A Software Engineering Environment (SEE) for Weapon System Software

H. G. STUEBING

Abstract—A software engineering environment (SEE) has been designed, developed, and used for the life-cycle support of weapon system software. This SEE consists of two types of facilities: software production and integration. The software production facility consists of a software system that runs on a commercial multicomputer configuration. The approach features increased management visibility of the software development process, increased programmer productivity through automation, reducing the cost-of-change during maintenance, and the use of automated regression testing to improve software quality. These facilities have been used for eight years to develop and maintain weapon system software for several projects. This paper describes accomplishments, refinements to the code and test functions, and a general approach to extend the capabilities into the requirements and design phases. Techniques are described that simultaneously allow different methodologies, programming languages, and target computers to be implemented on the same host computer. Also discussed is the implementation of a SEE in a distributed computer network.

I. INTRODUCTION

SOFTWARE engineering is concerned with developing software systems that satisfy the requirements of the user over the life of the system; a SEE assists the accomplishment of software engineering through sets of computer facilities, integrated software tools, and uniform engineering procedures. The term “weapon system software” inherently implies a concern with software for embedded computer systems and support over the entire life cycle.

A generic view of the weapon system software life-cycle phases is shown in Fig. 1. This figure emphasizes the view that weapon system software is redeveloped several times during maintenance, the time after the initial version is delivered. The development process has overlapping phases, each with a measurable input and output. The phases overlap, showing an interaction between them. Within each phase, a set of activities is defined to systematically achieve the goals; the functions of management, quality assurance, and configuration management are included as activities in each phase. There are iterations horizontally, between activities of a particular phase, and vertically, between phases. During initial system development the work progresses through all phases. During maintenance the point of reentry is determined by the scope of the intended change.

At the U.S. Naval Air Development Center (NADC), Warminster, PA, facilities have been constructed to assist software engineering for weapon system software. Two types of facilities were built: software production and integration. The integration facilities were built for each project and consist of laboratory hot-mockups of the embedded computers with realistic simulation of external inputs. The software production facility is an integrated software environment hosted on a large scale commercial multicomputer configuration. The host configuration consists of five Control Data Corporation (CDC) computers, a CYBER 175, CYBER 720, and three CYBER 760's. This software production facility is called FASP (facility for automated software production) and it is described in [1] and [2]. The conceptual and architectural ideas of FASP were strongly influenced by [3] and [4]. FASP became operational in July 1975 and was the first integrated environment to be used for weapon system software and among the first integrated environments. FASP supports the activities shown in Fig. 1 from mission requirements to code and test; however, only the code and test phase is supported by an integrated environment. In the earlier life-cycle activities the support is provided by loosely coupled sets of tools, an important distinction discussed later.

II. ACCOMPLISHMENTS

In this section the accomplishments of FASP are discussed; the reported period of operation is July 1975 through June 1983, or eight years. These accomplishments are given to set the context for the discussion that follows and to give encouragement to those who are contemplating establishing such facilities regardless of scale. In judging these accomplishments one must consider the methods that were used before the introduction of FASP. Generally, before July 1975 the software development was done on the target computer itself; sometimes the same target computer was used for development and integration. The state of support software and peripheral devices for these target computers was primitive compared to the state of commercial computers. Nevertheless, a large industrial-based work force had established a way of “doing business” with these facilities and produced large amounts of weapon system software. The software problems of those days are well documented.

A. Integrated Environment Hosted on Commercial Computers

The FASP experience has shown that the concept of an integrated environment hosted on large scale commercial computers can be used as a true production facility. True production means that the availability and performance are adequate

Manuscript received Sept. 7, 1982; revised July 25, 1983.

The author is with the Software and Computer Directorate, U.S. Naval Air Development Center, Warminster, PA 18974.

U.S. Government work not protected by U.S. Copyright.
Fig. 1. Generic system development process.

Fig. 2. Key FASP parameters.

to produce weapon system software. FASP was contractually specified as government-furnished-equipment with guaranteed performance; the usage was almost entirely contractor personnel located at remote sites. Fig. 2 shows some of the key parameters measured with FASP. The amounts of software shown are larger than the software delivered to the fleet since some projects keep several versions active at any given time. Also, the data refer to on-line software and do not include the amount of archived software.

This approach used a one-time development of support software that not only eliminated such tasks from the contracts but also used less in-house personnel than supporting separate facilities for each project.

B. Management Visibility and Control

FASP provided the dual functions of an advanced programming system and a management information system; this allowed management visibility into the software development process at a detailed level. Database controls allowed configuration management to be enforced from the beginning of projects, a welcome benefit. Also, FASP was a natural way to have work standards uniformly enforced over a group.
of projects. NADC was able to serve in various roles with regard to weapon system projects; these roles included system prime, validation and verification, and life-cycle support activity. FASP with its remote terminals could be used by contractors or other government laboratories regardless of their geographical location.

An advantage of developing software in FASP is that no host computer change was required when transferring the project to the maintenance phase. Furthermore, since the software was in a government operated facility with all management information, test data, and documentation in hand, the maintenance of the software could be competitively procured in a realistic way. This was a major change from the times of being captured to one vendor for the life cycle.

When software was not developed in an integrated environment it was found to be poorly organized and impossible to recreate without the original developers. The development of interface control documents proved invaluable when such software had to be transferred to FASP from another development facility. When the developed software did not follow the interface control documents the effort to transfer the software was sometimes large. Transition efforts took a few days when the interface standards were followed and varied from one-half to seven man-years when they were not.

IV. “TO THE SEE”

In [1] the productivity, measured in delivered source lines per man-month, was well over 400. Those data were a two-fold increase over published industry data for real-time embedded computer software. Those data were measured before significant interactive features were added to FASP; the productivity data are believed to be greater with interactive features. The data before FASP are sparse and were not consistently measured. There are local examples of turnaround time varying from one to several days with target computers being used for the development facilities. These times also do not include the courier travel time to and from the facility. With FASP the turnaround time is measured as viewed from the remote terminal. The FASP speed improvement and the complete elimination of courier travel time reduced turnaround time by a factor of 20.

With FASP we have shared large amounts of software between projects. The sharing is easy when both projects use the same facility; it is simply a matter of copying a database. Furthermore, the use of a common facility by a large group of people tends to result in better communication among the group as a natural byproduct.

A. Quality

The quality of the software produced by FASP is significantly better than that previously produced. Some of the reasons are better tools, enforcement of standards, comprehensive unit testing with software emulators, and improved management visibility into the software development process.

Consider the following example. NADC was tasked to do a verification and validation effort on weapon system software that was developed by a contractor in a separate facility. The software was scheduled to be delivered as functional increments, each adding to the previous delivery and giving additional functional features. Fig. 3 shows a plot of the delivered source lines (not counting comments). An interface control document defined the software formats of both FASP and the contractor facility. This software was quickly installed in FASP and subjected to many unit tests, using regression testing with path coverage analysis. As new deliveries were received, additional tests were added and run with all the previous tests. Fig. 4 shows the number of unresolved software errors (per 1000 lines) that was recorded after each delivery. At the end of the effort 89 percent of all paths were tested. Fig. 4 is significant for the following reasons.

First, before the delivery to NADC the contractor believed the software to be suitable for fleet use. However, the contractor had not used an integrated environment with regression testing or path coverage analysis to test the software.

Second, the resultant error data were a factor in the computation of the contract award fee. However, the errors were reported to the contractor while the development team was still in place, resulting in the correction of many of the errors.
Third, all errors corrections were made before fleet delivery. Clearly, this was more cost effective than waiting until the errors were reported from the fleet.

B. Technology Transfer

FASP has been used to support projects in several of the Naval System Commands for airborne, surface, and subsurface applications. This was a much broader scope of use than originally expected. This usage has all been on the central computer facilities at NADC.

The entire FASP software system was successfully transferred to a major aerospace corporation. The corporation plans to use it for all Navy software that it develops. Today, FASP is only portable to other CDC CYBER computers.

A version of FASP was rewritten using the UNIX operating system on the Digital Equipment Corporation VAX 11/780 computer system (UNIX is a trademark of Bell Laboratories). This version of FASP supports several popular microprocessors, components that are rapidly appearing in weapon systems. It is planned to bring the UNIX version to the same level as the CYBER version, forming a product that will be easily portable to many other users.

IV. “TO THE SEE”

A. The Software Problem

The hardware, the physical embedded computer resources of the weapon system, is generally considered less of a problem than weapon system software. The “software problem” has been covered extensively in the literature; however, different aspects of the problem have been emphasized over time.

In the early 1970’s there was great concern about the quality of weapon system software. The performance, reliability, and user friendliness were poor; most errors occurred during coding and remained undetected after testing and integration. The quality has significantly improved through better design methods and comprehensive testing; most errors are now traceable to erroneous requirements, not coding.

Today’s paramount issue about software is productivity, that is, the achievement of a true economic increase in productivity over the life cycle (see [5] and [6]). It is well known that software is a labor-intensive field and that the life-cycle costs are both high and rapidly increasing. For a given weapon system about 25 percent of the software life-cycle costs are for development, 75 percent for maintenance. Software productivity in an economic sense has only increased modestly when measured over the life cycle. Most of the available labor is devoted to maintenance; the amount is rapidly rising because more and more systems are being deployed. The demand for labor with software skills has exceeded the supply, a trend expected to continue through the 1980’s. As the balance of labor continues to shift to maintenance, less and less labor is available for development. Therefore, to reverse this trend, in the future it must be cheaper and faster not only to develop software but also to change it during maintenance.

A SEE consists of sets of computer facilities, integrated software tools, and procedures that support a weapon system over the life cycle. A SEE serves as a unifying element to assist software engineering and forms a basis for attacking the software problems of quality and productivity. Testing remains the primary method for assuring software quality; a SEE can provide many automated aids to minimize the labor required for testing. Productivity is improved with a SEE not only by automating the testing of software but also by aiding all steps in the development process, including making it easier to reuse large amounts of software.

Each phase of the life cycle employs different engineering methods. Within each phase there is usually a choice of several methods for each activity. The term “methodology” refers collectively to a selected set of engineering methods. Ideally, one methodology consisting of uniform methods would exist for the life cycle. The transitions between phases would be smooth as well as the transitions between activities of a particular phase. An ideal SEE would be highly integrated both horizontally and vertically. However, such an ideal state is some time in the future. Therefore, the issue of today is choosing a way to evolve toward the ideal SEE; it is a matter
of implementing what is practical while continuing research into improved methods.

A key to success is to create a framework where new tools and techniques can be continually superimposed on existing work activities in a nondisruptive manner. Furthermore, it is better to allow multitudinal methodologies, to the extent possible, than to attempt to select the one “true” methodology. For example, in the code and test phase instead of implementing only the Chief Programmer Team approach, it would be better to choose a way that allowed several methods, one being the Chief Programmer Team. The activities concerned with code, test, and integration are better understood than the remaining phases; therefore, they offer a natural starting point. There are many methods and techniques currently available for the requirements and design phases; thus, a loosely coupled collection of tools is more appropriate for those phases.

B. SEE Versus PSE (Programming Support Environment)

Several terms have appeared in the literature that are similar to “SEE.” They are programming environment, programming support environment, and software environment. These terms have been generally used to describe the code and test activities, although frequent reference is made to the requirements and design phases. Furthermore, these terms have usually been restricted to the software concerns of a system and not the system as a whole. Therefore, the distinction is that the term “SEE” is more general and includes the above terms. A SEE refers to the support over the life cycle including aspects other than purely software.

Of course, the term “environment” itself can be somewhat confusing in this context. This term is so general it is difficult to determine its limits in some texts. In this paper the term refers to the “work environment” for the phases of the weapon system life cycle. The emphasis is on the facilities that support the work of each phase and the interface between the engineer and the computer. It is recognized that organizational and social factors are an important part of the work environment. These factors must be considered in the design of any computer-based support system, but they are not the main points discussed in this paper. An important point is to recognize that the engineer’s detailed view and use of the support facilities is different depending on the phase of the life cycle. The needs are different between coding and unit testing and integration of the weapon system software with the embedded computer. Likewise, the detailed view of the software is different between “development” and maintenance. The term “meta-environment” has been used to describe the aspect of environments that depend on the user’s view of the system and the organizational and social setting.

In industrial engineering there has been considerable work on facilities, both in concept and implementation. There are many valuable observations in this field that can be applied in part to a SEE. However, there are also some important limitations. For example, in industrial engineering the production facilities are oriented to rapid and automated replication of physical devices, devices that have been previously designed. Thus, there is an area called CAM (computer-aided manufacturing) where the computer has been applied to the task of automating the production of physical devices. Separately, there is an area called CAD (computer-aided design) where the computer has been applied to assisting the designer, making, for example, integrated circuit layouts. In the software field the work of coding and testing bears a similarity to production in the industrial engineering sense. This was the main theme in the development of FASP. Now we attempt to extend these facilities into the requirements and design phases; it is important to note that the analogy must be to “CAD” and not to “CAM.” In these phases the facilities must support both the cognitive processes of the designer and the more clerical aspects of recording the results of the processes. Therefore, the theme of the SEE in the early phases is to assist the engineer during the cognitive processes and to automate the clerical aspects of recording information and generating documentation.

C. An Integrated System

The term “integrated system” is also frequently used in the literature. Now, the dictionary definition of “integrate” is clear: “to make whole or complete by adding or bringing together parts, to put or bring parts together into a whole; unify.” Thus, in creating an integrated system the designer would do tradeoffs between the parts to achieve a unified whole. A SEE is referred to an integrated system and it appears to the engineer as a unified whole that assists the accomplishment of software engineering. The users include both engineers and managers; the work activities vary over the life cycle, and the user interface and capabilities vary accordingly. Conceptually, the SEE may be viewed as a single entity that presents to the user different views and capabilities according to the phase of the life cycle. The implementation is most likely to be several computer-based facilities that have similar user interfaces but different specific capabilities depending on the phase of the life cycle.

The two types of users, engineers and managers, mean that their individual needs must be traded off such that the final system represents a unified whole. The software engineers need advanced programming capabilities for the code and test phase; the managers need relevant, consistent information and a means to control the cost and schedule of the effort. The software engineers need compilers, linkers, system generators, and simulators to accomplish their work; these components are called “tools.” The managers need a means of identifying the end-items, a way of monitoring progress, and a way of ensuring that the agreed-on procedures are being followed. The database concept is a natural way of collecting such management information; it also provides a way of meeting many functions of the engineer.

Clearly, if an engineer were given just a tool set, the work could be completed. If the tools were compatible with one another, forming an integrated set, the work would be easier to accomplish. However, since this tool set executes directly on the host operating system, the engineer is thus free to create any arbitrary set of files. If many engineers are working on the same project the chances of them all retaining the same type of information in their files is small. In such a case the manager has no easy way to determine what end-
items are being produced or what progress has been made. Of course, this hypothetical team could agree to follow a set of standards, but changes in personnel, deadlines, design changes, etc. would quickly destroy the good intentions. Also, the end-item productivity of this team is lower because a large part of their energy is diverted to writing the job control statements for the file system and other support programs.

The computer is the natural and convenient place to integrate the needs of the engineer and manager and provide an integrated system to accomplish the work. A SEE is an integrated system in the above sense; it is much more than a tool set; it is a unified system that meets the needs of both engineers and managers.

V. Code and Test

In this section the characteristics of an integrated system to support the code and test phase are briefly described; [1] contains a more detailed description. The system that is described is not specifically FASP, but one that has been generalized and refined based on eight years of operational experience with FASP. During the eight year period, three major evolutions took place along with extensive feedback from the users.

A. A Dual System

The system should provide the dual functions of an advanced programming system and a management information system. The needs of the manager must set the top level framework of the system. This requires a selection of the engineering methods and procedures and deciding what information should be saved; it implies choosing a method or allowing only certain methods to be supported by the system. The methods must have a sound engineering basis and fit the organization's business methods. With FASP it was a conscious decision to support several methods with the same system. The FASP facility is owned and operated as a government facility and used by weapon system contractors to develop and maintain software. Each contractor had different methods and procedures for doing business; yet each was able to effectively use FASP. Some contractors have used the Chief Programmer Team approach, others a different team approach. All use some form of structured programming, although the details are different. Thus, as a government facility it was important to impose only reasonable constraints on the contractor and allow for the different ways of doing business.

An important concept in software engineering is incremental development. The idea is to first complete the software design and then to build the software in stages, or increments, such that each successive increment adds a new functional feature. This approach breaks the work into smaller pieces that are easier to manage. Thus, it is easier to judge progress on the total project and it has the additional benefit of allowing users to gain some early experience with the software system. Incremental development has proven to be valuable on large scale software projects and should be supported by the SEE.

An important management need is the enforcement of configuration management principles. Configuration management principles consist of identification, control, status accounting, and the establishment of baselines. Decisions must be made regarding what software elements will be subject to configuration management. This involves deciding what is the smallest unit of software that will be configuration managed. A "line" of code? A "module"? Is it the basic compilation unit of the compiler? Is there more than one language to be supported and is the definition of a compilation unit the same for both? These decisions have a significant impact on the final system and must be made at the outset.

Along with structured programming came the idea of include segments. Include segments are fragments of code that are used in many modules without change. The programmer identifies these segments by name and places them in the database; in the source code of a module the segment is referenced by name. The system automatically locates the segment and "includes" it into the source stream before compilation. Since these segments are fragments of code they cannot be separately compiled without errors; however, they are useful to the programmers. In FASP, during a typical month the databases contained 48 000 modules and 28 000 include segments, showing the wide acceptance and use of include segments. Therefore, although include segments add to the configuration management burden, they are recommended for a SEE. Another related area is access control. Management level decisions are needed regarding what aspects of the system are subject to access control. In FASP there are three dimensions of control: control over access to the software end products, control over software tools, and control over access to the computer system itself. In the latter case this implies cost control over the use of the computer system.

B. The Database

The database is the most critical component of the SEE since it serves as the unifying element for all other components. The database holds not only the weapon system software, but also related technical and management information that contains the genesis and status of the total effort. Furthermore, the database has a significant influence on the performance of the SEE, an important consideration in obtaining a true production environment.

Here the term database refers to a fixed number of libraries that are encapsulated and managed as a whole rather than distinct parts. The software for a particular weapon system is contained in several "databases." Each database contains the following libraries:
- The source library containing either the source code for modules or the source code for include segments, or both.
- The object library containing the object code corresponding to the source library.
- The test library containing test input data, previous test results, test directives, and system generation directives.
- The interface data library containing information such as linkages to external object programs or to shared source code.
- The production data library containing modification histories and a variety of management information.
- The documentation library containing all documentation about the weapon system software.

The encapsulated database is the basic unit of the SEE; several libraries are included because the relationship between
those libraries must be strictly enforced. Thus, source and object code must have a one-to-one correspondence with no exceptions. A consequence of this relationship is that if compilation errors occur, the database (source and object libraries) will not be updated! Likewise, test data and test results are synchronized. Most important, at any time in the development schedule the management data are consistent with the rest of the database; thus, managers always have access to accurate information.

An important feature of the SEE for large scale projects is the automatic recompilation of dependent modules when certain software is modified. For example, if an include segment is modified then all modules that use that segment should be automatically recompiled.

Commands should permit software to be shared between the databases, allowing the total effort to be divided among several databases and teams. Similarly, commands should divide databases into smaller ones or combine them into larger ones; other commands should allow the databases to be copied.

The integrity of the database must be assured; therefore, during interaction with the SEE the system should automatically create a backup copy of the database permitting instantaneous fall-back to the previous version. Additionally, commands allow archive copies to be made on magnetic tape for off-line storage. This level of protection is over and above that offered by the host operating system.

C. Procedures, Tools, Commands, and Processing

The use of the SEE involves sequences of tool and database interactions. To simplify the use of the system, a set of procedures is defined that is invoked by user commands. A procedure is a set of computer directives that automates a particular task, invokes the proper tools in the proper sequence, provides all database manipulations and correspondences, and automatically records statistics of all activities.

The software tools are programs that do certain functions for the software engineer. Examples are editors, translators (compilers and assemblers), system generators, test analyzers, software emulators (target computer instruction level simulators), data extractors, report generators, and documentation aids. A tool or set of tools is automatically invoked as part of the execution of a procedure. Some tools are visible to the user, such as the editor, and require communication in a language unique to the tool. Other tools are invisible to the user, such as the librarian, and are automatically invoked when certain actions take place with the database. In the latter case input and output data may be processed by other programs, but all such actions are hidden from the user.

User commands cause procedures to be invoked. A command is a procedure name followed by parameter values; these values give the user flexibility in directing the procedure to accomplish the specific desired function. All commands are validated before being executed. The commands can be grouped into two categories: immediate and queued, depending on whether the database is modified or not. In batch mode there is no distinction and all validated commands are executed in the order received. In interactive mode an “immediate” command is executed at once; all others are placed on a command queue. Once activated the system executes the queued commands.

As an example of the power of commands, consider the FASP command MODSW (modify software). This command creates or modifies software in the database. In the FASP on the CDC CYBER computers, MODSW causes 315 job-control-language commands to be executed; in the FASP on the VAX computer it causes 262 UNIX shell-script commands to be executed. These operating system level commands are all hidden from the user.

A general set of procedures has been developed and is described in [8]. Fig. 5 shows a list of these general procedures. The procedures are divided into functional groups and are described by process flow diagrams. These diagrams use a structured English description of the control flow for a procedure
and a dataflow diagram showing the process performed, the tools used, the database contents used and produced, and other information required by the procedure. When a procedure is performed a certain amount of standard processing is done before and after the main processing for that procedure. The standard processing for each procedure is shown in Fig. 6 using structured English. Two process flow diagrams are shown in Figs. 7 and 8.

### D. Testing

Testing remains the primary method for determining the quality of software. The SEE should support four distinct types of testing during the code and test phase; they are as follows.

- **Progression testing** that evaluates new or modified software operation.
- **Regression testing** that identifies changes to previously attained software operation.
- **Automated test analysis** that measures the effectiveness of a test by identifying the software source code paths exercised.
- **Trial testing** that provides for testing proposed software changes without modifying the database.

Progression testing is used during the development of new software or modifications to existing software; this form of debugging is frequently an intense creative process best performed interactively. It usually involves interactive use of the software emulator to make experimental changes to initial conditions, data, or instructions, and immediate rerunning of the test.

Regression testing is used once proper operation is achieved; it ensures that the software does not deteriorate (regress) because of later progression changes. Two forms of regression testing are used: explicit and automatic. In the explicit form the user specifies the tests that are to be performed. In the automatic form tests are automatically run whenever certain modules are modified. Test data, test results, and test directives are accumulated in the database during the life of a module; also, an index is kept that relates tests to modules. A change to the module triggers the automatic running of all associated tests and a comparison of all results; the user is able to identify those portions of the test results that are important.

Automated test analysis is provided to check the quality of the tests themselves. In this form of testing a tool called ATA
(automated test analyzer) scans the source code and inserts software probes at program decision points. This allows the decision-to-decision paths to be identified. When the instrumented code is run on the software emulator with the test input data, the system reports how many times each path was executed, flagging those not executed. Thus, the percentage of total paths tested is available along with indications of "dead code" and code paths most frequently executed. These data allow the user to devise changes to existing tests or to develop more effective tests. The data on the most frequently executed paths are valuable when optimizing the speed of the program.

Trial testing consists of syntactic and semantic checks before the software is entered into the database. This type of testing is used when changes are made to large existing bodies of software. In such cases there may arise uncertainty about the interactions between changes to the software and to tests. Also, uncertainties about the optimum changes that could be made may require that several different changes be tried before deciding on the best.

E. Interactive/Batch

In many ways the differences between interactive computer jobs and batch have disappeared; however, there are some fundamental differences that are important to the operation of a SEE. A batch job consists of a stream of user commands with all parameters and input data previously determined; an interactive job must have these items supplied on-line. Since the system is to be user-friendly, the interactive job must prompt the user for such items and provide some helpful information when incorrect data have been input. Thus, the SEE must distinguish between the two types of operation and provide some extra software for interactive usage.

In a SEE for weapon system software it is unlikely that all the tools will be interactive, especially the compilers. Therefore, it is a matter of judgement to determine what functions are best performed interactively; all functions should operate in the batch mode. There are three areas that should allow both interactive and batch operation; they are editing, debugging with the software emulator, and generating management reports.

With the editor there is a clear benefit to the user to be able to quickly inspect and change the software; full screen editors appear to offer the best advantages. When debugging, particularly during progression testing, there is also a benefit to the user; here errors tend to be discovered more frequently and once observed the remaining parts of the tests can be terminated, saving computer time. Management reports, especially the smaller ones, tend to be the most useful when they can be quickly and easily obtained by the manager whenever desired.

Just as interactive mode is best for progressive testing, batch mode is best for regression testing. Here the total running time increases as the project software grows, a case best left for overnight turnaround when computers are lightly loaded and costs are frequently reduced. For example, when FASP has been used for maintenance of large bodies of software the ratio of interactive-to-batch commands is about 3/1 on the average; however, in tests of intense regression testing the ratio becomes 1/2. If one considers tool invocation during the same period then the ratio of interactive-to-batch is about 1/3 on the average; during intense regression testing the ratio becomes 1/12.
F. Multilanguages and Multitarget Computers

The design of a SEE is greatly simplified if there is only one programming language to be supported for a single target computer. For weapon systems this is rarely the case. Fig. 9 shows the matrix of languages and target computers in FASP. The difficulties begin with the languages themselves; a SEE is dependent on the programming language, a point not generally understood. One problem is the definition of a module is different in all the languages. Also, there are different dependencies on the database between the languages. For example, if one language has structured programming constructs with an include segment feature built into the compiler, and another language does not, then clearly there is a significance difference in the way that a SEE would support each language. In the latter case it may be desirable to provide the capabilities by way of preprocessors; however, the way the SEE supported each language would still be different.

Other problems arise owing to structural differences in the languages. For example, the versions of the CMS-2 language have an order dependency on the appearance of declarative statements and executable statements. The declarative statements are dispersed throughout the program in blocks followed by blocks of executable code with an order dependency on the referencing of data. In this nightmarish state any change normally would mean that the entire program would have to be recompiled. However, in FASP a special modular compilation feature has been added such that the system keeps track of the dependencies and only the appropriate blocks are recompiled. This example might appear extreme but it is characteristic of the difficulties that can arise.

It is normal to support several high-order languages and several assembly languages for a weapon system. Therefore, it is best to present the SEE to the user as an integrated string of tools that apply to a language and target computer. This would correspond to the vertical columns of Fig. 9. During the log-on process the user identifies the desired language or target computer; later commands are restricted to the associated string of tools. Internally, many common tools, such as an editor or librarian, may be used to implement the different strings; however, to the user the system appears as a unified set of tools.

The consequence of having strings of tools is that there are in effect several different SEE’s. In FASP there are four such systems concurrently operating in the host computers at any given time. From a maintenance standpoint, FASP is maintained in a Fortran FASP. About 75 percent of the code is common across the four, the remaining unique to each language-dependent environment. The language unique portions are maintained separately from the common portion and combined when a new version is desired.

G. Management

The success of a SEE depends on the degree that management is satisfied. Although a SEE makes the job of producing or maintaining software easier, it also constrains the user to work in a somewhat rigid framework, a point the user is sometimes quick to make. However, the benefits in productivity, improved quality, and stability over the life cycle are great compared to any perceived loss of freedom by the user.

The degree to which management is satisfied depends on a great extent on the amount of involvement by management, the degree to which the system is understood by management, and how smoothly the SEE fits into the current methods of doing business. As with any management information system, the SEE requires that management become more involved with the operation at a deeper level than previously. However, once this commitment is made the gains are great.

The management view of the SEE is through the reports; therefore, it is desirable to generate very clear concise reports in terms that managers can understand. Reports that measure work progress and expenditures against planned profiles are of interest to management. For example, to report that the effort is on schedule and within funding regarding the number of modules, lines of code, storage size, and target computer execution time as measured through unit testing is obviously valuable to managers.

There is no general agreement across the industry on what precise software measures should be made; therefore, each organization must establish such measures and slowly refine them based on experience. It is important to allow a high degree of flexibility for SEE management reports.

Software complexity measures have been somewhat disappointing as absolute measures of software quality [9]. However, some are useful as relative measures and can be used for management control purposes. Generally, care must be taken in the selection of such complexity measures.

A promising area appears to be “earned value” reports and other related measures. These reports can be easily established in a SEE and have the benefit of being based on impersonal data directly from the software development or maintenance environment. Of course, considerable experience is needed to select the particular “value” that is earned; however, there is a reasonable expectation that this can be accomplished.

VI. The Extension to Requirements and Design Phases

It has been stated that the SEE should support a weapon system over the entire life cycle as shown in Fig. 1. It is intentional that the term “system life cycle” has been used instead of “software life cycle.” Today, software is so important that it must be taken into consideration at the system level. Here the term “requirements” is used somewhat loosely to cover the phases in Fig. 1 from mission requirements to software requirements; perhaps, the terms system requirements and system design are more accurate.

The requirements phase begins with high-level statements about the mission of the weapon system. During a subphase called concept formulation, a set of requirements is evolved...
that begins to express the requirements in technical terms. The activities at this point are not highly structured. The system designers use high-level tools such as analytic simulations and the methods of operations research to do tradeoff studies and to verify the conceptual design.

Once the system requirements are expressed in technical terms the system architecture must be determined in detail. The critical issue is the allocation of the system functions to hardware or software implementation. The system designers need tools to assist the tradeoff analysis. This activity is probably the most difficult of the entire process since the final system’s cost and performance are largely determined by these allocations. Once the allocations are made, any changes become not only increasingly difficult but also increasingly expensive as the system moves toward operational deployment.

When the hardware and software allocations are completed, the system development splits into two paths; the resultant hardware and software efforts come together at system integration time. The software requirements should be expressed in a formal requirements language so automated tools can analyze them for completeness and consistency, the two major sources of errors. At the end of the phase the requirements should exist in a computer database so all formal documentation can be automatically generated.

The software design process begins with the formal software requirements and results in a specific software design. This design is a specification for the code. During this process the designers synthesize a software system that satisfies the requirements. This involves considering several different designs and evaluating them according to performance, cost, and ease of change. A major output of the design process is information that will guide the unit and system level testing of the software.

There has been considerable work done in both the requirements and design areas. However, a uniform and consistent set of methods has yet to be developed that covers the entire process. To date, somewhat singular efforts have been pursued that usually focus on one small step. For example, there are several software design methods that have been developed. The result is that the individual methods do not fit smoothly together, particularly at the boundaries between phases. There are two primary reasons for these problems.

First, the efforts to date have attempted to address the software issues, ignoring the distinction between software requirements and design and system requirements and design. Thus, system design remains a hardware-oriented process and there is a considerably better interface between system design and hardware design.

Second, there has been a lack of thorough understanding of the requirements and design process. Recent work [10] suggests that if one forms a model of the work that identifies the cognitive processes that are used, then virtually all present methods have serious omissions. Future work utilizing this concept would seem to have good potential.

From a practical standpoint it is best to view the present state as a time of change. There are certainly tools and methods that can be profitably applied to the requirements and design phases. However, they do not fit well together and it is likely that new ideas and refinements will continue to emerge. It would be ideal to have methodological strings of tools to apply to the requirements and design phases, tools that assisted the engineers with the cognitive processes and that automated the recording of relevant information and generation of documentation.

A recommended approach to a SEE for requirements and design is to start with a highly integrated environment for code and test as previously described. Next, simplify Fig. 1 to reflect just the software concerns, as in Fig. 10. New weapon system developments would start at the top and progress through all the phases; each phase would have a support environment as shown in Fig. 11. It is understood that in the requirements and design phases the tools and methods would be loosely coupled, although relationships between the phases can be determined and recorded in databases. The tools and methods are chosen off the shelf and then force fit together; several methodological strings should be implemented to gain experience with each. This approach is judged to be the most practical in the short term. As new, better integrated methods are developed they can be superimposed on this structure. The same approach can be used to extend the capabilities to the system design phases.

VII. INTEGRATION FACILITIES
Integration facilities consist of a hot mockup of the weapon system computers with realistic simulation of external inputs. These facilities are used for hardware-software integration at the system level, evaluation of man-machine interfaces, and evaluation of hardware engineering change proposals. Typically, they form the hardware configuration baseline for the computer and associated subsystems. The simulation of realistic inputs allows the total system to be tested in a laboratory where sophisticated instrumentation can monitor the tests; this minimizes costly flight or shipboard testing.

Originally the integration facilities used special equipment or groups of minicomputers to simulate the external inputs; the capabilities of the test engineer were limited. Today, these facilities can take advantage of commercial computers to create an integrated test environment that both speeds the testing and takes it to greater depth. Modern integration facilities are full integrated environments and are electronically linked to the software development and maintenance computers for rapid loading of the mission software. Interactive capabilities allow symbolic debugging to be done on the target computer; extensive capabilities for storing test inputs and saving test outputs are now available.
Experience has shown that it is better to use separate computers to run the integration facilities than to attempt to use the host computer of the software production facility. This is because the CPU and I/O utilization can be high in the integration facility computer during intense periods of real-time debugging.

Furthermore, the target computer and its subsystems frequently require extensive hardware checkout, particularly when new hardware is being developed.

VIII. SEE ARCHITECTURE

The goal of the SEE is to support the weapon system over the entire life cycle. There are, of course, many ways to implement such facilities. The approach taken at the NADC was to coalesce the functions of Fig. 11 into two facilities as shown in Fig. 12. At NADC there are large central facilities capable of supporting these activities and several integration facilities distributed throughout the center. A larger scale approach is to form clusters as shown in Fig. 13 and to interconnect the software production facilities by communications networks. Similarly, the software production facility could support just one integration facility with less capable host computers. It is important that the production facilities be interconnected regardless of size because this communications capability will ultimately permit software sharing to take place between weapon system projects.

Alternately, the functions of Fig. 11 could be allocated to separate host computers that are interconnected. The choice may be dictated by the scale of the available host computers, a judgement that may vary depending on the expected workload. However, a word of caution; the software tools of today do not efficiently use computer resources; thus, it is easy to underestimate the size of the host computers. Software emulators used for unit testing take a large amount of computer resources, for example.

An emerging factor in SEE architecture is the availability of microprocessors and the expectation that networking is close at hand. An excellent example of a workstation for a software engineer is [11]. With this technology the problem becomes how to distribute the functions to retain the dual aspects of an advanced programming system and management information system. On the one hand, microprocessors appear to have the power for editing, small compilations, document gener-
Fig. 12. The NADC two facilities approach.

Fig. 13. Interconnection of facilities by networks.

...
Research on Structured Programming: An Empiricist’s Evaluation

IRIS VESSEY AND RON WEBER

Abstract—In spite of the widespread acceptance by academics and practitioners of structured programming precepts, relatively few formal empirical studies have been conducted to obtain evidence that either supports or refutes the theory. This paper reviews the empirical studies that have been undertaken and critiques them from the viewpoints of the soundness of their methodology and their ability to contribute to scientific understanding. In general, the evidence supporting precepts is weak. A framework for an ongoing research program is outlined.

Index Terms—Design, experimentation, human factors, languages, performance.

I. INTRODUCTION

A basis for improving the quality of software, the precepts of structured programming are compelling [22]. For the academician, the mathematics of software have a newfound elegance and rigor. For the practitioner, structured programming concepts have strong intuitive appeal. As a result, the area has spawned a multitude of disciples, and many books, articles, and courses have appeared, all offering some dose of the new elixir.

Whereas conceptual developments in structured programming have been forthcoming, corresponding empirical developments have been slower. Several researchers have bemoaned the unsubstantiated nature of the theory [2], [24]. Since ultimately programming is an empirical science, the acid test of a normative theory of programming must be whether or not the principles derived from the theory produce cost-effective changes in software practice. In the final analysis the theory of structured programming amounts to nothing more than an interesting intellectual exercise if these cost-effective changes do not result.

In this paper we examine the precepts of structured programming from the stance of the empiricist. Our purpose is singular: we seek to show that these precepts, which many re-