An industrial software engineering methodology supported by an automated environment

by MICHAEL S. DEUTSCH
Hughes Aircraft Company
El Segundo, California

ABSTRACT

In recent years, industry and government have sought to formalize software development by constructing automated environments that support the application of modern techniques and methodologies to the production of software. This paper describes the automated software development system being installed at Hughes Aircraft Company. This system is expected to be a major contributor to the orderly management of software development at Hughes.

This software development system consists of integrated development techniques over the life cycle, a set of software tools, and a physical facility for software development and test. Structured methodologies such as structured analysis, structured design, and structured programming are supported by automated tools. The configuration of the software development facility consists of a host software development system, the target machines, and the user display terminals.

Project planning and performance measurement are based on the rate charting technique and earned value assessment.
INTRODUCTION

The "software crisis" of which we are constantly reminded is connected with the vastly increased complexity of contemporary data processing systems and the limited ability of traditional software practices to deal with this complexity. Only within the last several years has this stagnation in software technology been generally recognized and accepted in government/industry circles with the realization that current software practices were of little help in attacking increasingly complex applications.

One of the more positive responses of industry to this situation has been to formalize software development into an engineering practice by developing automated software development environments that support the application of modern techniques and methodologies to the production of software. These software development systems have been built largely as proprietary products with exact characteristics varying from organization to organization. Regardless of the differences, these systems support the same thrust, i.e., that orderly software management and predictable results are based on methodologies of how software is specified, structured, and integrated into larger systems.

In this paper, the software engineering development system being installed at Hughes Aircraft Company's Space and Communications Group is described in terms of the development methodologies being used, the software tools that support the methodologies, and the facility that houses the tools. An overview of the development system is presented followed by a description of the engineering method within each life cycle phase. The project planning and performance monitoring approach is also described.

SOFTWARE DEVELOPMENT SYSTEM

The three required constituent elements of a software engineering development system are: (1) an overall approach of coherent methodologies covering the entire software life cycle, (2) a set of software tools that supports the consistent application of the methodologies, and (3) a computational facility that houses the tools.

The techniques and tools for the Hughes development system are delineated on Figure 1 for each life cycle phase. Note that several tools/methodologies span multiple life cycle phases and provide a unifying influence. Noteworthy is the system verification diagram (SVD) that is used for requirements and design verification and to guide the construct, test, and integration processes. (The SVD technique was originally conceived by Computer Sciences Corporation.)

The software development facility is shown on Figure 2 and consists of a host software development system, the target machines, and the user display terminals. The host development system consists of several PDP 11/70s and VAX machines. The 11/70s house the programmer's work bench (PWB), which is a facility for source code generation and word processing. The VAX hosts a set of requirements definition and design tools. Requirements engineering, design, and code generation are accomplished on the host system independent of the target machine.

Dynamic execution of code is performed on the target machines. The target machines contain machine peculiar tools including compilers, assemblers, linkers, debuggers, and automatic test tools.

User terminals are linked to an "intelligent" microcode driven switching device called a port contention unit. A terminal may be connected by user request at sign-on time to any of the host system or target machines. There is at least one terminal for every two programmers (located in their offices), thus providing practically unlimited access to computational resources.
Requirements Definition Phase

This phase normally consists of two constituent activities: (1) a specification activity that generates the system level functional requirements specification and (2) an allocation activity that generates a specification for requirements allocated to each computer program configuration item (CPCI).

The structured analysis methodology is used to analyze the requirements and produce structured specifications. A data flow diagram is constructed to present a logical model of the system functions. By successively decomposing the elements of the data flow diagram, the system is disclosed in an order proceeding from the most abstract to the most detailed. All the data items flowing between bubbles are defined in a data dictionary. Each function in the detailed level data flow diagrams is described by a process description.

SCG is a display interactive tool for creating, modifying, and maintaining structure charts. A hardcopy of the created structure chart can be output that is suitable for deliverable documentation. DQM analyzes a structure chart by identifying areas of the design that are overly complex using algorithms based on a hierarchical tree model. Highly complex sections are potential problem areas that are subjects for redesign on the next iteration.

Verification of the design versus requirements is accomplished by "threading" or associating each stimulus/response element of the SVD with a sequence of software modules that implement the stimulus/response pairing. Incomplete, missing, or extraneous associations suggest a nonresponsive design or misinterpretation of requirements. This allocation of modules to threads is the method by which a visible connection between requirements and design is maintained throughout the development cycle.

Detailed Design

Each of the modules identified in the structure charts is expanded into a detailed design. The design of each module is expressed in pseudo-code and input interactively into a programming design language (PDL) processor. Logic tree plots are automatically generated by a tool that uses the PDL syntax as a command language. These logic trees represent the PDL syntax in a graphical form that permits better comprehension of the abstract information and are suitable for design walkthroughs, design reviews, and deliverable documentation.

A development presently in progress will replace the existing conventional PDL with an Ada PDL. This will permit a module design to be developed in two major steps of refinement—specification and implementation. A module specification providing a functional definition of the module procedure and a definition of the visible data interfaces is produced first. The module implementation providing the design of the procedure that operates on the visible data is then generated. One of the major advantages of the Ada PDL is that it will provide automated verification of interface consistency between module specifications.

Software Construction

The software construction phase entails the coding, checkout, and preliminary qualification testing of each CPCI. A build plan for each CPCI is graphically depicted as a calendared network of threads that were previously defined on the system verification diagram. Because each thread is correlated with specific modules, the coding sequence of modules is defined by the build plan. Each thread undergoes a preliminary qualification test before being baselined.

HIFTRAN, a Hughes developed structured FORTRAN preprocessor, is used wherever possible as the source language.

Each thread is exhaustively tested with the assistance of the RXVP80 automatic test tool. RXVP80 "instruments" the code to determine the extent of the testing coverage. Reports are generated by RXVP80 showing which paths of the code have been covered by previous testing and which paths remain to be tested. Additional test cases are contrived to target on previously untested paths. This sequence is repeated until a
complete (or very close to complete) path coverage is attained.

Actual project experience has shown that this early emphasis on comprehensive testing using the automatic test tool reveals a significant number of errors during the construction phase that otherwise would have gone undetected until some later time. During subsequent periods, however, including operations, the detection rate of latent errors is lower. The cost of rectifying an error later in the life cycle is, of course, higher than if detected earlier. The comparative error detection profiles, depicting testing with and without the aid of the test tool, is illustrated on the left-hand side of Figure 3. Exhaustive testing assisted by the automated tool, while requiring a slight additional level of immediate testing effort, is believed to be a worthwhile investment that pays dividends in reduced life cycle costs.

The exhaustive test procedure applied on a recent project at Hughes detected an average of one additional error per thread (approximately 400 additional total errors). The cost, in schedule time, of performing the extended testing ranged from one-half day to three days per thread, with the average schedule cost close to one day per thread. Normally, two persons were involved in testing the thread at this point and, therefore, the incremental cost of the exhaustive testing effort was an average of two person-days per thread. The subject project consisted of about 400 threads. The incremental cost of finding and correcting the 400 additional errors was 800 person-days (400 errors × 2 person-days/error) as shown on the upper right of Figure 3.

The average relative cost to fix an error during the integration activity versus construct activity is four times as great, and the average relative cost to fix an error during the operations phase versus construct activity is nine times as great. To model the life cycle cost benefits, it is assumed that discovery of the 400 errors would otherwise have been evenly distributed over the integration and operations periods. This is probably a conservative estimate since the type of error overlooked during the construct activity is more likely to reappear during operations in which the software would otherwise undergo its first thorough exercise. The 200 errors found during integration would cost 1600 person-days (200 errors × 2 person-days/}

Integration and Test

An incremental integration and test philosophy is based on the "builds" technique. In this approach there is considerable overlap between system integration and CPCI construction. A major emphasis is to segment a complex system development into smaller, more manageable, functionally oriented segments called builds.

Testing at the system level is planned and organized in the same manner as the "thread" testing at the CPCI level. The series of system level tests will integrate CPCI versions and hardware CIs. An SVD derived from the system requirements specification is used to establish the content and order in which partial versions of CPCIs and CIs are developed and introduced into the integrated testing process. Adding only one new element at a time toward a deliverable system capability permits more efficient detection and correction of interface problems.

Builds are incrementally constructed from components of one or more CPCIs and hardware CIs. Each build augments a previously established baseline. Prior to build testing, a preliminary qualification test is performed on components of the CPCI to establish a CPCI baseline version. This baseline is subsequently augmented by additional components which extend the baseline to a complete CPCI. The key objective with this approach is to establish a logically complete system skeleton early in the integration period.

The merits of this approach include: (1) demonstration of key functional capabilities early in the development cycle, (2) early demonstration of the essential viability of the system, (3) early demonstration of key interfaces, and (4) minimization of special test bed environments required for test and integration.

PROJECT PLANNING AND PERFORMANCE MEASUREMENT

At this point, the software development process has been examined from a technical perspective. This development approach is now explored in terms of some of the accompanying management methodologies that complement the technical approach. A project planning approach, based on earned value reporting, that is a natural adjunct to the technical software engineering process will be described. This planning approach consists of methods for scheduling, reporting, and monitoring development progress.

Earned value measurement is directed toward assessment of progress through comparison of actual versus planned expenditures and schedule. The procedure involves decomposing a project into small work packages. Each work package is accompanied by frequent milestones with specific
completion criteria, a situation naturally supported by the engineering process previously described.

At periodic points, schedule and cost variance for each work package is determined and summarized at various levels of the work breakdown structure up to the total project level. Three parameters are used in this determination: (1) the budgeted cost of work scheduled (BCWS), (2) budgeted cost of work actually performed (BCWP), and (3) actual cost of work performed (ACWP). The cost variance is the difference of the ACWP and BCWP. The schedule variance expressed in dollars is the difference of BCWP and BCWS for the effective reporting date.

The basic management tool used in this project planning approach is the rate chart. It is a simple two-dimensional plot of the percentage of work planned and actually completed as a function of time.

As illustrated on Figure 4, planning begins by constructing a master schedule bar chart showing the time-oriented relationships among the various phases of the project. Although the bar chart is an effective tool for initial project planning, it inadequately portrays overall project status and production trends. Instead, a technique called “rate charting” is used. The composite rate chart plan shown on Figure 4 has been derived from the bar chart. The rate chart shows start and end planning dates (derived from the bar chart) and production rates. By weighting the work in each of the phases according to the allocated budget, a total project production rate can be planned as shown here.

![Figure 4: Rate charts: tools for monitoring progress](image)

Rate charts provide visibility for all levels of management—individual work areas, project management, and customer. By evaluating the slope of the actual production rate with respect to the planned rate, a manager is alerted to trends and can consider reallocation of resources.

Each of the development activities is broken down into several or more work packages. Planned versus actual accomplishments are monitored at the work package level and summarized on a composite rate chart. Individual work package contributions to the composite summaries are weighted according to the BCWS that has been allocated to each package. This is depicted on Figure 5.

![Figure 5: Work package contributions factored into composite rate chart](image)

Work packages are generally defined around the natural products of the engineering process. During requirements and design activities, these products and work packages are documents (specifications, ICDs, test plans, etc.). Progress is planned and measured by allocating points or work units to subchapter and chapters of the document. Work units are accumulated as each subchapter is completed. During coding, the complexity units assigned to each thread serve as work units. In integration, work units allocated to each functional component are accrued as each component is integrated into the baseline system.

The schedule variance (BCWP minus BCWS) and cost variance (ACWP minus BCWP) is computed monthly for each work package and summed up through the work breakdown structure to evaluate overall project status. This performance measurement is supported by an automated tool, the Performance Evaluation and Measurement System (PEMS). PEMS receives and archives actual expenditures weekly, provides an interactive interface for scheduling and recording of accomplishments, and performs earned value assessment. PEMS outputs automatically produced reports that document earned value accomplishment and variances.

CONCLUDING OBSERVATIONS

The software engineering approach described here has emphasized auditable verification and validation events in each life cycle phase that are directed toward early detection of errors. These verification and validation mechanisms are summarized on Figure 6. This reflects a sensitivity to the escalating cost of fixing errors as a function of the time in the life cycle that they are detected as shown on the upper right of Figure 6. The software engineering development process that has been outlined here has emphasized parallel verification and validation in all the life cycle phases as a cost-effective approach to guarantee product reliability and contain life cycle costs.

These parallel verification and validation mechanisms are recapped briefly here. During requirements definition, the CADSAT analysis tool is used to verify interface relationships, while the system verification diagram verifies the functional requirements. The SVD is later employed to guide design verification, construction, and test/integration.
verification is performed by “threading” the design modules against the stimulus/response pairings of the SVD and also by using the design quality metrics tool to evaluate design complexity. During detail design, an Ada PDL processor verifies interface relationships among the various modules by automatically checking Ada package specifications. In the construct activity, CPCI requirements are informally validated using threads defined by the SVD and assisted by the RXVP80 automated verification system to exhaustively test the software and maximize error detection at the thread level. CPCI and system requirements are formally validated during test and integration using the builds approach, guided by the SVD, and directed toward early establishment of a system skeleton.