Automating the software development process

by G. F. Hoffnagle
W. E. Beregi

Demand for reliable software systems is stressing software production capability, and automation is seen as a practical approach to increasing productivity and quality. Discussed in this paper are an approach and an architecture for automating the software development process. The concepts are developed from the viewpoint of the needs of the software development process, rather than that of established tools or technology. We discuss why automation of software development must be accomplished by evolutionary means. We define the architecture of a software engineering support facility to support long-term process experimentation, evolution, and automation. Such a facility would provide flexibility, tool portability, tool and process integration, and process automation for a wide range of methodologies and tools. We present the architectural concepts for such a facility and examine ways in which it can be used to foster software automation.

Software development challenges

Through the years, measurable gains have been made through the application of advanced software development techniques and tools. Nevertheless, we see the need for further improvements. Part of the underlying problem can be traced to a demand for software that existing resources cannot satisfy.[1] Another factor is a unique dilemma in the evolution of software technology: As the technology to develop larger and more complex software systems is realized, the use of these systems and the demand for increasingly complex systems exposes problems that exceed the capabilities of the technology.

The fundamental challenge facing software developers today is the achievement of uncompromising quality and increased productivity, where we define *quality* as the absence of any form of defect. Producing quality software under this definition is a challenge to the ability of system developers to interpret user requirements and transform them into a reliable, effective, and appropriate product. Productivity is a challenge to the ability of system developers to achieve the goal of quality with the minimum expenditure of resources.

Traditionally, efforts to improve software quality and productivity have centered around four approaches:

- Definition and separation of the software development life cycle into phases or steps, to control complexity and measure progress
- Development of methodologies for each phase to define the procedures by which the objectives of the phase are realized
- Development of supporting tools to assist in intellectual control of software volume and complexity and enforcement of methodological procedures and standards
- Development of software development support environments to integrate, monitor, and control the life cycle

The phased or staged development life cycle is based upon a divide-and-conquer concept, meaning that a task is partitioned into smaller units of work with

specific, understandable milestones for purposes of control and evaluation. Lack of consensus on the degree of phase granularity, overlap, concurrency, and emphasis in software development has led to a proliferation of life-cycle models. For example, the U.S. Department of Defense (DoD) life cycle is a complex set of multiple and often concurrent phases based on standardized documentation milestones.[2,3]

As the size and complexity of software systems increased, the phased approach to development based solely on such simple phase objectives as documentation milestones became insufficient to control and guarantee the quality and uniformity of the final work products. Methodologies were developed and introduced to identify the procedures and intermediate checkpoints that guided the developer from initiation of a phase to the final work products. In simple terms, *methodologies* attempt to define to the developer where to begin, where to end, and how to go from start to finish.

The introduction of *tools*, both manual and automated, served to augment and reinforce methodologies. Tools, especially if automated, provided additional intellectual control over the increasing volume and complexity of development data, which had to be collected, analyzed, and documented. Tools also supported methodological procedures and techniques. Tools, notations, and methodologies are often intricately linked and viewed synonymously, even though they are separable entities. Program design via stepwise refinement can be represented by, for example, pseudo-codes and Nassi-Schneiderman Diagrams (NSD).[4] NSDs were proposed to represent structured programs, but numerous other tools have been developed to support different notations.

Methodologies and tools have evolved in parallel with software engineering and hardware technology. From basic assemblers to optimizing compilers and from punched-card input to computer-aided graphics design, the availability of methodologies and tools produced by vendors and academic institutions has increased in recent years.[5] The proliferation of tools, techniques, and vendors in the marketplace is indicative of the magnitude of the software engineering challenge. The demand for solutions is coupled with a lack of consensus on software engineering approaches. It should be clear that any specific approach adopted today will quickly become obsolete unless it fundamentally supports experimentation and evolution of process, tools, and methodologies.

Within recent years, there has been a shift in research and development emphasis from methodologies and tools supporting low-level design and implementation to the areas of requirements, high-level design, and maintenance.[6] This shift has been partly delayed by a seeming intractability of these phases. However, the development of more effective cost and benefit measures across the life cycle through error tracking and productivity analysis allows the re-evaluation of life-cycle costs and a reassessment of accountability.[7] The evidence gained has resulted in the realization that the requirements, high-level design, and maintenance phases present significant opportunities for improvements in quality and productivity.

A significant challenge remains in the area of maintenance.

Emphasis on quality in the requirements definition and high-level design is moving toward defect prevention, as opposed to defect detection. Defects introduced early in the life cycle that are propagated and detected in later phases are significantly more costly to detect and correct.[8,9] A major concern thus far in the 1980s has been and continues to be that of tools and techniques to reduce the propagation of redundant, incomplete, and inconsistent requirements.[10,11]

A significant challenge, however, remains in the area of maintenance (error detection and correction and product enhancement), which consumes a major portion of life-cycle costs.[7,8] In what is often called the old-code problem, methodologies and tools oriented toward the development of new products are often found to be inappropriate or difficult to use unless the base product was developed using those same approaches and the original intermediate work products still exist. Although useful results are being realized by the reverse-engineering of existing code into abstractions representing the design or requirements,[12] other approaches center on discipline and formality for incremental enhancements to products.[13]

The introduction of tools and methodologies within the life-cycle phases can be justified[7] and shown to improve the quality and productivity of software.[14] However, the chronic software development challenges persist, despite the introduction and application of advanced methodologies and tools.[10] Apparently, not all of the factors involved in achieving significant quality and productivity gains have been discovered and applied. Also, development systems must support change, experimentation, and evolution as those factors are uncovered.

An examination of the current state of the art in software development environments shows that they are characterized by a proliferation of methodologies and tools that are essentially disjoint. In these environments, tools are brought together under a common invocation interface but remain essentially unrelated, sharing neither the data nor the methodologies that they support. The span of life-cycle phases supported by some facilities is very narrow. Similarly, process and tool expertise may be concentrated in a few organizations or individuals. This narrowness presents a challenge to further integration of facilities and to the collection of expertise to offer broader applicability and benefits. In addition, methodologies and tools are often developed, supported, and intricately tied to a single operating environment, which is usually the target system architecture for which the products are being developed. These discontinuities, a reflection of the phased approach to software development within multiple support environments, manifest themselves in several ways.

The span of life-cycle phases supported by some facilities is very narrow.

Tools and the users of tools may not be easily moved from one system environment to another. Because tools must be rewritten for portability, their maintenance is compounded, and users must be retrained on the multiple interfaces and environments.

Tools use many different data organizations and data base management systems to store their work products. As a result, tools do not readily share work-product data. An even greater difficulty is that tools do not support the view that the work products are interrelated results of an integrated life cycle, but rather a set of unrelated results from the phases.

Methodologies are similarly unrelated in that they do not have a common view of data as related work products and common conceptual models. The result is that the transition from one software development phase to another is inhibited.

These symptoms point to a need in current software development support environments to integrate and manage the development process so that appropriate supporting methodologies and tools can be selected or developed at either the location or project level. The development environment must also support the experimentation and evolution required to select and specialize the methodologies and tools. The integration and control of such a process implies that all process control functions must be removed from methodologies and tools. Also, the creation, flow, and relationships of work-product data must be defined and managed from one phase to another, backward as well as forward. The use of tools and the methodologies that tools support should be permitted only under conditions that are controlled, orderly, integrated, complete, and logical.

These symptoms also point to the need for software development environments and tools to be portable across system and data base environments. The lack of a portable support environment results in duplicated tool development and maintenance costs, architecture adaptation costs, and reduced adaptability of users to changing needs. The cost of adapting tools from one environment to another is often prohibitive and minimizes their use and effectiveness.

Current software development support facilities also indicate a need to provide for flexible process and tool evolution, for variation, and for experimentation. The capability must exist for local variations in process and the selection of appropriate tools to reflect differing product, organizational, and development environments. Software technology and implementation support bases do not remain stable over time. A software development support facility must be capable of flexibility as the process and tools evolve, either incrementally or radically. A lack of flexible process and tool evolution can lead to obsolescence or increasing maintenance costs.

The goals of increased programming quality and productivity suggest a need for an integrated software engineering support environment. Such an environment makes possible the explicit definition and control of a development process and its supporting methodologies, so that portable tools can be developed, integrated, tried, changed, and used to automate the process.

Even considering the great value of the existing work, productively developed quality software requires improved software development environments.

Motivation

We have addressed challenges facing software development organizations today. Under the heading "Rationale" in the following section, we present ways in which the industry has responded to those challenges. In the section under "Goals," we show what we believe can be achieved through a new approach to those same challenges.

Rationale. The challenges previously cited are limitations under which software developers are working today. We are motivated in our work on software engineering support facilities by a knowledge of what has already been accomplished and by the desire to answer more of those challenges.

Even considering the great value of the existing work, we believe that productively developed quality software[15] requires improved software development environments. Current environments[16–20] do not contain all of the needed software development capabilities of automated process control, integration, portability, and flexible process and tool evolution. Our ideas for a software engineering support facility are intended to allow the specification of and automated support for a well-defined and controlled software development process. The facility would also allow us to enhance the use of existing and evolving methodologies and tools. The result would be improved quality and productivity.

One of the earliest software development environments is the Software Requirements Engineering Methodology (SREM),[21] developed by TRW for the U.S. Army. SREM consists of a processable formal language, the Requirements Specification Language (RSL), and a validation technique, the Requirements Evaluation and Validation System (REVS). SREM is concerned with the software requirements phase. It incorporates the concept of an integrated data base surrounded by interrelated tools to be used within the structure of an informal process control mechanism.

The Computer-Aided Development and Evaluation System (CADES),[14,18] developed by International Computers Limited (ICL) of the United Kingdom, supports a broader segment of the life cycle, from high-level design through maintenance. CADES is an operational software engineering support system that addresses most of our objectives. It was used on the VME/B operating system project.[14]

Earlier software development support environments tended to use one tool or methodology that concentrated on one phase of the life cycle. More recent support environments[22,23] are beginning to span a greater portion of the development life cycle. However, these newer facilities are more like tool kits consisting of loose associations of individual tools and lacking a common and integrated view of development data and a rules-driven process control mechanism.

The most visible contributions toward software development support facilities are those of agencies within the U.S. Department of Defense (DoD). The Ada® (a registered trademark of the U.S. Department of Defense) Program Support Environment (APSE) calls for the integration of conventional software tools into a framework that is sufficiently open-ended to accommodate a wide variety of programming methodologies and automated software tools currently available or unused in military systems.[24] One of the primary objectives of Ada, besides its applicability to the implementation of complex, real-time software for embedded computer systems in military applications, is its intended machine independence. This provides portability across the wide variety of computing systems used in the DoD. Ada portability is also extended to APSE, the implementation of

which is being guided by the architecture detailed in the Common APSE Interface Set (CAIS).[19] CAIS establishes interface requirements for the transportability of Ada tool sets to be used in Department of Defense Ada Programming Support Environments (APSEs). The prime objective of the APSE is portability, which in turn provides the secondary benefit of continued APSE tool evolution. In contrast to our architecture, APSE objectives do not explicitly support process and tool integration or automated process control.

A more comprehensive proposal for an advanced software engineering environment is set forth in a U.S. Navy document.[20] This document defines the Naval Standard Software Engineering Environment (NSSEE) in which to build at least one integrated tool set that operates through all phases of the life-cycle model. The NSSEE defines a doctrine preliminary to a formal specification of acquisition requirements. The doctrine recognizes the need to identify all the activities and phases in the life cycle, the interfaces and control between them, and appropriate methodologies and tools to fit within the entire framework. This is in addition to portability and an Ada support environment.

The increasing demand for less expensive software products of higher quality will require improved production resources and increasing numbers of qualified personnel in the future. Given the predicted shortfall of qualified software personnel,[20] facilities must be built to automate software development and improve the productivity of the existing workforce. The software engineering support facility addresses that challenge.

Goals. The requirements for a software engineering support facility directly reflect the objectives of the architecture. Here, we explore the goals, objectives, and requirements for both the architecture and the resulting facility.

The goals of a software engineering support facility are to provide an integrated environment for the support of software development tools and software development process automation.

One of the elements we must provide for is variation and evolution of the process and the tools. The facility must be flexible so as to accommodate local process and tool variations. The facility must run in many operating environments, each using different processes, life-cycle methodologies, and tools.

The facility must also be flexible so that it can exploit evolution in software development technology and system support, software, and hardware. At any given time, the process must be formally and explicitly defined, yet we must anticipate change. The evolutionary process is expected to result in a finer granularity of the tasks and validation processes in the life cycle. This refinement is also expected to be reflected in smaller and more portable tools that can be more productively and reliably developed and maintained.

The architecture should specify a common data model.

This objective requires that the architecture specify a facility that is independent of any particular process, life-cycle model, methodology, or tool set. The architecture must prescribe a framework in which particular processes, methodologies, and tool sets can be defined to, embedded in, and easily modified within the facility.

Another goal is that of fostering the widest possible use of common tools in the facility. The operation of tools should not be impeded because they depend on particular system and data base environments. Also, the tools should be so designed that changes in system technology will not make them obsolete. This requires that the architecture specify a facility that isolates tools from system dependencies and supports the development of common tools that are reusable in multiple operating environments.

A major impediment to current tool integration is that few tools share data organizations or conceptual models of the product they help to describe. Therefore, we have a goal of ensuring that tools in the facility are integrated and cooperating to automate software life-cycle tasks. The architecture should specify a common data model to support tool integration and cooperation and to reduce the manual transformation required between process stages and

tools. The architecture should also specify a consistent user interface to support a single integrated view of the tools and process for users.

Still another goal is the automation of process control. Process definition and execution should be made explicit, formally definable, and machine processable. There should be an evolutionary capability to move from the current process, based upon paper guidelines that are manually monitored and administered, to an automated process control mechanism which uses explicit, formal process definition and rules-driven control. The architecture should specify a process mechanism that has functions and interfaces for defining a process, storing it in a facility as a set of rules and using these rules to control the execution of process tasks. Formal definition and mechanization will permit the recording and monitoring of the usage, performance, and effectiveness of the specified process. This information, which is used in analysis and research, is expected to contribute to improvements in the process and the tools as well as their evolution.

The objective of the architecture is to guide the development of a facility that supports a software development process and its associated tools. The software development automation architecture has the following characteristics:

- Process and tool independence to support flexible process and tool evolution
- System and data service isolation to support tool portability
- A common data model and a consistent user interface to support tool and user integration
- A process mechanism to support formal process definition and automated process control

The architecture must specify a facility framework, functions, data, interfaces, and event recording to support these capabilities.

The remaining architecture objective is to specify these capabilities in terms of a process framework. Such a framework is described in another paper in this issue.[25] The architecture specifies a facility that automates the software development process and process control, as defined by the process framework, in a way that allows for local variations to the resulting process definition. Thus the architecture specifies a scheme by which the process can be defined, encoded, and used to automate process control in a software engineering support facility.

Approach and structure

We now discuss means by which the previously presented software automation goals can be achieved. In the section under "Principles," we present a model of the software development environment that defines the structure, roles, and relationships among users, tools, methodologies, process control, and supporting system functions. This model captures the relationships among these elements that we expect to remain valid even as the software development enterprise evolves. Under the heading of "Architecture," we define the structure and organization of a software engineering support facility system base that supports this model and facilitates evolution toward software automation.

As an industry, we do not yet fully understand the software development process.

Principles. One approach to an automated programming environment is based on technology (e.g., a programming or design language), around which an environment to support development using that technology is defined and built. Another approach first assumes a particular software life-cycle model (e.g., a specified set and ordered sequence of task methodologies). In this approach, tools to automate the methodologies are selected and then bound in the sequence specified by the methodologies to provide an environment. Both approaches neglect such important process characteristics as the following:

- As an industry, we do not yet fully understand the software development process.
- The software development process will change as a result of new and improved technologies.
- Eventual process requirements cannot be predicted and accommodated with existing technology.
- Process change will require environments to be unbound and reconfigured to accommodate new methodology relationships.

- People use a variety of cognitive models to solve problems; some persons will find that any single technology impedes their problem-solving ability and requires alternatives.

These factors dictate that a software automation approach be based on two global principles: flexibility and separation of issues. The model proposed in this section recognizes that a software engineering support facility must flexibly accommodate project-level variation in process, methodology, and tools usage. It also recognizes that the facility must flexibly accommodate process and technology evolution. The model achieves this flexibility by separating the issues of process control from process methodology, methodology from tools, and tools from system functions. In this way, changes in any one item are isolated from the others. The architecture defines the interfaces between these elements that support their separation and interaction. These interfaces maintain the relationships among the elements as the elements evolve.

The model and architecture define a software engineering support facility.

Thus the model and architecture described in this paper define a software engineering support facility framework and mechanisms sufficient to allow an evolving family of programming automation schemes to be integrated. This approach does not depend on particular software development methodologies, tools, or languages. Instead, the approach provides the flexibility for a process and methodologies to be defined and modified. It provides for tools to be selected and interchanged in the facility to automate software development.

Software automation concepts. The goal of the software engineering support facility is to create an integrated, flexible environment and system base to support the following model for automating software development.

Members of a project define their process so that the tasks to be performed in the programming life cycle are identified. Each task can be performed manually or by a tool, if one exists. Alternatively, a tool can be built to automate the task.

The process definition specifies life-cycle control rules for the sequencing and control of these tasks. These rules can be formally defined, encoded, and captured in a data repository. A *process interpreter* engine in the facility can execute the rules to drive the process. The process interpreter can be used to gather and analyze process experience, to improve the process, and increasingly to direct the work of programmers and to control the tools based on these rules.

A project may tailor the facility for its methodology and environment by selecting the *agents* (human or tool) to perform each task in its life cycle. A project may be further tailored by defining the process rules to be used by the process interpreter to control the order of operation of the tasks.

New tools, methodologies, and process steps can be introduced by changing the task definitions, by embedding and interchanging tools, and by altering the life-cycle control rules. Process and technology evolution should be accommodated without disrupting the facility.

To achieve the flexibility required to exchange tools in the facility and still have widespread use, the tools should be portable. They should be capable of running in the operating system, in the data base, and in device interface environments in which the facility runs. To achieve portability, the tools should rely on common support functions provided by the facility to manage the data, the system functions, and the user interface. The common support functions should not contain unique, system-dependent code for that purpose.

To achieve integration, the tools should share a common view of the product data and the process description provided by the facility. All aspects of a product should be definable and manageable using a common data model. The relationships among data defined at various stages of the life cycle should also be definable, manageable, and shared by tools.

This model supports the separation of the automation functions that are necessary to promote flexibility. Process definition and control are separate from

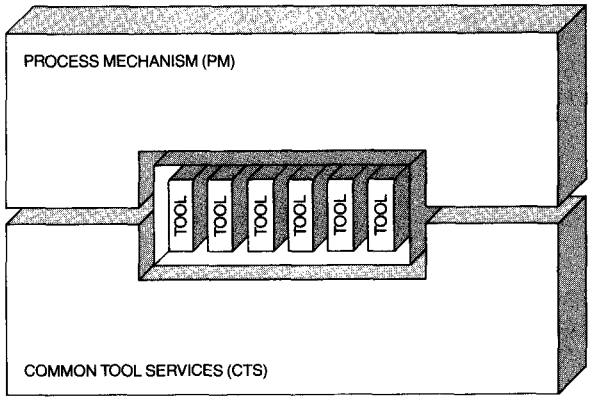
the actual process description. The process content (data) is separate from and modifiable by the process definition function. Process control is separate from the agents (human and tool) that perform the process. The process is described in terms of tasks. A set of agents can be defined and associated with the tasks to describe who or what performs the process. This association is maintained in the process description, rather than buried in the function of tools. The process automation function of a tool is separate from the system functions used to implement the tool. Thus, the tool is isolated from system dependencies.

Organizing these functions results in a software engineering support facility with the elements and structure depicted in Figure 1. A *system base*, composed of *process mechanism* (PM) and *common tool services* (CTS) components, provides common process control, service support, and integration functions to tools and users. A formal process description that is stored in the repository provides task descriptions and sequencing rules for life-cycle control. A set of portable process automation tools automates software engineering practices across the programming life cycle. Each of these elements must be developed and integrated to support our software automation approach.

The system base is the foundation and precursor for achieving our software automation strategy. By defining, providing, and stabilizing system base functions first, the architecture frees tool builders to concentrate on developing process automation functions. The architecture frees the analysts to concentrate on defining process, using system-base functions. The system base permits accelerated tool development, process experimentation, and eventual process automation. The system base should also encourage those with innovative software engineering ideas to build new tools that fit in the software engineering support facility.

Software engineering practices. Before discussing the architecture of a software engineering support facility in the next section, we describe here software engineering practices that should be automated in software automation tools. We have observed that in the past, the computer sciences have concentrated on the study of programming concepts, tools, algorithms, and languages.[26] A cadre of skilled programmers is now available with the training and talent to write computer programs. Now, however, there is another emphasis: software economics. Users expect

Figure 1 Architectural model of a software engineering support facility



a product to be delivered on time, within budget, and without errors. Intense competition also demands increased quality, function, price-performance, and shorter development schedules. For these reasons and because of the inevitability of human error, software development automation is now receiving increased attention.

This new emphasis is known as software engineering, which can be described as the systematic production and maintenance of high-quality software systems, delivered on time and within budget. It is a creative process supported by professional practices based on the best-known industrial experiences and computing theories. In support of software engineering, a software engineering support facility provides an environment in which software developers can manage the complexities of software development, and an integrated package of views, tools, functions, and controls that support the productive development of reliable software.

Figure 2 shows those software engineering practices expected by the authors to contribute most to success in automating software development. The figure also shows the ways in which the practices that benefit software quality and productivity can be realized.

In some cases, a facility may exhibit these characteristics directly. For the most part, however, they are exhibited by the processes, methodologies, and tools supported by the facility. The facility serves as an enabler or vehicle to encourage the use of these practices. The facility is designed with the expectation that the selected processes, methodologies, and tools are not in conflict with these practices.

Figure 2 Software engineering practices and benefits

Today, software development does not always make use of this set of practices in an organized and integrated way. This is thought to be the result of a lack of knowledge and experience, rather than a lack of desire or effort. This possibility lends credence to the need to evolve toward appropriate processes, methodologies, and tools, and to encourage such evolution through a properly supportive facility.

Architecture

This section describes the architectural concepts of a software engineering support facility. It elaborates on the organizing principles introduced earlier in this paper under the heading of "Software automation concepts." This section also gives an overview of the models, mechanisms, interfaces, and functions

that a software engineering support facility should provide to support software automation.

The concepts in this section are introduced in terms of perspectives or *views* of the facility. A view is a vantage point from which to examine a given aspect of the system. The architectural concepts can be examined in terms of the following three major views: system view, data view, and life-cycle view. These views expose the organization and major features of the software engineering support facility. A later section entitled "Direction" speculates on how such a facility could be used to direct the software development process toward automation.

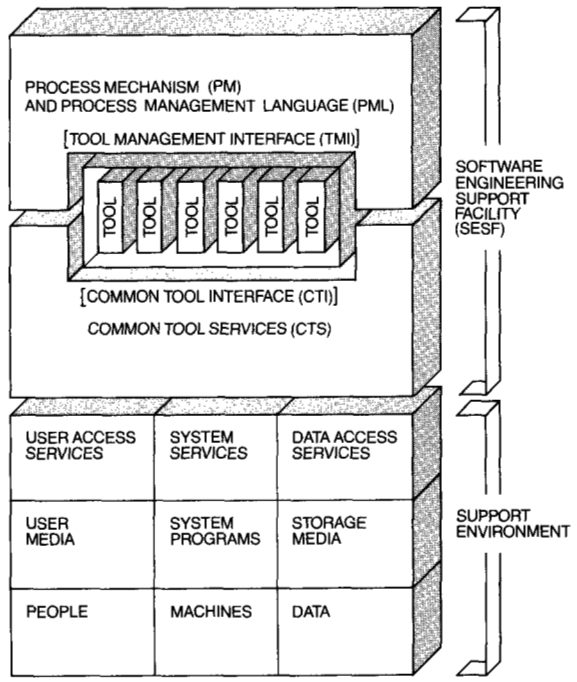
System view. An architecture should describe the functions, interfaces, and organization of a system, and how one system relates to another in the same environment. The *system view* examines the software engineering support facility from this perspective.

To achieve the flexibility and portability needed in the facility, the architecture introduces a hierarchy of function *layers*. This scheme separates the function of current monolithic tools into discrete layers. The scheme also describes how these layers interface with and make use of IBM program products and operating system facilities. The scheme is described under the heading "System base" later in this paper.

To understand the rationale for layering, we review the problems in the current tools environment that the software engineering support facility aims to correct. In most current tools environments, unique tools have evolved to automate or augment various stages of the software development process. Most of these tools are monolithic in the sense that each performs its own data base and system service management, presenting a unique interface to the user. Many tools have embedded functions that perform aspects of project control. Most tools are local, in the sense that they cannot share data or work products with other tools. In general, the structure of current tools presents such obstacles as the following to the merging of tools into an integrated, automated software engineering environment:

- Most current tools are written against a proliferation of system-dependent interfaces for data, presentation, and system services.
- Most current tools are not portable.
- As a group, tools present many interfaces to the user.

Figure 3 Architecture of the system base



- The tools share no common data representation of the product they help construct.
- The tools share no common view of, nor are they driven by, a common process.
- New tools are expensive to write for system-dependent interfaces.
- It is expensive to write versions of tools to support multiple system environments and data bases.
- It is expensive for tools to migrate to new system, data base, and device technologies.

The software engineering support facility approaches these problems by separating the process automation function in current tools from process control, data management, and system management tasks. The facility segregates process automation functions into tools. It separates the latter functions into a common support environment called the *system base*. The system base provides interfaces against which tools can be specified and an environment (whether interpreted at run-time or compilable) into which the tools can be embedded to automate process tasks.

System base. The system base organizes common tool support functions into a hierarchy of function layers (as shown in Figure 3) that supports tool portability and tool integration.

The system base is built on a layer of *system-dependent services* that make up the system environment in which the software engineering support facility operates. This layer can consist of IBM system program products, such as operating systems and data base access methods, that provide data communication, user media presentation, data base, and system services to current tools. This layer can be separated into user access services, data access services, and system services. The layer supports user and tool access to user display devices, data storage devices, and physical system resources.

The system base introduces a layer of *common tool services* (CTS) into the software engineering support facility. The common tool services provide the *common tool interface* (CTI), against which tools are specified. Tools specified against the CTI can run in any environment in which the system base runs. Within the software engineering support facility, the tools are portable and independent of operating system or data base environments. The objective in the software engineering support facility is to make possible the evolution of current tools that use system-dependent services into portable tools that use the CTI. The CTS layer achieves this by mapping tool requests for logical CTI services into system-dependent services. The CTS also provides a common data model that tools can use to describe and share product representations.

The system base provides an environment in which tools can be embedded. Each tool contains functions that automate one or more software life-cycle tasks. A tool may be specific to a phase in the development process, such as a debugging tool, or it may be used across several phases, such as a project measurement tool. Several tool functions may be packaged into a single tool for some purpose. In the system base, the tools are independent and do not communicate directly with one another. They perform no interface, data base, system service, or process management functions themselves. They use the CTI to obtain these functions.

Surrounding the tools in the software engineering support facility is a *process mechanism* (PM), which supports process definition and is responsible for controlling user access to and application of the tools on the basis of the process definition. The process mechanism uses a *tool management interface* (TMI) to invoke and control the tools. The process mechanism presents a *process management language* (PML) to the users, so that they may define their process and control its execution in the software engineering support facility. This process is encoded and stored (using CTI data services) in the repository as a set of life-cycle control rules (e.g., finite-state machine productions). The process mechanism executes these rules to drive the process.

The system base provides a software engineering support facility environment in which portable tools may be developed, integrated, and invoked to automate tasks in the software life cycle. The "data view" and the "life-cycle view" discussed in the next two sections explore the use of these functions provided by the system base to integrate tools and manage the software development process.

A key ingredient in managing the software development process is the distinction between the process and the product developed using the process.

The data view. One way to examine software development is as an information synthesis enterprise. The *data view* examines software engineering support facility requirements from this perspective.

In gross terms, the development of software involves synthesizing from a product idea (requirements) a series of increasingly detailed product descriptions. The process continues until we reach a description that runs on a machine and from which we can derive related product components such as documentation. Software development depends upon the creation, management, packaging, and presentation of information.

A key ingredient in managing the software development process is the distinction between the process and the product developed using the process. The process can be viewed as a description of the activities required to manage the development of a software product. The product or product description is the information created as a result of these activities.

For the most part, developers regard the final code that is released to users as the product of software development. To effectively manage the process, this view must be extended to include not only the released code, but also requirements, design, documentation, decision criteria, verification procedures, metrics, audit trails, other relevant information, and the relationships among these data that are produced during the development process. The software industry bases its intuitive, informal process management decisions on such data (or estimates thereof) today. To be more effective, data must be formally defined, captured, shared, and manipulated by tools to provide a basis for integrated, automated process management.

A product evolves from an idea to a released function.

In a sense, a product evolves from an idea to a released function, describing new information and deriving new relationships as it evolves. Our architecture regards the evolving product as an interrelated aggregate of all information derived during the software development process. This architecture also regards the process as a separate information aggregate consisting of rules for the creation and management of the information in the product.

A software engineering support facility must be able to create and manage these information aggregates in two important ways, to achieve product and process integration.

Product description sharing. Agents that produce work products for stages of the product description (e.g., requirements, system design, implementation, and test plan) must be able to share work products and information in the stages of the product description without transformation. This requires that the software engineering support facility provide all agents with a common, shared view of the product description. It also requires that the facility define and expose to agents the common data elements of which the product description is composed, so that they may share and manipulate these common elements.

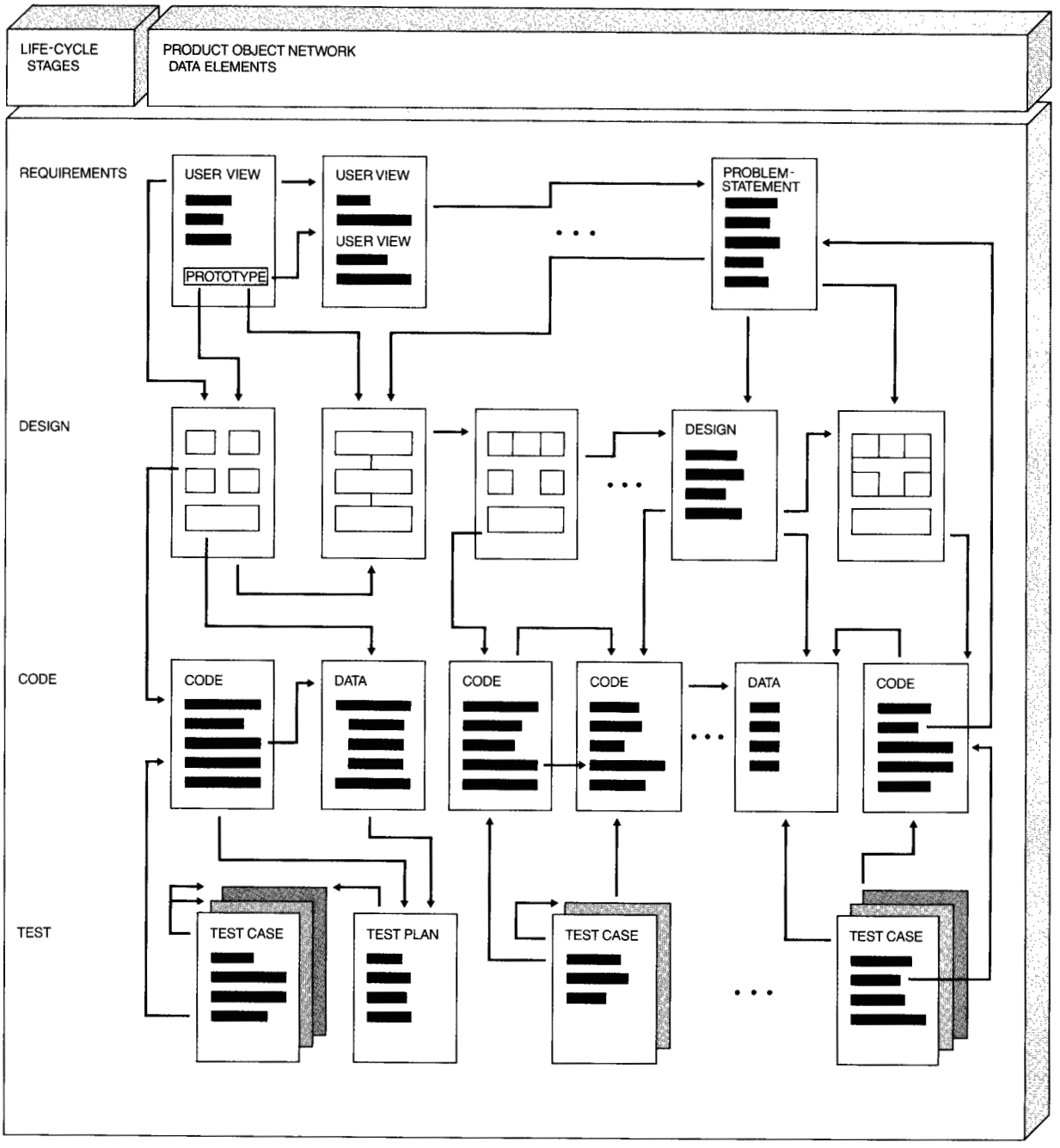
Traceability. The agents must be able to trace (track and backtrack) the relationships of product information to predecessor and successor information in the product definition. For example, a tool might need to determine which requirements statement motivated the creation of a fragment of program code. This capability allows tracking of the rationale, constraints, and relationships used to develop the product element. It allows analysis of the completeness and consistency of product elements and management of the process used to produce the product.

These objectives require that we adopt a broader scheme for data management than that supported in most current software development environments.

In these environments, the extent of product-description sharing that is achieved is provided by library management programs. These programs allow the user to define the structure of a software product as a hierarchy of parts or files. Some of these programs contain parts management functions that control aspects of software development (e.g., promotion of code into a test system). These functions are based on the existence of parts (e.g., code completed and stored in the appropriate file) and the completion of operations on parts (e.g., code successfully compiled). Parts management provides library users with some elements of project control. Library applications (e.g., build and integration functions) support the construction of a software product based on a parts hierarchy definition.

The deficiency presented by these current environments is that the part (file) is merely a carrier for many product data elements contained in the part. Parts management provides tools with information about the hierarchical relationships among parts only. The parts themselves contain a network of elements and relationships that are, for the most part, currently invisible to tools. For example, a file containing program source code contains many chunks of code. Some of these chunks are related to or derived from original product requirements. Other chunks are derived from change requests or fixes. Some chunks may not be code at all, but comments or assertions about program behavior for use in deriving test cases. To manage the software development process adequately and to promote sharing and traceability, each of these elements and

Figure 4 A product as an object network



relationships must be defined as part of a product, managed as part of process control, and visible to agents.

The architecture supports this capability by introducing a unified data view to support product description sharing and traceability. In the architecture,

a product is defined as a *network* of related data *objects*. An *object network* (see Figure 4) holds a product as an information aggregate. A process can also be defined as an object network. This network contains descriptions of the activities and rules used to create and manage the information in the product object network.

An *object* is a separately identifiable and named collection of data that contains its own identifier, actual content, attributes, and all defined relationships to other objects in the network. The objects and relationships required for a product are determined by the process description. The process description also dictates the granularity of the objects. For example, a very detailed process definition may require the tracking of individual lines of code as objects.

Thus, the architecture describes how a software product and related information can be defined and managed by agents as a unified network of related objects. On the common tool interface, the architecture provides an *object model* and a set of services for manipulating the model. This model, based on Entity-Relationship (E-R) theory,[27] provides a common, logical view of data to tools and users. Software developers can use these facilities to construct data models of the product to be defined and the process by which it will be built, both of which can be shared by all agents in the life cycle. Data sharing and tool integration are supported in the architecture by having all tools manipulate product and process data using the object model.

To support portability, the architecture isolates the use of the data model by agents from the physical organization and storage of actual data. Through the common tool interface, the architecture provides functions that allow references to the data model by agents to be bound to actual data stored in a variety of physical data organizations and data accessed through multiple data access methods and data base management systems. This capability allows an integrated tool set that shares the common data model to be portable across physical data base environments.

The life-cycle view. The software engineering support facility is designed to alleviate a major problem in current software development environments—inadequate life-cycle management. This problem has three aspects.

One aspect is that of isolated tools and manual methods that are used to create disjoint representations (e.g., requirements, design, and code) of a software product. Usually the representation is produced manually and is not processable by machine. Representations that are machinable usually do not share the same data organization. As a result, product description data cannot be shared or mapped across process stages without manual transformation. In turn, process management in these environments is fragmented. Without the means to analyze the content of a product representation or to relate one product representation to another, process stages and transitions from one development stage to the next cannot be effectively managed.

Mechanisms for controlling the order, sequence, and communication among tools are today embedded in the tools themselves.

In another aspect, the mechanisms for controlling the order, sequence, and communication among tools that collaborate to perform life-cycle tasks are today embedded in the tools themselves. These hardwired connections limit life-cycle management to the functions and order dictated by the tools. These connections also limit the flexibility of the environment to adapt to improved methodologies, better tools, and new technologies, because change requires the expense of breaking old connections and establishing new ones.

The third aspect of the problem is that the process is informal, and process experience is not adequately recorded. Without such information, there is no basis for process review, analysis, feedback, and improvement.

The architecture approaches this problem by introducing a flexible, integrated, and automated life-cycle control scheme into the software engineering support facility. The life-cycle view examines the facility from this perspective.

Our scheme for tool integration, communication, and life-cycle control is based on several concepts that we outline here.

Life-cycle control is separate from the automation of life-cycle tasks in the facility. Agents (both persons and tools) accomplish software development tasks. Life-cycle management is provided independently of these agents by process mechanism functions that span and control all of the agents.

Agents are mutually independent and they do not communicate explicitly or share private data. They use common facilities in the system base for communication, data sharing, and control.

Tools are designed to be pluggable and interchangeable in the facility.

Agents access a common data view of the product they describe and the process used to describe it. This view is provided by the object model.

The process mechanism spans the entire life cycle and controls all agents by means of a rules-driven scheme. It provides users with a process management language they can use to define a process control scheme to the facility. The process mechanism encodes this process description as a set of life-cycle control rules. Using the object model, the process mechanism stores these rules in a repository for use during process execution.

During execution, the process mechanism accepts all user requests for data and development tasks to be performed, and records actions taken for later analysis. It accesses the process rules to determine whether a requested action is permitted, based on whether the actions and conditions specified in the process as prerequisite to this request have been satisfied. The process mechanism then uses the tool management interface to invoke an appropriate tool to satisfy the request, or notifies an appropriate human agent to carry out the task. The tools are structured to respond to process mechanism action-on-object-through-agent commands, as specified using the tool management interface.

To support this scheme, the tools are designed to be pluggable and interchangeable in the facility. A programming site may choose the tool it prefers to use to automate a task in the programming process, assuming that alternative tools are available. The set of tools in a software engineering support facility configuration can be determined at installation binding, when specific tools can be chosen to map to the tasks specified in the local process description. Tool order, whether sequential or concurrent, is controlled at execution time by the process mechanism, using the precedence relationships expressed in the life-cycle control rules.

The key feature of this life-cycle control scheme is its independence of methodology and tools. Rather than hard-wiring a specific set of tools into the facility in a sequence designed to automate a specific methodology, the scheme allows local project administrators to change the set of tools contained in that project's facility. This allows local process managers to modify the process and, therefore, the sequence of tool use. This flexible scheme provides integrated, automated life-cycle management. At the same time, the scheme accommodates changing process, tools, and technology, as well as local process variation.

Direction

The software engineering support facility architecture discussed in this paper is designed to allow flexibility and adaptability across a wide spectrum of software development processes, local conditions, tools, and technologies. The architecture is expected to remain effective for many facilities, collections of tools, and process definitions. It has characteristics designed to make it stable in the presence of changes in tools and process technology. We expect its influence to be effective over a long period of time.

In this section we discuss the direction that we believe processes and tools will take under the influence of this architecture. We also consider the effects of evolutionary pressure on process definition and control, tools, and people. We begin with a brief look at software development today, in terms of evolution under this architecture. We then consider near-term practical implications of the architecture and a corresponding facility. Finally, we speculate on the longer-term direction that software development evolution might take under such an architecture.

The situation today. Large-system software development tools and processes in use today are characterized by the formality and power of large, unintegrated tools and by the relative informality of complex manual processes used to join and control them. Tools have evolved in this direction through the lack of a common support system. This encourages large tools that contain their own support systems (including user interfaces, system interfaces, and data base management). Also, there has been a desire to automate as much of the process as possible, while using unintegrated tools. This has led to process assumptions and limitations being built into large tools in fundamental and inflexible ways.

Automated processes for tools are quite controllable.

As a direct result of this tools-driven environment, the portion of the software development process that is automated today is almost precisely that portion which is built into the large but inflexible tools. The effect has been the development of paper processes that are both flexible and visible. Paper processes are, however, not very controllable. Automated processes for tools are quite controllable, but they are neither flexible nor visible.

The near term. What this architecture introduces to today's situation is a software engineering support facility, i.e., a common support system for tools and a mechanism for process automation. The presence of a *common tool interface* (CTI) in such a facility encourages tools developers to rid their tools of unique user interfaces, system service interfaces, and data base management systems, in favor of a common set of such services. Any such architected facility includes a mechanism for automated process control, called a *process mechanism* (PM), that exists above and beyond the individual tools. The presence of the PM encourages tool developers and process owners to remove process automation from the separate tools and place it in a common control mechanism where it is flexible, visible, and controllable.

The effect is twofold: (1) the tools change shape, content, and scale, and become integrated at the service level; and (2) the process takes on an automated life of its own, separate from the tools. The near term under the first effect is characterized by planned migration from existing tools to tools that make best use of the facility. The architecture and corresponding facility must accommodate such migration through provisions for levels of tool integration, from simple unintegrated user access to tools all the way to full and unrestricted use of the facility's capabilities. Such best use will be exemplified by tools that are divided in function into user interface, system interface, data base management, tool invocation, and *tool function*. By tool function we mean whatever is left over that describes what the tool actually does for a user. The facility provides all of these services except the actual tool function.

For existing tools, the change is principally one of removing unique code that provides the facility services today. For new tools, the advantage comes from not having to write such code.

With respect to the second effect, the process becomes an entity separate from the tools, with its own automation—the PM. The combination of removing the process from the tools (where it is relatively rigid) and providing automated support for what is basically a complex job in its own right creates a new situation for any software development project. That is, the process itself must now have the same kind of automation-oriented formality and sophistication that tools work already has. The process for any project must be codified and defined to the PM and executed under automated control. This new, separate, formal, automated process definition will require human effort, intelligence, and maintenance. Whether or not this is done by specialists, best use of the facility will require automation.

The process definition must match the project and process data available during the process with the available tools capabilities and with the capabilities of the persons who have roles in the process. The process definition includes all those dimensions. We call the matching of a formal definition with the available working parts of the process *complementarity*. Complementarity must always be maintained or the process will fail to function properly.

We now explore the anticipated effect of maintaining complementarity. Initially, the process definition may be at a very high level and made up entirely of human activities, regardless of whether the people involved use tools to accomplish some or all of a process step. Under such conditions, complementarity becomes a determination that people can perform the steps, and that the steps make sense to those who will perform them.

Best use of the facility implies that eventually, through a carefully planned migration, a fine-grained process codification will emerge. The codified definition will include both human and automated steps, and complementarity will become much more important and obvious. It is always true that the human steps must make sense to people. It is also true that complementarity must work for automated, tool-driven steps. Here the rigidity of tool capabilities melds with automatic invocation by the PM to yield a direct connection between process step and tool power, without human intervention, i.e., the complementarity of process and tool.

Thus, the near term is characterized by migration from existing large and unintegrated tools to tools utilizing the facility for services and process control.

The long term. The effects of this architecture and corresponding facilities over the long term are, of course, more speculative than those for the near term. The authors believe that some directions have been clarified by the discussion thus far.

The process definitions are expected to become more formal, more sophisticated, and more detailed over time. This is a natural result of the desire to use the capabilities of the facility and its PM to the maximum. As one part of the process comes under close scrutiny and definition in the PM, another part appears both more tractable and more in need. Complementarity, as just discussed, must be preserved at all times, but many of the human aspects of the process and some of the tools aspects could become far more formally defined and measured than they are today without disrupting the existing tools or process. As investment in tools and process is allowed to progress into the longer term, pressure will build naturally, first to define and automate the simple yet troublesome things. Then one will turn to whatever is at least tractable. This process could eventually lead to complementary investment in tools and process, where one cannot move forward without the other.

The tendency cited earlier in this paper to draw the process out of the tools and into the PM over time should lead to a more detailed process, being both finer-grained and more visible. Those progressive levels of granularity and visibility will likewise lead to more controllable and measurable processes, and tools to support them.

The other side of the increasingly refined process granularity is the long-term breakup of today's large tools.

The other side of the increasingly refined process granularity is the long-term breakup of today's large tools. As we described earlier in this paper, the services provided by the facility will replace similar services built uniquely into each tool. What has not yet been affected is the tool function itself, which remains as a large, multifunction black box. Over time, however, the desire to make more effective and efficient use of the facility, the desire to extract the process from those large tools to make it visible, and the investment pattern of new tools (which will utilize the facility from the start) are expected to cause the large-tools functions of today to break up. That breakup is expected to be along process-defined lines. That is, as the process definition becomes more granular, process definers look at each large-tool function as though it were a black box to be better understood. The understanding of a function causes its process substeps to become known. Once known, the tool function can be broken along those process lines, resulting in a yet-finer-grained process definition and smaller tool functions. Thus, instead of today's expectation that tools systems will become more and more monolithic as they become more and more integrated, our architecture anticipates a situation in which the tools become smaller and smaller carriers of single, discrete process steps wherein the process definition serves as the glue. The separate and complementary development of tools

and process, while preserving complementarity, is a cornerstone to foreseeing the effect of this architecture over time.

In the future envisioned here, new tools come as single functions for single process steps. Since tools are more discrete and no longer need expensive services or process knowledge built in, more alternative tools can be tried and tested at the local level. Also, more alternative processes can be tried and measured for effect. Because most tools and process ideas start at the desk of one person with a problem on a real project, the effect of our architecture should make ingenious solutions and progress more likely.

Without tool and process portability from one facility or project to another, process granularity and the resulting functional tools would be primarily of local value and not easily shared. With portability of processes and tools, the impact of even small improvements can be readily and cheaply achieved by many users.

Most important is the use of such facilities to cause people's tasks to change. As the process becomes visible, measurable, and granular, and as the view of software development becomes more process oriented and less tool oriented, people involved with a project or process will notice at least some of the following:

- People become task oriented rather than tool oriented, and can move more easily among projects.
- Tools knowledge becomes a specialty.
- Process knowledge becomes a new and highly automated specialty.
- Repetitious process steps that are done today to satisfy tools that cannot share information should disappear as steps are taken once in the process and the information is shared by tools.
- Process responsibilities become more visible, concrete, and understandable.

It is not known at this time whether such a future is more than speculation. Neither is it known whether such breaking up will lead to single-function tools. All possibilities await the creation and use of such facilities and the experiences that time alone provides.

Concluding remarks

This paper has presented an architecture for a software engineering support facility. We believe such a facility to be at or beyond the current state of the art in software development environments. We also believe such a facility would support the practice of advanced software engineering techniques. The objective of such a facility is to support software development tools and process automation with the following characteristics:

- Process and tool independence to support flexible process and tool evolution
- System and data-service isolation to support tool portability
- A common data model and a consistent user interface to support tool and user integration
- A process mechanism to support formal process definition and automated process control

Technologies are available today that make implementation of this architecture appear to be feasible. The use of this architecture to direct the implementation of a software engineering support facility should produce a new state of the art in software development environments that can answer the challenges raised by both software users and producers and addressed in this paper.

Acknowledgments

The authors acknowledge the contributions to the software development automation architecture discussed in this paper by many researchers and developers throughout the IBM Corporation. Special thanks go to Jerry Anderson, Sam Bailey, Dr. Dan Chang, Len Orzech, Jess Rowland, Jim Sagawa, and Fred Wilkes, who participated as a team with us to define and develop the approach described in this paper. In some cases, verbatim excerpts of their work from other related documents are included here.

Cited references

1. *Software Technology for Adaptable, Reliable Systems (STARS) Program Strategy*, Reports AD-A128957/8 and AD/A128918/0, Office of the Deputy Under Secretary of Defense for Research and Engineering, Washington, DC (March 1983).
2. *Specification Practices*, MIL-STD-490, United States Government, Department of Defense, Washington, DC (October 1968).
3. *Configuration Management Practices for Systems, Equipment, Munitions, and Computer Programs*, MIL-STD-483, United States Government, Department of Defense, U.S. Air Force, Washington, DC (March 1979).
4. I. Nassi and B. Schneiderman, "Flowchart techniques for structured programming," *ACM SIGPLAN Notices* **8**, No. 8, 12–26 (August 1973).

5. M. Schindler, "Today's software tools point to tomorrow's tool systems," *Electronic Design* **29**, No. 1, 6–15 (July 23, 1981).
6. D. E. McConnell, *An Investigation of the State of the Art Trends in the Life Support of Complex Embedded Computer Systems*, Naval Weapons Center, Dahlgren, VA (September 1979).
7. V. J. Crandall, "Enterprise-wide information management: A management perspective," presented at *Ken Orr's DSSD User's Conference, FEEDBACK '84* (October 1984).
8. D. Alberts, "The economics of software quality assurance," *AFIPS Conference Proceedings, National Computer Conference* **45**, (1976).
9. M. Knight, "Software quality and productivity," *Defense Systems Management Review* (Autumn 1978).
10. M. W. Alford, "Software requirements in the 80's: From alchemy to science," *Proceedings of the Annual Conference, ACM 80*, Nashville, TN, pp. 342–349 (October 27–29, 1980).
11. W. E. Beregi, "Architecture prototyping in the software engineering environment," *IBM Systems Journal* **23**, No. 1, 4–18 (1984).
12. D. E. McConnell, *Productivity Initiatives for Effective Lifetime Support in the Navy's AEGIS Program*, Naval Surface Weapons Center, Dahlgren, VA (June 1982). Copies of this unpublished report, which was presented at the European INDOS Project, Pisa, Italy, in 1983, may be obtained from the author.
13. R. G. Mays, L. S. Orzech, W. A. Ciarfella, and R. W. Phillips, "PDM: A requirements methodology for software system enhancements," *IBM Systems Journal* **24**, No. 2, 134–149 (1985, this issue).
14. D. J. Pearson, "The use and abuse of a software engineering system," *AFIPS Conference Proceedings, National Computer Conference* **48**, 1029–1035 (1979).
15. A. J. Thadhani, "Interactive user productivity," *IBM Systems Journal* **20**, No. 4, 407–423 (1981).
16. K. H. Kim, "A look at Japan's development of software engineering technology," *Computer* **16**, No. 5, 26–37 (May 1983).
17. E. W. Hubbard and D. W. Waugh, "Automation for the software engineering process," *Technical Directions* (IBM Federal Systems Division) **10**, No. 1 (1984).
18. R. W. McGuffin, A. E. Elliston, B. R. Tranter, and P. N. Westmacott, "CADES—Software engineering in practice," *Proceedings, 4th International Conference on Software Engineering, IEEE*, pp. 136–144 (1979).
19. *Common APSE Interface Set (CAIS), Proposed Military Standard, Version 1.3*, Report AD-A134825/9, Office of the Secretary of Defense, Ada Joint Program Office, Washington, DC (August 1984).
20. *Software Engineering Environment for the Navy, Report of the NAVMAT Software Engineering Working Group*, Report AD-A131941/7, Naval Material Command (NAVMAT), Washington, DC (March 1982).
21. M. W. Alford, "A requirements engineering methodology for real-time processing requirements," *IEEE Transactions on Software Engineering* **SE-3**, No. 1, 60–69 (January 1977).
22. R. J. Lauber, "Development support systems," *Computer* **15**, No. 5, 36–46 (May 1982).
23. W. Rauch-Hindin, "The software industry automates itself," *Systems and Software* (October 1983).
24. *Requirements for Ada Programming Support Environments, Stoneman*, United States Government, Department of Defense, Washington, DC (February 1980).
25. R. A. Radice, N. K. Roth, A. C. O'Hara, Jr., and W. A. Ciarfella, "A programming process architecture," *IBM Systems Journal* **24**, No. 2, 79–90 (1985, this issue).
26. D. Teichroew and A. Hershey III, "PSL/PSA, a computer-aided technique for structured documentation and analysis of information processing systems," *IEEE Transactions on Software Engineering* **SE-3**, No. 1, 41–48 (January 1977).
27. P. P.-S. Chen, "The entity-relationship model—toward a unified view of data," *ACM Transactions on Database Systems* **1**, No. 1, 9–36 (1976).

Gene F. Hoffnagle *IBM Information Systems and Storage Group, P.O. Box 390, Poughkeepsie, New York 12602.* Mr. Hoffnagle is currently a senior programmer with the Programming Quality and Process directorate in IS&SG. He joined the IBM Federal Systems Division in 1967 and worked until 1973 on the National Airspace System. From 1973 until 1977, Mr. Hoffnagle was part of a small team working with IBM Fellow Dr. Harlan D. Mills on software engineering and advanced information automation systems. He developed software engineering education programs for the Federal Systems Division until 1979, and subsequently for the IBM Data Systems Division. In 1981, he joined IS&SG as a systems architect and initiated this software development automation architecture. Since 1983, he has been the lead architect for this architecture, especially for its process mechanism. From 1980 until 1984, Mr. Hoffnagle was an adjunct instructor in software engineering at the IBM Systems Research Institute. Mr. Hoffnagle received a B.S. in mathematics from Case Institute of Technology in 1967 and an M.S. in computer science from The Johns Hopkins University in 1976.

William E. Beregi *IBM Information Systems and Storage Group, P.O. Box 390, Poughkeepsie, New York 12602.* Mr. Beregi is currently a development programmer on the Programming Quality and Process directorate staff in IS&SG. After graduating from Carnegie-Mellon University with a B.S. in mathematics in 1974, Mr. Beregi joined IBM at Kingston, New York. He managed software engineering and reliability, availability, and serviceability groups there. He also had technical assignments in system design, planning, and product quality assurance. His most recent responsibility was managing the Software Engineering Process Technology group, where he was responsible for the invention and development of system design and rapid prototyping tools. Mr. Beregi joined IS&SG as a systems architect in 1984 and established and directed the development team for this software development automation architecture. Mr. Beregi was lead architect for the architecture's common tool interface. He has published articles in the *IBM Systems Journal* and in the IEEE conference proceedings on software engineering and rapid prototyping. He received an M.S. in computer and information science from Syracuse University in 1977.

Reprint Order No. G321-5242.