CONF-9105125--1

Progress Report: Automated Computer Software Development Standards Enforcement

Herbert P. Yule and John W. Formento
Advanced Computer Applications Center
Argonne National Laboratory
9700 S. Cass Avenue
Argonne, IL 60439

CONF-9105125--1

DE91 007328

Abstract: The Uniform Development Environment (UDE) is being investigated as a means of enforcing software engineering standards. For the programmer, it provides an environment containing the tools and utilities necessary for orderly and controlled development and maintenance of code according to requirements (DoD-STD-2167A). In addition, it provides DoD management and developer management the tools needed for all phases of software life cycle management and control, from project planning and management, to code development, configuration management, version control, and change control. This paper reports the status of UDE development and field testing.

Goals

To improve management of its software acquisitions and its production software, a sponsor within the Department of Defense (DoD) requested a study on the feasibility of automated implementation of software engineering standards. As a result, the Uniform Development Environment (UDE) was developed as a means of establishing unilateral software engineering standards for DoD and its software contractors. Software development activities under control of the UDE may be required to conform to DoD-STD-2167A. The extent and nature of activities necessary to comply with this standard vary from job to job, and thus a major requirement of the UDE is flexibility. Other important aspects of software development managed by the environment include configuration management, version control, change request management, and so forth.

Some of the benefits the UDE is designed to achieve are to:

o improve the productivity and efficiency of software development and maintenance
o support orderly and controlled software development and maintenance throughout the entire software life cycle
o allow communication and cooperation between developers, managers, users, and other persons involved in project support
o provide an integrated framework for accommodating development and analysis tools
o provide effective configuration management
o help the software developer involved in extensive development efforts by providing a common interface to tools and utilities, communications with team members, etc.
o facilitate utilization of CASE tools
o provide flexibility for incorporation of new and different software technologies, methodologies, and procedures

UDE Background

In the early stages of this study, the sponsor's primary concern was the possible imposition of software development standards on contractors developing software; hence the project name, Uniform Development Environment. The software development environment chosen for this study was the Atherton Software BackPlane together with Atherton's Project SoftBoard (PSB). At the time that selection was made, no other software provided so many of the required capabilities. The BackPlane is a software development environment specifically designed for large programs, programming in the large, and programming in the many. Atherton designed the BackPlane to provide a seamless, flexible environment furnishing the user with insulation from the hardware platform. Clearly, no software environment can provide complete insulation from the hardware, but the software environment will improve many aspects of code development and can improve software transportability across platforms.

A complete description of the features provided by the BackPlane and Project SoftBoard is beyond the scope of this paper; a partial list of features follows:

o Flexibility via a programmable interface
o User portability

- Configuration management
- Seamless integration
- Software tool interoperability
- Data interchangeability and sharing
- Partial software portability
- Tool integration and utilization
- Large-scale data management
- Support for the full software development life cycle
- Workflow methodology enforcement

For further information on the Atherton BackPlane and Project SoftBoard, see references [1 and 2].

The BackPlane maintains and manages all files as objects in an object-oriented database. Access to objects is only available through the environment. In addition, it ensures file and version immutability, and can be implemented to require users to provide documentation of their activities such as code development or modification. The amount of documentation required can range from extensive to none. When documentation is not mandatory, the user has the option of inserting documentation as deemed appropriate.

Project SoftBoard manages and automates Component Change Request (CCR) procedures. Change requests pertaining to the managed software system are entered directly into the UDE, and appropriate staff members are automatically notified of the CCR and of its disposition. Change deadlines are provided on an alert basis, and notifications of implemented changes that could possibly affect other programmers and managers are sent to all those concerned.

Required software tools (compilers, source code editors, debuggers, etc.) and CASE tools can be integrated into the environment so that programmers have access to any requisite software tool or utility within the environment. Equally important, a development methodology that imposes compliance with a given set of development standards can also be integrated into the BackPlane.

Software tools to enforce standards could be integrated into the BackPlane in accordance with 2167A-tailored requirements. Of course, documentation required by the enforcement tools would have to be supplemented by judgment, and management acceptance of these required documents would have to be obtained before incorporation of documents into the database as an immutable object.

The major thrust of this work has been the design and development of a prototype UDE that could be imposed on contractors who develop software. Developers could be required to develop programs under this environment. The environment can have integrated enforceable workflow methodology to compel adherence to standards. In this way, developers would be required by the environment to adhere to a desired set of software engineering standards. Thus, program baselines could be created in accordance with acceptable standards, and once established, such baselines are immutable. Change requests leading to new versions and releases would be completely documented as to (1) the rationale for the change, (2) how the programmer implemented the change(s), (3) how the changes were verified and validated, and (4) when a new version is created and released. Parallel development efforts would be similarly treated, including merging of parallel development branches. All documentation would be maintained in the UDE and would be extended as further documentation becomes required.

The UDE is more than just a development environment; it will support all phases of the software life cycle. The field test of the prototype, described below, was an exercise in very large code maintenance for an already developed code.

Standards

A discussion of a programmer's development environment and which components it should comprise would be incomplete without mention of the role of standards. A wide variety of standards exist for software development and related purposes. These include collections from government, industry, and academia [3,4].

Probably the most well known set of software development standards has been produced by the Department of Defense (DoD) in its series of automated data systems standards. Although other organizations have developed similar standards, the focus of this paper is on the DoD standards and methodologies since the work presented herein has been sponsored by DoD.

The UDE was conceived as a means of prescribing and enforcing uniformity of software development methodology and techniques under a variety of development scenarios and conditions. For this reason, it was necessary that the environment offer a great deal of flexibility of enforcement since different situations call for differing degrees of standards implementation and administration.

The organization sponsoring this research is charged with providing technical support to a number of technical divisions that perform extensive analyses using simulation models. The sponsor must maintain several models and oversee the development of new models. This effort is carried out through the use of many subcontractor organizations; the sponsor does not provide these services directly. A major requirement this work attempted to address was that of allowing the sponsor organization to have a degree of control over the way its subcontractors did software development.

This requirement led to an investigation of how standards should be enforced as well as which standards should be enforced. The obvious standard to enforce was DoD-STD-2167A, "Defense System Software Development" [5], since this sponsor was involved only in defense-related analyses. However, 2167A is a comprehensive standard, and unilateral enforcement of all its provisions was deemed to be counter to optimal operation of this organization. A much more sensible enforcement strategy was that of tailoring a subset of military standards to each situation.

The DoD-STD-2167A is based on a software life cycle model. This model is a linear process, often called phase-oriented. It basically recommends that software development consist of separate and distinct stages. In the DoD standard, these stages are:

a. system requirements analysis/design
b. software requirements analysis
c. preliminary design
d. detailed design
e. coding and software unit testing
f. software component integration and testing
g. software configuration item testing, and
h. system integration and testing

The life cycle model was originally derived from the hardware production model of requirements, fabrication, test, operation, and maintenance. As such, it reflects management concerns in production. It is essentially a linear process, although the verification and validation functions occur in parallel with other activities. While this model works very well with hardware production, its appropriateness for software development is questionable. Instead, it was desired that the UDE be able to selectively enforce those parts of the DoD standards that are applicable on a case-by-case basis.

The UDE could have been developed as a standards-based programming support environment. This is one in which major aspects of the environment are based on the application of a set of standards or an ordained development paradigm. Traditionally, developers have worked under conditions and standards as set by the company developing the system. Compliance with standards was enforced manually, and team members had much discretion in their degree of use of standards and techniques. As the number of programmers employed on a project increased, so did the opportunity for deviation from the mandated standards. Standards-based environments are not considered to be particularly flexible.

The ability to selectively enforce various subsets of 2167A as well as or in addition to any other standard or development attribute considered important to a particular development paradigm was central to the design of the UDE. The range of scope of various development and maintenance projects active at any time led to recognition of the need for selective standards enforcement. The UDE is able to provide this flexibility of enforcement and tailoring of capabilities.

The UDE is a programmable environment. Inherent in it is a capability for requiring software development personnel who work under it to adopt certain practices and standards. For instance, the person who has the task of adding a function to a software unit may be required to check out a controlled version of the software unit from the UDE and document what he/she intends to do. In order to check the completed unit back in, he/she could be required to update the system data flow diagrams, user documentation, and provide evidence of testing of the unit.

Another project could require software engineers to perform these same tasks and update entity relationship diagrams as well. The UDE can be programmed to require one set of tasks for one project and a second set for a different project. Thus, the standards can be applied on a case-by-case basis. An automated method exists for evaluating the requirements of a software development project and determining the most relevant aspects of 2167A. The results of this process can be used to provide input into the tailoring of the UDE to the needs of the separate projects within DoD.

A software project may entail development standards that require the use of an approved computer aided software engineering (CASE) tool (or set of CASE tools) in system development. It may entail the use of a certain compiler or set of configuration management tools mandatory on the project. The key to the UDE is flexibility of standards enforcement rather than strict adherence to an arbitrary set of standards.

The Argonne UDE Prototype

UDE Prototype Configuration. The prototype UDE was assembled by Atherton for testing at Argonne. It consisted of the BackPlane, Project SoftBoard, and the following integrated tools: a Fortran compiler, linker, debugger, the (Unix) VI editor, a DEC VAX EDT editor emulator, a project management system (SUN's PMS), and a commercial CASE tool (Software through Pictures). The hardware system was a SUN workstation model

3/260 running under SUN O/S 3.5 Unix. It should be noted that the SUN workstation supports a windowing environment. The UDE can be operated from a windowing display or an ASCII terminal. The majority of operations are better performed in the windowing environment, although a few operations are easier on an ASCII terminal. Operation of the UDE without windowing was found to be cumbersome and is not recommended.

UDE Features and Operation. The UDE typically contains a number of databases, one for each code package. To facilitate understanding by the reader, consider the following scenario. A programmer is tasked with placing a code package under UDE management for configuration management. Once the baseline is established, a team of programmers will proceed with bug fixes and with several major modifications. A programmer will start up the UDE and create a new database for the code package. He/she will then "import" into the database requisite source files, text files containing documentation, and any other files needed to generate the executable file(s) and documentation. All of these files become parts of the baseline version, which is immutable.

To fix a bug, the programmer obtains a copy of the immutable source code and proceeds to debug it. The original baseline is not changed by this process. At some later time this bug fix and others will be reviewed and, if acceptable, incorporated in an updated (and immutable) version. Acceptance of the revision could require that sufficient documentation be provided for incorporation with the new version.

Major changes can utilize parallel branches for simultaneous, totally independent code modification. When all branches are finished, differences between branches are manually resolved, documented, tested, validated, and finally made into yet another version.

Only (immutable) versions are released for distribution. Because each version is identifiable through its unique identifier, released versions are identifiable unless deliberate efforts are made to delete identities.

UDE Prototype Testing. The prototype was tested extensively at Argonne for ease and speed of learning, ease of use, adequacy of manuals, information supplied by the UDE, configuration management capabilities, action documentation, report generation, release management, version control, and integrity of system releases.

Typical large software packages, including the UDE, follow a fairly rigid logical operation procedure. The typical user learns to conform to this procedure fairly quickly; the learning process is both facilitated and shortened by receiving instruction from Atherton either at scheduled classes at Atherton in Sunnyvale, CA, or from instructors on site. One week of instruction is sufficient for the user to attain adequate mastery. After receiving instruction, the UDE is relatively straightforward to use, within the procedural constraints indicated above. (Such constraints are necessary to provide the configuration management capabilities required.) Without instruction, the time required to achieve adequate mastery would probably be considerably longer. The manuals are adequate as reference material, but are of limited use to the novice.

For configuration management, version control, and integrity of system releases, the UDE provides almost all of the capabilities required for mandating compliance with 2167A. Report generation and action documentation were cumbersome at the time we tested the UDE, but in subsequent releases, this feature is much improved.

Our initial reactions to the UDE were favorable, and it was decided to evaluate the UDE under realistic conditions at a contractor site. Rather than perform a lengthy evaluation, which would be required for code development, it was decided to evaluated the UDE for code maintenance, version control, and CCRs.

The Field Prototype

A prototype similar to the Argonne prototype was established at a contractor site for field evaluation. Funding constraints precluded integration of a production Fortran compiler into the BackPlane and limited the magnitude of the field evaluation. The field test and evaluation were arranged for code maintenance of an existing, 600,000-line, Fortran source code. In brief, evaluation was planned for baseline establishment and for CCRs. All results obtained using the UDE were verified by comparison with code maintained and changed following established procedures. Specific details of contractor tasks are provided in the next two major sections of this paper.

Baseline Establishment Methodology

Database design. Because a database is being created by the UDE, care in database design and creation to assure completeness, flexibility, and expansion is mandatory. The structure of the database must be consistent with the directory structure previously used for maintenance. Thus, although baseline creation occurs only once for a given code and version creation may occur several times, well planned initial baseline creation is very important. The structure of the initial database will affect versions created later. Special attention will have to be paid to the source code, which specifies files in other directories (e.g., INCLUDE statements for source files and open statements for data or output files). File naming conventions will require special attention to avoid identity complications caused by duplicate file names (native operating systems permit duplicate file name for files in different directories, while the BackPlane does not).

Database creation and loading. The contractor was charged with actually creating the database, loading it with all requisite files (i.e., "acquiring" files from the native operating system as objects in the database), and demonstrating that a correctly operating code can be generated from the UDE.

Native operating system O/S) accessibility and utilization. Access to the native O/S, from within the UDE, must permit utilization of software tools and CASE tools not integrated into the prototype UDE. For instance, access to system information is provided by native operating systems, and could be made available through the UDE by simple integration procedures. In a UDE with a minimal number of integrated tools, facile access to the native O/S would almost certainly be mandatory.

Implementation of event logging. In placing (acquiring) a large number of files into the UDE, a log file should be generated to provide a permanent, detailed record of the process so that successful acquisition may be verified, and problem areas adequately documented and readily detected.

Internal collection type utilization. The native O/S term "directory" or "sub-directory" becomes "collection" or "sub-collection" in the BackPlane. A collection is analogous to a directory. However, a collection has properties, unlike a directory. These properties include fortran_project, c_code, internal_text (the catch all property), binary, method_map, isb_pool, etc. Other collection properties are possible. Properties serve to keep similar objects together. A collection is an object in the database, and in the Object Oriented Programming (OOP) paradigm, an object must have enough intelligence (i.e., properties) to respond to messages sent by the program.

In the UDE, a component (of a collection) is indivisible: the smallest possible object. It is equivalent to a file on the VAX. For example, a component may contain a Fortran subroutine or function. Unlike files, components have properties such as ADA, binary, c_source, fortran_main, fortran_subroutine, text, and so on.

Collection and component properties simplify the job for the user. Fortran_subroutine components, at user direction, will be candidates for editing or compiling. Fortran_main components can be compiled, edited, executed or debugged.

Database access control. The context feature, together with read and write permissions, grants or denies access to database objects. Different contexts may access different portions of the database. For example, the project manager's context would have access to all collections and components. Program users utilize the code for their own purposes, while programmers maintain the code. Thus, users' contexts would only permit them to access executable objects, data components, and some documentation. Programmers' contexts would permit them to "see" source components, object components, etc. Quality Assurance (Q/A) staff would have contexts allowing them to view their documentation components. The UDE's flexibility permits the project manager to define contexts to suit his/her particular needs. An important design requirement of the database is to establish contexts.

Immutability. After the entire source code is imported into the UDE, tested, verified, and validated, it is ready for "checkin". Once checked in, the baseline is permanently established. It is immutable. Clearly, "checkin" should require prior management approval. Further code modifications are easily performed by "checkout" of objects to be modified. Modifications can be incorporated into new versions checked in later -- the baseline cannot be modified. The contractor was asked to demonstrate that the objects in the baseline were indeed immutable.

Historical records. The UDE tracks and maintains records of every "checkin" and "checkout", together with the user-supplied comments accompanying these actions. The contractor was required to determine which level of enforceable documentation should be required and to demonstrate that necessary and sufficient documentation was thereby generated.

UDE Change Procedures, Management, and Tracking

With the establishment of the immutable baseline complete, the next step is to set up change procedures: CCRs. Project SoftBoard was designed by Atherton to manage these requests.

Simple change implementation and tracking. A simple change request might be a minor bug fix. The user detecting the bug would submit a CCR documenting the bug using the PSB facility. Then the designated software person, following established procedures, would determine the course of action to be taken to respond to the CCR. He/she could designate a certain programmer through the UDE to investigate the bug; a priority could be assigned to the CCR; a deadline could also be created; all of these actions, as well as tracking, are supported by PSB.

Typical operations with CCRs include 1) opening new CCRs, 2) displaying existing CCRs, 3) displaying newly created CCRs, 4) querying CCRs on specific criteria, 5) printing CCRs, and 6) closing existing CCRs by changing their status to resolved, rejected, or deferred. In addition, CCRs may be linked (related) to one another, may be linked to users, delegated to other users, or forwarded to other owners. Tracking of CCRs may be achieved through querying CCR status and/or history.

Multiple change implementation and tracking. For more multiple changes, each programmer might check out a branch. Within his/her own branch a programmer could modify code without affecting modifications made in another branch by another programmer. Thus, multiple, parallel changes can be made independently. All of the change management capabilities described above are available for this more complex case.

Reconciliation of multiple branches for new baseline or version establishment. When modifications are completed, the project manager may decide to create a new version for release. The initial baseline is called version 1 by the UDE. To generate version 2, differences between different branch versions must first be reconciled. Unfortunately this is a manual task. Q/A and validation and verification (V&V) are required. After formal approval, the "checkin" procedure will create version 2. Now both version 2, the new version, and version 1, the baseline, are immutable and available. The contractor was tasked to study multiple changes and branch reconciliation only if time permitted.

Documentation of changes. PSB provides capabilities for complete, printable records of CCRs and their dispositions. The report shows the status of each CCR: completed, open, closed, etc.

Differences between collection versions and component versions. Differences between component versions can be displayed using the native O/S "differencing" facility. Differences between collections are indicated as differences in version numbers of objects attached to the collections. Differences are important in documenting changes.

Correspondence establishment and utilization; Deadline notification. A correspondence is a relationship between two component versions. If one component is changed, a notification message is sent to the other components for which correspondence is established. Thus, the programmer is notified of changes that may affect his/her work, and other objects are notified when a programmer makes changes that affect the corresponding documentation object, etc. Due dates or deadlines are set when CCRs are created. Thereafter, users are automatically notified when the due date arrives or when corrective action is overdue.

<u>Baseline Establishment Findings</u>

Database design. It was found that the BackPlane's collection structure is very similar to directory structures commonly found with modern file management systems. "Include" files were thought to be a possible source of difficulty, but that could not be verified during the evaluation. The BackPlane will not permit two files in a database to have identical names even though located in different collections or subcollections. This problem did not arise in the evaluation; in any case, good configuration management practices normally require all files to have unique names. The contractor found no problem in database design, or in the actual database creation.

Database creation and loading. Creation of the database was no problem, but loading the database with all the source files proved somewhat difficult at first. By experimenting with different methods, it was found relatively easy to create "script" files (known as "command" files on some systems and "batch" files on microcomputers) that performed the task (as a background job) satisfactorily while generating a log file to indicate any trouble spots.

Native O/S accessibility and utilization. Ready access to the native O/S is available from within the UDE. Further, it was found possible to create directories in operating system that mirrored collections maintained by the BackPlane. Hence, it was possible to take extensive advantage of the native O/S using these directories. Changes in directories are mirrored in the collections.

Implementation of event logging. The BackPlane generates logs of user activities. Also, it provides numerous opportunities for the user to comment further on activities. The challenge to the user and manager is to limit these records to a meaningfully short discourse. Otherwise, the logs can become so voluminous that desired information is virtually impossible to find, even using sophisticated text editors. The contractor chose to provide comments only when tasks were completed, feeling that this method provided manageable documentation.

Database access control. In the UDE, contexts are used to determine who may access which database object. The contractor found it possible to establish contexts for this purpose, but had no time to explore the context capability at length.

Immutability. The UDE was found to create immutable versions successfully.

Historical records. By providing comments at "checkin" time, satisfactory *event* documentation was created. To create a readable document for future usage, we feel that this documentation could provide the basic facts for an edited document having additional explanatory text and statements relating different, sequential paragraphs.

Component Change Request Findings

Simple change implementation and tracking. The contractor was able to simulate a CCR issued by the sponsor to the code maintenance contractor. The contractor felt that an electronic CCR was far superior to a hand-written piece of paper, which could get lost or might be illegible. The user can attach a comment to the CCR as a permanent part of the CCR documentation. A new version of the entire code was easily generated.

Multiple change implementation and tracking. Time constraints did not permit full evaluation of this capability.

Reconciliation of multiple branches for new baseline or version establishment. The contractor felt that this procedure could be performed using commercially available software that could be integrated into the BackPlane, time and funding permitting. No further work was done.

Documentation of changes. This topic is covered above in the simple change evaluation report.

Differences between collection versions and component versions. This topic is addressed above under multiple branches.

Correspondence establishment and utilization; Deadline notification. These features of Project SoftBoard were found to be very useful.

Further Evaluation Observations

General. Management involved in the evaluation at the contractor site had a favorable reaction, especially noting the merits of CCR management by PSB. We have been told by a reliable source that the testing contractor has approved a request to acquire the BackPlane and Project SoftBoard, subject to the speed limitations reported below.

Even more encouraging is the reaction of the programmer who performed testing in accordance with the scenarios. Initially he was opposed to the UDE, but when the evaluation was completed, he stated that he liked the UDE and was in favor of its utilization.

Speed limitations. Most of the evaluation was performed using a common but dated super-minicomputer that performs slowly in comparison with current engineering workstations that have been on the market for several years. Atherton no longer markets its software for that hardware platform. In the preliminary evaluation at Argonne, the UDE, running on a state-of-the-art workstation, responded promptly except for the initial startup of the UDE which requires one or two minutes.

Terminal limitations. The UDE was designed to be operated in a windowing mode, although it can also be operated in the ASCII mode. Utilization of the UDE without a bit-mapped terminal that supports windows was determined to be unsatisfactory.

Learning curve problems. During the first half of the evaluation, the contractor perceived numerous problems, but they were resolved as he moved up the learning curve.

Other problems. Other problems are being resolved by Atherton as it refines and extends UDE capabilities. Only one or two minor problems reported by the testing contractor have no currently planned resolution.

Disadvantages. Being compelled to work under the UDE places a burden on the programmer. This burden has been estimated at a 6% overhead of his/her workload. This seems a small price to pay to ensure that proper methodology and procedures are followed.

Conclusions

What has been accomplished thus far is to demonstrate that the UDE has potential for standardizing software development procedures. The BackPlane and Project SoftBoard have the flexibility to be tailored to the degree of procedure enforcement desired by the sponsor. Tools to enforce workflow methodology and to require the user to supply requisite documentation can be devised and integrated into the UDE, as desired. Workflow methodology enforcement will require clever software tool integration. Unfortunately, time and money limitations prevented exercising the UDE in this way.

Similarly, we were unable to test other advantages offered by the UDE such as interoperability, transportability of code, transportability of users and programmers, data interchangeability, and utilization of

CASE tools to aid in development and maintenance of codes.

We believe that the UDE can provide improved control for DoD such that its burden of inoperative and flawed software provided by contractors could be substantially reduced.

Acknowledgment

Work supported under a military interdepartmental purchase request from the U. S. Department of Defense through the U. S. Department of Energy contract W-31-109-Eng-38.

References

1. Paseman, W., "Architecture of an Integration and Portability Platform", Compcon 87, page 1.

2. Black, E., "Software Configuration Management with an Object-Oriented Database", USENIX, January 1989, page 1.

3. Marco, A., and Buxton, J., "The Craft of Software Engineering", Addison-Wesley Publishing Company, Wokingham, England, 1987.

4. Hekmatpour, S., and Ince, D., "Software Prototyping, Formal Methods, and VDM", Addison-Wesley Publishing Company, Wokingham, England, 1988.

5. Department of Defense, "Military Standard DOD-STD-2167A", 29 February 1988.

- END -

DATE FILMED

04 / 04 / 91