1059

Bell Laboratories **Cover Sheet for Technical Memorandum**

The information contained herein is for the use of employees of Bell Laboratories and is not for publication. (See GEI 13.9-3)

Title- **The Programmer's Workbench - A Machine for Software Development.**

Date- **May 19, 1975**

TM- **75-9152-4**

Other Keywords- **computer configurations**
computer networks
software development
software maintenance
UNIX

Author	Location	Extension	Charging Case- 39373-090
Evan L. Ivie	**RR 4C-800**	**7969**	Filing Case- 39380-6

ABSTRACT

On almost all software development projects the assumption is made that the program development function will be done on the same machine on which the eventual system will run. It is only when this production machine is unavailable or when its programming environment is totally inadequate that alternatives are considered. In this paper it is suggested that there are many other situations where it would be advantageous to separate the program development and maintenance function onto a specialized computer which is dedicated to that purpose. Such a computer is here called a Programmer's Workbench. The four basic sections of the paper introduce the subject, outline the general concept, discuss areas where such an approach may prove beneficial, and describe an operational system utilizing this concept.

Pages Text 14	Other 0	Total 14	
No. Figures 4	No. Tables 0	No. Refs. 9	

Bell Laboratories

Subject: The Programmer's Workbench - A Machine for Software Development.
Case- 39373-090 -- File- 39380-6

date: May 19, 1975

from: Evan L. Ivie

TM: 75-9152-4

MEMORANDUM FOR FILE

1. Introduction

Perhaps the two most significant trends in the computer industry in the past fifteen years have been the marked decrease in hardware costs and the significant increase in human costs. Both of these trends have caused software development and maintenance costs to become a major factor in the total cost of a system. In fact, for some large projects, development and maintenance costs now far outweigh hardware and operational costs.

A number of steps have, of course, been taken to increase the ease and efficiency with which software systems are designed, built, and maintained. There has been a general shift toward the use of higher level languages. New techniques are beginning to be used such as structured programming and chief programmer teams. Some commercial packages are starting to be marketed (e.g. PRIDE[5], LCP[9]) that provide a methodology for software development. Also, the general level of experience and expertise of people in the profession has been increasing. All of these factors have had, and will have, an effect on our ability to produce software systems.

Unfortunately many software systems still seem to suffer from schedule slips, cost overruns, high bug rates, insufficient throughput, maintenance difficulties, etc. Evidently any progress that has been made, has been offset by such things as the increased complexities of the projects undertaken.

One might well ask why more progress hasn't been made? Although this is a very difficult question to answer fully, it may be appropriate to suggest a few possible reasons. First, the importance of the programming environment in the software development process is often either not properly recognized, or else is ignored because it is so difficult to properly evaluate. Take, for example, the machine selection process. Since hardware throughput is in some sense measurable, it is much easier to use that as the selection criterion than to try to evaluate the programming environment that comes with the machine. Thus, hardware and application considerations generally take precedence, and the programmer may be left with a set of development tools that is totally inadequate.

Even when some attention is paid to the importance of good programming tools, one is still faced with the fact that programming is as yet very much of a trial and error process. There is no accepted standard way of designing, building, and testing a software system. There are at this point only the beginnings of what might be called a programming methodology or discipline.

By trial and error and at considerable expense each project does, however, eventually develop a way of getting the job done. Then it's usually time to either upgrade the system to a new operating system or machine, or else to move on to a new project. And the programmer

is again faced with the job of developing a new set of tools and procedures because the old ones don't apply to the new application or run on the new machine. And finally he is faced with the loss of his tools at the very time he is trying to learn about the new project and the new machine, so that there is little time for adequate tool and procedure development.

What can be done to move the programming industry toward a more professional and stable approach to software development? Certainly education (courses, books[1], conferences, etc.) will play a part in the long run. And, of course, the development of new techniques in programming and program management will contribute if these techniques are accepted and put into use. More compatibility and standardization of hardware, operating systems, languages, and programming procedures would be of great value, but there is little hope that much will happen along these lines, except perhaps in the language area. Also, an increased investment in the development of programming tools and procedures must occur, but much of this will continue to be lost as computer hardware and operating systems evolve.

2. The Programmer's Workbench Concept

In this paper a very different approach to improving the development process is proposed. It is suggested that the programming community develop a program development "facility" (or facilities) much like those that have been developed for other professions (e.g. carpenter's workbench, dentist's office, engineer's lab).

Such an approach would help focus attention on the need for adequate tools and procedures, it would serve as a mechanism for integrating tools into a coordinated set, and it would tend to add stability to the programming environment by separating the tools from the product (the current approach is equivalent to a carpenter leaving his tools in each house he builds). In addition to these rather general advantages to a "Programmer's Workbench" there are a number of specific benefits to a dedicated program development machine, which will be discussed in Section 3. Before launching into a detailed discussion of the pros and cons, however, it would be appropriate to first define a little more clearly what is meant by a Programmer's Workbench.

Figure 1 shows the separation between the Workbench which performs the development and maintenance function and the host or target computer on which the production system will run. The link shown between the two machines represents a physical connection which is used to transfer data, run tests, etc.

Figure 1. Division of Functions Between Workbench and Host.

The idea of splitting off a well defined and cohesive function onto a separate dedicated computer is certainly not a new idea. Front-end computers for message concentration and line discipline control are in wide use, having proved to be not only economical, but beneficial for other reasons also[4]. Back-end computers for data base management are just beginning to impact the data base management field[2,3]. Specialized computers for the control of peripherals

(disks, terminals, photocomposers, etc.) have become commonplace. Indeed the availability of inexpensive microprocessors is certainly going to increase the pressure to go to networks of interacting computers with each processor performing certain specialized functions. The proposal that there should be computers which have been designed and configured to perform just the program development function is merely a further step in this direction.

2.1 Workbench Capabilities

The term, "Programmer", in Programmer's Workbench, should be taken in its most general sense and should not be restricted to the coding function. To emphasize this, the steps which go into the development and maintenance of a computer based system will be briefly outlined:

Step 1. Define what the application system is to do (e.g. system specification, functional description).

Step 2. Design the system. This is normally done at numerous levels of detail (e.g. system, subsystem, program, subroutine) and for various components of the system (e.g. the software, the hardware configuration, the user interface, the operating procedures, etc.).

Step 3. Implement the system through installation of the hardware, coding of the software, writing of users' manuals, etc.

Step 4. Test and evaluate the system. This occurs at numerous levels and for various components as in Step 2, and includes the integration of all of the pieces of the system into a working whole.

Step 5. Convert to the system and cut it over to live operation. This is generally a much neglected but significant part of the total job.

Step 6. Operate, support, and maintain the live system. This involves machine operation, user consultation, software bug fixing and enhancement, hardware maintenance and change, etc.

Some of the capabilities that the Programmer's Workbench should have to perform these steps are:

(1) Generation, modification, and production of specifications, manuals, catalogs, reports, and documents in general.

(2) Creation, editing, and control of programs and test data.

(3) Compilation, execution, and debugging of programs (either directly or through the host computer).

(4) System generation, integration, and installation.

(5) Regression testing and load testing of subsystems and of the total system.

(6) Analysis and reduction of test results.

(7) Tracking of changes to the system (e.g. trouble reports, enhancement requests).

(8) Evaluation and monitoring of system performance. Also, system modeling and simulation.

(9) Conversion of data files and the loading of the data base into the host for live operation.

(10) Production of lists, reports, and statistics for use by management in the control of each phase in the development and maintenance process.

3. Areas of Applicability of Workbench Concept

Four situations will now be described where a dedicated machine for program development would be of particular merit. Some arguments relating to why the Workbench might or might not be a good approach in general will then be presented.

3.1 Multi-vendor Installations

Many companies operate a variety of different computers. This diversity might occur across several manufacturers or it might be across several different models of a given manufacturer. For example, in the Business Informations Systems Programs (BISP) area of Bell Laboratories, where the Workbench is in use, there is an IBM 370/158 and 168, a UNIVAC 1108 and 1110, two XEROX SIGMA 5's, plus a number of minicomputers.

Initial efforts in BISP concentrated on development of a programming environment for each vendor line (IBM, UNIVAC, XEROX). Programming tools were implemented on two and sometimes three machines so that the tools would be available to all of the projects. An attempt was also made to keep these tools fully compatible across machines, but this met with only partial success.

Part of the motivation which led to the implementation of the Workbench was based on the realization that a much better set of tools could be provided for less money if only one set of these tools had to be developed. As a specific example, Figure 2 compares the approximate costs for developing and maintaining a specific programming tool on the Workbench versus doing it directly on two host machines. Development and maintenance costs for the Workbench version actually turned out to be less than half that of the dual implementation, due perhaps to the better programming environment. The savings would be even more significant if more than two hosts were included, or if more than one tool were considered.

	Workbench	Implementation on IBM and UNIVAC
Development (not including design)	$50,000	$120,000
Maintenance (per year)	$22,000	$56,000

Figure 2. Comparison of Development and Maintenance Costs for a Specific Programming Tool. (Includes both staff and computer time)

A second benefit of the Workbench approach is the uniform programming environment that is possible, even across projects which run on different computers and under different operating systems. Such a standard environment offers the following advantages:

(1) Training - It has been found that basically the same training courses on programming tools can be offered to programmers on, for example, both IBM and UNIVAC projects.

(2) Documentation - Only one set of users' manuals is needed for describing programming tools. (There is, however, a section in the Workbench User's Manual for project and host machine dependencies.)

(3) Standards- The development of standard policies, procedures, and methods relating to program development and maintenance are greatly simplified because one no longer has to take account of a multitude of environments every time a standards decision is made. The enforcement of standards is also simplified.

(4) Programmer Mobility and Retraining - Programmers can become productive much more quickly when they are transferred from one project to another.

In addition to the advantages offered to existing projects, the Workbench approach also offers substantial benefits to new projects. If a particular Workbench system has been accepted as the standard approach at an installation, and if all the programmers there are trained in its use, then a new project can bypass that lengthy "getting started" period during which tools and standards are developed, adopted, and learned. This should save not only money but also provide a quicker start, and thereby shorten schedules.

3.2 Installations in Transition

Even companies that make it a policy to use only one type of hardware find it necessary to periodically upgrade their equipment or to change operating systems. The transition to the new equipment or to the new operating system is usually a very painful period. If the installation were using the Workbench approach, it could make the transition a less painful process in several ways.

First, a change in the program development environment is no longer inextricably tied to a change in the production equipment. That is, changes in the two machines can be scheduled independently. Indeed, one would probably want to avoid a change in the Workbench at the time the production machine is changing so that all of the programmer's attention could be concentrated on the new equipment and so that he does not have to simultaneously worry about changes in his tools. This should shorten the transition period in bringing up the new equipment significantly. It should also eliminate that difficult "uncovered" period on a new machine when adequate tools are not available, and one has to make do with whatever happens to be provided by the vendor. Then, after the new production equipment is installed and operational one could consider upgrading the Workbench equipment and/or software.

In addition to being able to stagger changes in the production equipment with upheavals in the program development environment, the Workbench approach also allows one to independently decide upon the frequency with which changes in these two areas will occur. Indeed, if one has a good development environment on a Workbench, he might well decide to adopt a much less frequent change cycle, thereby providing more stability to standards, less retraining, smaller tool development costs, etc.

One also has more flexibility in upgrading the host (production) machine when it becomes obvious that such a move is advantageous. This is true since such a decision is no longer clouded by considerations of what it will do to the programmer's ability to develop code.

3.3 Projects in which Needs of Developer and User Conflict

One of the most critical decisions in the development of a computer-based application is the selection of the computer on which it is to run. In many cases a formal selection procedure is not undertaken because the decision is based on machine availability or other compelling factors. In those cases where an actual choice can be made, there generally is a conflict between the needs of the developer and the user.

If the selection is based strictly on the ability of the computer to perform the eventual application, then the software developer may have to survive in an environment that is poorly suited to his needs. If the selection is based entirely on the needs of the developer then the target system may perform its functions expensively if at all. The third possibility is that both the developer and the user compromise their needs in the machine selection process. This may, of course, leave neither very happy.

The development programmer is looking for a machine that has a powerful command language, sophisticated editing tools, flexible and easy-to-use file structure, terminal access (time shared), quality document production facilities, and good human engineering in general.

The end user is, on the other hand, more concerned with sufficient throughput and size, appropriate peripherals and equipment to support the application, hardware and software options to optimize certain critical features (e.g. access methods, block sizes, physical placement),/special needs in the areas of availability and reliability, and quality maintenance.

In addition to these potential functional conflicts, opposing needs can also manifest themselves because of a desire to utilize existing equipment or because of the experience and background of the programmers, the operators, or the terminal users.

All of these conflicts are based on the assumption that the development of the software for a project will be done on the same machine as the one on which the project will finally run. The Workbench approach helps to eliminate these built-in conflicts by providing an independent choice of the computer for the developer and for the user. No compromises are necessary and each can choose the machine best suited to his needs and experience. Here again not only the initial choice but the frequency with which a change is to be made is decoupled. This also eliminates possible down-stream conflicts and compromises.

3.4 Terminal Oriented Systems

One of the most time consuming and critical parts of software development is testing, especially total system testing. If the application being developed services terminals the testing is additionally complicated. It is very unsatisfactory to perform such testing by stationing people at terminals and having them type in data and examine output. Aside from the cost and frustration, it also is a non-repeatable and error prone approach. A reasonable solution to this dilemma is to provide a canned scenario of user interactions that can be fed into the system in a timed manner. However, if the insertion of the input messages and the capturing of the output messages is done in the internal queues, then the total system is not really being tested (e.g. terminals, lines, controllers, line servicing software, etc.). To help circumvent this problem various "loop back" devices can be developed which allow one program operating within the computer to send out data which is returned as though it came from a terminal.

Such an approach provides more complete testing but still suffers from certain side effects. For example it may be difficult to isolate whether a failure occured because of an error in the test driver or in the application being tested. This is because of the various possible interactions that can occur between two systems operating simultaneously within the same computer. Also if the application operates on a dedicated computer, it is impossible to effectively load test it to determine its total capacity since the test driver is consuming a portion of the resources. Thus, there are special reasons why the system test part of the Workbench needs to be on a separate computer.

3.5 General Advantages

It's probably safe to assume that every installation will periodically upgrade to a new computer. It is also fair to say that user and developer needs always conflict to some extent. Thus, to a greater or lesser extent, the advantages ascribed to the Workbench approach in the preceding paragraphs apply to all installations.

Additionally there are potential economies which may accrue to the Workbench because of specialization. Applications computers are typically large, general purpose computers with complex operating systems having many options and features. The Workbench consists of a specialized set of functions running on a dedicated system. The hardware configuration for the Workbench should be much simpler, the operating system should be less cumbersome, and the actual tools should therefore be smaller and faster. Front-end and back-end computers have been found to be economically attractive for the same reasons.

Another general advantage to the Workbench is the fact that it encourages the development of machine independent programming tools. Each tool must now function for program-

mers developing code for a number of different vendor machines. There is no room for shortcuts which are dependent on the idiosyncrasies of a given machine (or a given project). One is thus forced into a more stable and generalized software development approach which should be more applicable to new machines.

It has already been noted that a separate program development facility will help focus attention on the importance of the programming environment in the software development process. It should also provide some stimulus for the integration of the programming tools operating on the Workbench into a coordinated set of interconnected functions.

3.6 Potential Disadvantages

Thus far a number of arguments favoring the Workbench concept have been presented. The other side of the ledger will now be examined. One disadvantage to the Workbench approach is that another machine is needed which costs money and which is one more link in the system that can fail. The cost may be counter-balanced by the fact that the host computer(s) may not have to be available quite as early in the development cycle, they may not have to be as big, and fewer may be needed. Having an extra link that can fail is a problem if the components cannot do useful work on a stand-alone basis (e.g. program editing and document production on the Workbench when the host is down) or if there is insufficient redundancy (e.g. other Workbenches to shift the load to when one fails).

In addition to the question of the actual purchase or rental cost, there may be other "costs" in having a second (Workbench) machine. For example, the Workbench machine may be manufactured by a vendor which is different from the host machine vendor. This will duplicate all the problems associated with contracts, maintenance procedures, operator training, system programming support, etc. It will also force the programmer to be aware of two machines as he works and not just one. However, if the Workbench has a good user interface it may actually be easier for the programmer to keep track of it and the operational aspects of the host than to use the host tools.

A second general problem relates to the fact that data and functions are now split between two machines. This may manifest itself in slower response to some requests for data and cpu processing because of transmission and queuing delays. It may also result in some duplication of data to avoid these delays. The best solution to this problem is a judicious choice of what data is stored on each machine and what processing is done on each machine. Also, the use of a high speed link will minimize the delays.

A third problem is the fact that machines use different character sets, number representations, etc. Thus, there is a conversion cost in shipping data back and forth between machines. (However, in the current implementation this amounts to a very small fraction of the total Workbench cpu load.)

A final problem may occur because a fixed part of one's computational power is dedicated to a given function (i.e. program development). Some flexibility is thereby lost in being able to balance the computational load. Potential imbalances can occur in any of the available resources: cpu, disk, printers, etc. Imbalances may also develop at certain points in the development cycle. For example, one may find that the host is being under-utilized during the initial phases of the project. How serious this problem is depends a great deal on the relative costs and sizes of Workbench and host machines. If the Workbench machines are small and constitute a small fraction of the total computational budget then adding or deleting a Workbench machine to match the total Workbench load may be adequate. Also, because of the link, some functions such as printing and disk storage can be shifted from one machine to the other to help balance the load.

4. Description of Current Workbench Implementation

In considering the potential benefits of, and the possibly difficulties with the Workbench approach a number of questions arise. For example, are there unforeseen problems in the implementation or in the use of a Workbench system which will outweigh all of the projected advantages? Can such a system be implemented in a reasonable amount of time and with a modest expenditure of resources? Will programmers be willing to try something new and give the approach an honest trial? Can such a system be assimilated into an ongoing software development organization? How many of the benefits will turn out to be real? How serious will the potential problems be?

The idea of the Workbench was first conceived in April 1973. The Business Information Systems Programs (BISP) area in Bell Laboratories appeared to be an ideal environment in which to try the idea because all of the conditions described in Section 3, for which the Workbench approach would be benificial, were present. Thus, the decision was made to try out the approach on an experimental basis. The first Workbench machine was installed in October 1973. Two additional machines have since been installed (Oct. 1974 and May 1975).

The machine currently being used as the Workbench machine is the Digital Equipment Corporation PDP 11/45. The decision to use PDP 11's was based mainly on the fact that the UNIX [7] time sharing system operated on the PDP 11. UNIX was developed at Bell Laboratories by Ken Thompson and Dennis Ritchie. It is an outgrowth of the MULTICS [6] system, but much simplified and streamlined. It offered an ideal base on which to build program development tools. One of the current Workbench configurations is shown in Figure 3. Monthly rental for such a system is in the neighborhood of $6,000.

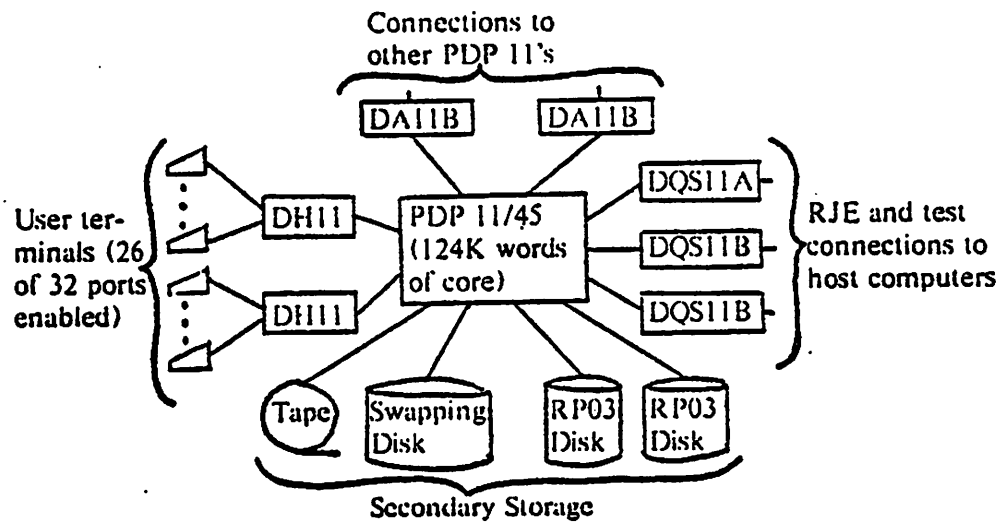

Figure 3. Configuration of one of the Workbench Computers.

Before discussing the components of the Workbench that are in operation a brief comment on implementation philosophy is in order. The idea of designing and building a complete and fully integrated Workbench system was rejected for a number of reasons, not the least of which is the fact that no one in the programming field knows what that system should look like at this point in time. Instead, every effort was made to identify the immediate needs of the BISP projects and to develop pieces of the Workbench that could satisfy those needs quickly. This approach provided the Workbench designers with much valuable user feedback quickly, and it allowed the projects to start picking up pieces of the Workbench to satisfy their

most critical needs immediately, and thus to start a phased transition to the complete Workbench.

4.1 Basic Components

It is not the purpose of this paper to fully describe the current Workbench implementation. Other papers are planned which will describe in detail each of the Workbench subsystems. However, a brief description of each of the five Workbench components currently in operation will be presented here to establish the fact that the basic Workbench concept has in fact been given a live shakedown with a useful subset of functions. The five basic components of the Workbench which were selected for initial implementation are job submission, module control, change management, document production, and test drivers.

1. **Job Submission:** The whole Workbench concept depends on having the capability to transfer data easily and quickly between the workbench and the host machines. The Bell Labs Workbench system currently provides this capability by operating as a remote job entry (RJE) station. Figure 4 shows the number and speed of the current links. There are also links for the test drivers.

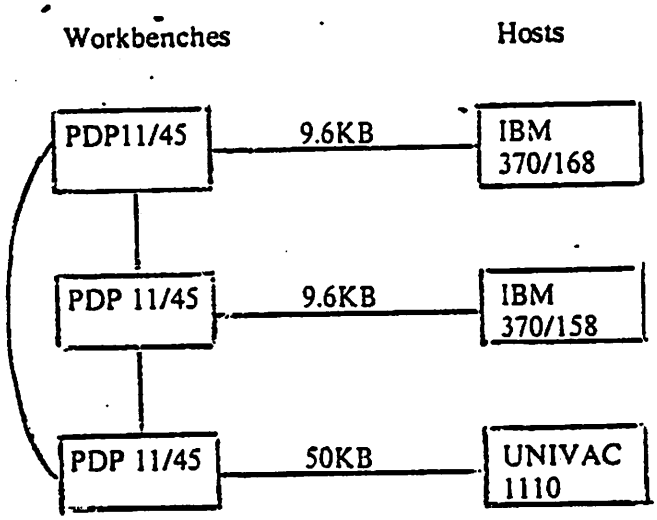

Figure 4. Current RJE Connections
(Inter-Workbench links provide access to a host not directly connected to a given Workbench.)

The RJE facility can be broken into four components:

(1) Job preparation- this step performs file concatenation (e.g. combining the job control language (JCL) file with the data file(s)), character conversion (ASCII to EBCDIC), and queuing.

(2) Transmission - this component empties the transmission queue, monitors the status of the communication line, and receives results from the hosts.

(3) Status reporting - facilities are provided so that Workbench users can determine the overall load on the host machine and the progress of each individual job. Each user can select what status he wants sent to him automatically. He can also initiate requests for status information.

(4) Post processing - the output from each run can be returned to the Workbench or selectively routed to a host printer or elsewhere. That which is returned to the Workbench is placed in the appropriate file for examination. Various scanning programs have been developed to help determine such things as the success of the run, etc. The RJE link does not, of course, provide direct interaction with a program executing on the host. This capability is part of the test driver.

2. **Module Control:** In the development of a software system, particularly a large system, each program goes through a number of revisions (releases, versions). In fact, at any one point in time there will probably be several revisions of a program in use simultaneously. The revision in use in the field may be different from the one in trial, which in turn may be different from the one in system test, which finally may be different from the one the programmer is working on. Also, there may be differences caused by different operating systems, data management systems, and user needs. Keeping track of all of these revisions is a major task. An even larger task is to make sure that the right modifications are applied to the right revisions and not to the others. The module control system developed for the Workbench provides:

(1) Creation of any revision of a program from any previous point in time.

(2) Protection against accidental tampering and change.

(3) Selective propagation of each change to a module to each of its revisions which should contain that change.

(4) Identification of object and source (revision number, date created, etc.).

In addition to programs, all sorts of other documentation on a project also go through many revisions. The Workbench module control system was generalized to handle not only source but any type of text. In fact, it is currently being used heavily for keeping track of the evolution of user manuals, test plans, error lists, etc. The module control system is described in [8].

3. **Change Management:** When a software system goes into production it becomes necessary to formalize the way changes are made to it. For a large software system, some type of formal change control is necessary fairly early in the development cycle and increases in importance as time progresses. The mechanism for change control is usually some type of trouble reporting form describing the reason why a change (in programs, hardware, or documents) is needed. To this form is added information as to who will make the change, what it consists of, and when it will be made. The Bell Labs Workbench currently provides a facility for entering trouble reports into a project data base and of subsequently editing and updating them. Facilities for generating summary, status, and other reports have also been developed.

4. **Document Production:** Having accurate, up-to-date, and understandable documentation (e.g. design specs, test plans, user manuals) is vital to the success of a project. On a large software project it is generally a larger and more difficult job to produce such documentation than it is to produce the source programs.

A wide variety of document production tools have been built for the UNIX time sharing system on which the Workbench operates: text editors, formatters, typographical error finders, etc. [7]. A phototypesetter which provides multiple character fonts and point sizes has also been connected to the system. A document writer has the option of having his text printed on a terminal, routed to a host machine for high speed printing, or sent to the phototypesetter for high quality output.

Over the past year a number of "macro packages" have been developed for the UNIX formatters. These packages automate the production of many of the standard document formats used by the BISP projects by producing headings, footings, labels, tables of contents, etc. Users can thus produce highly structured documents with minimal input effort. Typing pools, document production centers, programmers, and managers are all heavy users of this

UNIX/Workbench facility.

5. Test Drivers: Two test drivers have been implemented on the Workbench thus far. The IBM Test driver simulates IBM 3271 cluster controllers, each with several 3277 terminals. It serves as a driver to projects which use the data management systems operating on IBM/360 and IBM/370 computers. The IBM test driver is used for both load testing and regression testing. The UNIVAC test driver simulates a Teletype cluster controller with up to four terminals and is used for regression testing.

4.2 Possible Extensions

All of the above components are under continuing revision and improvement. In addition, efforts are now underway to integrate the current components into a more closely cooperating set of tools. For example, each change made to a module should be related to the trouble report(s) that initiated that change. This will allow one to supply a list of all the trouble reports that are to be resolved by a given release of a system and to have the Workbench automatically select which revisions of each program are needed in that release. Besides integration, several of the other functions noted in Section 2.1 are in various stages of planning. The system generation and configuration control area is one example. Also, efforts are now underway to connect the Workbench to the XEROX SIGMA 5 or 9 as a host computer.

In addition to these short term efforts, some long term objectives are being studied. For example, an attempt may be made to develop a uniform job control language (JCL) which can be translated into each of the host JCL languages. Also, a standard programming language might be designed which has code generators for each of the host machines. The value of this would be enhanced considerably if a set of Workbench system calls were devised that could be mapped into system calls to each of the host operating systems. These steps toward software portability are obviously very difficult, but might well be eventually achieved in an environment like that offered on the Workbench.

4.3 Workbench Usage

In addition to describing the capabilities currently offered on the Workbench it may also be of interest to describe the present usage of the system. As of May 1975 there were 225 authorized users of two Workbench machines (the third machine was just beginning operation). These users were dialed into the machines about 1500 hours a week for an average of 7 connect hours per user per week. There were about a half million lines of source under module control and about that much documentation on the system also. All of the twelve BISP projects were using at least parts of the system. Requests for use have been received that will double the usage of the system during the coming year.

The initial training session for using the Workbench takes three hours. Typically, a person can be submitting jobs, editing programs, etc. the first day that he starts on the system. After about a week he is able to operate without having to keep referring to the manual.

Workbench development costs to date (exclusive of UNIX) have been about ten staff years and $120K in machine (IBM, PDP, and UNIVAC) usage. To this should be added another ten staff years for user support and training.

4.4 Example of Use

Perhaps the most effective way to describe the current operation of the Workbench is to give a very simple example of how it might be used. Assume a trouble report has been written describing a bug in the "sum" program. After the programmer responsible for the maintenance of the program dials in the following interaction might take place:

(The characters typed by the programmer are in bold type. The UNIX prompt symbol is the "%".)

login Jones You have mail.	*The programmer identifies herself and is told she has mail.*
% mail From smith Fri Mar 14 12:48 1975 tr number: a-75-83-3 originated by: R. H. Johnson description: The value printed by the sum program is incorrect.	*She prints her mail and finds she has been assigned a trouble report.*
% get sum -r2 -e 2.4 129 lines	*She extracts the "sum" program at release 2 (r2) for editing (e).* *The get command responds with the release and level of the module (2.4) and its size.*
% ed sum 8463 **/put data/p** **put data (subtotal);**	*She invokes the editor to make the modification.* *The editor responds with the number of characters in the file.* *She locates the line containing, "put data", and prints it. Note that within the editor there is no prompting.* *(It is assumed at this point that the programmer realizes that she should have written "total".)*
s/subtotal/total/p **put data (total);** **w** 8460 **q**	*She changes the variable name from "subtotal" to "total".* *She writes the buffer and quits the editor.*
% send jcl sum 153 cards queued as /u3/hasp/xmit07	*She sends the module to the host machine for a compile and go run to test the fix. (jcl is a file containing the necessary job control statements and test data.)* *The send command responds with the total lines (cards) sent and the position of the job in the queue.*
15:40 /u3/jones/rje/prnt1 996.ldgo ready	*The rje system notifies the programmer that the listing from her run has been returned to the Workbench.*
% ed prnt1 26845 **/TOTAL/p**	*She scans the output listing for the total line and verifies the value.*

```
TOTAL=93,467
q

% delta sum
history? a-75-83-3 The output variable
    should be total, not subtotal.
128 unchanged
1 inserted
1 deleted
```

She makes the change (delta) a permanent part of the module's second release.

The delta program summarizes the amount of the change to the module.

5. Conclusions

The programming profession has yet to produce a software development methodology that is sufficiently general so that it can be transferred from one project to another and from one machine to another. The development of a Programmer's Workbench, a machine dedicated to the software development and maintenance function, can serve as a vehicle for the development of such a methodology.

By achieving application and machine independence, the Workbench approach can offer significant economic and other benefits to companies with several different machines and to companies that are in the process of installing a new computer. Since the Workbench concept allows an independent selection of the development and the production machines, the capabilities and environment available to both the development programmer and to the end user can be optimized. By running on a separate machine, the Workbench provides additional advantages to the testing function.

The Workbench concept has been partially implemented at Bell Laboratories. In open competition it has met with enthusiastic user acceptance. Both IBM and UNIVAC programmers now take advantage of the UNIX time sharing system and the same set of Workbench development tools. It is suggested that this initial success has, in fact, provided some evidence of the utility of the basic Workbench concept in at least the BISP environment. The extent to which this approach can be cost-effective at other installations and under other conditions is still an open question.

Acknowledgement. The concept of the Programmer's Workbench, its implementation, and its acceptance within Bell Laboratories are due to the efforts of the members of the Support Products and Systems Department. Much credit is also due to the members of the Computing Science Research Center who developed the UNIX time sharing system and the document production tools. Finally, special thanks go to the many users of the Programmer's Workbench who were willing to try something new and who have provided invaluable feedback and support.

Bibliography

1. Brooks, F. P. Jr., *The mythical man-month*, Addison-Wesley, 1975.

2. Canaday, R. H., Harrison, R. D., Ivie, E. L., Ryder, J. L., and Wehr, L. A., A back-end computer for data base management, *Comm ACM 17*, 10(Oct. 1974), 54-180.

3. Ivie, E. L., "A back-end computer for data base management", presentation at Data Base Management Session of 1973 National Computer Conference.

4. Feinroth, Y., Franceschini, Y., and Goldstein, M. Telecommunications using a front-end minicomputer, *Comm ACM 16*, 3(Mar. 1974), 53-160.

5. McMillan, W.E., Step-wide management controls, *Computers and People*, Apr. 1974, pp. 8-9,17.

6. Organick, E. I., *The MULTICS system: an examination of its structure*, MIT Press, 1972.

7. Ritchie, D. M. and Thompson, K. L., The UNIX time sharing system, *Comm ACM 17*, 7(July 1974), 365-375.

8. Rochkind, M. J., "The source code control system", submitted to the First National Conference on Software Engineering, Sep't. 11 and 12, 1975, Washington, D.C.

9. Warnier, J.-D., *Logical construction of programs*, H. E. Stenfert Kroese B. V., Leiden, Netherlands, 1974.

Evan L. Ivie

RR-9152-ELI-pw

Evan L. Ivie