A User's Viewpoint on the Programmer's Workbench

M. H. Bianchi
J. L. Wood

Bell Laboratories
Piscataway, New Jersey 08854

Keywords: Software development, programming aids, UNIX.

Abstract: The Programmer's Workbench boasts a broad set of highly useful features aimed at the application program developer. It claims to be a "human-end" computer providing tools and services to ease the load on the application system designer, programmer, documenter, tester, and delivery personnel. This paper shows the benefits of using the PWB tools, individually and in combination. Through specific examples drawn from the history of a software project, evidence is given that the use of the Programmer's Workbench can be a major contributing factor in the successful development of a software project.

1. RELATION OF CDS DEVELOPMENT GROUP TO PWB

The Circuit Design System (CDS) group was getting ready to write code at about the time that the Programmer's Workbench (PWB) was starting to accept customers. An informal arrangement was made to allow the authors to try out the new system. After about a month of discovering the tools that PWB offered, the arrangement was made official. For a while, CDS was the heaviest application development user on the Workbench, and hence we were the first to ask many questions and make many comments.

We have seen the Workbench grow and have been users for over two years. Unlike the developers of the PWB, we are only users. We will demonstrate through discussion of our experiences that the Programmer's Workbench *concept* is viable. Moreover, we will show that the actual PWB at Bell Laboratories is a most important contributor to the successful development of a project that ultimately runs in an unrelated environment.

Many of the PWB's facilities can be found on other systems in some form or other. From the user's viewpoint, the PWB provides an unusual variety of program development tools in a single, uniform, and easy to manage environment. This paper is *not* intended as a catalog of new or exotic facilities, but as a summary of one group's experience in using the tools provided.

We will be talking about PWB strictly from the viewpoint of a user who does not see, and is generally unconcerned with, the details of PWB implementation. The reader should be familiar with [DOL76A], which provides an overview of the PWB and a rationale for its existence. [RIT74A] describes UNIX, the time-sharing system on which the PWB is based. Finally, the discussions in [MAS76A] may improve the reader's understanding of some of the more complex examples presented here.

2. CDS—A QUICK OVERVIEW

The Circuit Design System mechanizes certain functions performed in the day-to-day activities of a Bell System operating telephone company. It uses other software systems written at Bell Laboratories to provide data base information, but its own emphasis is strongly in the engineering field. During the development cycle, the majority of our personnel were communications engineers and not data processing professionals.

The system itself must coexist with another system that utilizes IBM's Information Management System (IMS) to provide hierarchical data base management and transactional telecommunications [IBM75B].

The purpose of the first development cycle was to test the feasibility of the engineering process. Therefore, we had two secondary objectives. The first was to use the cheapest equipment possible, and the second was to minimize overall experimental costs. Our final product contained 196 PL/I program modules and 6 data bases accessed from dial-up terminals.

The developers themselves had to perform all the tasks involved in program maintenance. PWB allowed us to set up procedures that drastically reduced the amount of time required to maintain the system. In many instances, whole tasks, such as partitioned dataset compression, were made totally automatic.

3. ENVIRONMENT BEFORE PWB

The primary program development tools available at our location were Applied Data Research's LIBRARIAN [ADR73A] and IBM's standard utilities [IBM72A]. The source editing features of these programs do not lend themselves to making complex updates to modules. Moving blocks of code from one section of code to another is almost prohibitively difficult. Thus, as a program is modified, it becomes riddled with branches that have nothing to do with the implementation of the algorithm. Also, it is a batch system and an error in editing can ruin a half-day's work.

4. EARLY PWB TOOLS

The first version of PWB to which CDS was exposed was a DEC PDP-11/45 running UNIX plus a Remote Job Entry (RJE) capability. Many of the programs that were to enhance the PWB concept were still in development.

But the early support provided by this one system was of great value to us. We found it relatively easy to use, extremely reliable, and adaptable to many of our needs with little effort.

4.1 The Text Editor

At first, the major tool used was the UNIX text editor, *ed*, with its very terse syntax and surprising flexibility. Previous experience with the QED style of editor was a definite advantage in learning about *ed*. The first real work done with *ed* was to enter two small PL/I programs for use in the CDS project. These were thought out and entered at the terminal by the programmer. This early exercise convinced us that *ed* was going to be a valuable tool. We were impressed by the ease of editing and moving code around, the time saved by entering code directly into a computer rather than using coding forms and keypunches, and the ability to "desk check" while at the terminal.

4.2 The UNIX File System

UNIX presented us with a true tree-structured file system that allowed us to build logical relationships between its files and directories (leaves and nodes).

The CDS directory became the root of our "program tree" which we present in part in Figure 1. We built personal directories ("doug", "joe", etc.) and directories that were the repositories of related modules of source code. Programs that relate to CDS concepts are found in directories "af01", "cr01", and "ed01". Documentation is found in "doc". Files of Job Control Language are in "jcl". Test data is in "test". Directory "r3" contains directories of CDS Release 3 files. The ability to create

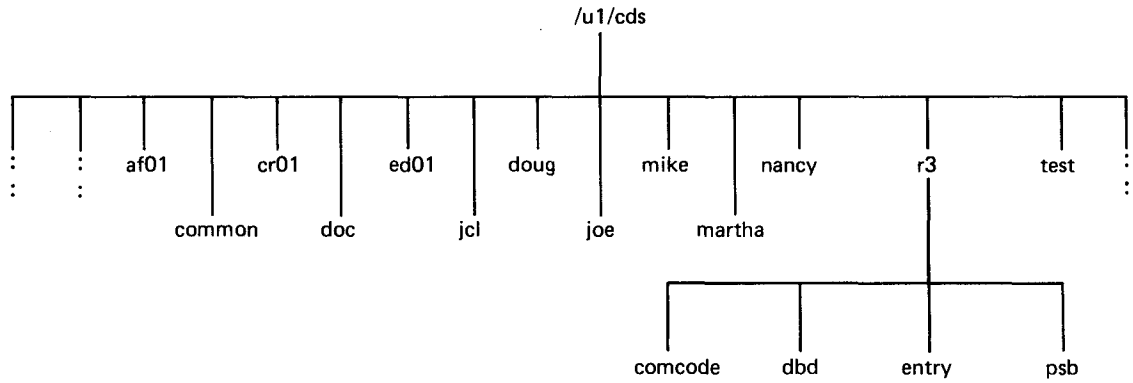

Figure 1. The CDS File System

meaningful collections of data files, reasonably named, under directories, also reasonably named, proved to be a major asset. It was now possible to produce quickly lists of all programs coded, those related to a particular concept, those which adhered to particular naming conventions, and those related to a particular concept that adhered to particular naming conventions.

In theory, this type of grouping and classifying of ideas or programs is possible on other computer systems through naming conventions. But PWB provided the tools that allowed one to go right into the file system and make the computer do the searching. By the time CDS consisted of 140 independently compiled procedures, this feature became invaluable. Our cross-reference listing procedure is shown in detail later.

4.3 Send—the RJE Program

The *send* command allowed us to communicate with the IBM target computer via Remote Job Entry. This is about all that the first version did. The command line

```
send jobcard plijcl source
```

would send the file "jobcard", followed by the file "plijcl", followed by "source". But *send* also assigned special meaning to the "tilde" (~) character. In particular, a line of the form:

```
~filename
```

read that file as the source of text, and a line of the form:

```
~~
```

read from the terminal, with a prompt of "input:", for the text. The "~~", in what came to be called "send-speak", was put to work. Rather than have the programmer enter all of the file names to be sent on the command line, we had *send* prompt for each input item, and lines like "~plixclg" (for PL/I Optimizer compile, linkage edit and go) and "~cdsed01" (for the source code for program CDSED01) were the responses.

We used the "~filename" form embedded in our code to implement the idea of common code, "comcode" for short. Our "comcode" was stored in a directory by that name. Each file contained one program concept. In the majority of cases these were PL/I DECLARE statements of structures that represented data base segments, input formats, and IMS control structures. There were 200 comcode items averaging eleven lines each by the time we were operational. Each module has an average of 4.2 "comcode" references for a saving of 46 lines per procedure.

There was another common directory called "entry" that contained DECLARE statements for each of the external procedures in CDS. By entering the following lines:

```
~/u1/cds/r3/comcode/orderno
~/u1/cds/r3/comcode/bodyin
~/u1/cds/r3/comcode/icferr
~/u1/cds/r3/entry/plitdli
~/u1/cds/r3/entry/prtdate
```

the programmer could reference 48 lines of code that declared the ORDERNO and BODYIN segments of our ICF (Incomplete Circuit File) data base, the code defining the ON CONDITION ICFERR that handled errors in calls to the ICF, and entry declarations for the external procedures PLITDLI and PRTDATE.

4.4 The Text Formatter

Documentation support was provided by the text formatter, *roff*. This program made it possible to sit at a terminal and enter a draft document directly, along with an occasional format command for paragraphs, headings, etc. When finished, one asked *roff* to print the document in formatted form. The raw document was entered and edited using the same text editor that was used for entering code, *ed*. Not only did this save having to learn a separate editor for use with the text formatter, but we could easily include sections of code in our documents, and vice versa, without having to retype them.

We found that the job performed by *roff* was acceptable for the documentation and day-to-day business of building a project. We started preparing program documents, such as explanations of CDS error messages, with this program and found it fairly easy to keep them current and available.

Some people question the "waste of time" of typing one's own documents. We feel that for anyone with a moderate amount of typing skill, it takes about the same amount of time to type as to write by hand. Many authors develop their documents at the terminal from a few notes. So there is no time lost, and the result is as good as or better than that from a typing pool.

Roff was even more helpful for large documents with several co-authors. Each author could have an up-to-date and readable copy of the entire document at all times. Our clerk/typist found it more rewarding to be able to correct errors or rearrange paragraphs without having to retype an entire page or "cut and paste," because all the time spent working on the document was productive.

5. RECENT PWB TOOLS

More recently, PWB has increased in potential and CDS has made use of that potential.

The *ed* program has not changed significantly in the last two years, but the few changes have increased the ease with which it can be used to do the more esoteric editing that the experienced user inevitably desires.

Send, however, has grown in capability to the point that it is a major tool in easing the "nuisance work" most programmers have to deal with.

Early modifications to *send* added the ability to establish "keywords" that would prompt the user, who would then respond with appropriate answers. These were substituted into "canned" Job Control Language files, creating custom JCL for the particular purpose at hand. The answers to the keyword prompts would also be displayed prominently in comments, so that if there was

a problem, it was not necessary to try and dig out their values. At the same time we made the prompts for file input request the appropriate type: JCL, PL/I source, run data, control cards, etc.

The major benefit achieved here was that it was *impossible* to forget a substitution because you were *always* asked. The result was that we had engineers, technical assistants, and clerks who, by learning the correct responses, repeatedly sent jobs to do testing without ever seeing a JCL statement. Two JCL "gurus" managed everything; JCL syntax errors became so infrequent as to be curiosities. Care was taken to make the prompts for both key words and file input consistent with the *intent* of the job, and not the details of the JCL or IBM file setup. The people thought of *what* they wanted to do; PWB performed the actual work of implementing those intents. By use of these prompts, our non-JCL oriented users could have great flexibility in sending their jobs and still not worry about the details.

Our early JCL files emulated many of the features normally provided by catalogued procedures. However, as our experience increased, we found that we were doing things with our JCL files that were not easily accomplished in catalogued procedures. In particular, one keyword could be used to specify fields anywhere on the JCL statements and in the source text. Also, we could write JCL which referenced other JCL files, thus avoiding duplication and easing maintenance.

5.1 The Source Code Control System

Over the years there have been attempts to provide a means to store, control, and document code as it is being developed. In almost every case these systems incorporate the means of editing the code. The PWB Source Code Control System (SCCS) does not [ROC75A]. The programmer requests a copy of the code for editing and SCCS locks out any other edit requests. The programmer then edits the source, which is an ordinary text file, by whatever means available, usually *ed*. When convinced that the new form is what is desired, the user asks SCCS to record the changes and unlock the master file to other editing. SCCS also records a statement of why those changes were made.

SCCS puts no restriction on what the text is or how it is generated and edited. Thus, when CDS started using the SCCS, our programmers had only to learn the initial request, *get*, and the final request to record the changes, *delta*.

5.2 NROFF—the New Text Formatter

Nroff, the new *roff*, has actually been available the entire time CDS has been on the Workbench. However, its much greater power was gained at the cost of syntax and features which are difficult to learn. Recently, the availability of a comprehensive set of "macros" for doing documentation has made *nroff* as easy to use as *roff* [MAS76B]. All of the late CDS documentation was done using *nroff*, giving superior document appearance and content in considerably less time than was previously possible.

6. HOW CDS USES PWB

PWB consists of many different processors, some of which perform quite primitive functions. It is the user's responsibility to put these programs together in imaginative and useful ways. It takes a while to get used to the idea that most of the work to accomplish a particular task has been done for you and that your work consists mostly of piecing together the features and little programs you need to produce the desired effect.

For example, when we want to produce a sorted list, we call on *sort* to do it. We do not know or care how it gets its job done or how much machine core or time it takes. We just call it with arguments that indicate what is to be sorted and by what rules. It then goes and does it. The same goes for printing, editing, file searching, string searching, etc. The little programs do some little thing in a reliable and flexible way. We piece them together to do what we want.

Thus the Workbench becomes a human oriented computer system. We spend our time working on what we want to do and how, but on a very high level. The implementation details are not our concern.

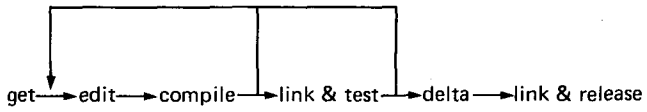

Figure 2. The Program Development Sequence

6.1 An Example of PWB Working for CDS

To show just how much work PWB does and how easy it is to get it to do that work, we will trace through a basic terminal session to change a program module, test it via compiles and runs on the target computer, and then make the changes official and permanent. The sequence is shown in Figure 2.

The first step in our example terminal session is to retrieve a copy of the original program module and to have SCCS restrict access to that module to non-editing only. The entered command line is:

```
get –e /u1/cds/cr01/s.ccanal
3.1
106 lines
```

That *get*s the SCCS source module "s.ccanal" for editing. The computer's response is printed in **bold**. A file "ccanal" is created for the programmer. SCCS tells us that the current release and level of the module are three and one, respectively, and that the created file has 106 lines.

We will not show the editing process which is fairly standard. Suffice it to say that there are no special considerations that the programmer must make for SCCS while editing the module.

send ../jcl/mhb	a job card file
CLASS=b	"B" class job (sets core and time limits)
JCL:	request for job control cards
~../jcl/plixc	file for basic compile
RELEASE=3	release of "comcode" to be used
pl/i source:	request for code to be compiled
~ccanal	reference to file to be compiled
pl/i source:	request for further code to be compiled
~.	conclusion of "plixc"
JCL:	request for further job control cards
~.	conclusion of "JCL" prompt and job stream
125 cards.	user information from send

Queued as /u1/hasp/xmit120.

Figure 3. User Conversation to Compile for Error Messages

When the programmer wishes to compile this program to check for compiler messages, the "conversation" in Figure 3 is held with PWB. What the user does *not* see is nine lines of JCL which include customized comments to help identify this job and 116 line of source code.

The "JCL:" and "pl/i source:" prompts are nested and repeated, allowing multiple compiles in one step and/or multiple steps in a job. The "~." discontinues the current level of nesting.

6.1.1 Compile for Testing. The edit and "plixc" cycle is repeated until the program compiles cleanly. The next step is to put it out where it can be run in a test environment. This is done with the conversation shown in Figure 4. We use the same job card and the prompts have the same meaning. The file "../jcl/compile_test" contains JCL to do a compile and linkage edit into a target machine library, R9411.CDS.R3.TEST.

```
send jcl/mhb          a job card file
CLASS=d               "D" class job (sets core and time limits)
JCL:                  request for job control cards
~../jcl/compile_test  file to compile for testing
FILE=ccanal           the file to be compiled
PROC=ccanal           the name it will have when released
RELEASE=3             the release it is to be tested in
JCL:                  more JCL requested
~../jcl/testlib       file to linkage edit into the test library
DIRECTORY=cr01        the directory of the main procedure
MAIN=cdscr01          the main procedure name
RELEASE=3             to be tested at this release
JCL:                  more JCL requested
~../jcl/btsbatch      execute the test library
d class job           a reminder that this must be run class "D"
RELEASE=3             the release of the test library to be used
BTS input:            request for input data
~../data/cr01_test    input data file
BTS input:            request for input data
~.                    conclude test data input
JCL:                  more JCL requested
~.                    conclude jobstream
406 cards             user information from send
Queued as /u1/hasp/xmit590.
```

Figure 4. Compile and Linkage Edit for Testing

The linkage editor has the ability to identify the load module with a "stamp" and to provide aliases for the module. In all source files, as a convention, there appear lines of the form:

```
~!echo " identify *ccanal1('%R%.%L% %D% %T%')" > ccanal_i
~!echo " alias  ccnext,cccompi" >> ccanal_i
```

The portions within double quotes are linkage edit control cards. These lines, when read by *send*, cause a file "ccanal_i" to be created containing the card images. Then "compile_test" reads that file at the appropriate moment, and the cards become part of the job stream. The "ccanal_i" file is removed later. The "%R%", "%L%", "%D%", and "%T%" are used later for SCCS release, level, date and time. Since we are not using SCCS here, but are only compiling for test, they will not be changed and their presence in the module "stamp" signals that this module is an unofficial version.

The "../jcl/testlib" file also contains a linkage edit step. It links the main module with all of its supportive modules to create an executable load module in the target machine's dataset called "R9411.CDS.R3.TESTLIB". We also stamp it with the date and time the job originated.

Finally, we add the file in "../jcl/btsbatch" to the jobstream which exercises the test library with the data in "../data/cr01_test".

Both "compile_test" and "testlib" reference the shell procedure "auto_compress". This eight line "program" keeps count of how many updates have been sent to the partitioned datasets used by the linkage edit steps. When the count gets to a certain number, currently eight, a job is kicked off that compresses that particular dataset. Since we have begun using this technique we have not had to worry about the problem of doing "garbage collection" on our partitioned datasets. Thus PWB is overcoming a deficiency of the target computer and relieving us of work we really should not have to do. The lines listed below are from "compile_test". They reference the "identify" file (~PROC_i), remove it, and execute the automatic compress. The same type of "send-speak" appears in "testlib".

```
//lked.sysin    dd *
~PROC_i
~!rm -f PROC_i > /dev/null
~!auto_compress test RELEASE; exit 0
```

```
send jcl/cds          job card for administrator
CLASS=b
JCL:
~jcl/compile
DIRECTORY=cr01
PROC=ccanal
RELEASE=3
3.2                   new release and level from SCCS
114 lines             new line count from SCCS
JCL:
~jcl/pgmlib           linkage edit into official program library
DIRECTORY=cr01
MAIN=cdscr01
RELEASE=3
JCL:
~.
238 cards.
Queued as /u1/hasp/xmit088.
```

Figure 5. Final Compile and Linkage Edit for Release

6.1.2 Compile for Release. When the testing is completed, the module must be made "official". We use *delta* to provide protection and keep a history of the changes. The user would type the SCCS command line:

```
delta ../cr01/s.ccanal
history? Change the choice code analysis – tr 76061–1
87 unchanged
27 inserted
19 deleted
```

The final steps are to compile the *delta*ed module and to link it into the "official" executable library. This is done by the CDS program administrator, who sends the job stream shown in Figure 5. The information from SCCS appears because the "jcl/compile" file contains a *get* (without edit) to obtain the PL/I source for send. The command line within that file is:

```
~!get /u1/cds/DIRECTORY/s.PROC –rRELEASE –l –p
```

The "–l" asks for the complete history of this file to be put in a file called "l.PROC" (in this case, "l.ccanal") and the "–p" causes the output of *get* to go directly to *send* without using any intervening file.

The %R%, %L%, etc. we saw earlier now come into use. SCCS changes %R% to the release number, %L% to the level, and %D% and %T% to the date and time the *get* was done. This information is scattered through the source code on comments to help the user, and appears on the identify card we saw earlier in the "PROC_i" file, ccanal_i:

```
identify *ccanal1('3.2  76/03/13 17:39:09')
alias  ccnext,cccompi
```

So now our load module is stamped with the release, level, date, and time of our module. We now can easily determine if a particular version is up to date.

The "jcl/compile" has an extra step in it to print the history that SCCS provided as part of the compile listing. Since that history tells when and who did what to this module, the listing produced is a complete document of this module to date. That's a handy thing to have, especially if people get into the habit of giving reasonable histories to *delta*. Histories that say "debug" are not all that useful.

The "jcl/pgmlib" file is much like the "jcl/testlib" we saw earlier. However, the read permission is resticted to the CDS program administrator so that only that person is able to send it. It also has an extra step after the linkage edit to create a listing of all the "identify" stamps that we put on our load modules. Thus the linkage edit listing includes a complete list of all CDS modules in the executable module, including their SCCS release,

level, and the date and time they were retrieved from SCCS, which was a useful thing to have when we were not sure what version was last compiled.

Again, "jcl/compile" and "jcl/pgmlib" both use the "auto_compress" to keep things on the target machine clean.

We would point out that most of the automatic processes we show would have to be performed by hand or by writing special programs if we were to use the facilities available on the target computer. PWB has totally relieved us of the drudgery of the manual process and even hidden the work being done.

6.2 Using PWB To Analyze Output

The RJE process permits the returning of output to PWB instead of having it printed. The "big file scanner", *bfs*, is used to scan large files. The authors have used this to some advantage. We have a *bfs* script that searches PL/I compiler output for the significant diagnostics, linkage editor complaints, and the printout from the actual run. When the project goes into "panic mode," this is very useful for first compiles and test case drivers. We also use it on a casual basis during the normal work day. Being able to look at twenty lines of significant diagnostics rather than twenty pages of output is convenient and we still can go back for details.

7. IS PWB WORTH IT?

That is the real point of all this: is it really worth learning another system to gain the benefits of PWB?

7.1 Productivity

PWB increases programmer productivity in a number of significant ways.

7.1.1 Fewer Steps in Coding. Generally, one step is eliminated in transferring an idea into the code of a program. The usual sequence of events for a batch card-oriented system is:

- Rough draft the idea into a flowchart, or some code or shorthand form.
- Expand into code on a coding form, hand written.
- Keypunch the code (either by the author or by a keypunch service).
- Wrap the code in a JCL deck and take to the computer center to be compiled.

In CDS we have observed that the second step is frequently skipped. The programmer arrives at the terminal with a rough draft of what is intended and refines it while entering it via the editor. So the terminal serves the purpose of the coding form.

Obviously, there is no wait for the keypunch service nor are any physical cards generated. And finally there is no need to go to the computer counter to push the deck across. The *send* command does that for you.

When that first compile comes back with its inevitable diagnostic messages, the next savings are realized. The programmer can log into UNIX and directly add that missing comma, include the forgotten argument, or move a misplaced statement. There is no need to write up another coding sheet, or duplicate cards, or shuffle cards. Just log in, correct, and *send* again.

Thus, in terms of the productivity to be gained through reductions in duplicated effort and trips to the computer center, PWB provides significant enhancements relative to the card-oriented environment.

7.1.2 "Automatic" Documentation and File Maintenance. Since the Source Code Control System keeps both the code and the history of updates, and since the Job Control files are flexible and yet always consistent, the Workbench performs all the functions normally assigned to a Program Librarian. We have a "daily daemon" which runs every weekday at 5 am. It "mails" reports to programmers on files added and removed since the previous working day. If something disappears, either through a system failure (which is rare), or a programmer error (much more common), we usually know about it within 48 hours and can get it backed up. On Fridays, it sends a job to produce a usage report on all of our IBM datasets so that we can stay ahead of our requirements, rather than reacting to crises. Before we built the "auto_compress" discussed in Section 6.1.1, the daily daemon also sent a job to compress all of our partitioned datasets.

In terms of the productivity gained by automating the work associated with staying ahead of the demands that CDS was making on the target machine, PWB was again very helpful.

7.1.3 Non-Programer Productivity. As we mentioned earlier, there was a fair amount of telephone engineering being done in the CDS project. The people who were involved did not know, and did not wish to learn, the various intricacies of JCL and IMS which the programmers live with. By using the Workbench as a filter, they were not forced to learn these extraneous systems, and thus could concentrate on designing CDS.

The programmers and those of us providing the JCL and IMS support also found the layer of filtering helped us concentrate on getting the programs working. Far less time was spent chasing down JCL syntax errors, recovering from dropped or misplaced decks, or counseling people on how to read cryptic messages.

True, there was a price to pay: learning enough about the key features of the PWB to make it work for us. But relative to the extraneous education we avoided for eight of our people, the time spent learning about UNIX was not very significant.

7.2 Better Code?

It is one thing to do a job faster. Does PWB help to do it better?

7.2.1 Program Style and Structure. The popular concepts of program structure and style are much touted in the literature, but we suspect most software shops are finding them difficult to implement. It is just a pain to have to re-code working code, "just to make it look pretty". We found that the use of PWB helped and even encouraged our programmers to write new code using the "good style" concepts. It was also possible to take existing code and "structure" it without changing a single character of actual code. One simply spaced it out, indented, and blocked as required. That did not change anything as far as the final compiled machine code was concerned, but made it *much* easier to maintain.

The ability to plagiarize well written code and modify it just a little bit was quickly discovered. This technique was used when a piece of common code could not be constructed for a particular purpose. A generalized solution to the problem would be made available and each programmer would adapt it as needed.

7.2.2 Sharing Code. The "comcode" idea mentioned earlier provided consistency in naming and usage of CDS concepts everywhere they appeared, and they appeared everywhere. Programmers working on opposite ends of the system had no trouble talking about data concepts that they had to share since they shared the same "comcode" for those concepts. The first time that inter-program communication was attempted via our data base, it worked! Also, since our "comcode" items were liberally and intelligently commented, every program that used them benefited.

Just as important was the effort saved. On the average, the programmer saved ten lines of coding every time a comcode was referenced. Thus, the naming conventions were easy to enforce since it was easier to use them than not.

7.2.3 Keeping Names Meaningful. Many more times than once in CDS we were faced with this problem: This variable no longer means what its name implies. It should really be changed to be more meaningful. "But it is used all over the place! How can you be sure you've gotten every occurrence?" Normally that is

a sticky problem. It is even worse when it is not just the name, but the nature of the data it represents that changes.

With the PWB we could, and did, find every occurrence of a variable and change its name and nature, without causing the usual catastrophies. To change the size and structure of the root segment of our principal data base in 100 independent PL/I procedures took one evening's worth of work for the authors. We were up and running the next morning. As we recall, there were three bugs associated with that change, all due to an oversight on our part, and they were all found and cleared by the end of the week. The code to accomplish this is shown in Section 7.3.3.

7.3 Because It's Only Impossible, We'll Give You Until Tomorrow

In any job there are those aspects which can best be described as a nuisance. And yet they take up time and effort and must be dealt with. Sometimes, they are extremely difficult to do, other times just plain tedious. It was pleasantly surprising to find that the Workbench could relieve the tedium and ease the difficulties.

7.3.1 How Many Statements in CDS? Programmers will never understand the fascination that the managers of software projects have with "statements of code." Normally when faced with the request for a count, programmers cringe, as did we. But when pressured we discovered that it was not all that bad. We found there were already programs for extracting lines containing particular characters (*grep*) and counting lines (*wc*) and we could pipe the output of one into the other. We built a shell procedure that looked at all of our code and counted semi-colons, the statement delimiter in PL/I. Thus the Workbench reduced a tedious job to a trivial one.

7.3.2 Where Did That Extra Character Come From? In any job, no matter how hard you try to avoid it, something that you did not anticipate takes almost as much time as planned activities. In CDS, the terminal we selected provided the "diversion." Simply stated, the relationships between the cassette tape drive and the data line to the computer were not as advertised. Characters which were recorded on the tape would not get to the computer. Characters which were not on the tape would be transmitted. Since these were non-graphic and control characters, we had some difficulty isolating the problem.

But UNIX came to our rescue. By turning off the special meaning that characters had to UNIX, that is by setting the input processor to "raw" mode which is within the user's power, we were able to record exactly what the terminal sent down the line in a file. The *od* program, octal dump, was then used to see exactly what was present. Through this technique we were able to determine the specific terminal deficiencies that were causing our problems. We then provided the manufacturer the specification for the correction which we required.[1]

This discussion brings up an important point. We have used several brands of printing and CRT terminals. In each case we could tailor the character set and response times from UNIX so that the terminal could keep up. If the terminal is fast, as in the CRT types, UNIX can be made faster to save time. It should be understood that we are not talking about changes to system code; we mean that the user enters a simple "set teletype" command (*stty*) to provide the features required.

7.3.3 Change the World. We talked earlier about finding and making massive changes reliably. That process is not trivial. It requires a thorough understanding of what needs to be done. However, given that understanding, PWB provides the tools to reliably search out and change what is needed. The keystone of those tools is the ability to create files of commands to drive the text editor. Moreover, it is a simple matter to make the files dynamic, changing according to need. All of this is done at the command language level, and is readily learned if one is willing to spend the time to read the programmer's manual and experiment a little.

For example, the need came up recently to change all occurrences of the character string "%M%" to the name of the file in which that string was found. There were 130 files spread around nine directories. The problem was to create the commands to *get* the file out of SCCS, edit it, and *delta* the result back in. The following was the heart of the solution.

```
if ! { get -e $1 } exit
gath -s III=$1 - ; delta $1 "-ychange %M% to proc name"
~$ed III.a > /dev/tty ; exit 0
~$f
~$g/%M%/s//III/gp
~$w
~$q
```

What it does is:

- When called with the file name as argument ($1), does a *get* of the file for editing (-e) and if that fails, exits.
- *Gath* establishes the string "III" as the file name and reads the lines that follow (-).
- The "~$" lines are read by *gath* and "III" is replaced with the file name. The resultant lines are given to the shell (*sh*) and thus the *ed* command performs the "f", "g", "w", and "q" commands which edit the file.
- The *delta* is made on the file with the given history.

A driver already existed for finding all files in the needed directories. It took about a half hour to develop and test the above code and about 50 minutes during off-hours for the PWB to perform the actual work. We've kept the skeleton around, since this type of thing comes up about once every other month.

7.3.4 Compile the World. In *send*, the keyword capability is not limited just to prompting. The few times it became necessary to compile everything we had we would satisfy the keywords in advance, using files of definitions or shell procedures, and then use the same JCL files as we did every day to do the work. The result was 100% consistency with day-to-day practice.

```
rm -f /u1/cds/xrefi
: loop
if $1x = x  goto wrapup
chdir /u1/cds/$1
get . -r3 -s
grep "~" ?[0-z]*  >> /u1/cds/xrefi
rm -f ?[0-z]*
shift
goto loop
: wrapup
chdir /u1/cds/r3/comcode
grep "~" *  >> /u1/cds/xrefi
chdir /u1/cds/r3/entry
grep "~" *  >> /u1/cds/xrefi
ed - /u1/cds/xrefi
                    1,$s/://
                    w
                    q
reform +16 +8 < /u1/cds/xrefi | sort +1 > /u1/cds/xrefs
rm /u1/cds/xrefi
mail jw
xref completed
```

Figure 6. Shell Procedure for Cross-Referencing Comcode

7.3.5 Cross-Reference the World. The Programmer's Workbench gave us the tools that allowed us to build a shell procedure to find all occurrences of the "tilde references" and tell us the

1. The final result of the investigation was that line-feed characters recorded on the tape cassette did not get sent to the computer and every block from the tape that was transmitted terminated with two carriage-return characters. IMS teleprocessing required the line-feed character and could not tolerate the extra unwanted carriage-returns.

program module they occurred in sorted by the order of the comcode files. Time from conception to tested procedure was one day. We list it in Figure 6 to show its size. Briefly, it:

- Removes the temporary file xrefi.
- From ": loop" to "goto loop" *gets* all programs in the directories specified. All the lines which have "~" in them are put into "xrefi" along with the file name.
- From ": wrapup" to "grep "~" * >> /u1/cds/xrefi" does the same for the "comcode" and "entry" directories.
- From "ed" to "reform" produces a formatted, sorted list of all the references and puts it into "xrefs".
- Remove the temporary file "xrefi".
- Mail to the login "jw" the message "xref completed".

It is a bit complicated. But this was created when we were still novices. A newer version, which produces identical output, is faster, more robust, and the user does not have to provide the list of directories to be used.

The "xref" has proven valuable time and time again, when a CDS system concept was changing and we wished to see what modules were impacted. More importantly, we do not feel it would have been feasible, with the development schedule we had, to build a similar system for use with LIBRARIAN or any of the other tools available on the IBM machine. Certainly, it would have been difficult to do it in one day.

8. DEFICIENCIES

8.1 Education

When the authors first started using the PWB we were told that our login code would be "cds" and were handed a copy of the UNIX Programmer's Manual. This eight part volume was all the education that was initially provided. It soon came to be known as "The Book", a phrase taken from the expression often heard in our office, "It's in The Book." Truly all the information required to properly use the system was in the book but in an incredibly terse format intended for reference only. The entire writeup on *ed* consists of four pages. A part of the problem is that in almost any software project, PWB included, the paperwork comes after getting the product out the door.

Recently the user community has expanded to such an extent that the PWB developers do not have enough time to answer all the user's questions individually. This has led to a series of useful tutorials, memoranda, and classes.

8.2 File System Management

As we have seen in our example, use of the full path name of a file aids in adding robustness to the code. The shell procedure or tilde reference will always work. However, the amount of space that exists in a file system such as "/u1" is limited and sometimes it becomes necessary to move a project to another file system. Currently, there is no support for such a move, and we were stuck with the job of finding all occurrences of "/u1" and changing them to "/u9". It was not all that difficult to do, but the PWB "super-user" (system administrator) could have done it for us in much less time.

In PWB all storage is on-line. Although daily backups are taken, there is no convenient way for users to archive old material. Source code files in SCCS format can have a lot of built-up fat. Old releases which are not currently in the field could be "crunched" out from the bottom if there were some way to store the historic copy offline.

8.3 Independent PWB versus On-Line

For most of the users of PWB, it is an interface to a larger target machine. This immediately brings up the objection that PWB is no better than being in a batch environment. There is no general capability for on-line compilation and testing. These functions might be gained if the target provided a time sharing system, such as IBM's Time Sharing Option (TSO).

But would it be worth while to build a PWB facility under TSO (or other target operating system)? Certainly the ability to do on-line compilation and testing would exist. This approach may be appropriate at some installations. The general issues involved in using an independent PWB are discussed at length in [DOL76A].

If CDS were developed on a TSO Workbench, we would be sharing the target machine with a large batch environment and an on-line IMS environment which is already having difficulty servicing its terminals adequately. With PWB serving a strictly on-line environment, and the vast majority of its customers using its editing, RJE, and documentation facilities, we feel it is doing a better than adequate job. From our particular experience with CDS, the lack of on-line compilation and testing seems more than compensated for by the other advantages of the PWB. Also, in our installation (as in others possessing finite resources), we have often seen contention for priority of use between batch and on-line applications, and we welcome PWB's ability to provide service without getting involved in such contention.

9. RELIABILITY

The reliability of the PWB system has been very good. In over two years of use, only one file of CDS source code was damaged. It was restored to the previous day's version an hour later.

File space availability is a recurring problem. Unlike other systems in which users have a fixed allocation, PWB shares the available space across a group of users. Some users are not as nice about staying under their "paper" allocations as others; about every six months a file system runs out of space.

Because our location currently has four PWB machines, if any one malfunctions files can be switched to an alternate machine giving a degree of "fail-soft." Thus even when a machine is broken the users have service, though admittedly degraded. Even so we have found that the system uptime is very good. It is generally available during working hours, although at about 2:00 pm access can be limited due to line congestion. Usually it will stay up throughout an entire weekend without any attendant.

10. CONCLUSIONS

We have shown a small sample of the experiences of one group using the PWB to aid in developing a real-life application.

Since we first joined, the Programmer's Workbench community has grown at an astonishing rate. It is being used by development projects and maintenance projects, documentation centers and typing pools, clerks, typists, programmers, engineers, and supervisors. It is probably safe to assume that no two use it in exactly the same way.

But there is little doubt that the availability and capabilities of the UNIX Programmer's Workbench are having major impact wherever it is used. It reliably provides useful computing power to a large and diverse community at a very low cost. Moreover, this power is available in the form most people want it: a human oriented system which is easier to use than not.

This type of software design, where the system does not drive the user, but rather the user easily drives the system, will, in our opinion, have great and favorable impact on professional and public acceptance of future computer technology.

REFERENCES

All references cited in this paper appear at the end of *"An Introduction to the Programmer's Workbench,"* by Dolotta, T. A., and Mashey, J. R., in these Proceedings.