SRI ARC Journal Number 29292

April 1977

A SOFTWARE ENGINEERING ENVIRONMENT

Kenneth E. Victor
Research Engineer

Augmentation Research Center
Stanford Research Institute
Menlo Park, California 94025

Proceedings of AIAA/NASA/IEEE/ACM
Computers in Aerospace Conference,
Los Angeles, CA, October 31-November 2, 1977, pp. 399-403

SRI ARC Journal Number 29292
April 1977

A Software Engineering Environment

Kenneth E. Victor
Research Engineer

Augmentation Research Center
Stanford Research Institute
Menlo Park, California 94025

Proceedings of AIAA/NASA/IEEE/ACM Computers in Aerospace Conference, Los Angeles, CA, October 31-November 2, 1977, pp. 399-403.

Introduction 1a

This note describes a novel approach to improving the productivity of people who design, develop, and maintain computer software. The productivity of these people is important because overruns in cost and delivery time for software are common, quality is low, and maintenance cost is often greater than that of the original development. Barry Boehm, at the 1973 Tri-Service Symposium on the High Cost of Software held in Monterey, California, estimated the 1972 Air Force software costs to have been between $1 billion and $1.5 billion, while hardware costs were in the range of $Ø.3 to $Ø.4 billion. He predicted the ratio of software to hardware costs will increase until it reaches a level of about 9 to 1 in the 198Øs because hardware costs will continue to go down while the costs of personnel increase. 1a1

We believe that by making more and better tools easily available to software engineers, by making the tools consistent with each other, and by developing better techniques and methodologies, we can reduce the costs associated with the software life cycle and improve the quality of the products. This note describes a method for achieving these goals: by building upon the Augmentation Research Center's NLS system, by defining and implementing a few new tools, and by providing consistent access to already existing tools. 1a2

A number of tools and techniques are currently available to assist in the various phases of the software life cycle. We use the term 'tool' to refer to a software system or program that helps someone develop other systems or programs, much as a hammer is a tool used to build a house. We use the terms 'techniques' and 'methods' interchangably to refer to the manner in which the tools are applied. The term 'methodology' is used to refer to a systematic use of a collection of tools and techniques. Finally, we use the term 'environment' to refer not only to the collection of software tools, techniques, and methodologies available to a software engineer, but also to the people with whom the software engineer interacts, and the hardware tools and systems available (such as computers and terminals). 1a3

Background 1b

The software life cycle, as defined by Boehm (in the December 1976 issue of the IEEE Transactions on Computers), consists of the following seven phases: 1b1

(1) System requirements 1b1a

(2) Software requirements 1b1b

(3) Preliminary design 1b1c

(4) Detailed design 1b1d

(5) Coding and debugging 1b1e

(6) Test and preoperations 1b1f

(7) Operation and maintenance. 1b1g

In addition to these distinct phases, there are threads that pervade the entire life cycle: the management and control of people, resources, and the cycle itself; the production and distribution of documentation; and the training of designers, developers, maintainers, and end users. 1b2

The later in the life cycle an error is detected, the more expensive it is to correct. Or conversely, the more effort that is expended in the frontend of the cycle (Phases 1-4), the less time and funding are necessary for implementation and integration (Phases 5 and 6) and for operation and maintenance (Phase 7). 1b3

A number of tools and techniques are already available to aid in all phases of the software life cycle: 1b4

* Formal languages exist for specifying system and software requirements, and tools are available to validate these requirements to ensure their completeness and correctness; 1b4a

* Formal languages exist for specifying the preliminary and detailed design of a program, and tools are available to prove the completeness and correctness of these designs; 1b4b

* High-level languages for implementation are abundant, and a number of other tools and techniques, such as the use of program support libraries and the use of programming teams, help the implementation process; and 1b4c

* A wide variety of debuggers are available, including systems for automatic testing of programs, automatic generation of data with which to test programs, symbolic program execution, and automatic correctness proving of programs. 1b4d

Until there exists one tool that will automatically generate a program given the end users' requirements (stated loosely and in a natural language), no single tool will be sufficient to guarantee correct, error-free, software to meet all the users' (changing) requirements and intents. Additionally, many existing tools have both practical and theoretical limits as to what each can accomplish. For example, many program proving tools are capable of proving only small to moderate length programs, and only when such programs are written in highly specialized languages. Tools that exercise all statements or control paths of a program, in addition to having many of the above practical limitations, are theoretically incapable of determining the absence of a necessary control path. 1b5

Thus, for the present, and the foreseeable future, we are left with the need for a number of tools and techniques. This collection is needed for each individual phase of the life cycle and for the threads that pervade the cycle itself. Current practices (for the most part) consist of the haphazard application of the tools known to local groups of programmers. What is needed is the widespread knowledge about existing tools, and a well-defined methodology for applying these for maximum payoff. 1b6

It is not sufficient merely to provide a wide spectrum of tools. The tools should be used. A tool might not be used for many reasons: it may not solve the problem it was designed to attack; it may create more problems than it solves; it may be too expensive to use; it may be too difficult to use and/or learn; it may be solving a problem that is not perceived by those who must use it (e.g., the imposition of a tool on programmers by a manager). Conversely, we may define the ideal tool as one that solves a need perceived by the users of the tool without creating any new problems and is cheap to use. The difficulties

associated with learning and using it should be commensurate with the perceived payoffs to the user. 1b7

If a multiplicity of tools is needed, then these tools certainly should complement each other. The results produced by a tool that assists in one phase of the cycle should be directly usable by other tools that assist in the same phase and by the tools that support the next phase. Ideally, the techniques and discipline would be the same for using all tools. Thus, knowing how to use one tool, would be almost sufficient for using all tools. The specifics of a new tool may be different but the discipline would be the same. This might be analagous to increasing one's vocabulary (a relatively easy task) rather than having to learn an entirely new language (a relatively more difficult task). 1b8

A large project consisting of a number of modules and being worked on by a number of individuals requires tools just to keep track of who is doing what to which module and where in the cycle each module currently is. Such information ought to be maintained in a data base associated with the project. The individual tools that support specific phases of the cycle could then interact with this common "project data base" to indicate where a module currently is in the life cycle. It is easy to imagine other tools that would then interact with this data base to extract information such as who is working on what module, what percentage of the system has been designed, what percentage of the system has had its module specifications verified, what percentage of the system has been coded, etc. 1b9

We propose to define, implement, and study a software engineering environment containing a collection of consistent and complimentary tools, technigues, methodologies, and supporting data bases. 1b10

Basic Approach 1c

The tasks required to provide a software engineering environment fall roughly into the following three areas: 1c1

1) Defining and implementing the environment; 1c1a

2) Providing the tools that will be used by the software engineers; and 1c1b

3) Providing consistent access to the above tools from the software engineers' environment, and ensuring that any results produced by one tool are useable by other tools, including the project data base maintenance tool. 1c1c

We intend to build upon the NLS system (see below) to provide the framework for a software engineering environment. NLS was designed and implemented with its own evolution as one of its primary goals. This fact will make it possible to expand NLS to provide the framework for a software engineering environment. NLS already provides many of the capabilities that would be needed in such an environment and thus by using NLS we reduce the number of new tools that need to be developed. 1c2

We intend to define a project data base and design and implement the tools for the maintenance and management of project data bases. 1c3

Since many good tools already exist to assist in all phases of the software life cycle, we do not intend to redevelop such tools. Rather, we intend to adopt existing tools into our environment by using NLS as a frontend for these tools. This approach will ensure consistent access to these tools. A means will be defined for using NLS as a frontend system and for capturing the results of a tool for incorporation into the appropriate project data base. Ultimately, tools will be modified and designed to fit cleanly in the environment. 1c4

The NLS System 1d

For the past 15 years the Augmentation Research Center (ARC) at Stanford Research Institute (SRI) has been developing a large interactive system, NLS, to help people work with information. NLS is currently in wide use by secretaries, managers, editors, and a variety of researchers. We propose to use NLS as the basic framework within which to build the software engineers' environment. 1d1

NLS provides a wide spectrum of tools and techniques for the creation, viewing, editing, and dissemination of textual and pictorial information. At the user level, NLS is organized into a number of subsystems. Each subsystem contains a number of related commands. Although the semantics vary widely from subsystem to subsystem, the syntactic rules for all commands are the same, and the user interaction discipline is the same for all subsystems. 1d2

NLS is an extendable system. By writing new subsystems, individual users can extend the basic capabilities of NLS to meet the needs of specific goals. Those user generated extensions that have been found to be of general utility have been incorporated into basic NLS and thus made available to a wide number of users. In writing a new subsystem, the user takes advantage of a number of tools already part of NLS. The use of these tools ensures that the new subsystems will continue to provide a consistent and coherent user interface. 1d3

Internally, NLS is organized into a number of hierarchically structured modules. At the most basic organizational level, NLS is split into two distinct modules: a Frontend (FE) for all interaction with a user; and a Backend (BE) for actually performing the commands specified by the user in his/her interaction with the FE. Communication across the FE-BE interface is supported by protocol modules in both the FE and the BE. This organization allows the FE to be executing on a machine close to the user (to provide responsive command interaction and feedback) and allows the BE to be executing on a different (perhaps more powerful, perhaps geographically remote) machine. 1d4

The FE and BE are themselves composed of a number of functional modules. For example, the FE consists of a Command Language Interpreter (CLI) for parsing

user-specified commands, a virtual terminal handling module, the above-mentioned protocol module, an Operating System Interface (OSI) module for performing operating system and machine dependent operations, a physical terminal handling module, and other modules. The BE includes the above mentioned protocol module, an Operating System Interface (OSI) module, a file system module for handling the NLS hierarchical file system, a formatting module for formatting the information to be displayed to the user, a subsystem backend module for each supported subsystem, and other modules. 1d5

Most of these modules are themselves divided into a number of different levels. At the topmost levels are those routines and data structures that are available for use by other modules. These routines and data structures maintain a constant interface to the outside world. Underneath these levels are the necessary supporting data structures and routines. 1d6

This internal structure of NLS, i.e., the organization into a number of functional modules, with each module divided into a number of levels, has proved to be valuable for the maintenance and development of a large system -- NLS itself -- that is under constant evolution by a variety of individuals. It also provides a readily available wealth of existing code upon which users can build to create their own special purpose subsystems. The evolution of NLS has included the continual addition of new features and subsystems, more efficient implementation, and the transporting of NLS to a number of different computers and operating systems. 1d7

NLS Use by Software Engineers 1e

NLS is and has been used extensively by the software engineers at ARC. It is used for both its own maintenance and development as well as for the maintenance and development of other systems developed by ARC. In addition to the many features of NLS itself which support software engineers, ARC has developed a number of tools and techniques to assist in the software life cycle. Some of the capabilities and features of NLS, available to all users, and particularily useful for software engineers, include: 1e1

The consistent user interface. Having a consistent user

interface makes it possible, with minimal effort, to extend one's knowledge about the available system, making it possible to use more of the system itself. lela

The NLS hierarchical file system. The NLS file system naturally supports structured programming techniques. Additionally, the ability to apply "clipping" functions (to see only certain levels) and content searches to a file as it is being viewed makes it very easy to move around in the information space of programs and documentation, including design documents. lelb

The "partial copy" mechanism for editing files. With this mechanism, all edits actually change a separate file, the partial copy (much like marking up a transparent overlay). When a user is finishes editing, he can either discard the overlay, or he can incorporate the edits into the basic file. Only one person is allowed to be editing a file at a time. However, even though a file is being edited, other users may read the basic file. This approach, in addition to providing minimal loss of work across system crashes, successfully avoids the problems associated with multiple people trying to update the same file simultaneously. lelc

The "statement signature" feature. All NLS statements have associated with them a "statement signature". The signature consists of the time and date that the statement was last edited and an identifier of who last editted the statement. This feature is very useful when maintaining code to find out who was last responsible for a piece of code and when changes were made. Other NLS capabilities provide mechanisms for finding statements editted in a certain time frame and/or by a certain (group of) users. leld

The NLS JOURNAL subsystem. Under this recorded dialog system, documents can be distributed to a group of people. When a document is distributed it is assigned a "journal number". This number then becomes a permanent attribute of the document and documents can be retreived at any future time by using this number. Indices and catalogs are also generated automatically at the time the document is "journalized". These indices are generated by author, title, and keyword. Thus, there exists a data base for retreiving documents by content, author, etc. Frequently design documents are journalized. Code is sometimes journalized, thus providing snapshot frozen views of a software system. lele

In addition to the capabilities of NLS useful for all information workers, there are a number of facilities specifically for software engineers. These include: 1e2

A programming subsystem. This subsystem provides a coherent interface between the compilers and languages used at ARC and the NLS system. A semi-incremental compilation facility exists that allows dynamic compilation and link loading to take place at the procedure level, rather than having to compile an entire file and loading an entire mew system. This subsystem also provides an interface to an interactive debugger. This debugger is part of NLS, and is knowledgeable about many of the data structures supported by L1Ø, the implementation language for NLS. Once again, the user interface to the debugger follows the same discipline as all other NLS subsystems. It is thus a very natural tool to be used by thus who use NLS for all there work. 1e2a

An experimental language editor. This editor currently knows the accepted templates for system documentation and for the programming structures of L1Ø, the implementation language for NLS. By using these templates, newly written code is guaranteed to have the proper lexical format, and many syntax and typographical errors are avoided. It is easy to envision extensions of this capability so that an editor is knowledgeable not only of accepted templates, but also of the proper syntax and semantics of a language. Such an editor could easily be used as a training aid by programmers in learning a new language. 1e2b

An L1Ø programming support library subsystem. This subsystem provides for automatic compilation, indexing, and printing of source code and documentation files that have been changed since they were last compiled, indexed, etc. Audit trails are maintained by this subsystem so it is possible to determine who has been working on what and when. Additionally, if desired, this subsystem can produce a catalog of the procedures composing a system. This catalog contains not only procedure names, but information about the formal parameters for each procedure and automatically extracted documentation about each procedure. This catalog can (and is) used by other NLS commands for moving around the information space of an entire system that (frequently) consists of more than one source file. 1e2c

Other tools and techniques used by ARC software engineers

include: meta-compilers; an informal team approach to programming with informal walk throughs; standards for coding styles and for documenting changes to NLS; a feedback system for dealing with user suggestions and complaints; the exclusive use of high level languages. 1e3

Most software groups have their own sets of tools, techniques, and methodologies. What is unique about ARC is that it provides a systematic approach to the problems. New tools and techniques are not developed in a vacuum; rather, they are designed with the goal of being incorporated in the existing system. The payoffs have been obvious: tools are used readily (and thus costs reduced) because they fit in naturally with the other tools the users are accustomed to using and fulfill needs perceived by these users. 1e4

We propose to use NLS as the basic framework upon which to build a software engineers' environment. We need to examine and determine how NLS has to be expanded in order to enable better support of this environment. 1e5

Necessary Developments 1f

The tools and techniques developed and in use at ARC are heavily oriented towards the later phases of the software life cycle. Other groups (at SRI and elsewhere) have complementary (and competitive) tools and techniques, e.g., specification languages and program proving tools and techniques. We need to be able to provide consistent access to these tools and techniques. Such tools may at first be encapsulated under NLS to demonstrate both the feasibility and desireability of providing a coherent environment at the possible expense of efficiency. This would involve using NLS as a frontend for these tools. Later, existing tools may be re-written to mesh more closely with the environment, and we would expect that tools would be written explicitly for the environment. 1f1

Central to the concept of a software engineers' environment is the concept of the project data base to keep track of the status of the modules comprising a system. All tools in the environment should either participate in the generation of this data base or take advantage of the information in the base. Information is often easily available only at the time a tool (e.g., a compiler) is executed. Thus, tools should either interact directly with

this data base, or the act of encapsulating a tool should include capturing data known to the tool. 1f2

Conclusions 1g

A rich environment for software engineers can decrease the costs associated with the software life cycle, while at the same time providing higher quality products. We propose to provide the basic framework for such an environment by expanding and building upon the NLS environment. We propose to define a means for incorporating existing tools and techniques into this environment so that these tools can all be accessed in a consistent and coherent manner. We will initially encapsulate a few tools developed by groups other than ARC to demonstrate the feasibility and benefits of our approach. Finally, we propose to define the project data base, and the means for interfacing to this data base. We are not proposing to develop any new tools, other than those for the maintenance and manipulation of the project data base. We feel that there is already a wealth of existing tools, and that much can be gained by providing access to a wide variety of tools in one consistent environment. 1g1