On the Transformation from a Fragment-Based Design into a Module-Based Implementation

Bruce I. Blum
Johns Hopkins University/Applied Physics Laboratory
Laurel, MD 20723-6099
bib@aplcomm.juhapl.edu

This paper characterizes the essence of the software process as a transformation from a need to a software product that responds to that need. An application development environment is described that manages this process. It builds a conceptual model for the desired response (in the form of a fragment-based design) and automatically generates commercial-grade, module-based programs that implement the response. A decade of experience with a large application demonstrates the viability of this approach. The paper concludes with some observations about the software process.

Background

I open this paper with a very personal introduction. I began developing information systems in the early 1960s. Since 1980, all development has used the environment described in this paper. Clearly, my view of the software process is strongly biased by this experience. Yet, because I believe that we cannot improve the process until we truly understand it, I have tried to evaluate my experience in the context of the essence (in the sense that Brooks uses the term [Broo87]) of the software process. As a result of this research I have come to question many of the practices that are fundamental to the conduct of software engineering, and I now believe that significant progress will not be made until we break with those practices.

At the highest level of abstraction, we all agree that the software process begins with the identification of a need and concludes with a product in the implementation domain [Blum92]. Because the product is installed in the application domain, it modifies the original need and thereby initiates requests for change [Lehm80]. Thus, the flow shown in the figure is but one cycle in a continuing process of evolution. Two types of model are identified. The conceptual models describe the desired response in application domain terms (which may include domain formalisms). The formal models are formal in the sense that they prescribe the implementation. The conceptual models capture the understanding of those with the need, and the formal models establish exactly how the product should respond to the need.

The essential software process model, shown below, depicts the process as starting with the identification of a need in the application domain and concluding with a product in the implementation domain [Blum92].

The software process must control the tension between deciding what a system is to do and realizing that decision.

The traditional approach to this problem has been to formalize the validity within the specification and then to ignore the validity concern except as far as it relates to the specified product. The thesis of this paper is that this is an oversimplification fraught with difficulty. Moreover, alternatives now are possible. Automation can eliminate the underlying tension when the software process is interpreted as a knowledge-based activity. One approach, which will be described in this paper, involves the modeling of the software goals as fragments. The resulting fragment-based design is then transformed into a module-based implementation. In this way, the form used for the representation reflects the character of the object being represented.

The essential software process model, shown below, depicts the process as starting with the identification of a need in the application domain and concluding with a product in the implementation domain [Blum92].

Because the product is installed in the application domain, it modifies the original need and thereby initiates requests for change [Lehm80]. Thus, the flow shown in the figure is but one cycle in a continuing process of evolution. Two types of model are identified. The conceptual models describe the desired response in application domain terms (which may include domain formalisms). The formal models are formal in the sense that they prescribe the implementation. The conceptual models capture the understanding of those with the need, and the formal models establish exactly how the product should respond to the need.

This work was supported in part by the U. S. Navy, Space and Naval Warfare Systems Command (SPAWAR) under contract N00039-91-C-0001, task VMAR9 with the Office of Naval Research (ONR) and the Naval Surface Warfare Center (NSWC).

0-8186-2905-3/92 $03.00 © 1992 IEEE
This essential model can be further decomposed as three transformations.

From the need to a descriptive statement of a response to the need (i.e., the conceptual model).

From the conceptual model to a formal model that specifies how the need shall be met.

From the formal model to an implementation that satisfies all the conditions of its specification.

The focus of traditional computer science has been on the third transformation. A project begins with a requirements specification, terminates with the successful delivery of a correct and valid software product, and operates in a domain of formal abstractions. Progress is made by removing the domain-specific concerns. Turski put it this way,

Logic—a calculus of formal systems—plays an important role in software development from specification to implementation.... That logic plays no discernible role in descriptive theory formation—an act always serendipitous and contingent on invention—should not worry us too much, as long as we do not wish to claim possession of the philosopher's stone. [Turs85, p. 400]

From the sponsor's point of view, of course, the software process includes both the descriptive theory formulation (i.e., conceptual modeling) and the formal modeling of logic. For every need there are many classes of conceptual model, each of which represents an adequate response. Similarly, for every conceptual model there are many classes of formal model, each of which specifies an appropriate response. The formal model that initiates the implementation activity is but one of many potentially valid responses to the identified need, and there seldom can be any assurance that it is the best response. The serendipity and invention occur in the first two transformations, and distortions here will be magnified in the delivered product. Furthermore, because we know that the cost to maintain a system exceeds its initial development cost, it follows that the formalisms that begin the third transformation will be subject to change. Using a mathematical analogy, we know that once we use our theorems we will want to change our axioms.

Resolving the Tension

The essential software process divides the models into two categories: conceptual and formal. The conceptual models can employ formalisms (e.g., there is an algorithm for computing FICA payroll deductions), and the formal models may contain extralogical requirements (e.g., timing and storage constraints are properties of the delivered system). The main distinction between them is that the conceptual model derives from the users' perception of the need and understanding of the problem domain, whereas the formal model exists in a well defined and highly abstract universe. The former is a description of a desired response to a need in a particular domain, and the latter defines the essential properties of any product that implements the response. Given the inherent uncertainty in the process, there always is a danger that the interpretation of the need (and therefore the desired response) is inaccurate and that the formal specification of the response is inappropriate (i.e., invalid).

Thus, there are two dimensions of uncertainty: understanding the real problem and detailing an appropriate solution. Barwise, speaking of the match between mathematical models and physical reality, describes this tension as the Fallacy of Identification.

The axiomatic method says that our theorems are true if our axioms are. The modeling method says that our theorems model facts in the domain modeled if there is a close enough fit between the model and the domain modeled.... As a philosopher might say, applied mathematics may not guarantee knowledge of the facts about the physical world, but it can lead to the next best thing—justified true belief. [Barw89, p. 847]

The formalisms of software engineering offer justified true belief, but with a major limitation. The physical world represents an external, testable, and repeatable universe, and the goal of the modeler is to define its behavior. Software design, on the other hand, is a response to real or perceived needs in an adaptive, dynamic, and social environment. Although wrong answers can be isolated, there seldom are "best" answers. Our goal is to satisfice, and we must acknowledge that our perceptions of need change as our experience grows.

We are on the horns of a dilemma. Identifying the need and a response to it is a human cognitive activity; implementing the product given a specification is a logically constrained task. Both may be viewed as creative problem-solving activities in which the former uses judgment to identify errors, and the latter bases decisions on justified true belief. But how do we merge these two models—one based on feelings, the other on precision? If we are vague in our specifications, our formal models will be meaningless; if we are wrong in our specifications, our
products will be useless.

In the early days of software engineering this dilemma was of little consequence. The perception was that most requirements could be defined precisely, and the process adopted the specify-first approach associated with hardware. The process begins with a requirements document and then builds a product that is correct with respect to that document. The specify-first model was first formalized in a waterfall model [Royc70]. As applications became more complex, uncertainty regarding the specifications' validity fostered techniques to improve confidence in the requirements before building the application. Rapid prototyping was used to refine the definition of the response [GoSc81], and the spiral model integrated prototyping-as-risk-reduction into the waterfall flow [Boeh88]. Incremental development, which divides the application into smaller (and therefore easier-to-implement) builds, offered another technique for risk reduction [MiLD80, Gilb88]. Each of these process models preserves the concept of "first we specify, and then we build."

Recent attempts to improve the specify-first approach include the use of formal methods and automated tools. Formal methods prove that each refinement of a specification preserves correctness (e.g., VDM [Jone86], Z [Spiv89], and the Cleanroom [MiDL87]). These methods typically have limited automated support. In contrast, in the area that emphasizes automated support (i.e., Computer-Aided Software Engineering, or CASE), the tools tend to mechanize manual tasks without increasing the level of formalism. Consequently, it seems unlikely that a specify-first process model will resolve the verify-validate tension.

There are alternatives to the specify-first model. In the operational approach, the specification can be executed as a prototype [Zave84]. In this way the developers can experiment with the specification's behavior before committing to its implementation. Balzer proposes a two-phase model in which one first accepts the prototype specification and then optimizes its performance by applying behavior-preserving transformations [Balz85]. In this specify-optimize model, the specification is preserved, and all maintenance is done at the specification level. A trail of the specification refinements is used to facilitate optimization after the specification has been modified. Neighbors has developed an environment, DRACO, that allows the users to model application domains (which define the principal application concepts) and modeling domains (which contain the transformations that produce efficient implementations) [Neig89]. Finally, there is the megaprogramming model in which application development is performed at the component (as opposed to the program) level, thereby maximizing the benefit of reuse [Boeh90]. As with the operational approach, components support prototype experimentation and can be refined to become production-quality applications.

The approach to be described in the next section is an instance of the second type of process model. It provides an automated environment that will:

Maintain a conceptual model that allows the designer (i.e., the environment user) to express concepts about the need and automated responses to it.

Transform the conceptual model into a formal computational model that exhibits the desired behaviors in some target machine. (The conceptual model serves as a specification for the delivered product.)

Support experimentation with incomplete and partial conceptual models (i.e., prototypes) and also maintain the complete model throughout the application's effective life.

Exploit the availability of automation to improve the software process.

In summary, the environment permits the designer to model the application at the conceptual rather than at the formal level. That is, he records what the software product should do in the context of his understanding of the application domain needs. Implementation details are hidden except as they may affect performance.

There is little experience with this level of conceptual model. For example, to the database community an entity-relationships model (ERM) is a conceptual model. It is conceptual in the sense that it is richer than the logical model it helps derive, but it also is very informal. Although there is a formal syntax for the ERM, the tokens used within the model are informal. Indeed, one of the advantages of the ERM is that it suppresses detail in favor of conveying information. One cannot take an ERM and generate a complete DBMS logical model; it lacks format definitions for example. Moreover, the ERM provides no information about the use of the data in the model; for this one may have to reference a data flow diagram (DFD) and its associated minispecs. However, what I refer to as a conceptual model includes everything known about the application under development. It is organized as a comprehensive, integrated database (e.g., it links the information that would be found in the ERM with that displayed in the DFD). Because that database is to support reasoning about the application, the conceptual model must be formal; because the conceptual model also is to be used for program generation, it must be complete. How this has
been accomplished is the topic of the next section.

An Application Development Environment

Since 1980, TEDiUM* has been used to create and maintain interactive information systems [Blum90]. The first version of TEDiUM, which is still in production use, was frozen in 1982, and a new version, which uses an extended representation scheme [Blum91], is under development. Both versions have been developed with TEDiUM, and the description that follows merges the features of both systems.

The basic idea behind TEDiUM is simple. It maintains an application database (ADB) which contains almost everything known about the application under development. The ADB is segmented into applications, and every item specified within an application (e.g., programs and relations) can reference all other items specified for that application. This sharing of specifications within the application avoids redundancy. To facilitate the sharing of specifications, the ADB is organized as fragments, which can be composed to form other, more complex fragments. For example, a program is composed of commands, each of which is stored as a fragment. A command may reference a relation, which is specified as a set of fragments that name the relation, define the key and nonkey attributes, identify relationships with other relations, and so on. Each attribute, in turn, is specified as a set of fragments that name it, define its format and validation criteria, etc. The environment provides an interface that hides the internal structure of the ADB from the designer and displays fragments at the appropriate level of detail. From the designers' perspective, the specification appears to be a conceptual model; internally, the environment manipulates knowledge at the fragment level.

Although the ADB provides a holistic, fragment-based specification of the application, the designer must navigate through the ADB at a higher level of granularity. To support the designer, the knowledge in the ADB is grouped into two categories.

Objective. This is formally defined knowledge used to generate the implementation. Examples are the program specifications and the data model. There also are derived objective data, such as cross references between programs and the data they read or write.

Subjective. This is descriptive knowledge that is used to document the application or to provide a structure to the design. Subjective knowledge is used to describe properties of the application, but it has no effect on the implementation.

The objective knowledge is used by the program generator to create a production-quality implementation. The subjective knowledge is used to guide the perusal of the ADB and the generation of documentation. Links among the various fragments (both objective and subjective) are maintained by TEDiUM. A specification listing, for instance, will combine both categories of knowledge.

The designers maintain the contents of the ADB. Their primary items of interest are the program specifications and the data model, which consists of relations (tables), attributes (elements), relationships, and complex structures. Internally, these specifications are stored as independent fragments. At generation time, the fragments are assembled and transformed to produce the desired products. For example, a program specification or a table definition may reference an element. The definition of that element, including its type and validation criteria, is available in ADB fragments. Thus, the generator has immediate access to knowledge about the element necessary for the local generation task. Because definitions are shared, changes to a fragment will affect references to that fragment throughout the application.

There are several features of this fragment-based organization that must be emphasized.

Definitions in the ADB are global unless they are restricted explicitly through the specification of a context for their use.

All knowledge in the ADB is available to both the designers and the program generator. Display of this knowledge is done at a level that presents a unified concept while hiding extraneous details.

As will be discussed below, the program generator has access to the knowledge necessary to produce a complete and efficient implementation. This knowledge, initially organized as fragments, is integrated within the generator.

There is no inherent structure imposed on the ADB contents. Its organization is viewed as a holistic network of fragments, and there is neither a top nor bottom, an inside nor outside.

The environment does not establish an architecture for the generated application. Although the subjective contents may identify a structure to be used for documentation, this organization has no effect on

* TEDiUM is a registered trademark of Tedious Enterprises, Inc.
the implementation.

The contents of the ADB (and hence the implementation) is adaptive. Altering fragments in the ADB will affect the implementation of all computational units that utilize those fragments.

The ADB relies on logical inference but does not support any diagrammatic notations. (See "Diagrams Considered Harmful" in [Blum90].)

In a crude sense, the organization of the ADB mimics the schema and chunks found in the models of human memory. There is a holistic structure in which the fragments are linked to each other. New fragments must be associated with existing fragments, and they are assimilated automatically. Fragments can be composed of other fragments, or they can reduce to atomic concepts. Finally, knowledge can be represented in procedural or declarative fragments. I assert, therefore, that this holistic, fragment-based conceptual model is ideally suited for capturing the designers' understanding of what a software product should do.

The conceptual model in the ADB is of limited value unless it can be transformed into an efficient computational model that will execute in the target computer. To accomplish this, TEDIUM accesses three domains of knowledge.

**Application Knowledge.** This is the explicit knowledge of what the application is to do.

**Application-Class Knowledge.** This is knowledge of the implicit behaviors of all applications of this general class (e.g., for an interactive information system, all inputs must be validity tested, all interactions must have help responses).

**Software-Tool Knowledge.** This is the knowledge of how to refine conceptual models as production-quality implementations (e.g., modeling domains in the DRACO terminology [Neig89]).

The application knowledge is stored in the ADB, and both the application-class and software-tool knowledge are integrated into the program generator (which is itself a TEDIUM application). The application knowledge specifies what the implementation must do, the application-class knowledge adds to this the implicit behaviors expected of all applications of this class, and the software-tool knowledge transforms this into an efficient implementation.

Every generated program is both efficient and complete (i.e., it incorporates all the explicit and implicit properties expected from programs of this class). Therefore, every program is of production quality. Application development takes advantage of this fact by employing an iterative method called system sculpture. The method derives its name from a sculptor metaphor. Unlike the architect, who must specify completely before he builds, the sculptor modifies his work until—when it is aesthetically pleasing—he considers it finished. With TEDIUM, the designer develops a portion of the conceptual model, generates the programs that implement that model segment, experiments with this prototype, and—once all the desired properties are observed—makes the program available for production use. The designer always works with the conceptual model (i.e., the specification); when that model is accepted as valid, the task is complete (i.e., "debugging" is done at the conceptual rather than the implementation level; because the programs are generated from the conceptual model specifications, their correctness is ensured). Thus, the software process is reduced to one of conceptual modeling, and the evolution of the product is viewed as the revision of the conceptual model. In this paradigm, the specification does not describe what is to be built (even though it may not work). Instead, the specification describes what has been delivered, is known to work, and will be maintained.

This method may be clarified by recourse to a simple example. Consider the command

**Input Patient_Id.**

This is a fragment within a larger program fragment. Also within the ADB are fragments that indicate that the external name for Patient_Id is Patient Identifier, that it is an 11 character numeric field, and that the standard description of the identifier (also in the ADB) should be used in response to a request for help. The application-class knowledge within the program generator specifies that the prompt for an input should use its external name, perform all validity tests identified in the ADB (e.g., tests for type and length), and respond to a request for a help message. The software-tool knowledge in the program generator indicates that the Input command should be decomposed into commands that first set up to receive an input, then test for validity, and then perform the final process (e.g., update the state or abort). Each of these commands will be decomposed in turn until they reduce to a primitive that transforms an ADB pattern into a pattern in the target language. Without going into the next level of detail, it should be obvious that the knowledge in this paragraph can be used to transform the above Input command into a block of code that will prompt for Patient Identifier and accept a valid value of
Patient Id. Moreover, if the links between the definition of Patient Id and the fragments that use it are maintained, then any changes to the specification of Patient Id will be propagated automatically through the regeneration of the linked fragments.

In summary, TEDIUM separates the conceptual modeling, which relies on human judgment, from the process of creating a software product, which can be automated. The designer models his understanding of how the software should respond to the need as fragments in the ADB, the designers of TEDIUM specify the application-class and software-tool knowledge to be included in all products, and the program generator transforms the fragment-based design in the ADB into a module-based implementation that will execute in the target environment. Clearly, these knowledge-based methods are not valid for all application domains. They require a well-understood class of applications to capture the application-class knowledge, a relatively mature technology to automate the generation of efficient implementations, and a collaborative setting that will commit resources before the final specification is available. Nevertheless, TEDIUM does instantiate a non-specify-first process model, and—as will be demonstrated next—it has been validated in an operational setting.

Experience in the Use of TEDIUM

Because the methods used by TEDIUM are so complex, it is difficult to present more than an overview in a paper of this length. Nevertheless, it is possible to demonstrate by means of an example that they do work. The largest application developed with TEDIUM is the Johns Hopkins Oncology Clinical Information System (OCIS) [EnLB89]. Considered one of the most complete information systems for tertiary care, the system operates on five networked computers with a distributed database and more than 200 terminals located throughout the Center. It provides online access to 25 million data points, which represent a half-million days of care for the 20,000 patients treated at the Center. OCIS was designed to support the clinical functions required by the Center without duplicating any of the services available from the hospital systems. Its network is linked to the Johns Hopkins Hospital computers that manage admission-discharge-transfer, inpatient and outpatient billing, the clinical laboratory, and radiology reporting. OCIS provides the following functions.

Clinical data display. OCIS collects and manages virtually all clinical data that can be displayed in a table or plot. Inputs come from various Hospital and Center laboratories. Some are transferred automatically; others may be entered manually.

Transfusion services. Because the toxicity of the antitumor therapy may result in very low platelet and white blood cell counts, the Center maintains its own transfusion services. This involves HLA typing, matching the available products to the patients in need, and evaluating the effect of each transfusion.

Daily care plans. Because most patients are treated using well established plans (or protocols), OCIS produces daily care plans that identify the tests, therapies, and actions for the current day based on previous patient history, current patient status, and general therapy plan. Functionally, this is similar to the service provided by the ONCOCIN expert system [HiSB85].

Pharmacy. OCIS includes a pharmacy system for inpatients that supports the special needs of medical oncology.

Clinical research and the Tumor Registry. In addition to maintaining and exchanging data for internal and external clinical trials, OCIS supports a registry of the 50,000 Hospital patients diagnosed as having cancer.

Administrative functions. Although OCIS was designed to augment the functions provided by the Hospital systems, it is used for a variety of administrative purposes including scheduling, planning, and auditing.

From this cursory review, it should be clear that OCIS is a comprehensive, complex application. Each day more than 100 staff members rely on OCIS in making life-threatening medical decisions, and they perceive the system to be error free.

Development of OCIS began in the mid 1970s, and the conversion to a two-computer, TEDIUM-based system took place between 1980 and 1983. During the initial years of operation, growth was constrained by limitations in the target operating system. This problem was corrected in 1986, and the application has grown at an annual rate of 10% since that time. The present system contains 15,000 MUMPS routines or approximately a million lines of code. The MUMPS programs are generated from the TEDIUM specifications, and the designers need only maintain the specifications. The table illustrates the growth of OCIS over time. It identifies the date the data were recorded, the size of the ADB in terms of the number of programs, tables, and elements,
the cumulative number of effort years devoted to OCIS at the recorded time, and the gross productivity as the number of production programs per effort day (P/ED, which is the number of production programs divided by the cumulative effort in days). These data demonstrate that the system is large and that productivity is high.

<table>
<thead>
<tr>
<th>Date</th>
<th>Programs</th>
<th>Tables</th>
<th>Elements</th>
<th>Effort</th>
<th>P/ED</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aug. 1983</td>
<td>3,662</td>
<td>848</td>
<td>2,025</td>
<td>19</td>
<td>.66</td>
</tr>
<tr>
<td>Dec. 1984</td>
<td>5,024</td>
<td>1,045</td>
<td>2,398</td>
<td>24</td>
<td>.93</td>
</tr>
<tr>
<td>Sep. 1986</td>
<td>5,541</td>
<td>1,375</td>
<td>2,613</td>
<td>33</td>
<td>.75</td>
</tr>
<tr>
<td>Jun. 1988</td>
<td>6,605</td>
<td>1,635</td>
<td>2,924</td>
<td>40</td>
<td>.73</td>
</tr>
<tr>
<td>Dec. 1989</td>
<td>7,695</td>
<td>1,920</td>
<td>3,281</td>
<td>48</td>
<td>.71</td>
</tr>
<tr>
<td>Feb. 1992</td>
<td>9,257</td>
<td>2,273</td>
<td>3,823</td>
<td>59</td>
<td>.70</td>
</tr>
</tbody>
</table>

Growth of OCIS

The data in this table count certain classes of fragment in the conceptual model. The justification for working with a conceptual model is that it should be effective for representing an application under development. Studies show that the functionality implicit in a 15-line TEDIUM specification would require a 300-line COBOL program to implement [Blum90]. Obviously, a compact specification will be easier to write and maintain, but what can be said of the difficulty in creating valid specifications? One measure of the expressiveness of the conceptual model (i.e., the ability of a designer to express his or her computational intent) is the number of times a program is edited prior to its use. If this number is low, then it can be concluded that the designer has had little difficulty in expressing the intent. (If, on the other hand, the number is high, no conclusion can be drawn.) In studies conducted from 1984 through 1989 the median number of edits was 10. This number includes all changes to a specification for debugging and maintenance—corrective, adaptive, and perfective—throughout its active lifetime. The low number suggests that the conceptual model offers an effective representation for specifying what the application must do.

Next we consider what special training is required by the users of the environment. Three groups of people were responsible for developing OCIS. One group, composed of seven software professionals, supported the initial development and left the project once OCIS became operational in 1983. A second group consisted of four Oncology Center employees, two of whom had no prior computer experience. These people participated in the initial development and assumed responsibility for the system's evolution. A third group of four developers was hired in 1990 and 1991; two of these people began work as computer operators. In summary, half of the developers learned to use TEDIUM on the job, because most of the original system designers were no longer available, the staff had to modify specifications they did not design; and—in 1991—half the development team was new to the project. Nevertheless, the complete conceptual model in the ADB facilitated continued high productivity. An analysis of the change data show that:

70% of all program specifications were edited at least one year after their initial design. Of these edits, 44% were made by persons other than the original designer.

26% of all table specifications were edited at least one year after their initial design. Of these edits, 38% were made by persons other than the original designer.

These data show that designers have little difficulty maintaining programs that they did not design and that the ADB is effective in transferring knowledge from one generation of designer to the next.

Finally, there is the question of the OCIS architecture. The presentation of the previous section stated that the structure of the application adapts as the ADB evolves. If the conceptual model is truly holistic, then one would expect changes to ripple throughout the ADB. Although some of these effects will be transparent to the designers, many application extensions will require changes to other portions of the application to take advantage of them. The following illustrates the degree to which the ADB fragments are linked.

In the 13-month period from January 1991 to February 1992, 1049 programs were added to OCIS and another 3045 programs were edited. That is, 44% of the 9257-program system was affected during that time. This work was accomplished by a staff averaging 6 FTEs, and the system was considered to be error free throughout this period.

Another analysis showed that half the programs invoked by the programs in the 1983 system were defined in 1984 or later. For the programs defined before 1987, more than 10% of the programs they invoke were defined at least two years after they were defined.

Approximately 20% of the tables read by the programs in the 1992 production system were not defined until at least a year after the program was defined.

Certainly, these data indicate that OCIS possesses a highly dynamic and integrated conceptual model. The
analysis also suggests that most changes have global effects.

In summary, we may conclude that a decade of experience with this environment demonstrates that:

Program generation from a fragment-based design can be used to develop and maintain a large, complex information system that is perceived to be efficient, reliable, and error free.

When design reduces to conceptual modeling, productivity is high. In this example, the system grew at the rate of .64 programs per effort day during the most recent five years of development.

The availability of complete and up-to-date specifications in a holistic ADB permits the ready modification of specifications developed by others and the transfer of application knowledge from one generation of developers to the next.

The fragment-based, conceptual model is deeply integrated. It is robust in the sense that small conceptual changes require only small changes to the specification, but the data also suggest that most enhancements to the application initiate global changes to exploit the new features.

Concluding Observations

This paper began with a review of the software process (i.e., the domain of software engineering) and postulated that (a) a specify-first approach could never be extended to support the process fully and (b) the process begins with conceptual modeling. Experience with an alternative method was described. It represents the conceptual model as fragments (which are organized like chunks in human memory), and there is a program generator that can transform this fragment-based design into a module-based implementation. The environment is based on a separation of concerns. The designers model product behavior in the context of the application needs, and the environment transforms this model into an efficient implementation by reusing existing application-class and software-tool knowledge.

Much has been left out of this paper. For example, how is a fragment represented internally, and how are fragments linked? What is the organization of a program specification? What does it mean to have a 9000-program application? Some of these questions are answered in [EnLB89] and [Blum90], others are the subject of ongoing research. But the goal of this paper was not to provide a comprehensive introduction to one specific environment; rather it was to provide a high-level overview of an alternative approach to software development. Because this technique exploits a new paradigm, it confronts the reader with a strange vocabulary. One could define these terms using a standard software engineering vocabulary, but that would mislead rather than instruct. Therefore, I conclude this paper with a few observations about the central theme of this paper: the essence of the software process.

The process involves the implementation of a conceptual construct (see [Broo87]), and the form of the initial and target constructs are very different.

The conceptual construct is evaluated on the basis of validity (i.e., is this the right system), and the implementation is verified (i.e., is the system right [Boeh-84]). There is an essential tension between these concepts (cf. the Fallacy of Identification).

It is possible to build and maintain complex conceptual models that are used to generate large, complex, real-world applications. If a fragment-based model is used, then an automated support environment with a knowledge-rich generator is mandatory.

There are many levels of reuse, including program libraries, object libraries, and abstract programs. For the TEDIUM approach, it is sufficient to have the generator reuse knowledge of the application class and of software tools.

Although the software process can be interpreted as a knowledge-based activity, much of the knowledge regarding how software currently is constructed is irrelevant outside the specify-first paradigm.

One final note of caution. The operational version of TEDIUM was frozen in 1982 and generates programs only in the MUMPS programming language; the new version of TEDIUM is not yet operational. Thus, TEDIUM is not proposed as a solution to the overstated “software crisis.” An analysis of experience with its use, however, does provide insight into a process that many consider to be in a state of crisis.

References


