Software Production: From Art/Craft to Engineering

T. Richard Hsueh
Thomas F. Houghton
Joseph F. Maranzano
Gerald P. Pasternack

Software-development organizations throughout industry are striving for better ways to develop, deploy, and maintain their products. Changing an organization's well-established operating procedures—to improved methods and techniques—is always a substantial challenge. But, quite often, change is crucial for maintaining competitiveness and ensuring long-term success. This paper analyzes the challenge, and identifies key elements of a typical migration path to creating a new operating paradigm. The paper also provides a framework for *technology insertion*, a methodology that helps develop a disciplined engineering process, allowing software projects to advance systematically in their journey to success. Experience with the AT&T mosaic of integrated software processes, known as MOSAIC, and the software-development environment (SDE), shows how MOSAIC and SDE support project teams undergoing organizational change.

Introduction

Since the dawn of the computer era, nearly 40 years ago, software-development organizations have been striving to improve their products and services. Yet, these same groups frequently have been criticized for delayed product deliveries, exceeded budgets, released products containing "bugs," and a failure to meet customer requirements. New tools, methods, and approaches for software development are introduced every year. Nevertheless, many development organizations complain that technology keeps improving, but they still cannot fully benefit from it.

As the size and complexity of software increase—and the time-to-market intervals are shortened—an organization's competitiveness increasingly depends on its ability to identify and adopt improved methods. To develop this ability, the software industry must abandon the traditional art-and-craft nature of product development. In such an environment, each software program is usually custom designed, and depends heavily on the ideas and expertise of only one or two "star" programmers.

Recently, software-industry researchers and practitioners have devoted much of their efforts to encouraging organizations to abandon old methods and procedures and adopt new ones. The software-process maturity model[1], software factory[2,3], and total-quality-management approach[4,5] are just a few examples of such efforts. The scope and emphasis in addressing productivity and quality gains vary. However, there exists a consistent and growing recognition that moving toward a more disciplined engineering process is vital for success in today's highly competitive environment.

Software production employs many processes and tools that are defined, used, and continuously improved. Moreover, many avenues of approach are available in advancing toward the goal of establishing engineering discipline in the production process. This paper focuses on the migration aspect in achieving this goal.

A Systematic Approach to Migration Success

Most of today's software-development projects have implemented specific quality and productivity initiatives. Typically, these initiatives reach a modicum of local success. Unfortunately, successful implementation throughout a large organization is often

Panel 1. Abbreviations, Acronyms, and Terms

BCP—best current practice

CASE—computer aided software engineering

cimgr—an annotation editor that records points of irregularity and inspectors' comments

code inspection—a technique used to ensure that source code conforms with all design specifications

FrameMaker—a document-preparation system that integrates word processing, page layout, graphics, tables, and other functions into a single, easy-to-use application

MOSAIC process—a generic life-cycle process model that contains information about such items as task descriptions, tools, practices, and methods

nmake—a software tool used in the "build" process when constructing a new version of a product

Sablime—a source-code-version control system that supports configuration management

SDE—software-development environment

SMARTsystem—software-development, maintenance, and reverse-engineering tool

T-aspects—five factors that a software-development organization must consider when migrating to an engineering discipline—"task," "technique," "tool," "training," and "transition"

technology insertion—a framework for the systematic migration to engineering discipline in software development

considerably more difficult. Such difficulty is usually due to the lack of a comprehensive migration process.

Leading a software-development organization into a state of engineering discipline is not an easy task. Such leadership requires continuous attention and commitment to the implementation and management of both technical and non-technical changes. Integrating these changes into day-to-day operations often challenges an organization to:

- Analyze its current environment to identify and close performance gaps;
- Investigate all applicable state-of-the-art techniques, methods, tools, procedures, practices, and approaches;
- Create a process and technology framework to incorporate applicable elements, as well as guide operations; and
- Orchestrate the adaptation of new methods and techniques so they become routine business practices.

To meet the challenge, a framework for the systematic migration to engineering discipline must be developed. This technique, as previously noted, is known as technology insertion.

Technology Insertion and Its Five T-Aspects. By definition, the word "technology" is a scientific method of achieving a practical purpose. Within the context of software production, technology insertion is meant to:

- Identify appropriate methods, procedures, and tools for "seeding" into an organization by means of a migration path;
- Nurture and institutionalize the selected methods, procedures, and tools so they become routine within the organization; and
- Increase local expertise on each method, procedure, and tool so they can be introduced easily into other organizations.

Technology insertion requires that a software-development organization carefully consider five factors, known as *T-aspects*, when migrating to an engineering discipline. "Task," "technique," "tool," "training," and "transition" represent the five T-aspects. Technology insertion also requires the synchronization and integration of the T-aspects, from both the planning and implementation perspectives. Figure 1 shows a high-level view of major tasks in the software-development life cycle.

The interrelationship among T-aspects can be illustrated by the following scenario:

> Consider the requirements work that is illustrated in Figure 1: It is a *task* of understanding, documenting, and confirming customer needs and expectations. To carry out this task, one can employ various *techniques* or methodologies. For example, the requirements can be documented in text-only form, or by using structured analysis or object-oriented techniques. Each technique may be supported by several different *tools* on various hardware platforms. To use any technique and tool, adequate *training* is needed. To shift from one technique or tool to another, *transition* must be well planned and executed, such that previous investments in the embedded base are properly secured.

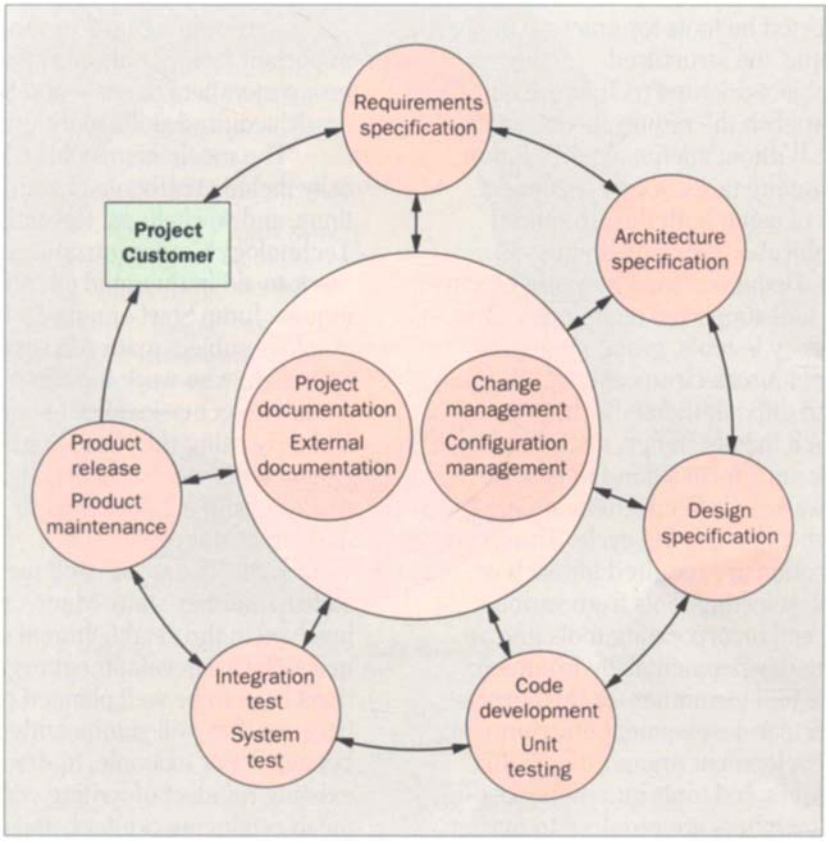

Figure 1. This illustration of the software-development life cycle shows a high-level view of all major tasks and how they are interdependent. Major tasks can be separated into subtasks. For a software-development organization striving for engineering discipline, defining and streamlining tasks are the first steps.

Each T-aspect is further detailed as follows:

- *Task.* This T-aspect can be viewed as specific steps in a process that brings about an end or result. As shown in Figure 1, requirements, architecture, design, coding, and testing are common high-level tasks in the software-development life cycle. From a process perspective, all tasks are interdependent. Each one must pass information to the following task. For example, the design task becomes meaningless without requirements. Tasks can also be separated into subtasks. For a software organization striving for engineering discipline, defining and streamlining tasks are the first steps.[6,7] However, it is usually not easy to capture and define the tasks of an organization's software life-cycle process. These tasks often require different methods of information encapsulation. Hence, group members could become deadlocked in seemingly endless debate over the information-encapsulation process.
- *Technique.* This T-aspect can be viewed as a method of accomplishing a desired purpose. In the software-production environment, many techniques have been developed and practiced to help improve software productivity and quality. For example, code inspection is one technique used to verify conformance with design, implementation correctness, and applicable standards. Quality-function deployment is a technique used to place in priority order the crucial requirements in the product-realization process, to ensure that end-products are properly aligned with customer needs. Structured analysis and design are techniques used to facilitate the development and expression of architectural and design specifications. Software quality assurance, discussed in another paper contained in this issue, is a technique that helps organizations proactively measure, utilize, and improve development processes.[8] When a technique is employed to accomplish certain tasks, the interchanges and hand-offs that must occur among the tasks are crucial to the success of both the technique and the tasks.
- *Tool.* Some techniques can be applied manually, while

others must be supported by tools for practical implementation. For example, the structured analysis/design and object-oriented techniques can each be used to accomplish the requirements, analysis, and design tasks. Without adequate tool support, passing information among tasks is very inefficient, and the effectiveness of using both the structured analysis/design and object-oriented techniques is significantly reduced. Documentation is another example of manual versus tool supported techniques. Due to the lack of consistency in tools, group members often create documentation electronically, but then circulate more costly and difficult-to-handle printed copies. To prevent such inconsistency, all the tools, techniques, and tasks must form a bond to ensure maximum "communications" effectiveness among the various segments of the software life cycle. Thus, careful planning and execution are required for each of these important steps: selecting tools from various vendors; introducing and incorporating tools into an organization's existing developmental environment; and, shifting from one tool to another for the purpose of establishing a particular development environment.

- *Training.* Software-development organizations often introduce new techniques and tools into their existing environment. Group members are required to master the skills associated with each technique and tool, learn about their benefits and drawbacks, and understand how they can be used.

There is, however, a common pitfall that has been observed in the training aspect. Individuals sometimes are too quickly satisfied that they have learned enough about a tool to solve a particular problem. As a result, they devote much less time than they should trying to understand the fundamental technique the tool supports. Also, they often neglect to consider the impact of the technique and tool on a specific task. For example, systems engineers, who have learned about a particular tool, may have used it to generate data-flow diagrams to help articulate the interaction among software modules. Sometimes, the data-flow diagrams may represent a new form of intermediate deliverables between project tasks. In such a situation, the transition to adopt the new discipline has to be managed very carefully, because implementing the new approach has a significant impact on an organization's operations.

Determining how to shorten learning time is an important factor in organizational training. This allows group members to use—and benefit from—their newly acquired skills more quickly.

The mechanisms of the learning process typically include textbooks, lectures, seminars, consultations, and workshops. Recently, the AT&T Software Technology Center introduced a method called Jump Start, to aid in the rapid adoption of new teaching techniques. Jump Start emphasizes learning by doing. It employs subject-matter experts, utilizing the hands-on approach, who work together with the trainees on actual project tasks for which the trainees are responsible. By using the Jump-Start method, an organization can benefit from the shared learning experience of working with a subject-matter expert, and thus avoid making mistakes.

- *Transition.* The word itself means shifting from one state to another state. Many crucial transitions are involved in the establishment of engineering discipline in a software-development organization. These transitions have to be well planned and carefully executed, because they will significantly affect group members' behavior. For example, under the new discipline, the existing mindset of writing code will be modified to mean producing product, and maintaining it throughout its life cycle. Such transitioning—from one set of techniques and operations to another—often reinforces feelings of constraint and fear among many group members. Some may even embrace the not-invented-here syndrome in an attempt to dissociate themselves from any new approaches. To effectively overcome these common reactions, management must demonstrate strong leadership, communication, and commitment during the transition.

The three tactics that follow have proven to be very useful in facilitating the transition process:

- *Establish and agree on a change-control mechanism at the beginning of any transition.* This allows group members to maintain control of the transition, rather than being controlled by it.
- *Introduce and manage necessary changes by using incremental, small steps.* Although the final objective can be bold, this tactic can help minimize the difficulty of the transition and the natural resistance to change.
- *Establish the software process-engineering function in the organization as early as possible.* This function is to

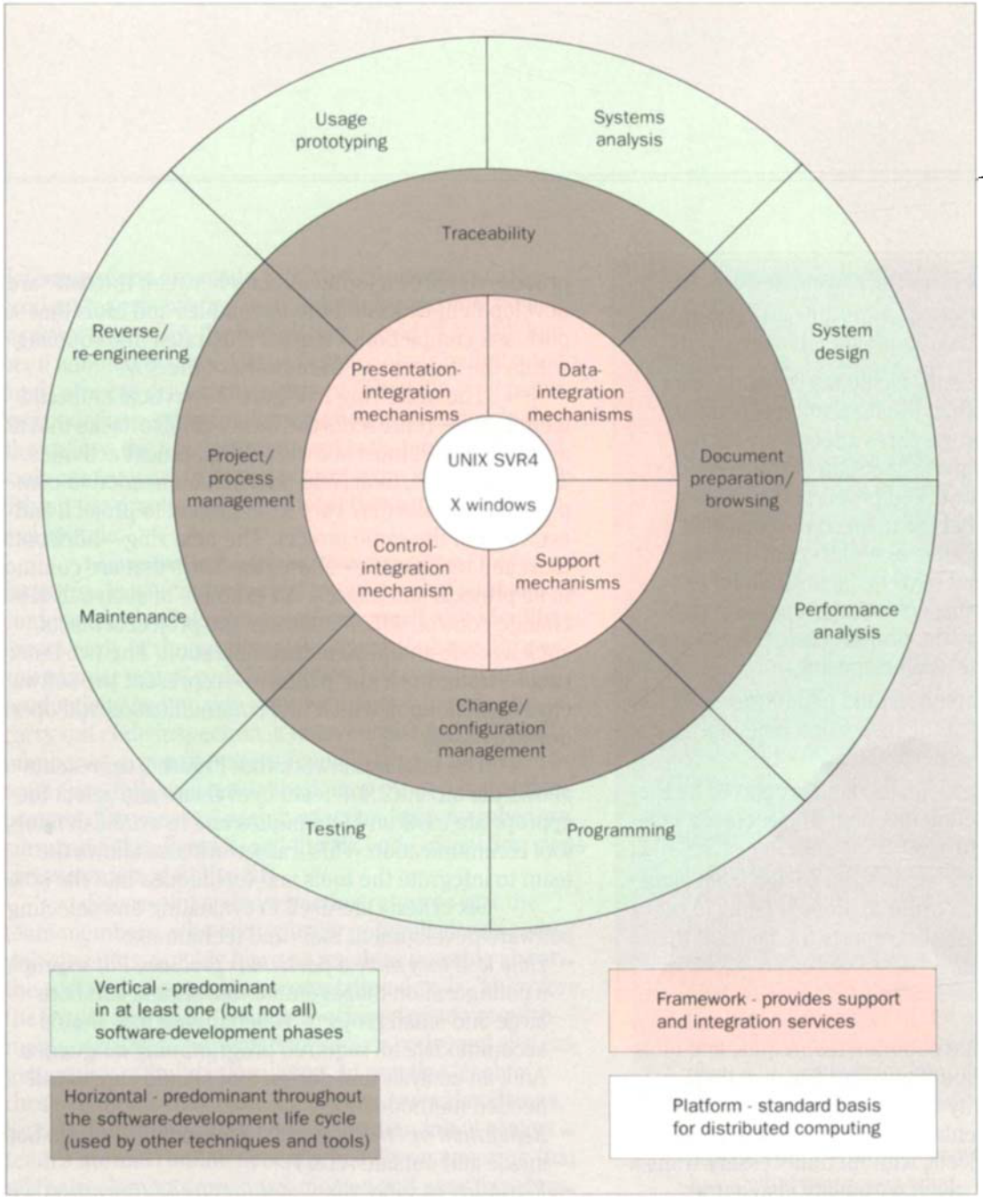

Figure 2. Effective implementation of the T-aspects requires a tight coupling between the software-development tasks and the techniques and tools that support task completion. The outer ring represents the basic, familiar tasks that are associated with most software-development activities. The two inner rings represent the software environment upon which tool communication and operation are based.

a software-development organization what the architecture function is to a product the organization builds. Software process engineering plays a major role in the synchronization and integration of all the T-aspects described earlier.

Synchronization and Integration. Establishing the engineering discipline in software production has a degree of difficulty that can be likened to changing clothes while jogging along an uneven path. In the transition journey, few organizations have the luxury of starting from the very beginning. Momentum on the current operating paradigm continues, project change-requests must be accommodated, and existing commitments must be honored. Therefore, in order for an organization to advance successfully toward engineering discipline—while maintaining team-member harmony—careful synchronization and integration of all the T-aspects must be maintained. This will benefit an organization three ways:

- Team members become more process focused, and have a better understanding of their role in the organization's migration journey.
- Team members working on single or multiple projects can use common terminology and follow specifically defined tasks, helping to establish a repeatable and stable software-production process and allowing more accurate measurements.

Panel 2. Roles of Code-Inspection Team Members

Author. The author develops the code unit submitted for inspection.

Moderator. The moderator facilitates the code inspection and ensures that: the inspection is planned properly; the team prepares adequately for the inspection meeting; the meeting is run properly; the code unit is reworked properly after the inspection; and, that the inspection results are reported properly after rework is complete. In summary, the moderator is responsible for ensuring that the code inspection conforms with the process defined by the project. Also, the moderator may participate as an inspector.

Reader. The reader interprets and paraphrases the code for the team at the inspection meeting. The reader is also an inspector.

Recorder. The recorder logs the faults reported by the team at the inspection meeting. The recorder is usually an inspector.

Inspector. The inspector studies the design, code, and coding standards in order to identify faults in the code. The inspector also reports the faults at the inspection meeting.

- A set of consistent and common techniques and tools can be used throughout the entire organization. Increased productivity results because libraries, data models, and documentation can be transferred and applied to other projects, without unnecessary translation or reformatting. Such portability also helps reduce training costs, and improves team-member mobility across projects.

MOSAIC and SDE Implementation

In order to address implementation of the T-aspects and the associated synchronization and integration requirements, an SDE—and related supporting services—have been developed. This section discusses the construction and implementation of an SDE. It also presents an example of how the code-inspection technique is inserted into the code-generation task through proper support of integrated tools.

SDE Construction Using the T-Aspects. Effective implementation of the T-aspects in the production process requires a tight coupling between the software-development tasks and the techniques and tools that support task completion. Figure 2 illustrates this coupling within the conceptual framework of the SDE.

The outer ring of Figure 2—vertical tools and techniques—represents the basic, familiar tasks that are associated with most software-development activities. The techniques, tools, and information needed to complete these tasks may vary from project to project, and even within the same project. The next ring—horizontal tools and techniques—shows the tasks that are common to all projects and phases. An example of such a task is change control, which monitors the project artifacts, such as code and design documentation. The two inner rings—framework and platform—represent the software environment upon which tool communication and operation are based.

The total framework that Figure 2 represents allows the MOSAIC/SDE team to evaluate and select the appropriate tools and techniques and to establish inter-tool communication. This framework also allows the team to integrate the tools and techniques into the SDE.

Six criteria are used in evaluating and selecting software-development tools and techniques:

- *How well they solve a particular problem.* For example, a configuration-management tool should suit both large and small projects. A debugging tool should accommodate all required programming languages. And, an analysis-and-design tool should support all needed methodologies.
- *Reputation or "track record."* This includes usage both inside and outside AT&T.
- *Flexibility in supporting additions and integrations without source-code modifications.* Examples include project-specific screens that can be added to the front end of a configuration-management tool; sufficient flexibility of the user-interface tool in supporting additional user defined operations; and, a tool having a published interface that allows information to be extracted from, or input to, the tool.
- *Vendor support.*
- *Existence of effective training courses.*
- *Competitive pricing.*

To help ease the transition of any project, tool-assessment and process-engineering services are also provided. These services assist in determing which tools, techniques, and platforms are appropriate. The

determinations are made with special attention to the product-teams' existing staff, schedule, and cost constraints. Currently, the foundation provided by SDE has been adopted by over 100 projects on their path toward a more process driven environment. The following paperless code-inspection implementation is an example that shows how the SDE integration of task, techniques, and tools can be used to ease a project team's effort in establishing the practice of code inspection as part of its normal routine.

Implementing Paperless Code Inspection. *Code inspection* is a technique used to ensure that source code conforms with all design specifications. It also identifies errors early in the product-development phase. Proper and regular use of code inspection has resulted in both productivity and quality improvements.[9] In order to carry out code inspection, a team consisting of a small number of group members (usually five) is formed. The team then follows a predefined, step-by-step inspection process. The roles of each inspection-team member are summarized in Panel 2, and the six code-inspection steps are summarized in Panel 3.

Successful code inspection requires that the team members, who are typically unfamiliar with code-unit specifics, quickly learn as much as possible about the code's design and implementation details. Therefore, the inspectors must frequently access written requirements, design documentation, source code, and system-generation or build information. In practice, each of these elements may have undergone several revisions. Successfully integrating code inspection into a project team's normal routine depends heavily on ensuring that all these elements are synchronized and easy to access.

Paperless code inspection under SDE consists of the following two elements and related subelements:

1. *Establishing a foundation.* Five tools are typically used in the inspection of source code. These tools include:
   - *The SMARTsystem* (software-development, maintenance, and reverse-engineering tool). (SMARTsystem is a registered trademark of Procase Corporation.) It provides capabilities for C programmers and facilitates the analysis, modification, and management of code.
   - *FrameMaker* (FrameMaker is a registered trademark of Frame Technology Corporation), which is a document-preparation system that integrates word processing, page layout, graphics, tables, and other functions into a single, easy-to-use application. It has a what-you-see-is-what-you-get user interface that makes document browsing easy.
   - *nmake*, which is a software-generation system that facilitates system development from source code.
   - *Sablime®*, which is a source-code-version control system that supports configuration management.
   - *cimgr*, which is an annotation editor that records points of irregularity and inspectors' comments.
2. *Using the tools in support of techniques.* The following six steps detail the code-inspection process:
   - *Planning.* The author or moderator uses *cimgr* to invoke the calendar manager that resides on the Sun (Sun is a registered trademark of Sun Microsystems, Inc.) operating-system distributions. The calendar manager allows the author to: view

Panel 3. The Six Code-Inspection Process Steps

Planning. The author and moderator assemble the team, plan the meetings, and distribute the inspection materials to the team.

Overviewing. The author presents the design to the team.

Preparing. The inspectors individually compare the code to the design to identify inconsistencies and code-implementation problems. The inspectors also ensure that the code conforms with all applicable project-coding standards.

Meeting. The reader guides the team through the code. While preparing for the meeting, the inspectors report the faults they found. During the meeting, the inspectors report any additional faults, while the reader guides the team through the code. The recorder logs the faults as they are reported.

Reworking. The author modifies the code unit according to the fault list developed at the code-inspection meeting. The moderator audits the rework to verify that the author resolved all the faults.

Reporting. The moderator submits a summary of the code inspection to the process owner, and informs the project manager of the inspection's disposition. The process owner ensures that data is placed in archives for later analysis.

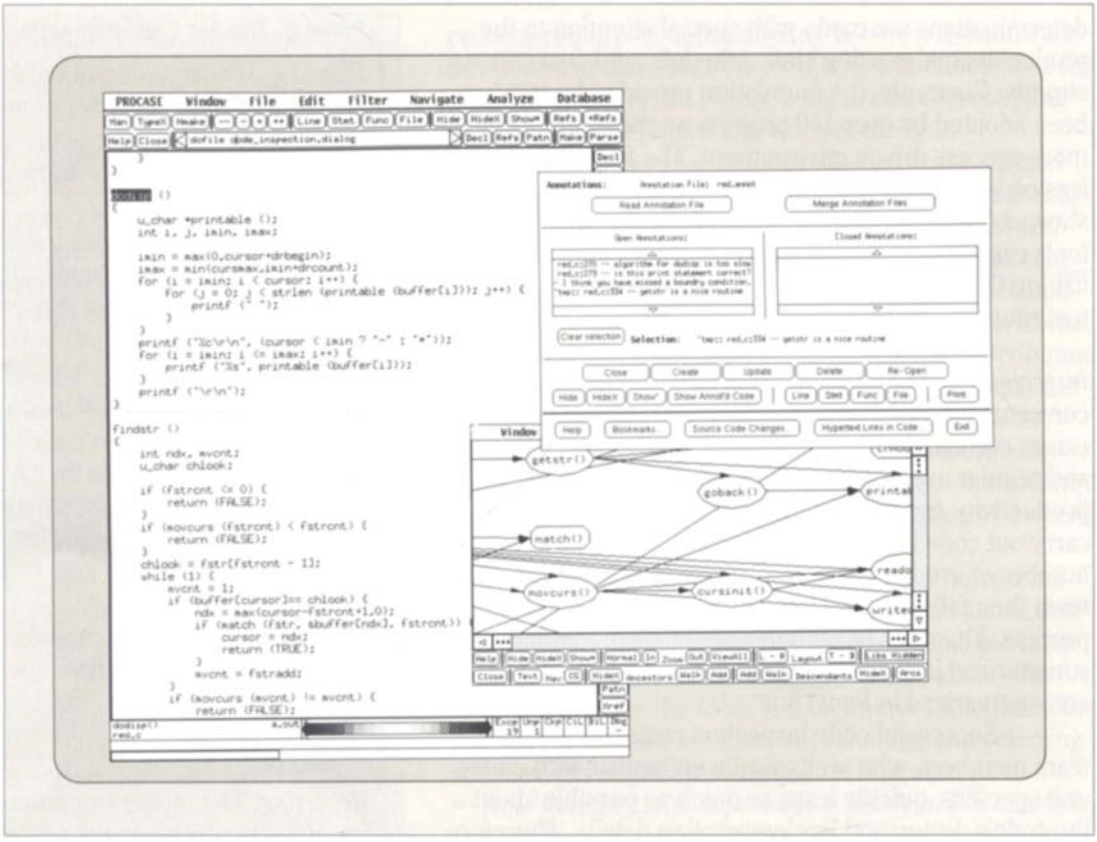

Figure 3. This is a typical screen image of the "meeting" code-inspection step. The moderator can use *cimgr* to merge all annotation files from the inspectors and remove any redundant annotations before the meeting. The moderator, author, and inspectors use *cimgr* and the SMARTsystem during the meeting to help resolve any outstanding annotations.

the calendars of the inspectors and moderator; allocate an available time slot; and, send electronic mail, informing all participants of the scheduled meeting.

- *Overviewing.* The author uses *cimgr* and the SMARTsystem to store the new code that is to be reviewed. Through integration of *nmake* and the SMARTsystem, the author can ensure that the correct code will be available for the inspectors. Such integration enables the inspectors to view the "build" directives that were used to create the code being reviewed. In addition, *cimgr* supports the linking of code to FrameMaker documents. Using this link, the author can direct inspectors to the appropriate requirements and design documents. Once these items have been collected, the author can use electronic mail to inform the inspectors that the material is ready for review.
- *Preparing.* The *cimgr* editor will automatically place each inspector into the environment that was created by the author. While in that environment, the inspectors can use the features of the SMARTsystem, FrameMaker, and Sablime to understand the code and to review documents. If necessary, the inspectors can extract the official source code from Sablime and compare the new code under review with the previous official version. Using *cimgr*, each inspector can also create an annotation to highlight potential errors or to indicate comments. *cimgr* links the annotation to the correct source-code line.
- *Meeting.* The moderator can use *cimgr* to merge all annotation files from the inspectors and remove any redundant annotations before the meeting. The moderator, author, and inspectors use *cimgr* and the SMARTsystem during the meeting to help resolve any outstanding annotations. Figure 3 is a typical screen image of these activities.
- *Reworking.* The author performs this step once a decision is reached. The file is extracted from Sablime, by means of the SMARTsystem, and integrated into Sablime. The author then makes any

necessary changes. Finally, the new official file is placed under Sablime.

- *Collecting.* For this final step, *cimgr* can be used to collect data for future analysis. This data might include the amount of code reviewed, number of faults, preparation time per inspector, meeting length, and author-rework time. Such metrics as preparation and inspection rates, fault density, and fault-detection effort can be derived from the data.

Equipped with the paperless capability, a project team can train its group members more effectively with the essence of the code-inspection technique and the use of supported tools, and plan and manage the transition of introducing and practicing this technique.

Conclusion

This paper discusses the challenges faced by software-development organizations in transforming production operations from art and craft to disciplined engineering. To be successful, an organization must systematically address the T-aspects—task, technique, tool, training, and transition—in technology insertion.

An example of the coupling between software-development tasks—and the techniques and tools that support task execution—is the MOSAIC/SDE team's experience in constructing and implementing the SDE. The SDE provides a foundation upon which organizations can build a specifically defined, process driven environment for software production.

Two overarching concepts are also discussed:

- Planning and defining a new software-development environment are only the first relatively simple steps in the migration process. Managing the transition to the new environment is the real challenge.
- Proper integration and synchronization of all the tools, techniques, and tasks will facilitate the transition.

References

1. W. S. Humphrey, *Managing the Software Process*, Addison-Wesley Publishing Company, Menlo Park, California, ISBN 0-201-18095-2, 1990.
2. J. R. Johnson, *The Software Factory*, Second Edition, QED Information Sciences, Inc., Wellesley, Massachusetts, ISBN 0-89435-348-9, 1991.
3. M. A. Cusumano, *Japan's Software Factories*, Oxford University Press, New York City, ISBN 0-19-506216-7, 1991.
4. L. J. Arthur, *Improving Software Quality*, John Wiley & Sons, Inc., New York City, ISBN 0-471-57804-5, 1993.
5. H. T. Yeh, *Software Process Quality*, McGraw-Hill Inc., New York City, ISBN 0-07-072272-2, 1993.
6. *ISO-9000 International Standards for Quality Management*, Second Edition, International Organization for Standardization, Geneva, Switzerland, ISBN 92-67-10172, 1992.
7. *Capability Marketing Model for Software*, Version 1.1, Software Engineering Institute, Pittsburgh, Pennsylvania, 1993.
8. M. H. Fallah and A. M. Jrad, "SQA—A Proactive Approach to Assuring Software Quality," *AT&T Technical Journal*, Vol. 73, No. 1, January/February 1994, pp. 26–33 (this issue).
9. R. Sharma, *Proceedings of the 6th Annual Software Quality Week Conference*, San Francisco, California, May 1993.

(Manuscript approved December 1993)

T. Richard Hsueh *is a technical manager in the Software Technology Center at AT&T Bell Laboratories in Murray Hill, New Jersey. His efforts are focused on two main areas of responsibility: improving software productivity through establishing the process-engineering discipline; and large-scale technology reuse, transfer, and insertion for the software-product-development life cycle. Mr. Hsueh received a B.S. in electrical engineering, as well as an M.B.A., from the National Chiao-Tung University in Taiwan. He also received an M.S. and a Ph.D. in industrial engineering from the University of Nebraska in Lincoln. Mr. Hsueh, a senior member of the Institute of Industrial Engineers, joined AT&T in 1984.*

Thomas F. Houghton *is a technical manager of the Software Development Environment group at AT&T Bell Laboratories in Murray Hill, New Jersey. He is responsible for defining software tools and associated processes that can increase developer productivity and reduce development intervals. He has also been involved in UNIX system development and testing since 1978. (UNIX is a registered trademark of UNIX Systems Laboratories.) Mr. Houghton received a B.S. in electrical engineering from the University of Delaware in Newark, and an M.S. in computer science from Northwestern University in Evanston, Illinois. He joined AT&T in 1978.*

Joseph F. Maranzano *is head of the software process and architecture department at AT&T Bell Laboratories in Murray Hill, New Jersey. He is responsible for designing a reusable-process-asset library, and for managing the deployment of best current practices (BCPs). Mr. Maranzano has a B.S. in electrical engineering from the Polytechnic Institute of New York in Brooklyn, and an M.S. in electrical engineering from New York University. Mr. Maranzano, an AT&T Bell Laboratories fellow, joined the company in 1964.*

Gerald P. Pasternack *is the department head in the Software-Development-Environment Technologies group at AT&T Bell Laboratories in Murray Hill, New Jersey. He is responsible for software engineering, environments, technologies, and associated tools including computer aided software engineering (CASE), object oriented technologies, and the C++ programming language. Mr. Pasternack received B.S., M.S., and Ph.D. degrees in electrical engineering, all from the Polytechnic Institute of New York in Brooklyn. He joined AT&T in 1961.*