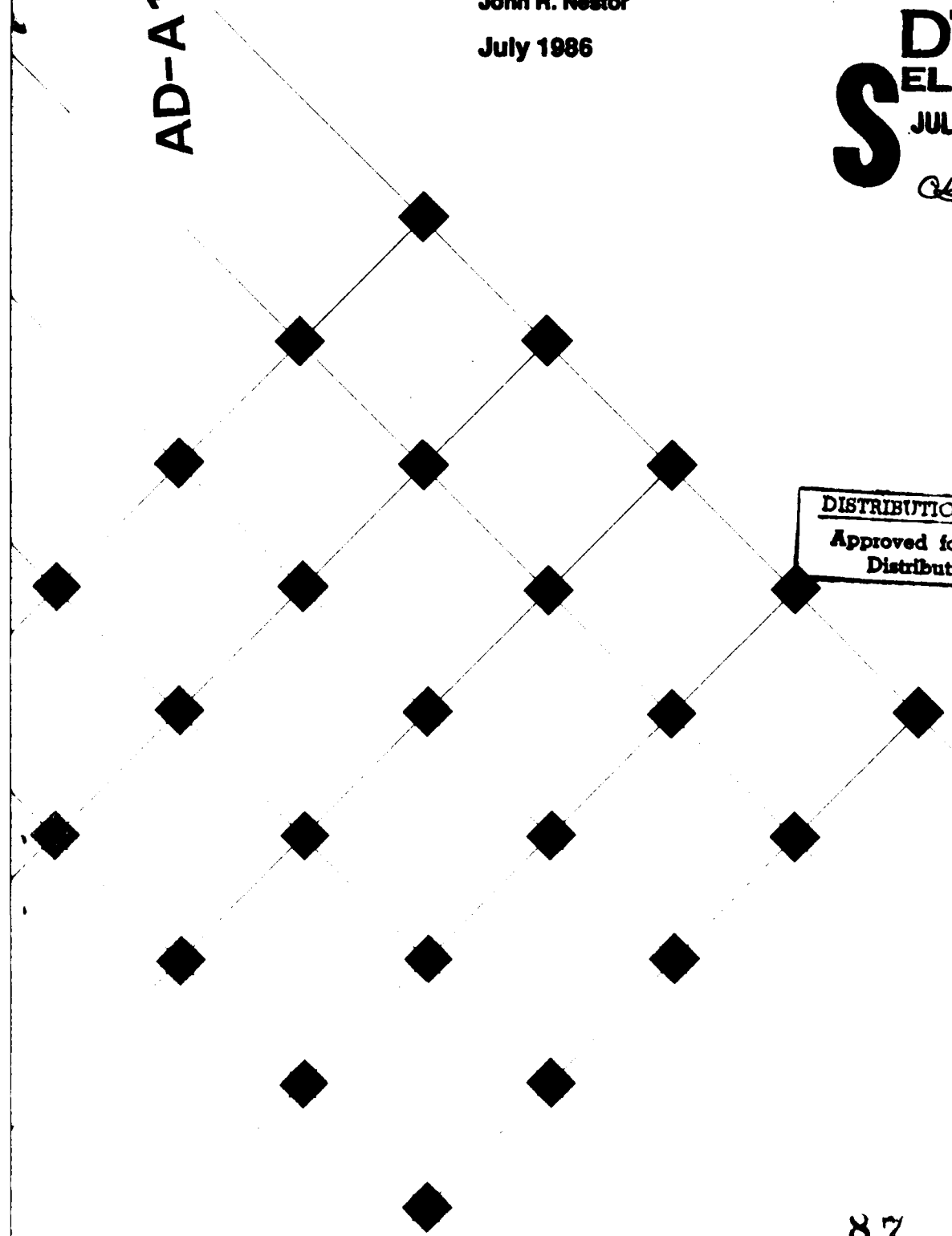
Technical Memorandum
CMU/SEI-86-TM-8

Carnegie-Mellon University
Software Engineering Institute

Toward a Persistent Object Base

by
John R. Nestor

July 1986



Technical Memorandum
SEI-86-TM-8
July 1986

Toward a Persistent Object Base

by

John R. Nestor
Software Engineering Institute

This work was sponsored by the Department of Defense.

Toward a Persistent Object Base

John R. Nestor

ABSTRACT To better understand the needs of future programming environments, two current technologies that support persistant data in programming environments are considered: file systems and data base systems. This paper presents a set of weaknesses present in these current technologies. These weaknesses can be viewed as a checklist of issues to be considered when evaluating or designing programming environments.

1 Introduction

Every programming environment must support not only transient data that is used during computation but also support persistent data that is kept over some period of time. Two widely used current technologies support persistent data: file systems and database systems. There is increasing recognition that neither of these technologies alone will provide an adequate basis for the next generation of programming environments. Most new environment efforts are moving toward a more object oriented approach that is a synthesis of ideas from file systems and databases. Some examples are CAIS [DoD 85], the ESPRIT Portable Common Tool Environment [ESPRIT 85], the Common Lisp Framework [CLF 85], and Arcadia [Taylor 86]). This next generation of technology will be referred to as persistent object bases.

To better understand the nature of the technology needed by future programming environments, this paper considers the weaknesses that will have to be eliminated in traditional file systems and database systems to create a first class persistent object base. Section 2 sets the context for later sections by discussing the character and needs of future programming environments. Sections 3 and 4 cover, respectively, the weaknesses of traditional file systems and database technologies. Section 5 presents conclusions.

2 Context

Modern software technologies allow software engineers to automate many of the processes that are often implemented by inefficient manual or semi-automatic procedures. Such improvements increase our expectations, leading to larger software projects. Larger projects, in turn, require improved communications among managers, users, designers, and maintainers of such projects.

As the software to be produced grows in size and complexity and the communication requirements increase in scope, the tools required to develop and support such software must become more powerful, and the computational system to support the software tools must grow proportionately in scope. In place of a single batch or time sharing machine, increasing emphasis is being placed on use of workstations, distributed computation, and networks to integrate

previously separate computer systems into a single vast communication and computational system. Not only must the hardware evolve, but the environment itself must be developed, upgraded, and enhanced over a lifetime of many years.

Programming environments have become a focal point for much of the work directed toward improving the practice of software engineering. Such environments provide support for software development, management, and maintenance. There are some primitive programming environments already available; there are many next generation environments currently being designed; and work on environments will be a major technical thrust of software engineering for many years to come. There are two top level design goals that will make future environments successful: openness and integration.

Openness refers to the ability to incorporate tools, methodologies, and technologies into the environment as needs and opportunities arise. For an environment to be open, it must provide a set of interfaces that permit new features and tools to be easily inserted. The degree to which an environment can be extended to support a wide variety of new tools and methodologies is one measure of the degree to which that environment can be considered to be open. Openness can also be enhanced by the way in which the interfaces are made available; public availability (as opposed to proprietary control), quality documentation, ease of use, acceptable performance, stability, portability, and standardization can all contribute to the openness, in actual practice, of an environment.

Integration means that the components of the environment work together through a uniform interface, style of operation, and communication medium. The cooperation of the components allows for better use of information sharing, resulting in an intelligent environment.

Though openness and integration are important to software development environments, most systems to date have emphasized openness over integration, or vice versa: There are few existing examples of systems that achieve both. Nevertheless, this tension can be resolved in a way that will enable both goals to be achieved; the key lies largely in the design of the infrastructure of the environment, the kernel parts on which all other tools, features, and methodology support are built. If the infrastructure is not properly designed, increasing complexity of our environments and the systems they are used to construct will make quality increasingly difficult to achieve.

In earlier systems, the infrastructure was provided mainly by the operating system, in which the primary concern was resource allocation and scheduling. As a result of improved hardware technology, new software engineering tools, evolving views of the software development process, and ever increasing expectations, a shift of emphasis has occurred in our view of the role of the infrastructure.

Persistent object bases are a key part of the infrastructure of future programming environments. Providing a high-quality persistent object base is a necessary, although not sufficient, condition for achieving the full potential of future programming environments.

3 Weaknesses of Traditional File Systems

This section considers five areas where traditional file systems are inadequate for persistent object bases: organization, abstraction, history, attributes, and synchronization. Unix[1] [Ritchie 74] is used here as an example of a traditional file system. Other traditional file systems differ in their details from the Unix file system but display essentially similar weaknesses.

3.1 Organization

The Unix file system is organized as a tree of files, each of which is either a directory or an ordinary file.[2] Within the tree, directories appear as inner nodes and files appear as leaf nodes. The root of the tree is a unique directory from which all directories and files can be reached. Each directory is a mapping between file names and the files themselves. Each file has a unique path name given by the path from the root directory to the file. For example, the path name `/usr/bin/man` is for a file named `man` that is reached from the root directory via first the `usr` directory, then via the `bin` directory.

One problem with a tree structured file system is that the user is forced to represent a system in a way that does not reflect the structure of the data in the system. A related problem is that as a system evolves the user periodically must do major reorganizations of the data within the file system. These reorganizations are needed because the preexisting hierarchical structure increasingly deviates from the actual logical relationships.

Consider, for example, a system being built as part of some project called Q_Development. A directory is built for the project.

```
/projects/Q_Development
```

Initially, all files for the project are placed in that directory. Soon the number of files in that directory has grown to where more structure is needed. Suppose that both documentation files and program files exist. To provide more structure, two new directories are created.

```
/projects/Q_Development/documentation
/projects/Q_Development/program
```

All of the files are moved into one or the other of these two directories. Not only is there the extra work involved in moving the files into the two new subdirectories, but any shell scripts that referred to `Q_Development` must now be changed to refer to one or the other or both of the two new subdirectories. For a persistent object base, no moves should be required and existing shell scripts should remain unchanged. Additional information would be added on top of the existing structure.

---

[1] Unix is a trademark of AT&T.

[2] There are also special files and links that for simplicity are not discussed here.

Consider next that it is time to release the Q system to users. Users should have the Q executable file and the Q user manual, but not the Q source code or the Q internal documentation. These files are a subset of the files in the two subdirectories. Since users should not have to know about the substructure of the Q project directories and be confused by all those other files that don't matter to them, a new directory is created to hold copies of the files that the users will need.[3]

```
/release/Q
```

Moving files was bad enough, but in this case there are actually two copies of the same files.[4] For a persistent object base, information would be added, but files would not be moved or copied.

Finally, consider that it is time to produce a new version of the Q system while leaving the previous version of Q around. To do this, the directories must be split.

```
/projects/Q_Development/documentation/V1
/projects/Q_Development/documentation/V2
/projects/Q_Development/program/V1
/projects/Q_Development/program/V2
/release/Q/V1
/release/Q/V2
```

Here all the old files are moved into the V1 directories. The V2 directories will be used for the new version of the system. A simple way to do this is to start by copying all the V1 files into V2. Work on the new version then can be done by changing the V2 file while leaving the V1 files intact.[5] Furthermore, when versions were introduced, why wasn't the directory tree split in one of the following ways?

```
/projects/Q_Development/V1/documentation
/projects/Q_Development/V1/program
/projects/Q_Development/V2/documentation
/projects/Q_Development/V2/program
/release/Q/V1
/release/Q/V2
```

```
/V1/projects/Q_Development/documentation
/V1/projects/Q_Development/program
/V1/release/Q
/V2/projects/Q_Development/documentation
/V2/projects/Q_Development/program
/V2/release/Q
```

---

[3]At least in some file systems symbolic links could be used to avoid the copy. In Unix, however, hard links can only be made between a directory and a file on the same physical volume. Symbolic links can cross volumes but result in an asymmetrical specification of a symmetrical situation.

[4]A well known software engineering "rule" states that when there are two identical copies of the same file at least one of them is different!

[5]Some file systems provide a search list mechanism where the V2 directories are initially empty and a search list is set that searches first V2 then V1. Any time a file not in V2 must be changed it is first copied into V2. This is again sort of a solution but is tedious, confusing, and error prone.

The answer is that that there is no strong reason to prefer one of these structures over the others. In a persistent object base, all three of these forms should be indistinguishable.

3.2 Abstraction

Unix starts with the assumption that all files exist on the same physical volume (typically a disk). In order to deal with multiple physical volumes, Unix has a mount command. The mount command has two arguments: an existing directory and a new volume that itself holds a file system consisting of a tree of directories and ordinary files. A mount causes the file tree on the new volume to be "pasted" into the file tree in place of the specified existing directory. The net effect is that there is a strong coupling between the path name of a file and its physical location. As Unix has come to be used in distributed networks, several network file systems have been proposed, including Apollo DOMAIN [Leach 83], Sun NFS [Sandberg 85], and AT&T RFS [Hatch 85]. In all of these systems, the path name is coupled to the physical placement within the network.

Modern data abstraction [Shaw 84] shows that considerable benefits can be achieved by separating the logical structure of data from its representation. As can be seen above, the Unix file system blurs together the logical concept of path name with the representational concept of physical location. Representational properties frequently influence the logical structure of data. Since physical volumes have a *finite maximum* data size, the number of files within the subtree for a volume is constrained. When the data size exceeds the physical space, the user is forced into modifying the logical structure. In networks, data on a local disk is often faster to access than data on a remote node. By changing the physical placement of data within the network, and therefore its logical structure, a user can get faster file access. In both these cases, the user who wants to deal with the logical structure of the data frequently spends considerable time also dealing with the physical constraints of the file system.

The Unix file system does not support flexible physical representations. For example, there is no way in Unix to transparently store a file in a compressed format using text compression [Welch 84] or as a data relative to some related file [Rochkind 75, Katz 84]. This kind of transparency would eliminate the user burden of explicitly invoking a decompressing program before each use of the compressed file.

Another kind of flexible representation is the use of multiple cached copies of the same file [Schroeder 85]. Within Unix, caching can be provided only by modifying the Unix kernel.

In a persistent object base, data abstraction should be practiced so that logical concepts are decoupled from physical representations; richer representations should be possible by providing the ability to program the implementation of file abstractions. The Apollo extensible streams mechanism [Apollo 86] is an example of such a data abstraction mechanism grafted on top of a Unix file system.

3.3 History

Two related history concepts are considered here: source versions and re-creation.

Every time a source file is edited, logically a new version is created, so that over time a linear sequence of versions is created. When alternatives occur, such as when a bug is fixed in an old release while work continues on the next release, the sequence can fork, and when alternatives come together separate sequences can join. Abstractly, a directed acyclic version graph is formed. Not all points in the version graph are equally important; in practice, users impose additional structure at one or more levels of granularity and do not preserve versions below some minimum level of granularity. The finest granularity corresponds to every edit. A coarse granularity would be at major release points. Intermediate granularities are frequently defined to aid the management of a development project. The concept of versions can be usefully extended to multiple related source files which may be considered to be progressing in parallel along a version graph.

One common way of handling source versions is through the use of naming conventions: either at the directory or the file level. Earlier in this paper, directory naming conventions were used as a way of representing versions of related sets of files. For example, the two directories below would hold all the Q system source files associated with each of the two versions.

```
projects/Q_Development/V1
projects/Q_Development/V2
```

A method for dealing with individual source files is use of a generation mechanism. Although Unix provides no special generation mechanism, the same effect can be realized by file naming conventions. For example, two versions of the same file could be named using a version extension.

```
/projects/Q_Development/Q_Control.ada.V1
/projects/Q_Development/Q_Control.ada.V2
```

One disadvantage of this approach is that all shell scripts need to be aware of the generation naming conventions, and any `V1` shell script needs to be edited before it can be used for `V2`.[6] When using conventions for representing version relationships, the entire burden for ensuring consistency rests with the user. Although a convention for representing linear version relationships is obvious, conventions representing forks and joins in the version graph are less clear.

A more sophisticated source version system is provided by the Unix SCCS tool and by a similar but improved tool RCS [Tichy 82]. SCCS keeps track of all the versions of a single source file. It provides support for both forks and joins.[7] The SCCS implementation holds all versions of a source file in a single file called the s-file. Before any use of a particular version of that source can occur, it must be extracted explicitly from the s-file. Typically, shell scripts will contain calls to

---

[6] The edit could be avoided by passing the version as a string parameter which is then concatenated to all file names.

[7] The SCCS documentation suggests that forks be kept to a minimum to avoid structural complexity.

SCCS for this purpose. The big disadvantage of SCCS is that it is an *ad hoc* data encoding scheme implemented on top of the file system, rather than as part of it. In addition to its logical properties, SCCS also uses the representational method of source deltas to encode the versions. This is yet another example of how logical properties and physical representation have been blurred together.

In a persistent object base, SCCS functionality would be provided in a transparently integrated manner. Versions and data compression would be handled by orthogonal mechanisms.[8] The DSEE system [Leblang 85] is one current example of how this could be done.

Re-creation is the ability to be able to go back to an old version of a system and repeat all of the steps that were involved in its creation. Re-creation implies that all information about system creation is captured. Traditionally, a lot of the system creation information was held only in the heads of the development team, making re-creation difficult. Re-creation is important for two major reasons. First, if a system is re-creatable, important structural relationships between the files of the system are captured. System maintainers can use the relationships directly and use support tools that depend upon having the relationships available. Second, if an old version of a system has a bug, re-creation means that a minor variation of it can be constructed in which the bug is fixed. To better understand re-creation, the concept of a derivation graph is used. Derivation graphs were used in Toolpack [Osterweil 83]. The definition used here is a somewhat simplified form of the model presented in [Borison 86].

Those files that make up a system can be divided into primitive and derived files. A primitive file is either a source file of the system or some file from outside the system that is used in its construction. A derivation step consists of an invocation that accesses a set of input files to produce a set of output files. The invocation includes a tool consisting of an "executable" file and a set of actual parameters to that tool, which could be either constants or files (or their names). It is assumed that the output files depend only upon the input files and the invocation involved in the derivation step.[9] Derived files are those that are output of some derivation step. The inputs of a derivation step and the file holding the tool being run in the derivation step must be either primitive files or output files of some earlier derivation step. The combination of all the derivation steps for a system is its directed acyclic derivation graph. A system is re-creatable if all of its derived files can be re-created identically. In terms of a derivation graph, re-creation is possible if each derivation step is known, the set of primitive files is known, and the primitive files have not been changed since the system was first created.

In Unix, creation is often accomplished using the Unix tool Make [Feldman 79]. Make applies a set of heuristics to a *makefile* that contains a list of explicit commands to determine and run a set

---

[8] Version information, however, could be used to guide heuristics that identify candidates for delta compression.

[9] In practice, it is necessary to deal with things like steps that read the system clock or that interact with the user. For current purposes such problems are ignored.

of invocations.[10] Make is concerned with invocations, not with the more general concept of derivation steps. In general, it is not possible to tell the complete set of input files and output files of each derivation step of a system by looking only at the *makefile*; therefore, it is not possible to determine the set of primitive files of a system. By convention, users normally include information about these file sets in their *makefiles*, but there is no check to be sure that this information is complete or even correct. The reason for this deficiency goes deeper than just the Make tool. In Unix, when a tool is invoked, there is no way to tell what files are opened for input and/or output. Such an inquiry is essential to guarantee re-creation but when arbitrary tools can be invoked during derivation[11], this inquiry can only be implemented by making modifications to the Unix kernel.

For re-creation, the set of primitive files must be determined, and each file in the set must be checked to ensure that it has not been changed since initial creation. Unix provides some assistance here in the form of a time stamp for each file that gives the time that each file was last modified. As long as the last modified time on a file is older than the creation time of the system, then it would seem that it is a correct file. The problem occurs when the Unix move command, `mv`, is used. This command moves a file between two directories and preserves its last modified time. So when `mv` is used, the primitive file may not be correct and re-creation can not be done.[12] Worse, there is a system call, `utimes`, that can be used to change arbitrarily the last modified time.[13]

The use of SCCS protects the user from changing old versions of a source file. This is a step forward, although the problem is still present at a deeper level because the s-file itself is subject to all the previous problems.

In a persistent object base, re-creation would be achieved by immutable objects and unique object identifiers, such as those provided by the Cedar System Modeller [Lampson 83]. All primitive files and the full derivation graph would be stored as immutable objects whose content cannot be changed by any user. Each object is assigned a unique identifier at creation in a way such that no two objects ever will have the same unique identifier. The derivation graph would refer to primitive objects by their unique identifier, not by their file name; so move and copy operations would not confuse the identification of the primitive objects.

Keeping the information needed to re-create all the old versions of all systems on rotating magnetic media is generally considered too expensive. Write-once laser disks are just starting to

---

[10]Although not discussed here, Make also uses heuristics to avoid rerunning those derivation steps whose inputs have not changed since they were run last.

[11]Tools where the set of input and output files for an invocation can be determined easily present no problem. The C compiler, which can read arbitrary include files, involves moderate difficulty.

[12]In practice, it often looks as though re-creation happens correctly. Users frequently spend many confusing hours when the re-created system is subtly different from the original system.

[13]Tool designers have been known to use this call constructively to fake Make into doing "the right thing".

become available [Fujitani 84]: They offer the ability to hold extensive historical information at acceptable costs. When write-once laser disks are combined with the use of compressed representations, there is no reason why all past versions of all source files cannot be kept available on-line [Katz 84].

3.4 Attributes

Unix provides a fixed set of attributes for each file as part of its directory entry. These attributes include the name of the owner of the file, a set of file protection control bits, and times of file creation, modification and use. These attributes are mostly set and used by the system, although there are commands and system calls that permit the user to set and use them.

When additional attributes beyond those provided by Unix are needed, the user must find alternative ways of representing them, since there is no way to add new attributes to a directory. The set of additional attributes that could be of use in an environment is unlimited, being determined by the needs of a development effort and the tools it uses. A persistent object base must be able to support arbitrary attributes. Some examples of additional attributes include the unique identifier for the file, a string attribute that gives the reason why the file was created, and a boolean attribute that indicates whether the content of the file has been compressed.

Attributes can be used also to relate files. For example, the version graph can be represented by each file having as an attribute a set of the unique identifiers of the files that are its immediate version predecessors. Since the version graph is symmetrical, another attribute that is a set of version successors could be added also, but these two attributes contain redundant information. To avoid the redundancy and to preserve the symmetry, a better way of representing the version graph would be with a version relationship that relates predecessors with successors but that is not an attribute of either. So in addition to attributes, a persistent object base should support arbitrary relationships. Other examples of relationships include the derivation graph, and a relationship between C program files and the include files they reference.

In addition to files having attributes, a persistent object base should also permit relationships to have attributes.[14] For example, the version relationship could have an attribute that says why a successor version was produced from some predecessor version, and a derivation step within the derivation relationship might have an attribute for the time at which it was run.

Typically, when tools are built on top of Unix that depend upon attributes and/or relationships, then *ad hoc* encoding means are used. These means range from special purpose file encodings such as the s-files of SCCS, through special purpose database systems such as that used in Gandalf [Gandalf 85], to general purpose database system such as that used in DSEE. Not only is considerable effort wasted in building tools which each must do their own attribute and relationship support, but even greater problems occur when tools that use different *ad hoc* schemes must

[14] It even makes sense to have relationships between relationships.

be integrated. Consider, for example, two systems, each of which uses its own special *ad hoc* scheme and where the information used partially overlaps so that redundant information must be synchronized between the two different schemes. The net result is that integration is very difficult, if not impossible.

3.5 Synchronization

When two or more users are working on the same system and therefore the same set of files, some means of synchronizing that use is needed. When two people are editing the same source file without synchronization, the changes made by one may overwrite the changes made be the other without either being aware of the problem. In the absence of automated support, users frequently do such synchronization by manual conventions. For example, a specific set of files are agreed to be "controlled" by some specified user who may change any of the files while other users may read but not modify the files. The weaknesses of this approach are that it is time consuming, error-prone, and often overly restrictive in limiting modifications. Under Unix, the SCCS tool provides support for synchronization at the level of each source file. SCCS has two basic operations for synchronization: `get` a file for editing from the s-file; and `merge` the edited file back into the s-file. Only one user may have a given s-file in the editing state, between a `get` and a `merge`. This is overly restrictive because multiple edits could be proceeding safely on independently forked alternatives. The RCS tool solves this problem by permitting one edit to be occurring on each alternative fork. Both SCCS and RCS require explicit extra action by the user to `get` and `merge` a file when editing it. This is often enough of a burden to discourage users from using either SCCS or RCS. The DSEE system provides the synchronization in an integrated fashion that is less of a burden for the user and that is harder to subvert.

Another problem with SCCS and RCS is that the default mode of operation for `get` is to extract the most recent version on the main version line. At first this seems like a desirable feature, since most of the work on a system is with the most recent version. Problems can occur, however, when multiple people are producing new versions of the primitive files of a system. When a user changes some primitive files and then does a build based on most recent versions of all primitive files, the resulting system will incorporate not only the user's changes but also possibly arbitrary other changes made by other users to other primitive files. The net result is that the behavior of a most recently built system will often change over time in subtle ways that are not under the user's control. Normally, under Unix no derivation graph is recorded; thus it is difficult, if not impossible, to figure out which set of primitive files have changed since the previous system build. Time stamps are one clue to what has changed, but due to problems discussed earlier, they are not always reliable. Recall that primitive files include not only source files that belong to the system under development but also libraries and tools that are part of Unix. Normally, Unix libraries and tools are not version controlled; nevertheless, they are changed during periodic operating system releases and by Unix system software maintainers at arbitrary times to fix perceived "bugs". These changes cause not only unpredictable behavioral changes in the current system builds, but can also destroy the ability to re-create previous system versions.

A high quality programming environment must support synchronization that is simple for users, place no unnecessary restrictions on simultaneous access, support a version control system for both user and system files, record the full derivation graph, allow the user to control explicitly which versions of primitive files to use, and support inquiry so that users can determine easily which primitive files of a system have been changed. A persistent object base should provide the basic support layer on which such environments can be built.

4 Weaknesses of Database Systems

This section considers five areas where database systems are inadequate for persistent object bases: types, decentralization, time, distribution, and performance. Relational database systems [Codd 70] will be used as examples. Other database systems will differ in their details from relational systems but display essentially similar weaknesses.

Engineering databases, particularly those used for CAD/CAM, share many basic requirements with programming environments. Many of the weaknesses that have been identified in these applications [Hallmark 84, Hartzband 85] are similar to those discussed here.

Relational database systems are now just starting to be used within programming environments for applications including source program tree representation, dynamic execution behavior, and version and configuration control [Ceri 83, Snodgrass 84, Linton 84].

4.1 Types

Types in relational database systems are considered here from three perspectives: primitive types, structural types, and abstract types.

Relational database systems typically have a small predefined set of primitive types for attributes.[15] This set is often quite constraining when used for programming environments. For example, consider using a relation to represent the version graph.

```
Version : relation
        old:integer,     -- old version number
        new:integer,     -- new version number
        why:string       -- reason for change
      end
```

A value for this relation might be as follows.

[15] In database systems, the term domain is used to refer to a set of values for some attribute. The term type is used here instead of domain to emphasize analogies with the type mechanisms of programming languages.

Version

old	new	why
1	2	"Added a new feature that permits inverted input"
2	3	"Fixed the bug introduced in v 2"

Here the why field is a string of arbitrary length. The only string type provided by many relational systems is a fixed length string. The effect of varying length strings can be achieved, but only by subverting the system.

```
Version1 : relation
        old:integer,     -- old version number
        new:integer,     -- new version number
        count:integer,   -- string index
        why:string(20)   -- reason for change
      end
```

Version1

old	new	count	why
1	2	1	"Added a new feature "
1	2	2	"that permits inverte"
1	2	3	"d input "
2	3	1	"Fixed the bug introd"
2	3	2	"uced in v 2 "

Not only is the structure of the data obscured, but both space to store the data and time to access it are degraded. This kind of subversion only gets worse when trying to represent more complex programming environment data such as program source and relocatable, documentation, and graphics. Although all of these could be built up from the primitive types of a relational database system, the effort is large and the representation would be neither natural nor efficient.

Structurally, a relational database consists of a set of named relations. Consider, for example a database with two relations.

```
Version : relation
        old:integer,     -- old version number
        new:integer,     -- new version number
        why:string       -- reason for change
      end

Source : relation
        version:integer,-- version number
        day:integer,     -- day created
        month:integer,   -- month created
        year:integer     -- year created
      end
```

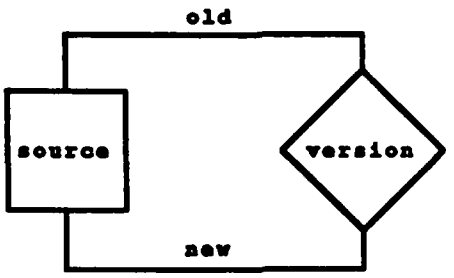
All of the relationships between relations are expressed implicitly. Typically, two relations are related by using the same type for some attribute in each so that they can be joined. For example, `Version.old` could be joined to `Source.version`. It is not the case, however, that if two relations have attributes with the same types that it always makes sense to join them. For example, joining `Version.old` to `Source.month` is not a sensible operation. One way to introduce more structure is by stronger typing such as that provided by the Modula-2 type declaration [Wirth 85].

```
type version_type = integer;
type day_type = integer;
type month_type = integer;
type year_type = integer;

Version : relation
        old:version_type,     -- old version number
        new:version_type,     -- new version number
        why:string            -- reason for change
      end

Source : relation
        version:version_type,-- version number
        day:day_type,         -- day created
        month:month_type,     -- month created
        year:year_type        -- year created
      end
```

Another structural approach is to use graphical entity-relationship diagrams [Chen 76].



Extended relational models are another way of introducing more structure. One such system is RM/T [Codd 79]. In RM/T, system generated surrogate keys are attached to every tuple of every relation so that no two are ever identical.

```
Source : relation
        version:integer,-- version number
        day:integer,    -- day created
        month:integer,  -- month created
        year:integer    -- year created
      end

Version : relation
        old:key[Source],-- old version
        new:key[Source],-- new version
        why:string      -- reason for change
      end
```

Source

key	version	day	month	year
#003	1	10	5	85
#004	2	20	10	85
#005	3	4	3	86

Version

key	old	new	why
#001	#003	#004	"Added a new feature ..."
#002	#004	#005	"Fixed the bug ..."

The **key** attribute is automatically supplied and initialized by the system. Since joins now are based on unique surrogate keys, structural relationships are specified fully. Surrogate keys are closely related to the unique object identifiers discussed earlier and to the typed pointers of Modula-2.

If surrogate keys are placed not only on tuples but on entire relations, then relations can be used to relate other relations. For example, a directory tree like that of a file system can be represented. First, a relation type for directories is introduced.

```
Type Directory = relation
        name:string,
        file:key
      end
```

As an example, the following directory tree is used.

```
/Q_Development/V1/documentation
/Q_Development/V1/program
/Q_Development/V2/documentation
/Q_Development/V2/program
```

That tree is represented by the following instances of the **Directory** type.

#001 :Directory

key	name	file
#002	"Q_Development"	#003

#003 :Directory

key	name	file
#004	"V1"	#006
#005	"V2"	#009

#006 :Directory

key	name	file
#007	"documentation"	#012
#008	"program"	#013

#009 :Directory

key	name	file
#010	"documentation"	#014
#011	"program"	#015

The lack of abstract data types [Shaw 84] in relational database systems is perhaps the most significant type weakness. All data in a relational database exists at a structural level. There is no way to define a new abstract type in terms of its abstract properties and then define its implementation in terms of existing types. Reconsidering an earlier example, varying length strings could be defined as a new abstract type that used a variable number of fixed length strings as its representation. This kind of abstraction becomes even more important for complex objects such as those that represent graphic images.

Abstract data types gain much of their power from considering not just data in isolation, but data together with the set of operations. In relational database systems, the data specification written in some schema language is separated from the operations as expressed in some query language. Not only are the specifications physically separate, but often they are expressed in an incompatible language.

Another aspect of abstract data types is that the implementation can be changed without impacting the users of the specification. Database systems normally provide users with a limited level of control over the way in which the data is represented. For example, many database systems allow users to specify those places where redundant inverted indexes are to be created. When the user needs a representation that is not supported by the system, the only alternative is to modify the source code of the database system itself. Even in those rare cases where source code is available, the complexity of most database systems makes this a formidable task. A solution is to put more of the control for representations in the hands of the user via an abstract type mechanism.

A persistent object base system should provide a rich set of primitive types, enable expression of rich structural relationships such as those of the extended relational models, and provide a full abstract data type mechanism. There are obvious parallels between the needed future direction for database systems and the past evolution of type support within modern programming lan-

guages. Many of the same type features found in modern languages need to be brought into database systems; however, database systems face special problems brought about by the persistence of data that were not faced by the designers of programming languages.

4.2 Decentralization

Database systems typically have a single centralized schema that is maintained by a database administrator, DBA. For programming environments, it must be possible to define and control data locally. This need is demonstrated below by several examples.

As an initial example, consider the set of documentation files in a system including help files, user manuals, implementation descriptions, and even the source files of systems. These represent online versions of information that each user would previously have had in hardcopy. One advantage of hardcopy is that it is easy for each user to write in personal comments. The same approach could be used online by letting each user make a copy of the document and edit in personal comments, but it would be better to have a single copy of the document and let each user be able to have a separate "overlay" that contains personal comments. This kind of ability is becoming available though a class of environments called hypertext systems [Yankelovich 85]. Consider the following simplified relation types.

```
type Document = relation
        line_number:integer,
        line:string
      end;

type Comments = relation
        document:key[Document],
        line_number:integer,
        comment:string
      end;
```

Considerable progress toward decentralization is already implicit in the use of type definitions and surrogate keys. Type definitions permit multiple instances. Surrogate keys enable an object and its attributes to be stored separately. This separation is important not only for local control but also because it permits data, such as an instance of `Comments`, to be added to a preexisting data structure, such as an instance of `Document`, without modifying either the type definition or contents of that preexisting data structure. Not only must the database permit the right kind of definitions, it also must permit the needed operations. As basic operations, each user must be able to create locally an instance of `Comments` and to control the use of that instance. As a more general operation, users should be able to define their own relation types for their own local use. In many database systems, these abilities are centralized with the DBA. Making a user go through a DBA for these kinds of operations is not only bothersome but also logically unnecessary.[16]

---

[16]Analogies exist in the file system area when users are forced to go through a centralized system administrator to have structures created or modified that should be under user control.

As a second example, consider what happens when a new tool is added to a programming environment. This tool may need the ability to create and access new attributes and relationships of existing objects. The need for local definition, instantiation, and control are similar to those of the previous example. For tools, decentralization also can be an aid to integration. Since each tool can manage locally the attributes and relationships for that tool, independent tools will not place conflicting constraints on centralized data. Conflicts can be representational, such as multiple tools wanting to use word 23 of some control block, or naming, such as multiple tools wanting to use the attribute name `Next`. Particularly severe conflicts can arise when two versions of a single tool are being supported simultaneously. For example, both versions might have a `Next` attribute, but give it slightly different semantics. By giving each version its own instance of the `Next` attribute, multiple versions can coexist without interference.[17]

As a final example, consider integrating two previously independent databases. Such integration could occur when two isolated programming environment systems are connected via a network and a transparent network file system is installed. When centralized schema are used, integration will require merging these two schema into a single new schema. Conflicts are virtually certain to occur, forcing either massive recoding or a less transparent integration in which the two independent schema continue to exist.

For a persistent object base, decentralization of definition, instantiation, and control is essential. This implies that there will not be a database administrator doing all data definition. Another implication is that traditional kinds of normalization that are based on a single centralized schema cannot be done. Since normalization is a method of removing redundancy and since controlled redundancy can be used to improve the engineering of software systems, full normalization may not only be limited but also undesirable.

4.3 Time

The basic relational data model views the database as having values that vary over time. Every attribute is considered to be variable and only its current value is available. As was previously discussed, programming environments must provide a history mechanism to record source versions and to support re-creation.

In many simple database systems that are used to support programming environment tools, the only way to preserve history is by making a complete copy of the entire database. In more powerful database systems, transaction journals are used to preserve history. A similar capability is provided in file systems by periodic backup of all the files on a system. All of these approaches have a common weakness: To go back in time, it is necessary to manually substitute a previous version of the data for the current version. This substitution can be either physical or logical. In a

[17]Local attribute instances of course do not solve all integration problems. Mechanisms, such as those in [Garlan 86], are needed for integrating tools that have logically related attributes where setting the attribute of one tool should modify the value of attributes of other tools.

physical substitution, the current version is copied to a safe place and then the old version is brought back in the place of the new version. For logical substitution, the new version is left in place, but operations are logically changed to operate on the reconstituted old version.

For a persistent object base, the ability to record previous states and to change back easily and transparently to an arbitrary previous state is essential. To capture this ability cleanly and safely requires more than just the addition of attributes that can take on time values. A persistent object base must include time as an integral part of its underlying formal semantic model [Clifford 83].

Work on temporal database systems [Snodgrass 86] shows that time can be modeled with two dimensions whose axes are transaction time and real time. The transaction time axis measures the actual state of a system over time. By backing up along the transaction time axis to some previous time, the system state is logically returned to what it was at that previous time. By backing up along the real time axis, the system state is logically returned to the state of reality at that previous time (as determined by our best current knowledge). Transaction and real times will differ when either there is delay between the time at which an event occurs and the time at which it is first entered into the database or when some event is incorrectly entered and later corrected. Programming environments are unlike conventional database applications because the reality that is being modeled is data within the database itself. This implies that for programming environments transaction time is identical to real time. Suppose, however, that the concept of exact modeling of reality is replaced by the concept of exactly correct program behavior. Now forward progress of some program under development along the transaction axis represents increased (or modified) functionality, while forward progress along the correctness axis represents an increase in the number of bugs fixed.

4.4 Distribution

Most currently available database systems require that all data be kept within a single machine. Future programming environments will be based on multiple machines connected via many kinds of networks. Although considerable work is now being done on distributed database systems, current systems are still rather limited [Ceri 84]. Two aspects of distribution are considered: how data is distributed among multiple machines and how multiple users on different machines can share the same data.

The goal of data distribution is to place data physically so that it is available easily and quickly to its users while satisfying the hardware size constraints. The simplest way to distribute a relational database is to place different relation instances on different machines. The placement can be static, determined when the instance is created, or dynamic, changeable at any time. Independently, the placement can be manual, under the control of the user, or automatic, under the control of the system.

A more complex distribution would place different parts of the same relation on different machines. For example, consider the version graph relation. Accesses to that relation will tend to be to tuples for recent versions. Older tuples can be placed on remote, slower, and/or larger

physical devices of the system.[18]

When two people are using the same data, then in general no one place is best for both. A solution is to permit separate copies of the data to exist at locations that are good for each user. When the users are reading the data and neither is modifying it, then permitting multiple copies is easier. A special case of read-only data is immutable objects. Many network file systems are now providing caching, a dynamic automatic mechanism for transparently creating and managing multiple copies of data [Schroeder 85, Morris 86].

Not all data in a programming environment can be immutable. At least some data must be mutable for progress to be made. A simple mechanism for dealing with mutable data in the presence of multiple users is to use a central server. A server is a specific machine that controls write access to data.[19] The server ensures that only one user is writing the same data at the same time. Before a user can modify data, a lock is set on the server so that other users cannot modify that same data.

Servers limit effective distribution. The problem can be reduced by either minimizing the frequency with which a user interacts with the server or by modifying data in ways that do not require the use of a central server.

To understand how to minimize server interaction, it is instructive to consider how multiple users working on the same system interact when using a programming environment that provides no synchronization for data modification. In this case, the users often invent manual methods for synchronization. Other than failures that occur when someone forgets the state of the manually set locks, such methods work just fine. An important distinguishing characteristic of these manual methods is the frequency of the synchronization operations. While common automated systems often operate with a frequency of many synchronization operations per second [Ousterhout 85], manual methods may have a frequency of only a few operations per day. By implementing analogues of these manual methods, server interaction rates can be lowered. As an example, consider the directory tree of a network file system. Every time a new file name is created, a synchronization operation is needed. Most users on Unix systems are creating and destroying files at a high rate. To lower the rate involves completely rethinking the role of global name spaces in programming environments.[20]

As an example of how data can be modified without involving a server, consider the version graph relation.

---

[18]This includes migrating old tuples to magnetic tape.

[19]In practice, there can be multiple servers as long as each data item is handled by exactly one server.

[20]Names in Unix serve two independent purposes, connecting uses to their definitions and communicating information between users. Uses can be connected to definitions by using unique identifiers instead of names. Each unique identifier may still have a name, but that name is used for local display purposes only; the connection is made via the unique identifier. A global name then is needed only in those relatively less frequent cases where information is passed between users. A single global name may be communicated for an entire system that internally contains thousands of local unique identifiers.

```
Version : relation
        old:key[Source],-- old version
        new:key[Source],-- new version
        why:string      -- reason for change
      end
```

Suppose that two users each want to create a successor of some existing version. Each will create a new source object and add a new tuple for it to the `Version` relation. There is no basic conflict between these additions. Since the tuples of a relation are an unordered set, the order of the additions will not affect the final value of the relation.

Since a network imposes finite delays[21], time within a network is relativistic [Lamport 78]. In relativistic time, there is no system-wide absolute clock. Each machine within the network is assumed to have its own clock that progresses at its own rate. In such a system, there is no total ordering of events. Consider again the two users, `A` and `B`, each of which has a machine within the same network, both trying to create a successor of the same existing version. To `A` it may seem as though the new `A` tuple appears before the new `B` tuple, while to `B` the order appears to be the `B` tuple followed by the `A` tuple.[22]

Now suppose that each new version is to be given the next new integer version number. This can only be done by having a single server that assigns those numbers. Many version management systems have a similar problem. When several alternative versions are present, one of them is designated as the "primary" version. A central server is needed to control which new version is to be the primary version.

For a persistent object base, automatic dynamic placement and caching of relations and parts of relations is needed. Various methods must be used to avoid high interaction rates with central servers.

4.5 Performance

The performance of database systems is tuned to access patterns that may be quite different from those expected in programming environments. Performance is considered here in terms of what is accessed and who is accessing it. A thorough understanding of the performance issues of using databases for programming environments can occur only after many more experiments are carried out and much more analysis is done. Based on what is still very limited experience, this discussion speculates on areas where performance problems seem most likely to occur.

Relational database systems typically are tuned to emphasize the performance of operations that deal with entire relations, such as join and projection. In a programming environment, operations

---

[21] These delays can be quite long when lots of data is transmitted over a dial-up phone line or when some link in the network is broken.

[22] A different approach that avoids a single central server for versions is given in [Ecklund 85].

that deal only with a single tuple from each of many relations may be more frequent. In programming environments, access patterns that traverse trees or directed graphs are common. Such traversals must extract a single tuple, from the relation that represents the tree or graph, at each step. Relational database systems typically assume that all tuples of a relation are equally likely to be accessed. In the version relation, for example, tuples for more recent versions are more likely to be accessed. Graph transitive closure operations are common in programming environments. For example, a query might be to determine all versions that are direct or indirect predecessors of some given version. Database systems are not normally tuned to make transitive closure efficient. Furthermore, relationally complete query languages cannot in general even express transitive closure.

Many database systems are designed mainly to interact with people. In a programming environment, most of the use of the data will be by programs. Most database query languages are interpreted, not compiled. While users may accept small delays due to interpretation, the heavy use of data accessing programs may produce unacceptable performance degradation in programming environments. Of special performance significance is the use of surrogate keys. These keys serve exactly the same role as pointers do in most programs: They are used to build linked list structures such as trees and graphs. The efficiency of the pointer dereference operation is known to be a major factor in determining the execution speed of most system programs. Unless surrogate keys can be implemented with an efficiency approaching pointers, then relational databases may prove to be an unacceptable basis for programming environments.[23]

5 Conclusions

This paper has examined from several perspectives the weaknesses of file systems and database systems as a basis for persistent object bases of programming environments. Neither current file systems nor current data base systems are adequate to support a first class persistent object base. In many areas, however, current research and development is progressing toward systems that correct at least some of the weaknesses.

This paper provides designers and evaluators of persistent object bases with a checklist of issues to be considered and a list of problem areas where further work is needed. However, the real work of building a persistent object base may be less concerned with finding novel solutions to specific problems and more concerned with effectively integrating current technology.

6 Acknowledgements

Many of the ideas presented here originated in discussions with Joseph Newcomer and Ellen

[23] A possible answer is to use memory addresses as the representation that programs see. Considerable engineering is obviously needed.

Borison. I would like to thank Haavard Eidnes, Purvis Jackson, Richard Snodgrass, Donald Stone, and Chuck Weinstock for providing many useful comments on earlier drafts of this paper.

7 References

[Apollo 86] Apollo Computer. *Using the Open System Tool Kit to Extend the Streams Facility.* To appear April 1986.

[Borison 86] Ellen Borison. A Model of Software Manufacture. *International Workshop on Advanced Programming Environments.* Trondheim, Norway. June 1986.

[Ceri 83] S. Ceri and S. Crepi-Reghizzi. Relational Databases in the Design of Program Construction Systems. *SIGPLAN Notices,* Volume 18, Number 11, November 1983.

[Ceri 84] S. Ceri and G. Pelagatti. *Distributed Databases Principles and Systems.* McGraw-Hill, 1984.

[Chen 76] Peter Pin-Shan Chen. The Entity-Relationship Model - Toward a Unified View of Data. *ACM Transactions on Database Systems,* Volume 1, Number 1, March 1976.

[CLF 85] CLF Project. *Introduction to the CLF Environment.* USC Information Sciences Institute, 1985.

[Clifford 83] James Clifford and David S. Warren. Formal Semantics for Time in Databases. *ACM Transactions on Database Systems,* Volume 8, Number 2, June 1983.

[Codd 70] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM,* Volume 13, Number 6, June 1970.

[Codd 79] E. F. Codd. Extending the Database Relational Model to Capture More Meaning. *ACM Transactions on Database Systems,* Volume 4, Number 4, December 1979.

[DoD 85] Draft Military Standard Common APSE Interface Set (CAIS). Proposed MIL-STD-CAIS. NTIS AD 157-587. January 31, 1985.

[Ecklund 85] Earl F. Ecklund, Jr., Darryn M. Price, Rick Krull, and Denise J. Ecklund. *Federations: Scheme Management in Locally Distributed Databases.* Technical Report CR-85-39, Computer Research Laboratory, Tektronix Laboratories, November 1985.

[ESPRIT 85] ESPRIT. *PCTE, A Basis for a Portable Common Tool Environment, Functional Specifications.* Third edition, Bull, The General Electric Company p.l.c., ICL International Computer Limited, Nixdorf Computer AG, Olivetti SPA, Siemens AG, 1985.

[Feldman 79] S. I. Feldman. Make - A Program for Maintaining Computer Programs. *Software Practice and Experience,* April 1979.

[Fujitani 84] Larry Fujitani. Laser Optical Disks: The Coming Revolution in On-Line Storage. *Communications of the ACM,* Volume 27, Number 6, June 1984.

[Gandalf 85] Special Issue on the Gandalf Project. *The Journal of Systems and Software,* Volume 5, Number 2, May 1985.

[Garlan 86] David Garlan. Views for Tools in Integrated Environments. *International Workshop on Advanced Programming Environments.* Trondheim, Norway. June 1986.

[Hallmark 84] G. Hallmark and R. A. Lorie. Toward VLSI Design Systems Using Relational Databases. *IEEE Computer Conference.* Spring 1984.

[Hartzband 85] David J. Hartzband and Fred J. Maryanski. Enhancing Knowledge Repesentation in Engineering Databases. *Computer,* Volume 18, Number 9, September 1985.

[Hatch 85] Mark J. Hatch, Michael Katz, and Jim Rees. AT&T's RFS and Sun's NFS, A Comparison of Heterogeneous Distributed File Systems. *Unix/World,* Volume 2, Number 11, December 1985.

[Katz 84] Randy H. Katz and Tobin J. Lehman. Database Support for Versions and Alternatives of Large Design Files. *IEEE Transactions on Software Engineering,* Volume 10, Number 2, March 1984.

[Lamport 78] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM,* Volume 21, Number 7, July 1978.

[Lampson 83] Butler W. Lampson and Eric E. Schmidt. Organizing Software in a Distributed Environment. *Proceedings of the SIGPLAN '83 Symposium on Programming Language Issues in Software Systems.* SIGPLAN Notices, Volume 18, Number 6, June 1983.

[Leach 83] P. Leach, P. Levine, B. Dorous, J. Hamilton, D. Nelson, and B. Stumpf. The Architecture of an Integrated Local Network. *IEEE Journal on Selected Areas in Communications,* November 1983.

[Leblang 85] David B. Leblang, Robert P. Chase, Jr., and Gordon D. McLean, Jr. The DOMAIN Software Engineering Environment for Large Scale Software Development Efforts. *Proceedings of the 1st International Conference on Computer Workstations.* IEEE, November 1985.

[Linton 84] Mark A. Linton. Implementing Relational Views of Programs. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments.* SIGPLAN Notices, Volume 19, Number 5, May 1984. Software Engineering Notes, Volume 9, Number 3, May 1984.

[Morris 86] James H. Morris, Mahadev Satyanarayanan, Michael H. Conner, John H. Howard, David S. H. Rosenthal, and F. Donelson Smith. Andrew: A Distributed Personal Computing Environment. *Communications of the ACM,* Volume 29, Number 3, March 1986.

[Osterweil 83] Leon Osterweil and Geoffrey Clemm. The Toolpack/IST Approach to Extensibility in Software Environments. *Ada Software Tools Interfaces: Bath Workshop Proceedings.* Springer-Verlag, Lecture Notes in Computer Science, Number 180, 1983.

[Ousterhout 85] John K. Ousterhout, Herve Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. *A Trace-Driven Analysis of the UNIX 4.2BSD File System.* Technical Report UCB/CSD 85/230, University of California, Berkeley, April 1985.

[Ritchie 74] D. M. Ritchie and K. Thompson. The Unix Time-Sharing System. *Communications of the ACM,* Volume 17, Number 7, July 1974.

[Rochkind 75] M. J. Rochkind. The Source Code Control System. *IEEE Transactions on Software Engineering,* Volume 1, Number 4, December 1975.

[Sandberg 85] R. Sandberg. The Design and Implementation of the Sun Network File System. *Proceedings Usenix,* June 1985.

[Schroeder 85] Michael D. Schroeder, David K. Gifford, and Roger M. Needham. A Caching File System for a Programmer's Workstation. *Proceedings of the 10th ACM Symposium on Operating System Principles.* Operating System Review, Volume 19, Number 5, December 1985.

[Shaw 84] Mary Shaw. Abstraction Techniques in Modern Programming Languages. *IEEE Software,* Volume 1, Number 4, October 1984.

[Snodgrass 84] Richard Snodgrass. Monitoring in a Software Development Environment: A Relational Approach. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments.* SIGPLAN Notices, Volume 19, Number 5, May 1984. Software Engineering Notes, Volume 9, Number 3, May 1984.

[Snodgrass 86] Richard Snodgrass and Ilsoo Ahn. Temporal Databases. *Computer,* To appear 1986.

[Taylor 86] Richard N. Taylor, Lori Clarke, Leon J. Osterweil, Richard W. Selby, Jack C. Wileden, Alex Wolf, and Michal Young. Arcadia: A Software Development Environment Research Project. *IEEE Transactions on Software Engineering,* To appear 1986.

[Tichy 82] Walter F. Tichy. Design, Implementation, and Evaluation of a Revision Control System. *Proceedings of the 6th International Conference on Software Engineering.* IEEE, Tokyo. September 1982.

[Welch 84] Terry A. Welch. A Technique for High-Performance Data Compression. *Computer,* Volume 17, Number 6, June 1984.

[Wirth 85] Niklaus Wirth. *Programming in Modula-2.* Third Corrected Edition. Springer-Verlag, 1985.

[Yankelovich 85] Nicole Yankelovich, Norman Meyrowitz,and Andries van Dam. Reading and Writing the Electronic Book. *Computer,* Volume 18, Number 10, October 1985.

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNLIMITED, UNCLASSIFIED	1b. RESTRICTIVE MARKINGS NONE
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT UNCLASSIFIED, UNLIMITED, DTIC, NTIS
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) CMU/SEI-86-TM-8	5. MONITORING ORGANIZATION REPORT NUMBER(S) ESD-TR-86-215
6a. NAME OF PERFORMING ORGANIZATION SOFTWARE ENGINEERING INST. 6b. OFFICE SYMBOL (If applicable) SEI	7a. NAME OF MONITORING ORGANIZATION SEI JPO
6c. ADDRESS (City, State and ZIP Code) CARENGIE-MELLON UNIVERSITY PITTSBURGH, PA 15213	7b. ADDRESS (City, State and ZIP Code) ESD/XRS1 HANSCOM AIR FORCE BASE HANSCOM, MA 01731
8a. NAME OF FUNDING/SPONSORING ORGANIZATION SEI JPO 8b. OFFICE SYMBOL (If applicable) ESD/XRS1	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F19628 85 C- 0003
8c. ADDRESS (City, State and ZIP Code) CARNEGIE-MELLON UNIVERSITY PITTSBURGH, PA 15213	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO.: 63752F PROJECT NO.: N/A TASK NO.: N/A WORK UNIT NO: N/A
11. TITLE (Include Security Classification) TOWARD A PERSISTENT OBJECT BASE	
12. PERSONAL AUTHOR(S) JOHN NESTOR	
13a. TYPE OF REPORT FINAL 13b. TIME COVERED FROM ... TO ...	14. DATE OF REPORT (Yr., Mo., Day) JULY 86 15. PAGE COUNT 28
16. SUPPLEMENTARY NOTATION N/A	

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB. GR.	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

TO BETTER UNDERSTAND THE NEEDS OF FUTURE PROGRAMMING ENVIRONMETNS, TWO CURRENT TECHNOLOGIES THAT SUPPORT PERSISTENT DATA IN PROGRAMMING ENVIRONMENTS ARE CONSIDERED: FILE SYSTEMS AND DATA BASE SYSTEMS. THIS PAPER PRESENTS A SET OF WEAKNESSES PRESENT IN THESE CURRENT TECHNOLOGIES. THESE WEAKNESSES CAN BE VIEWED AS A CHECKLIST OF ISSUES TO BE CONSIDERED WHEN EVALUATING OR DESIGNING PROGRAMMING ENVIRONMENTS.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED [X] SAME AS RPT [X] DTIC USERS [X]	21 ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED, UNLIMITED, DTIC, NTIS
22a. NAME OF RESPONSIBLE INDIVIDUAL KARL H. SHINGLER	22b TELEPHONE NUMBER (Include Area Code) 412 268-7630 22c OFFICE SYMBOL SEI JPO

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE

END

9-87

DTIC