Research in Software

David G. Belanger
Eric E. Sumner, Jr.
Peter J. Weinberger

Computer-aided software engineering (CASE) tools are providing new and more effective resources for product development in a rapidly changing software industry. They are a critical part of an increasingly competitive environment in which software has become a significant part of all telecommunications products, and in which complex problems require solutions that, paradoxically, reflect complexity of structure while making it easier to find quick and effective solutions to design problems. This paper examines trends in software research in the 1990s in terms of problems and the technologies to help solve them.

Introduction: The Problem of Change

Research is about change and finding opportunities, within the change, to do things better. In the past decade, several fundamental technical and non-technical changes have occurred in software production technology and in research to improve it. The specific requirements of product development—which include non-technical changes and the structure of research's relationship to the developer—are driving the following trends:

- Concentration on elapsed time, as well as on the traditional factors of cost, quality, and customer value.[2]
- Mass customization, i.e., the ability to deliver products tailored for individual customers without losing economies of scale.

These and other trends have had a significant effect on the technologies investigated by research. Indeed, the environment for software production research itself has changed, perhaps most dramatically in the emergence of a dynamic industry to provide tools for software production. Known as computer-aided software engineering (CASE), this industry supplies tools for the software life cycle, many of which would once have been created internally by research organizations.

The ways software production research interacts with its customers, product developers, and available set of problems also is changing. The current direction is toward forming small teams for direct interactions between an organization's development and research areas. These trends are changing both the crucial technologies that research investigates, and the mode of that investigation.

Unfortunately, some trends in software production are not changing. Among them are:

- The requirements for software within products continues to require developing more complex software. Solutions to yesterday's problems are insufficient.
- No telecommunications company can afford to see a competitor gain significant, lasting advantage in software production. What has changed is that the term "lasting" refers to a much shorter interval than in the past.
- Breakthrough ideas are rare, valuable, and still take a long time to mature.

Though it is true we can build some types of software far more efficiently than a few years ago, and that systems that would have seemed hopelessly complex a decade ago are now far more manageable, nevertheless the demands on software to provide product differentiation continue to drive toward higher performance, functionality, and complexity. This paper describes some approaches to software research in the 1990s in terms of the problems being addressed, some of the technologies currently under investigation, and the methods of addressing those problems and technologies.

Given the diversity of problems in software development, and the nature of the research, we attempt in this paper to sketch some promising current areas. Furthermore, we provide some insight into what is changing in software research. We hope, and even expect, that pleasant surprises will occur from directions we have not included here, and even from unanticipated directions. This paper, then, should be read as a view of some approaches that may be used in coming years to change the way software development is done.

The Problems and The Technology

As noted above, at least two trends are rapidly changing the notion of what issues are critical in system development: product development time, and mass customization. In this section, we examine some of the implications of these changes in software technology and some approaches currently being investigated in AT&T Bell Laboratories to address them.

Time. To understand where time is consumed in building a large software system, it is useful to think in terms of an activity model. Figure 1 is a simple model that allows us to understand the activities taking place. Most common software lifecycle models are made up of the cycles described below, executed with different packaging and ordering. For example, the classic waterfall lifecycle model consists of only one "Requirements" cycle, containing a single "Design" cycle. On the other hand, the spiral[5] consist of several, much shorter, "Requirements" cycles, each containing their associated "Design" and "Implementation" cycles.

System development activities are classified into three cycles or "loops" that are nested, one inside the other. During software system development, each loop is executed repeatedly, and the inner loops are executed more than the outer. To a first approximation, we can look at the technology's time-saving ability to reduce the required iterations of the loops, and to reduce the length of each loop. Many successes in this effort lie in the concept of keeping individual feedback loops short. For example, inspections and reviews provide earlier feedback than waiting until the end of a loop. Note that this model does not explicitly address the use of concurrency to reduce time. Some of those issues will turn up in our discussion of customization.

The loops described in Figure 1 are:

- *Unit loop.* This is the innermost loop, the most frequently executed in most projects. Simply stated, it involves creating, changing, or inspecting and testing a module. The module could be a document instead of code. This loop involves the least amount of communication between and among implementors, especially if the system being built has few design "side effects."
- *Design loop.* The entry to this loop is system design; the exit is a successful system integration test. Unlike the unit loop, the design loop nearly always involves extensive communication among implementors. Because it includes many iterations through the unit loop, the design loop tends to be long and sensitive to the architecture of the system being built.
- *Requirements loop.* This loop spans activities from system definition in terms of expected behavior through testing to ensure that it behaves as specified. Hypothetically, this loop only needs to be executed once. However, requirements are likely to change and errors found during a development project. Communication within this loop includes a significant amount outside the development organization (e.g., customers, communicating systems).

Performance in each loop depends to a great extent on the loop immediately outside it. A poorly designed system will be difficult to code quickly and well, and a poorly specified system will be hard to design. This problem is magnified as software gets older. Poor definitions or designs almost never improve merely by adding features to the system.

Unit loop. Within the unit loop, time reduction often is closely related to the speed and functionality of a few tools in the environment, and to the computer language used. The reduction is even more closely tied to the ability of the person executing the loop. Traditionally, research into reducing time spent in this cycle has involved creating tools to analyze code either statically (e.g., browsers) or dynamically (e.g., debuggers and interpreters). Faster computers generally have had a dramatic effect on this loop simply by reducing the time it takes to do compiling and other basic functions.

Much of the current research related to this area is in languages, software reuse, and language environments. Some of these are general-purpose languages, such as C++[16] and ML.[13] Others are special-purpose, such as Cymbal for database manipulation, and PRL

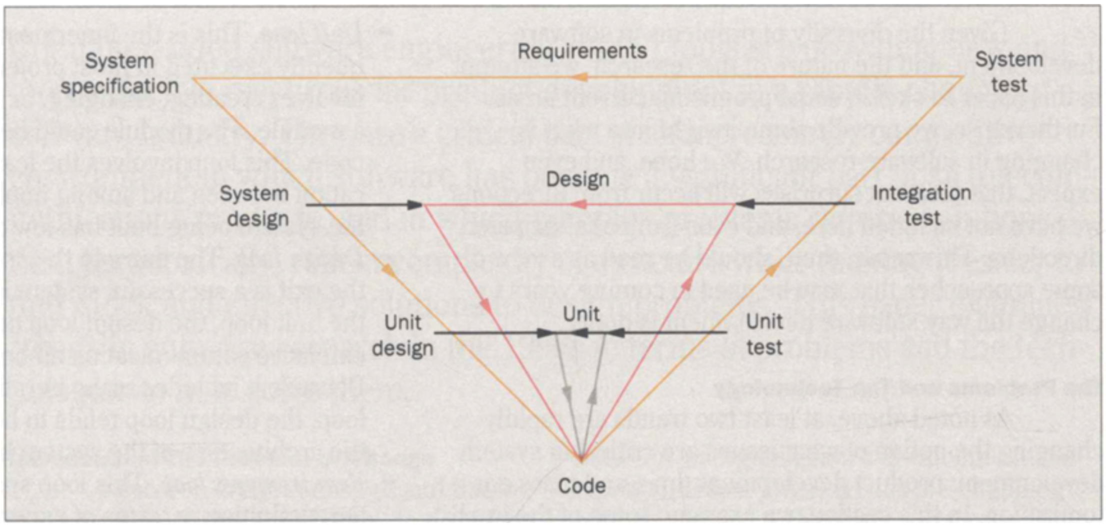


Figure 1. A representation of the software development cycle, showing the requirements, design, and coding cycles and the specific activities that take place within them.

(switching database integrity).[15] Their unified purpose is to reduce the lines of code that must be written to implement a function, and to reduce the defects in that code. This is often done, as with ML, by increasing the services the language automatically provides (e.g., garbage collection and concurrency), and by increasing the language's formality. With special-purpose languages, the increase in services usually includes services directly related to specific applications, e.g., data manipulation with Cymbal. This allows the language to look more like a natural specification of the problem than like a traditional programming language, thereby reducing errors in translation to implementation.

At most, reuse dramatically reduces the number of iterations through this loop by providing prepackaged, pretested "units." The present reuse levels are inadequate, and current research is paying serious attention to technology to increase them.

Design Loop. In most systems projects, especially where the waterfall model is practiced, the design through integration phases are viewed as a sequence instead of a loop. This is almost always a fiction, sometimes perpetuated by making all iterations but the first informal and invisible. The effect of this is reflected in a common observation: "the only accurate documentation is the code." That is, the system was redesigned many times, but the only visible artifact is the code itself. Although CASE tools are an approach to this problem, the more fundamental approach is to make visible the relationships among all the artifacts: design, code, and documentation, followed by careful tracking of program changes. As an example, Figure 2 contains a visible representation of the inheritance relationships between objects in a small subset of a C++ program. There are other representations that are also useful, including concise representations of actual code, and representations of replicated elements of a system.

The SeeSoft software visualization system[7] (Figure 3) allows analysis of up to 50,000 lines of code simultaneously by mapping each line of code into a thin row. The color of each row, shown here in gray, indicates a statistic of interest, i.e., the age of the changes. SeeSoft shows data derived from a variety of sources, such as version control systems, static analyses, and dynamic analyses. Using direct manipulation and high interaction graphics, the user can manipulate this reduced representation of the code in order to find interesting patterns. Further insight is obtained by using additional windows to display the actual code. Potential applications for SeeSoft include discovery, project management, code tuning, and analysis of development methodologies. This approach has the added advantage, where inspections and reviews are used, of conveying more information, more compactly, to reviewers.

Decreasing the design loop interval and number of loops involves reducing the amount of communication needed to build a system. This reduction makes the communication that remains more effective, and reduces the design decisions. One problem with both the design and requirements loops is that the feedback cycle is long.

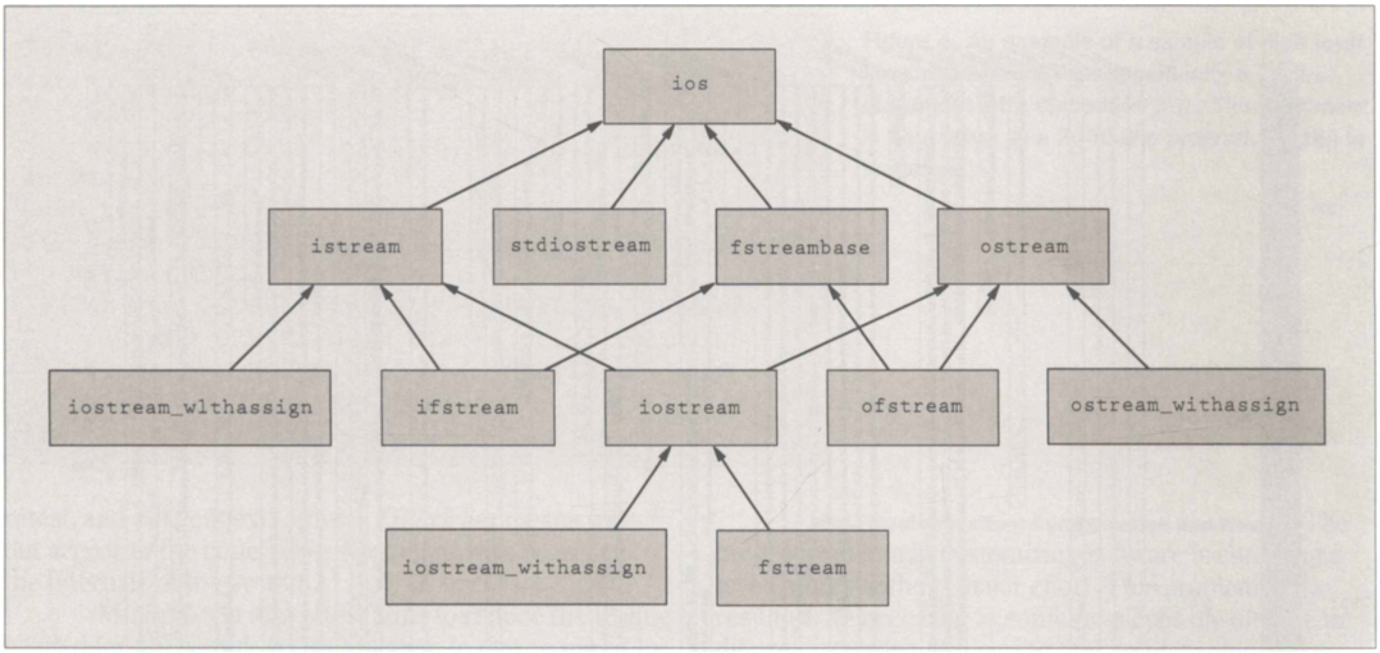


Figure 2. This drawing shows relationships among objects in a small piece of a C++ program. The relationships are of object inheritance. In an environment where inspections and reviews are used, this approach conveys more information, more compactly, to reviewers.

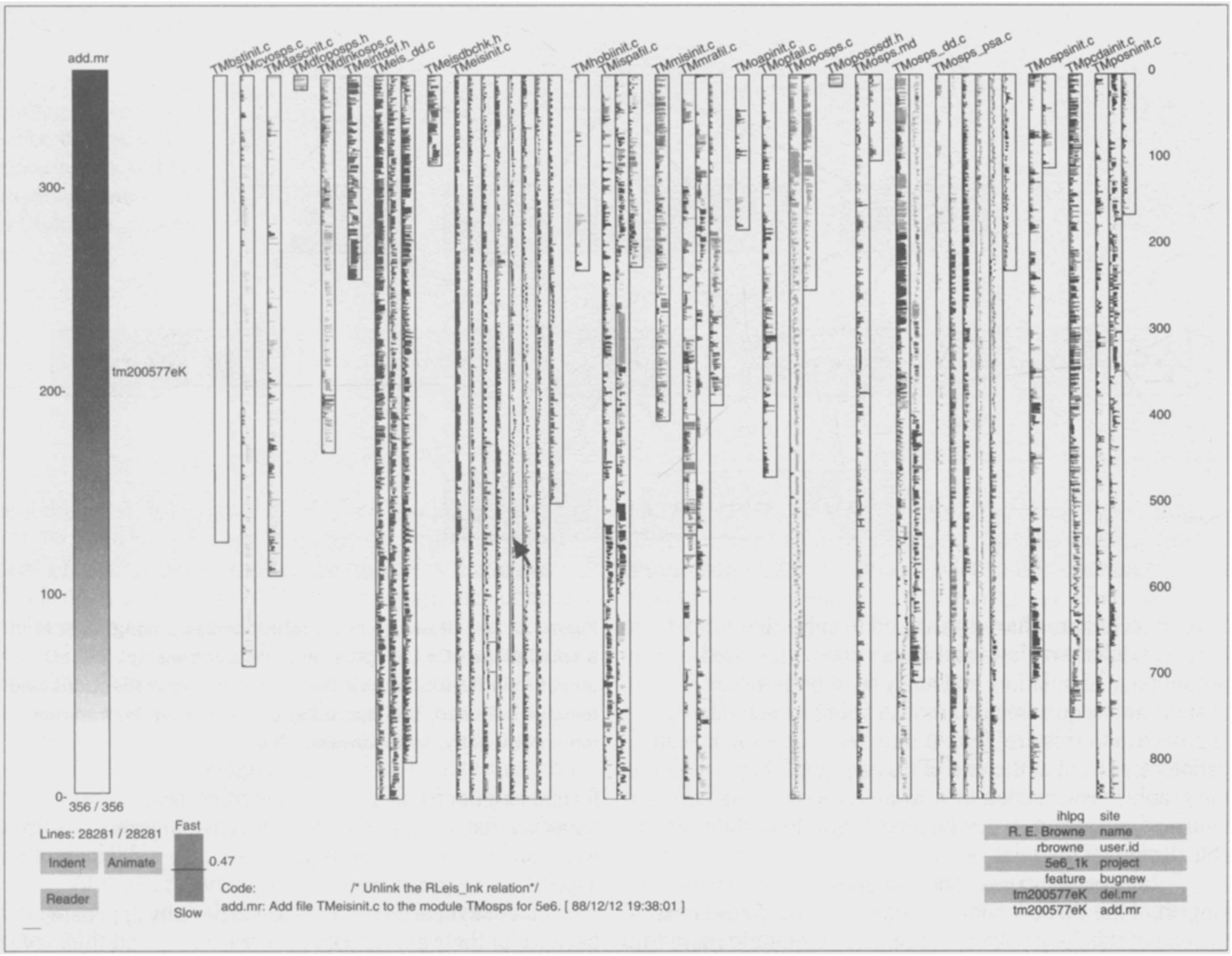
Several techniques have been used to solve that problem, particularly reviews and inspections. Increased prototyping, simulation, and analysis of both formal designs and requirements are also being investigated as approaches to this problem. A number of specific investigations are listed at the end of this section. They include the graphic visualization of software systems; software information systems[4]; design level languages; and reusable design techniques.

Requirements Loop. The requirements loop is the longest of the three feedback loops. An error found near the end of this loop will cost orders of magnitude more to fix than an error found near the beginning. This loop also contains the information most subject to ambiguity and the user's taste. The trend toward "ease-of-use" is particularly important in this realm. As with the design loop, creating short internal feedback loops is a primary way to reduce the loop's iterations. Moreover, because the testing effort for telecommunications products is a large proportion of development time and cost (30 to 50 percent), technology to reduce that effort is crucial.

In common with other loops, requirements loop specifications need inspections and reviews to reduce feedback intervals. In some areas, models of part of the software system (e.g., finite state machines) are well enough understood so specifications can be written in a formal language and analyzed for consistency. This approach has been particularly effective in protocol software, and several research projects address different aspects of the problem.[10]

Analytical techniques are especially appealing because of their guarantees of consistency, and their potential to significantly reduce testing efforts. We are not yet at the point, even in research labs, where we can develop a significant product using only a specification language. Even were that possible, the requirements of a system are often ambiguous enough so we must first learn what they should be.

Prototyping is an effective process to learn what a system should look like, assuming there are tools powerful enough to make prototype creation feasible. User interfaces traditionally have been a fertile area for prototyping tools, and research continues on graphical user interface creation, now largely based on the X Window System.®[11] (X Window System is a registered trademark of the Massachusetts Institute of Technology.)



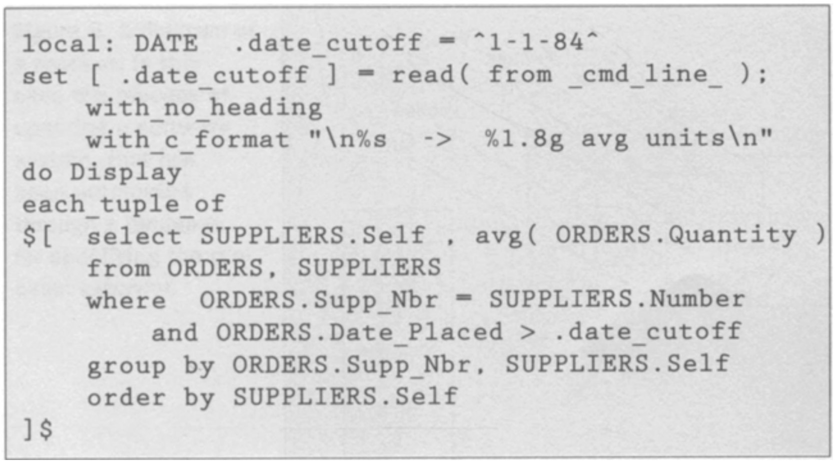
```
Code:                    /* Unlink the RLeis_lnk relation*/
add.mr: Add file TMeisinit.c to the module TMosps for 5e6. [ 88/12/12 19:38:01 ]
```

ihlpq	site
R. E. Browne	name
rbrowne	user.id
5e6_1k	project
feature	bugnew
tm200577eK	del.mr
tm200577eK	add.mr

Figure 3. A SeeSoft display of a directory with 20 files and about 28,000 lines of code. Each file is represented as a column and each line of code as a colored row (shown here in gray). The color of each row is determined by the age of the Modification Request (MR) that created the line.

Interfaces for foreign language character sets, because of AT&T's globalization, is an area of increasing interest. But systems require more than interfaces, and there has been much activity using high-level languages in areas such as database management (including object oriented and deductive databases), data communications, and artificial intelligence. (Figure 4 shows an example of a sample of high level language statements.) Happily, many of these languages are supported by systems with performance adequate enough for the product. This performance strength allows developers to avoid large parts of the development cycle's inner loops.

The above techniques aim to reduce iterations of the requirements loop, and may help to reduce testing by increasing confidence in the coding process. They do not *eliminate* testing, nor do they promise to displace testing in the foreseeable future as the most difficult part of development. Researchers are looking at other approaches to testing, both "white" and "black" box. We expect these approaches to address two issues: minimal test and



```
local: DATE  .date_cutoff = ^1-1-84^
set [ .date_cutoff ] = read( from _cmd_line_ );
    with_no_heading
    with_c_format "\n%s  ->  %1.8g avg units\n"
do Display
each_tuple_of
$[  select SUPPLIERS.Self , avg( ORDERS.Quantity )
    from ORDERS, SUPPLIERS
    where  ORDERS.Supp_Nbr = SUPPLIERS.Number
        and ORDERS.Date_Placed > .date_cutoff
    group by ORDERS.Supp_Nbr, SUPPLIERS.Self
    order by SUPPLIERS.Self
]$
```

Figure 4. An example of a sample of high level language statements, specifically a `Cymbal` **statement with embedded** `SQL`**. The statement is equivalent to a 3000-line program written in** `C` **language.**

retest, and efficient exit criteria. (*Black box* means without access to the code, *white box* means with access to the internals of the system.)

Minimal test and retest aims to reduce the testing needed for full confidence in a system to that required by the specific changes. Because changes to a large system usually affect only a small fraction of the system, this can result in significant time savings, and may include techniques for effective coverage with reduced input sets. Another approach uses information from code changes to determine the minimal set of necessary regression tests.

Exit criteria for testing focus on determining when enough testing has been done. The software reliability engineering field is receiving widespread investigation inside and outside AT&T.[14]

AT&T research toward interval reduction by reducing the number or length of the cycles described above include the following areas:

- General purpose languages and environments, e.g., ML, C++
- Special purpose languages for database management and artificial intelligence
- Software visualization and software information systems
- Design annotation
- Software reuse structure, objects, and tools
- Graphical user interface tools for prototyping and production
- Test minimization
- Specification languages
- Specification and design process measurement and improvement.

Mass Customization: Configuration and Reuse. The biggest problem in customizing software is changing a few features without major effort. This problem is the result of the necessity to combine pieces of software in different ways while assuring that no unwanted effects occur from new combinations. Effective customization thus allows one product to be delivered to several customers meeting their exact needs. A simple example would be to change from an Open Look® graphical user interface to a Motif™ graphical user interface. (Open Look is a registered trademark of UNIX Systems Laboratories; Motif is a trademark of the Open Software Foundation.) Another is to package features in different combinations, much as one purchases features packaged on an automobile.

One way to view this problem is from the standpoint of seeking high levels of reuse—i.e., more than 90 percent—based on a large existing product base. Another way to view it is in terms of the software asset base for a system being designed and controlled so several systems can be built from it. That is, the software assets are configured for quick and accurate manufacturing, testing, packaging, and shipping.

Software systems are amalgams of a "parts inventory" that must be controlled and manufactured according to precise rules. While a computer may have 50 thousand parts[3] that must be manufactured with strict quality control, it may have 10 million lines of system software, and many tens of millions of lines of application software whose connections to each other are even more elaborate than the physical connections. It will also probably have several million lines of documentation

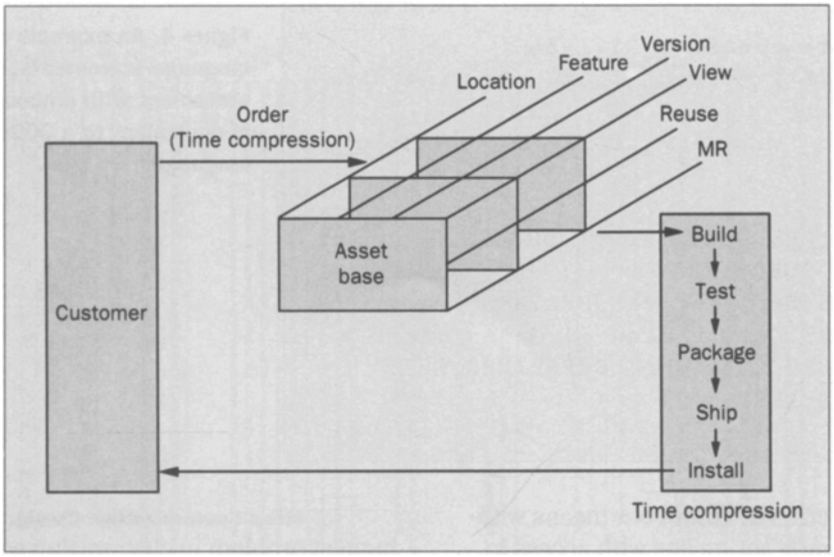


Figure 5. A collection of software assets (e.g., code, documentation, test scripts), and some of the multiple dimensions that affect how they are partitioned.

presented as text and pictures. These lines of code are typically packaged as modules with contents under change control. A large system may have thousands of modules, each with many versions. Thus, manufacturing amounts to choosing the right modules and assembling them into a product. We must be able to recreate old versions of the system, and must protect against arbitrary (but easily made) changes to the system.

Configuration Management. Figure 5 shows a collection of software assets (e.g., code, documentation, test scripts), and some of the multiple dimensions that affect how they are partitioned. The dimensions include:

- Location of the assets (including wide-area, global, distribution)
- Multiple versions of each module
- Multiple developers working in parallel (i.e., viewpathing)
- Reuse of the assets within the system, and in other systems
- Change control (tracking requests for modification and their resolution)
- Target machines, or portability.

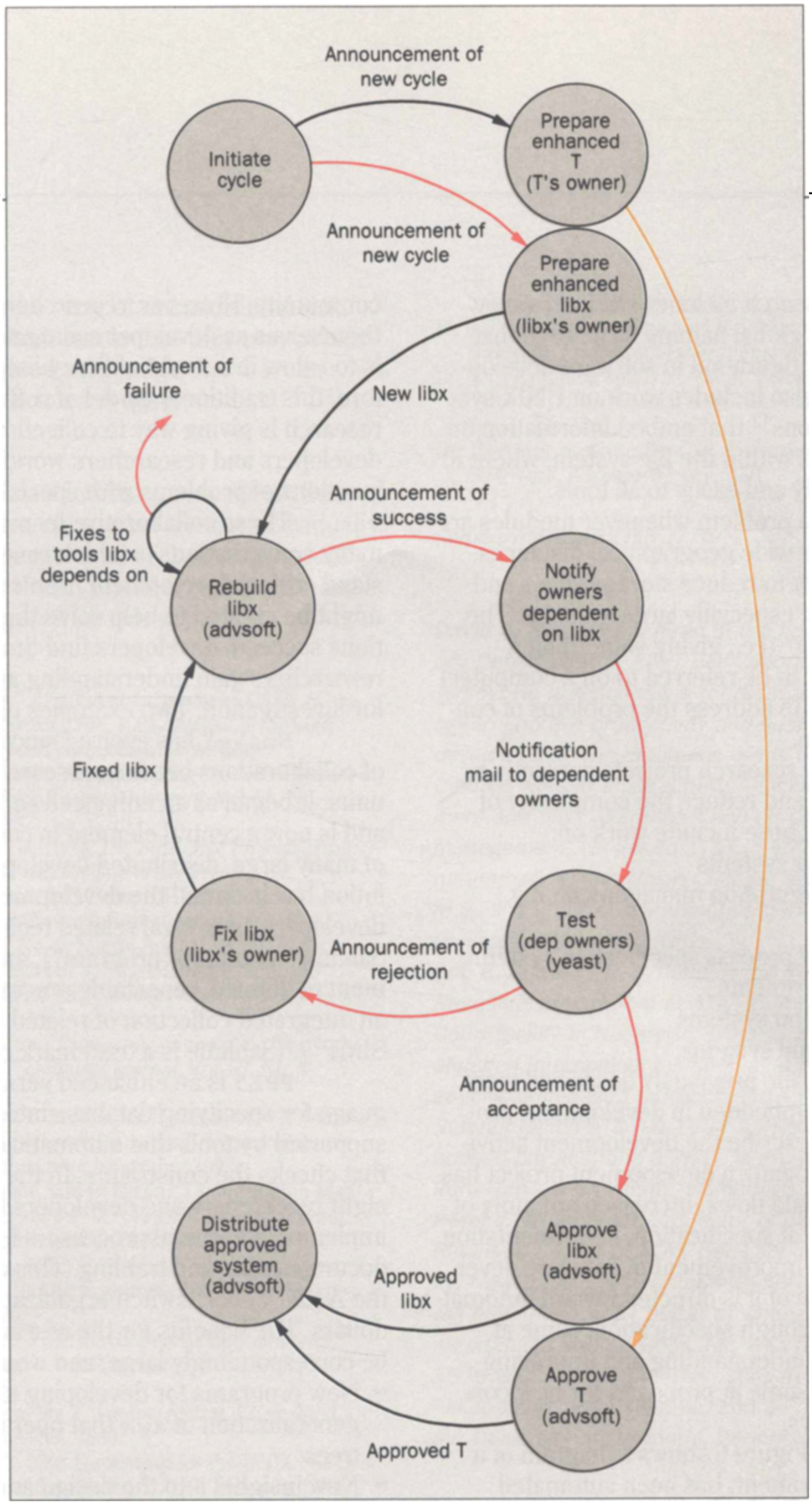
(Note: viewpathing is a technique of using file pathnames to provide users with specific logical views of their file systems.)

The configuration management function is the ability to quickly select appropriate modules from a large set, and forward them to automated systems for manufacture, testing, packaging, shipping to customer sites, and on-site installation. For example, we may wish to reconstruct a system that is scheduled for shipment late this year, and currently is in system test. This implies that we want the latest "official" (i.e., completely tested and released) version of the code (version management), as well as the modules that are new or changed, replacing corresponding "official" modules (i.e., viewpathing).

In a common but somewhat simplified scenario, some of the "official" or changed code may have been written by programmers at another location, so arrangements to find it must be made. In particular, this could be a set of objects being reused. Assuming this system must run on several versions of the UNIX® operating system we will have to select a form of the system appropriate to the target machine. (UNIX is a registered trademark of UNIX System Laboratories, Inc.)

Configuration management allows a developer to view and manufacture a single system. This is preferable to asking developers to reconstruct the system they need manually from a combinatorial explosion of possible systems. For example, in a simple system made up of four libraries updated quarterly, three tools updated semiannually, and where the system itself is shipped bimonthly, the explosion could amount of 12,288 possible combinations in any year (256 x 8 x 6 = 12,288). While many of the combinations are not feasible, this suggests the ease with which complexity of a large system can increase.

Classical version management techniques, e.g. the Source Code Control System,[9] are designed to view



Announcement of new cycle
Initiate cycle
Prepare enhanced T (T's owner)
Announcement of new cycle
Prepare enhanced libx (libx's owner)
Announcement of failure
New libx
Announcement of success
Fixes to tools libx depends on
Rebuild libx (advsoft)
Notify owners dependent on libx
Notification mail to dependent owners
Fixed libx
Fix libx (libx's owner)
Announcement of rejection
Test (dep owners) (yeast)
Announcement of acceptance
Distribute approved system (advsoft)
Approved libx
Approve libx (advsoft)
Approve T (advsoft)
Approved T

Figure 6. A diagram of a process, in this case the process of updating a software system, that has been automated through a language for specifying the process' behavior.

versions of files (small pieces of systems) and changes in those files, rather than an instantiation of the system itself.

A key research goal in this area is to reduce—even eliminate—the complexity caused by the explosion of possible systems, from the developer's view of the system. The developer will view a single system, chosen from the variety that are actual possible. Some of the issues are:

- Naming (OS and file system)
- Consistency (database)
- Compaction
- Notification (process, communication).

The current research addresses these issues from different directions, all aimed at reducing the feedback loop from design through code, build, test, and back to design. The research includes work on a new operating system[1] with global naming structures that should be useful for configuration in software development environments. It also includes work on UNIX system file system extensions[12] that embed information on software asset partitions within the file system, where it is available automatically and easily to all tools.

Consistency is a problem whenever modules are copied, particularly over wide geographical distances. Compaction is necessary to reduce storage space and traffic between systems, especially large systems. The experiments on "naming" (i.e., giving something a unique name so that it can be referred to on a computer) include new techniques to address the problems of consistency and compaction.

There are many research projects currently in progress to understand and reduce the complexity of software development. These include work on:
- Distributing operating systems
- File systems for configuration management, e.g., 3DFS
- Event notification and process specification systems
- Process capture experiments
- Wide area configuration systems
- Configuration and build systems.

Process. Finally, the previously described technologies and loops are embedded in development processes that accurately describe the development activities. Like a computer progam, a development project has both control flows and data flows. Increased amounts of research are now aimed at specification, instrumentation, analysis, execution, and improvement of software development processes. Some of it is directed toward automating parts of a process through specification, some at analysis techniques for understanding and improving existing processes, and some at proposals for new control and information flows.

As an example, Figure 6 shows a diagram of a process that, as an experiment, has been automated through a language for specifying the process' behavior.

Research and Development Collaboration

Two of the most important contributions of AT&T Bell Laboratories to software production methodology, the C programming language and the UNIX operating system, were created by researchers for researchers, and were gradually adopted by the development community. However, researchers can no longer use themselves as developer surrogates, and gradual adoption is too slow in a world of time-based competition. Therefore, this traditional model of software production research is giving way to collections of small teams of developers and researchers working on particular conjunctions of problems with specialized technical expertise.

These collaborative teams typically evolve out of many conversations in which researchers try to understand critical development problems and what technology might be created to help solve them. When the collaborations succeed, developers find timely solutions, while researchers gain understanding and a fresh set of areas for investigation. Two examples of this approach follow.

Nmake[6] has evolved since 1986 through a series of collaborations between research and multiple business units. It began as an enhanced software construction tool, and is now a central element in configuration management of many large, distributed development systems. This evolution has included the development, by researchers and developers, of several related technologies (e.g., the Sablime® computer program[6]), and finally, the development of defined, repeatable processes for efficiently using an integrated collection of related technologies (e.g., SIMP[7]). (Sablime is a trademark of AT&T.)

PRL5 is an enhanced version of the PRL4 language for specifying database integrity constraints. It is supported by tools that automatically generate C code that checks the constraints. In the last year a team of eight researchers and developers has defined PRL5, implemented several processors for it, and developed documentation and training. This collaboration may save the AT&T 5ESS® switch organization tens of millions of dollars. The benefits for the research organization may be correspondingly large, and would include:
- New programs for developing tools, such as Astar, a generalization of awk that operates on program parse trees.
- New insights into the design and implementation of specification languages
- New algorithms for deriving constraints on database updates
- A formal calculus for databases
- A system for reverse engineering of transaction-based C programs.

Researchers working in collaborative teams may be frustrated by the many project-specific issues that must be solved. Developers may be wary of the additional risk. However, it is through such teams that they may achieve what is perhaps the greatest advantage they have in working at AT&T: access to each other.

Acknowledgments

The work discussed in this paper reflects the projects of far more researchers than we can individually acknowledge. Our thanks go to all of them, but special thanks go to S. Eick, R. Greer, J. Grass, and B. Krishnamurthy for providing figures for this paper.

References

1. "Plan 9: The Early Papers," *AT&T Bell Laboratories CSTR [Computer Science Technical Reports] 158*, 1991. Available through anonymous FTP from **inet.att.com:/dist/plan9doc**.
2. G. W. Arnold and M. C. Floyd, "Reengineering the New Product Introduction Process," *AT&T Technical Journal*, Vol. 70, No. 6 (November/December 1992), pp. 12-19.
3. R. A. Bauer, E. Collar, and V. Tang, *The Silverlake Project: Transformation at IBM*, New York, Oxford University Press, 1992.
4. D.G.Belanger, R.J.Brachman, Y.F.Chen, P.T.Devanbu, and P.G.Selfridge, "Toward a Software Information System,"*AT&T Technical Journal*, Vol. 69, No. 2, March-April 1990, pp. 22-41.
5. B. W. Boehm, "A Spiral Model of Software Development and Enhancement," *ACM Software Engineering Notes*, Vol. 11, No. 4, August 1986, pp. 22-42.
6. S. Cichinski and G. S. Fowler, "Product Administration Through Sable and Nmake,," *AT&T Technical Journal*, Vol. 67, No. 4, July/August 1988, pp. 59-70.
7. S. G. Eick, J. J. Steffen, and E. E. Sumner Jr., "SeeSoft—A Tool For Visualizing Line Oriented Software Statisitics," *Transactions on Software Engineering*, forthcoming November 1992.
8. G. S. Fowler, J. E. Humelsine, and C. Olson, "Configuration Management for Large Software Systems," *AT&T Technical Journal*, Vol. 70, No. 6 (November/December 1992), pp. 46-61.
9. T. A. Dolotta, R. C. Haight, and J. R. Mashey, "The Programmer's Workbench," *Bell System Technical Journal*, Vol. 57, No. 6, July-August 1978, pp. 2177-2200.
10. G. J. Holzmann, "An Improved Protocol Reachability Analysis Technique," *Software, Practice and Experience*, Vol. 18, No, 2, February 1988, pp. 137-161.
11. Oliver Jones, *Introduction to the X Window System*, Englewood Cliffs, New Jersey, Prentice-Hall, 1989.
12. D. G. Korn and E. Krell, "A New Dimension for the UNIX File System," *Software, Practice and Experience*, Vol. 20, February 1988, pp. 19-34.
13. Robin Milner, Mads Tofte, and Robert Harper, *The Definition of Standard ML*, Cambridge, Massachusetts, MIT Press, 1989.
14. J. D. Musa, A. Iannino, and K. Okumoto, *Software Reliability: Measurement, Prediction, Application*, New York, McGraw-Hill, 1987.
15. J. C. Ramming and D. A. Ladd, "Software Research and Switching Software," *International Conference on Communication Technology*, September 1992.
16. Bjarne Stroustrup, *The C++ Programming Language*, 2nd Edition, New York, Addison-Wesley, 1991.

(Manuscript received July 17, 1992)

***David G. Belanger** is head of the Advanced Software Technology Department at AT&T Bell Laboratories, Murray Hill, New Jersey. He is responsible for research and transfer of technology to improve productivity and quality in software product development. The emphasis of his research is on robust, distributed system architectures—including database management and fault tolerance—and on software configuration management. He joined AT&T in 1979 with a B.S. in mathematics from Union College, Schenectady, New York; an M.S. and Ph.D. in mathematics from Case Western Reserve University, Cleveland, Ohio.*

***Eric E. Sumner, Jr.** is head of the Software Production Research Department at AT&T Bell Laboratories' Indian Hill Court facility in Naperville, Illinois. His department is engaged in empirical research in software production, e.g., specification-based methods and software visualization. Mr. Sumner joined AT&T in 1984 with an A.B. in engineering and applied physics, and a Ph.D. in engineering sciences, both from Harvard University, Cambridge, Massachusetts.*

***Peter J. Weinberger** is director of the Software and Systems Research Center of AT&T Bell Laboratories' Murray Hill facility, New Jersey. His center's primary activity is research that will help make AT&T better at writing the software it needs for its products and services. Mr. Weinberger joined AT&T in 1976 with a B.S. in mathematics from Swarthmore College, Philadelphia, Pennsylvania, and a Ph.D. in mathematics from the University of California, Berkeley.*