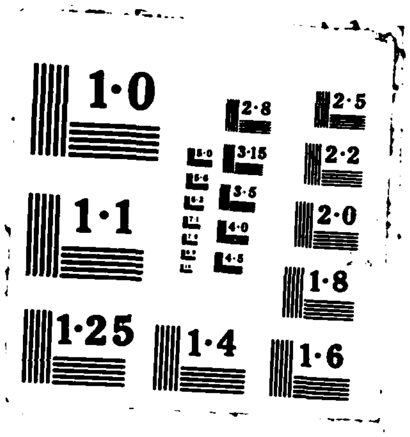
AD-A188 923 VIEWS FOR EVOLUTION IN PROGRAMMING ENVIRONMENTS(U) 1/1
CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING
INST J R NESTOR DEC 87 CMU/SEI-87-TR-45 ESD-TR-87-208
UNCLASSIFIED F19628-85-C-0003 F/G 12/5 NL


1·0
2·8
2·5
3·15
2·2
3·5
1·1
2·0
4·0
4·5
1·8
1·25
1·4
1·6

(2)

Technical Report
CMU/SEI-87-TR-45
ESD-TR-87-208

Carnegie-Mellon University
Software Engineering Institute

AD-A188 923

Views for Evolution in Programming Environments

John Nestor

December 1987

88 2 01 114

Technical Report
CMU/SEI-87-TR-45
ESD-TR-87-208
December 1987

Views for Evolution in Programming Environments

John Nestor

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

i

This technical report was prepared for the

SEI Joint Program Office
ESD/XRS
Hanscom AFB, MA 01731

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER

[illegible]
Karl Shingler
SEI Joint Program Office

This work was sponsored by the U.S. Department of Defense.

Views for Evolution in Programming Environments

John R. Nestor

Software Engineering Institute, Carnegie Mellon University

November 29, 1987

Abstract

Programming environments have become a focal point for much of the work directed toward improving the practice of software engineering. Such environments must provide mechanisms for recording and organizing the complex set of persistent technical and management data associated with all parts of the lifecycle of large software systems. This paper focuses on one important aspect of such persistent data: how to allow evolution when the existing information must be preserved without change to maintain history. First, the role of history in programming environments is discussed. Next, the additional demands of evolution are considered and shown to lead to a set of problems. View mechanisms are suggested as a solution to these problems. A simple example involving file system directory structure is presented to illustrate these problems. A simple view mechanism, called multidirectories, is introduced and shown to solve the illustrated problems.

This work was sponsored by the Department of Defense. The views and conclusions contained in this paper are those of the author and should not be interpreted as representing official policies, either expressed or implied, of the Software Engineering Institute, Carnegie Mellon University, the Department of Defense, or the U.S. Government.

1 Persistent Object Bases

The demand for software is steadily increasing both in the number of systems being built and in the complexity of these systems. Our ability to produce these systems has also been increasing in part due to an increasing number of qualified software developers and in part due to the increased per-person productivity resulting from improved technologies and methods. Unfortunately, demand for software has consistently been growing faster than our ability to produce it. Software is often delivered late, costs much more than originally predicted, and fails to satisfactorily do the job for which it was built. The combination of these problems is what many have characterized as the software crisis and has led to increasing emphasis on the field of software engineering whose goals are directed at solving these problems.

Modern software technologies allow software engineers to automate many of the processes that previously were 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. These automation and communication needs place increasing demands upon the underlying hardware. 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.

A *programming environment* is a set of hardware, software, and methods that assists software engineers with all aspects of software production. Programming environments have become a focal point for much of the work directed toward improving the practice of software engineering. The management of persistent data is increasingly becoming a central issue for such environments. Environments are being called upon to be the repository of all information, both technical and managerial, created throughout the lifecycle of software system from requirements though deployment and continued enhancement. Each phase of the lifecycle adds new information particular to that phase as well as information relating that new information to previous information. All of the information is modified and extended as work progresses. Not only is the current state of information needed but is often necessary to retrieve historical information as well.

Most programming environments now support persistent data by using a file system with one or more ad-hoc databases. Most new environment efforts are moving toward a more object-oriented approach that is a synthesis of ideas from file systems and databases. Such next generation systems are referred to as *persistent object bases* and consist of a set of persistent objects that capture both file-like concepts, such as source and executable programs, and database concepts, such as attributes and relations. These new systems will differ in several important ways from traditional database systems [1,2].

- **Data and Operations.** While traditional database systems provide support for large sets of small fixed size objects of a limited number of types, persistent object bases will need to support large variable sized objects of many types with complex interrelationships. While simple operations such as join and projection are common in traditional databases, persistent object bases require more complex operations such as transitive closure. Although persistent object bases can be built on top of existing databases, the differences in data and operations result in an unacceptable loss of performance.

- **Distribution.** Most current database systems run entirely on a single machine. Persistent object bases will need to run on large heterogeneous networks that include many workstations and special server nodes all integrated into a coherent whole.

- **Transactions.** While database systems typically process very short transactions lasting less than a second each, persistent object bases must support long transactions, such as modifying a source file, that can last hours or even days.

- **Abstraction.** Most database systems have only a small number of built-in types that are structurally composed by a central database administrator to form an overall schema for the entire database. In a persistent object base, full abstract data types must be supported and type definition ability must be extended to both tool builders and end users.

- **History.** A persistent object base must be be able to support historical versions of individual objects and of complex configurations. It must be possible to go back to an earlier release of a software system and re-create it in exactly the same way in which it was originally created.

Although the focus here is on the needs of programming environments, similar problems arise in other applications that require a large number of variable sized objects kept over a long period of time and shared by large groups of people. Both computer aided design and the document filing aspects of office automation have these characteristics. Solutions identified by research in software environments will also be applicable to these and other similar application areas.

This paper focuses on one important aspect of persistent object bases: how to allow evolution when the existing information must be preserved without change to maintain history. First, the role of history in programming environments is discussed. Next, the additional demands of evolution are considered and shown to lead to a set of problems. View mechanisms are suggested as a solution to these problems. A simple example involving file system directory structure is presented to illustrate these problems. A simple view mechanism, called multidirectories, is introduced and shown to solve the illustrated problems.

2 History

Two related history concepts needed in programming environments are introduced 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; 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 do not preserve versions below some minimum level of granularity. The finest granularity is a version for every edit. A coarse granularity is at versions that are major releases. 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.

Re-creation is the ability to go back to an old version of a software product and repeat all of the steps that were involved in manufacturing it [3]. Manufacturing takes primitives such as source modules and produces products such as executable programs by performing a set of manufacturing steps. All inputs to each manufacturing step, including the program used to perform the step must either be a primitive or the result of some previous step. The partially ordered set of manufacturing steps is captured by a derivation graph. Re-creation of a product is possible if its derivation graph and set of primitive components are known and they have not been changed since the product was originally created.

Re-creation is important because it captures key information about a product. If a product is re-creatable, then the relation between that executable product and the primitive source modules from which it was built will be known. Re-creation also makes it possible to produce a variant of some product.

3 Evolution

The way in which an object is viewed during the software development process will evolve over time. Even though its value may remain fixed, other kinds of information will be added or modified. An important trend in programming environments is to provide information not just about the software product but also about the software process. Process activities such as analysis and testing, reuse of a module in multiple products, and iterative development strategies all contribute substantial evolutionary information about program objects.

In environments based on traditional file systems, evolution is accomplished by moving, copying, editing, and deleting of information. This approach is not appropriate for next generation environments for two reasons:

- **History.** It is very difficult to record a complete version history and to ensure re-creation when the information involved has been moved or extended.

- **Identity.** When multiple copies of some object are created, the logical equivalence of the two copies is lost.

An alternate approach is considered here. The basic idea is to keep history information fixed and thus ensure re-creation, but to allow the way in which that information is viewed to evolve. A *view mechanism* is software that allows a map to be imposed between the user and the data. This map provides for various kinds of evolution:

- **Additions.** During evolution, new properties are added to objects. If the manufacturing process can be affected by this addition of object properties, then re-creation may not be possible. This problem is caused by treating the properties of an object as part of the object. In file systems, such properties include the fully qualified name of the file and attributes such as creation date. Most database systems have attribute mechanism that group attributes with objects. A solution is to store new attributes in some place away from the object to which they apply and then use a view mechanism to make it look as thought they are part of the object. In the same way that there can be versions of individual objects, there can also be versions of the system views which each view representing some point in time.

- **Restrictions.** As the amount of information in an environment increases, a user can easily get lost in the overall complexity. Views can be used to solve this problem by giving each class of user, or even each individual user, a personal view into the small subset of the total information that is relevant to the task at hand.

One general purpose approach to views is to allow arbitrary functions that map between the existing information and the way in which that information is viewed by the user. Although such an approach provides full generality, such generality is often only obtained with a considerable cost in lost performance. Special case view mechanisms are of interest when they provide for an important set of real needs without sacrificing performance. By combining such special purpose mechanisms together under a general view abstraction it may be possible to have the full power and generality combined with high performance for most common cases where it is used. One such special case view mechanism is presented here. The next subsection introduces a set of problems with directories as found in common file systems. The following subsection then shows how those problems can be solved by a simple special case view mechanism.

4 Directories

Most file systems are organized as a tree of files, each of which is either a directory or an ordinary file. 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. Not only is considerable user effort required to perform the reorganizations, but these reorganizations compromise re-creation.

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

```
/projects/Q
```

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/documentation
/projects/Q/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 all references to `Q` must now be changed to refer to one or the other or both of the two new subdirectories.

Consider next that it is time to release the Q system to users. Users need 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.

```
/release/Q
```

Moving files was bad enough, but in this case there are now actually two copies of the same files.

Next, consider that it is time to produce a new version of the Q system. If the previous version of Q is to be preserved, the directories must be split.

```
/projects/Q/documentation/V1
/projects/Q/documentation/V2
/projects/Q/program/V1
/projects/Q/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.

The directory tree shown above is only one of several possible ways of organizing the information. The following are alternative ways of structuring the tree:

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

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

The presence of several equally valid forms indicates each form contains some information that is a property of the representation of the form that has little to do with the overall logical structure of the information.

5 Multidirectories

A simple view mechanism responds to the problems illustrated in the previous section. The mechanism is based on an extended kind of directory, a *multidirectory*, in which each name can map to a set of objects. One use of a multidirectory is to divide a set of objects into a set of partitions, each of which represents some equivalence class. This can be further generalized by letting the named subsets overlap and thus produce a cover rather than a partition. Several multidirectories can be used to provide different orthogonal covers of a single set.

For example, suppose the Q system in the previous section included all of the following files:

```
Q.h.1           Q.h.2
x.c.1           x.c.2
        y.c.1
Q.exe.1         Q.exe.2
Q.mss.1         Q.mss.2
Q.doc.1         Q.doc.2
```

We can have several multidirectories of these files.

- **Name.** This partition groups all versions of the same object together under its name.

```
Q.h      -> {Q.h.1,Q.h.2}
x.c      -> {x.c.1,x.c.2}
y.c      -> {y.c.1}
Q.exe    -> {Q.exe.1,Q.exe.2}
Q.mss    -> {Q.mss.1,Q.mss.2}
Q.doc    -> {Q.doc.1,Q.doc.2}
```

- **Configuration.** This cover groups the files into configurations corresponding to two versions of the system.

```
V1       -> {Q.h.1,x.c.1,y.c.1,Q.exe.1,
             Q.mss.1,Q.doc.1}
V2       -> {Q.h.2,x.c.2,y.c.1,Q.exe.2,
             Q.mss.2,Q.doc.2}
```

- **Origin.** This partition distinguishes user written files from derived files.

```
User     -> {Q.h.1,Q.h.2,x.c.1,x.c.2,y.c.1,
             Q.mss.1,Q.mss.2}
Derived  -> {Q.exe.1,Q.exe.2,Q.doc.1,Q.doc.2}
```

- **Kind.** This partition distinguishes program files from documentation files.

```
Program       -> {Q.h.1,Q.h.2,x.c.1,x.c.2,y.c.1,
                  Q.exe.1,Q.exe.2}
Documentation -> {Q.mss.1,Q.mss.2,
                 Q.doc.1,Q.doc.2}
```

- **Access.** This partition distinguishes files released to users from files that are only available to developers.

```
Internal        -> {Q.h.1,Q.h.2,x.c.1,x.c.2,y.c.1,
                    Q.mss.1,Q.mss.2}
Release         -> {Q.exe.1,Q.exe.2,Q.doc.1,Q.doc.2}
```

These multidirectories can be composed in order to yield many different views of the system. The basic composition operator enables a multidirectory to be viewed `via` some set. Thus, `Name via Configuration.V2` is:

```
Q.h     -> {Q.h.2}
x.c     -> {x.c.2}
y.c     -> {y.c.1}
Q.exe   -> {Q.exe.2}
Q.mss   -> {Q.mss.2}
Q.doc   -> {Q.doc.2}
```

As a more complex example, `Name via Origin.User via Kind.Documentation` is:

```
Q.mss   -> {Q.mss.1,Q.mss.2}
```

An alternate way of thinking about a multidirectory is that it gives the value of some attribute of a set of files. For example, the `Kind` multidirectory, provides the following values for the `Kind` attribute:

```
Q.h.1   -> "Program"
Q.h.2   -> "Program"
x.c.1   -> "Program"
x.c.2   -> "Program"
y.c.1   -> "Program"
Q.exe.1 -> "Program"
Q.exe.2 -> "Program"
Q.mss.1 -> "Documentation"
Q.mss.2 -> "Documentation"
Q.doc.1 -> "Documentation"
Q.doc.2 -> "Documentation"
```

Observe that since the attributes are not stored with the object to which they apply, they can be added without modifying the object or compromising re-creation.

Multidirectories provide solutions to the evolutionary problems of normal file system directories:

- **Copies.** When a single file is used in several different ways, instead of creating multiple copies of the file, multidirectories can be used to create multiple views of the file. For example, instead of copying files from the `/projects/Q` directory to the `/release/Q` directory, the `Access` multidirectory was added to permit the views `Name` and `Name via Access.release`.

- **Evolution.** Multidirectories permit information to evolve without the need to modify or move existing files. For example, instead of having to split `/projects/Q` into

  ```
  /projects/Q/documentation
  /projects/Q/program
  ```

  all that was needed was to add the `Kind` multidirectory. No files were copied or modified, and therefore, re-creation was not compromised.

- **Alternate Representations.** Multidirectories avoid the problem of having to select some specific representation out of a set of equally valid alternatives. For example, the directory distinction between

  ```
  /projects/Q/documentation/V1
  /projects/Q/V1/documentation
  /V1/projects/Q/documentation
  ```

  disappears when multidirectories are used.

In summary, multidirectories allow users to organize their information in a more natural way than is possible in a conventional file system. Multidirectories allow information to evolve without forcing changes that could make re-creation difficult. Finally, multidirectories are a simple view mechanism that should have quite a efficient implementation.

6 Conclusions

Existing programming environments have the problem that they can not simultaneously guarantee re-creation, permit evolution, and avoid multiple copies. View mechanisms solve this problem by allowing historical information to remain fixed thus supporting re-creation but allowing the way that we view that data to evolve. Such view mechanisms will be a key component of the persistent object bases on which future programming environments will be based.

7 References

1. John R. Nestor, "Toward a Persistent Object Base", *International Workshop on Programming Environments*, Lecture Notes in Computer Science, Number 244, Springer-Verlag, 1986.

2. Philip A. Bernstein, "Database System Support for Software Engineering, An Extended Abstract", 9th International Conference on Software Engineering, IEEE Computer Society Press, 1987.

3. Ellen Borison, "A Model of Software Manufacture", *International Workshop on Programming Environments*, Lecture Notes in Computer Science, Number 244, Springer-Verlag, 1986.

UNLIMITED, UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

AD-A188923

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS NONE
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT APPROVED FOR PUBLIC RELEASE DISTRIBUTION UNLIMITED
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) CMU/SEI-87-TR-45	5. MONITORING ORGANIZATION REPORT NUMBER(S) ESD-TR-87-208
6a. NAME OF PERFORMING ORGANIZATION SOFTWARE ENGINEERING INSTITUTE 6b. OFFICE SYMBOL (If applicable) SEI	7a. NAME OF MONITORING ORGANIZATION SEI JOINT PROGRAM OFFICE
6c. ADDRESS (City, State and ZIP Code) CARNEGIE MELLON UNIVERSITY PITTSBURGH, PA 15213	7b. ADDRESS (City, State and ZIP Code) ESD/XRS1 HANSCOM AIR FORCE BASE, MA 01731
8a. NAME OF FUNDING/SPONSORING ORGANIZATION SEI JOINT PROGRAM OFFICE 8b. OFFICE SYMBOL (If applicable) SEI JPO	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F1962885C0003
8c. ADDRESS (City, State and ZIP Code) CARNEGIE MELLON UNIVERSITY SOFTWARE ENGINEERING INSTITUTE JPO PITTSBURGH, PA 15213	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO.: PROJECT NO.: N/A TASK NO.: N/A WORK UNIT NO.: N/A

11. TITLE (Include Security Classification)
VIEWS FOR EVOLUTION IN PROGRAMMING ENVIRONMENTS

12. PERSONAL AUTHOR(S)
JOHN R. NESTOR

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
FINAL	FROM ______ TO ______	DECEMBER 87	12

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB. GR.	PROGRAMMING ENVIRONMENTS, PERSISTENT DATA, VIEW MECHANISM MULTIDIRECTORY

19. ABSTRACT (Continue on reverse if necessary and identify by block number)
PROGRAMMING ENVIRONMENTS HAVE BECOME A FOCAL POINT FOR MUCH OF THE WORK DIRECTED TOWARD IMPROVING THE PRACTICE OF SOFTWARE ENGINEERING. SUCH ENVIRONMENTS MUST PROVIDE MECHANISMS FOR RECORDING AND ORGANIZING THE COMPLEX SET OF PERSISTENT TECHNICAL AND MANAGEMENT DATA ASSOCIATED WITH ALL PARTS OF THE LIFECYCLE OF LARGE SOFTWARE SYSTEMS. THIS PAPER FOCUSES ON ONE IMPORTANT ASPECT OF SUCH PERSISTENT DATA: HOW TO ALLOW EVOLUTION WHEN THE EXISTING INFORMAITON MUST BE PRESERVED WITHOUT CHANGE TO MAINTAIN HISTORY. FIRST, THE ROLE OF HISTORY IN PROGRAMMING ENVIRONMENTS IS DISCUSSED. NEXT, THE ADDITIONAL DEMANDS OF EVOLUTION ARE CONSIDERED AND SHOWN TO LEAD TO A SET OF PROBLEMS. VIEW MECHANISMS ARE SUGGESTED AS A SOLUTION TO THESE PROBLEMS. A SIMPLE EXAMPLE INVOLVING FILE SYSTEM DIRECTORY STRUCTURE IS PRESENTED TO ILLUSTRATE THESE PROBLEMS. A SIMPLE VIEW MECHANISM, CALLED MULTIDIRECTORIES, IS INTRODUCED AND SHOWN TO SOLVE THE ILLUSTRATED PROBLEMS.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED [✓] SAME AS RPT. ☐ DTIC USERS [✓]	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED, UNLIMITED
22a. NAME OF RESPONSIBLE INDIVIDUAL KARL 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.

UNLIMITED, UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE