Toward a Paperless Development Environment

Bruce I. Blum
Johns Hopkins University/Applied Physics Laboratory
Laurel, MD 20707-6099

This paper describes the process being used to implement a paperless environment for the design, construction and maintenance of computer software applications. The underlying paradigm for this research is not one of artificial intelligence. Nevertheless, the methods used are equally appropriate for the development of AI tools and environments. The paper begins with an overview of the problem being addressed, next describes an environment that offers a solution to that problem, and concludes with a discussion of the methods used to evaluate and improve the environment.

Software Development as Qualitative Modeling

The software process can be viewed as a knowledge representation activity in which analysts implement and maintain a qualitative model of a software application that provides some desired functionality. In a traditional setting, this application model is described in text and diagrams, and the model's realization is encoded in an imperative language that favors performance over flexibility. The process is one of mathematical derivation; complete requirements are necessary before design and implementation can begin.

By reason of historical necessity, computer program creation has been viewed as the central process goal. Most activities are linked to the specification and creation of the programs, and concern for formalism is restricted to the correctness of the programs with respect to their specifications. The validity of the highest level specification is extralogical; requirements are accepted as axioms. Progress is made in computer science by abstracting complexities in the application domain.

The impact of the computer program on software engineering thinking is illustrated in the research on reuse. This concept -- which is heralded as software engineering's major hope for productivity improvement generally is defined in terms of the reuse of code and design artifacts rather than of designer experience and domain knowledge. It would seem, however, that a greater payoff would result if we started at the beginning of the process by formalizing the knowledge about what the software application is to do rather than how it must be implemented.

ACKNOWLEDGMENT

This work was supported in part by the U. S. Navy, Space and Naval Warfare Systems Command (SPAWAR) under contract N00039-89-C-5301, task VMAR7 with the Office of Naval Research (ONR), and by the Air Force Office of Scientific Research (AFOSR) under grant AFOSR-89-0080.

Clearly, this view of software design as knowledge engineering is atypical. It parallels Chancey's analysis of expert systems as qualitative models. As will be shown, however, the techniques used in the research presented in this paper rely on information technology tools. But the distinctions between an information system and an expert system are narrowing. An information system can achieve its goals by retrieval or algorithmic calculations; in the case of an expert system, an inference mechanism is required. The appropriate tool for the task will depend on the environment in which it will be used, the state of the knowledge in the application domain, the context in which the tool will be used, and so on.

Future systems will require both computational paradigms, and there are more similarities than differences between the two system categories. Therefore, the remainder of this paper describes work with an information system environment. It is assumed that this experience will offer insights to AI tool developers regarding the refinement and improvement of their environments.

The Software Process and TEDIUM

TEDIUM is an environment designed to support the development of interactive information systems (IIS). The target products rely on a relatively mature technology, and it is possible to generate applications automatically once the requirements are known. That is, once knowledge of what the application is to do has been established, the application can be implemented without the recourse to a separate programming phase. The greatest risk in this type of application is that the delivered system will not correspond to the needs of the sponsor's environment, i.e., it will be an invalid IIS.

Although both TEDIUM and the fourth generation languages (4GLs) can be used to implement information systems, there are considerable differences between the two approaches in both capability and intent. The 4GL is a vendor-supplied label to a product designed to facilitate report generation, screen creation and database access -- often at a cost in performance or resource efficiency. The goal of TEDIUM, on the other hand, is to supply a comprehensive environment to manage all definition and validation tasks throughout the life of an IIS application.

This work was supported in part by the U. S. Navy, Space and Naval Warfare Systems Command (SPAWAR) under contract N00039-89-C-5301, task VMAR7 with the Office of Naval Research (ONR), and by the Air Force Office of Scientific Research (AFOSR) under grant AFOSR-89-0080.

* TEDIUM is a registered trademark of Tedious Enterprises, Inc.
With TEDIUM, the designers' primary task is one of recording the knowledge about the application to be produced and then generating programs to validate their understanding. The knowledge is stored in an application database (ADB) where it is represented in the form of declarative and procedural specifications, a semantic data model, and textual descriptions. A program generator translates the specifications into executable products. The deliverable documentation also may be generated from the ADB.

Although the ADB is structured as a database, it may be considered to be a knowledge base as well. It contains the formal specifications for the application to be generated, descriptive material that provides a road-map to the specifications, text for use in help messages and documentation, and a framework for initial analysis and design. The ADB specifications project the two dimensions of an IIS. There is a rich semantic data model that establishes the objects to be manipulated by the system and a specification language describes the functions to be performed by the application.

A program generator converts the ADB contents into executable programs that are perceived to be as efficient as custom-crafted code. The generator incorporates a system style that establishes the conventions and default operations used by the application, e.g., every input must be validated against all criteria in the ADB data model and every user interaction must reflect the chosen system style. In a sense, the generator's system style is a knowledge base that dictates how the executable programs should interpret the ADB. Consequently, the generated programs are complete in that they perform all functions implied by the system style and the state of the ADB at the time of their generation.

The executable programs establish a prototype to be tested and validated. Because each program is complete, the validated prototype is also a production-grade product. Moreover, because the ADB incorporates all knowledge about the application under development, new requirements are defined in the context of the existing ADB. Thus, the reliance on an ADB facilitates experimentation and evolutionary development. This implementation process is described with a sculpture metaphor and called system sculpture. One models the system dynamically, adds knowledge to the ADB, and when the results are aesthetically pleasing the development is complete. In contrast, the traditional method of specify-then-implement is consistent with an architecture metaphor.

With this paradigm for IIS development, each design and maintenance activity can be presented as a two step process. First there is the recording of knowledge about the desired product in the ADB. This is followed by the generation and testing of the implementation in the context of the application's intended use. Notice how this process parallels the knowledge engineering that precedes the implementation of an expert system with a shell. In both cases, the process concentrates on formalizing information (knowledge) about the application domain and then testing the validity of that knowledge. Very little emphasis is given to the creation of the code that becomes part of the operational product. That is, where implementation details are important, they are interpreted from the perspective of the application (rather than the implementation) goals.

The software process model used by TEDIUM is similar to the paradigm for developing knowledge-based systems described by Weitzel and Kerschberg. In both cases the emphasis is on recording what is known about the application domain with respect to achieving the desired goals. Of course, a system that generates IIS is not a knowledge-based tool. Neither is it a tool that applies AI for automatic programming, transformation or assistance. Nevertheless, TEDIUM has been shown to be effective in an operational setting for almost a decade. Thus, an examination of its development environment is worthy of further consideration.

TEDIUM as an Environment

TEDIUM is an example of an environment intended to capture everything known about the application under development (or being maintained). In the software engineering hierarchy, an environment is defined to be an integrated collection of tools designed to support a set of methods and practices consistent with a process model. The process model under consideration here is based on knowledge capture with incremental testing. To support this model effectively, the environment must provide a mechanism for the designers to explore and build models of the problem solution without regard for the implementation details. If the environment is comprehensive and complete, then there will be little or no need for the designers to record anything other than what is maintained in the ADB. Moreover, if the ADB truly contains all knowledge about the application, then any alternative or intermediate external representations will be unnecessary. That is, the environment will be paperless.

There are two levels of testing in the creation of a paperless environment. First it must be demonstrated that the knowledge representations are rich enough to achieve their goals and expressive enough to support the designers' conceptual modeling activity. Stated another way, it must be shown that any concepts previously expressed using a paper-oriented paradigm can also be expressed using the representations available with the environment. (Naturally, this may require some training.) Once the validity of the representation scheme is established, the environment must be examined to identify implementation deficiencies. This will suggest extensions to expand functionality and improve the interface.

This section concludes with a brief review of the evidence that the TEDIUM representations are satisfactory for their intended use. The results of more complete studies are summarized simple to suggest the kinds of questions that must be asked and the methods that may be used to provide the answers.

Are the representations rich enough to implement complex IIS applications? All application programs are generated from the contents of the ADB and the information integrated into the generator. The generated programs are treated as object code and never are edited. Thus, the representations in the ADB do describe the IIS products without recourse to other representations. The IIS applications so defined are considerably more complex than those typically implemented with a 4GL. TEDIUM was implemented with TEDIUM as were a major Oncology Clinical Information System (O CIS) and an intelligent interface for database query.
Is the knowledge base structured so that designers/knowledge engineers can explore its contents easily? All information in the TEDIUM ADB is organized as small chunks with links among chunks and descriptions.

Tools are provided for the designer to navigate through the ADB to examine either descriptive text or definitions. The size of an average program specification is 15 lines; it has been shown that these programs provide the functionality of a 60-line MUMPS program or 300-line COBOL program. Thus, the designer can view and screen the operations of a moderately-large program without concern for the implementation details.

Does the representation offer a natural way for the designer to express his intentions? Examinations of the editing history of the ADB contents indicate that the representations are effective for specifying the analysts' intent. In small (100 program) applications, an average of under two edits per program is sufficient to produce a program that is accepted as correct and valid. In the 6,000 program OCIS, studies have shown that the median number of edits for all programs over an eight year period was 10. These edits include development, initial debugging, error correction and product enhancement throughout those eight years.

Are the representations sufficiently unambiguous so that there is no loss in understanding or knowledge when designers interpret the work of others? Experience with a diversity of designers in a variety of settings indicates that the methods and representations are relatively simple to understand. For example, a small OCIS development staff is responsible for maintaining programs based on many definitions that they did not create. In one study of maintenance activity over an 18-month period, it was found that 1,000 programs were added to a 5,000-program OCIS baseline and another 2,000 programs were edited. This work was conducted by a staff of 4.5 full-time equivalents. Because the OCIS is used in a life-critical setting, there is the perception that there are no errors in the revised system.

Thus, one can conclude that the TEDIUM ADB is effective for representing the target class of application with trained users. Consequently, there is a justification for examining how well the environment interface supports the designers in their work.

Improving the Environment

Before considering TEDIUM in the context of a paperless environment, it must be pointed out that the version currently in use was frozen in 1982 and was designed to operate with "dumb" terminals. It now is being modified to incorporate modern features such as full screen editing, windowing, and a hypertext access to the ADB. However, these are simply cosmetic changes that will make the system easier to use; they will not alter any of the basic features. The previous analysis of the environment's effectiveness and expressiveness used data collected with the old interface. What was being evaluated was the representation scheme and not the interface. That 1982 interface now is a barrier for experienced computer users to overcome. But the simple elimination of these obvious defects will not make the environment paperless.

Two types of analysis are proving helpful in the extension of TEDIUM into a paperless environment. The first is the examination of why written materials are created during application development. Information is recorded on paper for several reasons: the dimensions and portability of paper provide features not available with a computer; the processing flow of the computer environment does not match the flow desired by the designer at that time; there is parallelism in the designer's problem solving that is not supported by the environment; the representations offered by the environment are not expressive enough for the problem under consideration; etc.

For several small applications all written notes created during the project were preserved and later reviewed. In almost all cases, the cause for the note was that the representations and flow supported by the environment did not match those used by the designer. Therefore, a notepad was used as an external memory. Obviously, it would be possible to provide an electronic notepad alternative, but this would only mask the real deficiencies. One must extend the environment so that the conceptual models used by the designers in their problem solving will map onto the representations available from the environment. For example, many of the notes were simply high level sketches of the database design recorded on paper because the normal TEDIUM processing flow demanded a more rigorous entry sequence. The solution to this problem was to support more incomplete definitions, to expand the listings of items not yet completely defined, to facilitate the editing of partial definitions, and to enable the use of different entry styles. In only one case was it found that the representations available were not rich enough to allow a paperless solution; as a result, a decision-table notation is being implemented.

The second technique used to refine the environment is an analysis of the changes to the ADB. With an ideal environment, all changes will reflect refinements in the understanding of what the system is to do and how it can be made to perform more efficiently; no changes should be related to errors that could have been recognized by inference from the ADB and system style. In two small projects, all changes in the evolving design were recorded and categorized according to type (semantic or syntactic) and domain (application or implementation). The implementation-syntactic errors identified validity tests to be integrated into the editor. A broad definition for these errors is used. For example, a reference to an attribute-relation pair that is not defined in the data model would be considered a syntactic error (which would ultimately be recognized by the generator).

Obviously, there are conflicts between the ability to work with incomplete definitions while, at the same time, immediately identifying all ADB contradictions and syntactic errors. The goal is to provide immediate feedback (without producing information overload) at the time of recording the conceptual model so that all errors except the application-semantic errors will be screened out. There is, by definition, no way of identifying such errors, and they must be identified by experimenting with the generated prototype.
To facilitate this kind of analysis, the present environment recently has been instrumented to record causal information with each change. At session logon the objectives of the session are identified. During the session, each change is logged with a coded reason. The hope is that these logs will offer insight into the problem solving associated with the software process as well as the appropriateness of the tool for this task. (Balzer uses a similar technique to record explanations for program changes; his goal, however, is to use this information as a guide for maintenance.) All revisions to the current environment now are monitored in this way.

Although some preliminary exercises have been published (e.g., reference 10), work on the refinement of the TEDIUM's interface is still at a very early stage. Thus, what is of interest here is not the extent to which TEDIUM is a paperless environment but the methods being employed to make it one.

**Conclusion**

This paper described a non-AI application that has a striking similarity to knowledge-based design. In both cases the emphasis is placed upon domain understanding and the representation of that knowledge. Programming and implementation issues are deemphasized. While the TEDIUM environment is primitive relative to most AI-based workstations, the methods used in the development and evaluation of TEDIUM are appropriate for the implementors of AI tools. Several questions were identified to help evaluate the environment's effectiveness:

- Does it provide a rich enough representation to achieve the desired outcomes?
- Is it expressive in the sense of being able to state the users' intent reasonably and rapidly?
- Is it compact in its ability to decompose the knowledge into manageable chunks?
- Is it organized so that users can navigate through it to find what is needed?

Given that the representation is satisfactory, further questions were suggested to aid in improving the environment's interface:

- When do the users go outside the environment for knowledge engineering tasks?
- What kinds of errors do the users make that available knowledge could either eliminate or identify?
- How does the environment support knowledge capture, and what are the essential characteristics of this activity?

Most AI tools are seen as objects intended for use by others. However, one of the most natural tasks for a tool is its recursive application for its own refinement. Clearly, TEDIUM in its current state is not the model of a modern development environment. Nevertheless, the methods being used to evaluate and refine that environment can serve as a model for AI tool developers working in another domain. In both situations, the goal is the creation of environments so powerful that their users can develop qualitative models without recourse to paper or any other external media. If we can learn how to do this for ourselves as users, then consider the kinds of environments we should be able to build for others.

**REFERENCES**