

Software System Engineering: A Tutorial

Applying system engineering principles specifically to the development of large, complex software systems provides a powerful tool for process and product management.

Richard H. Thayer
California State
University,
Sacramento

Software systems have become larger and more complex than ever. We can attribute some of this growth to advances in hardware performance—advances that have reduced the need to limit a software system's size and complexity as a primary design goal. Microsoft Word is a classic example: A product that would fit on a 360-Kbyte diskette 20 years ago now requires a 600-Mbyte CD.

But there are other reasons for increased size and complexity. Specifically, software has become the dominant technology in many if not most technical systems. It often provides the cohesiveness and data control that enable a complex system to solve problems.

Figure 1 is a prime example of this concept. In an air traffic control system, software connects the airplanes, people, radar, communications, and other equipment that successfully guide an aircraft to its destination. Software provides the system's major technical complexity.

The vast majority of large software systems do not meet their projected schedule or estimated cost, nor do they completely fulfill the system acquirer's expectations. This phenomenon has long been known as the *software crisis*.¹ In response to this crisis, software developers have introduced different engineering practices into product development.

Simply tracking a development project's managerial and technical status—resources used, milestones accomplished, requirements met, tests completed—does not provide sufficient feedback about its health. Instead, we must manage the technical processes as well as its products. System engineering provides the tools this technical management task requires.

The application of system engineering principles to the development of a computer software system produces activities, tasks, and procedures called *software system engineering*, or SwSE. Many practitioners consider SwSE to be a special case of system engineering, and others consider it to be part of software engineering. However, we can argue that SwSE is a distinct and powerful tool for managing the technical development of large software projects.

This tutorial integrates the definitions and processes from the IEEE software engineering standards² into the SwSE process. A longer version that includes a detailed step-by-step approach for implementing SwSE is available in *Software Engineering Volume 1: The Development Process*, part of the IEEE Computer Society's "best practices series."³

SYSTEMS AND SYSTEM ENGINEERING

A *system* is a collection of elements related in a way that allows a common objective to be accomplished. In computer systems, these elements include hardware, software, people, facilities, and processes.

System engineering is the practical application of scientific, engineering, and management skills necessary to transform an operational need into a description of a system configuration that best satisfies that need. It is a generic problem-solving process that applies to the overall technical management of a system development project. This process provides the mechanism for identifying and evolving a system's product and process definitions.

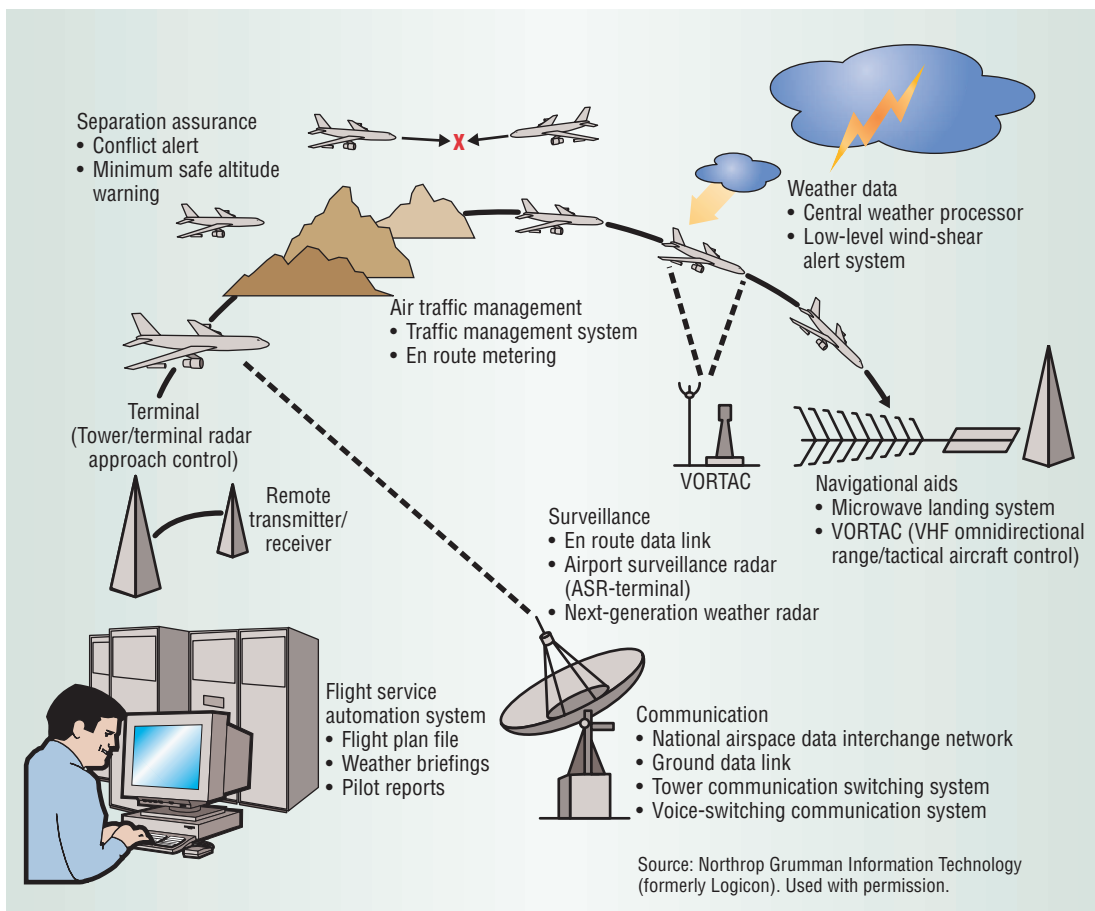


Figure 1. Air traffic control system environment. Software ties the elements of large systems together, and is frequently the most complex and technically challenging part of the system.

IEEE Std. 1220-1998 describes the system engineering process and its application throughout the product life cycle.⁴ System engineering produces documents, not hardware. The documents associate developmental processes with the project's life-cycle model. They also define the expected process environments, interfaces, products, and risk management tools throughout the project.

System engineering involves five functions:

- *Problem definition* determines the needs and constraints through analyzing the requirements and interfacing with the acquirer.
- *Solution analysis* determines the set of possible ways to satisfy the requirements and constraints, analyzes the possible solutions, and selects the optimum one.
- *Process planning* determines the tasks to be done, the size and effort to develop the product, the precedence between tasks, and the potential risks to the project.
- *Process control* determines the methods for controlling the project and the process, measures progress, reviews intermediate products, and takes corrective action when necessary.
- *Product evaluation* determines the quality and quantity of the delivered product through evaluation planning, testing, demonstration, analysis, examination, and inspection.

System engineering provides the baseline for all project development, as well as a mechanism for defining the *solution space*—that is, the systems and the interfaces with outside systems. The solution space describes the product at the highest level—before the system requirements are partitioned into the hardware and software subsystems.

This approach is similar to the software engineering practice of specifying constraints as late as possible in the development process. The further into the process a project gets before defining a constraint, the more flexible the implemented solution will be.

WHAT IS SOFTWARE SYSTEM ENGINEERING?

The term software system engineering dates from the early 1980s and is credited to Winston W. Royce,⁵ an early leader in software engineering. SwSE is responsible for the overall technical management of the system and the verification of the final system products. As with system engineering, SwSE produces documents, not components. This differentiates it from software engineering (SwE), which produces computer programs and users' manuals.

SwSE begins after the system requirements have been partitioned into hardware and software subsystems. SwSE establishes the baseline for all project software development. Like SwE, it is both a technical and a management process. The SwSE tech-

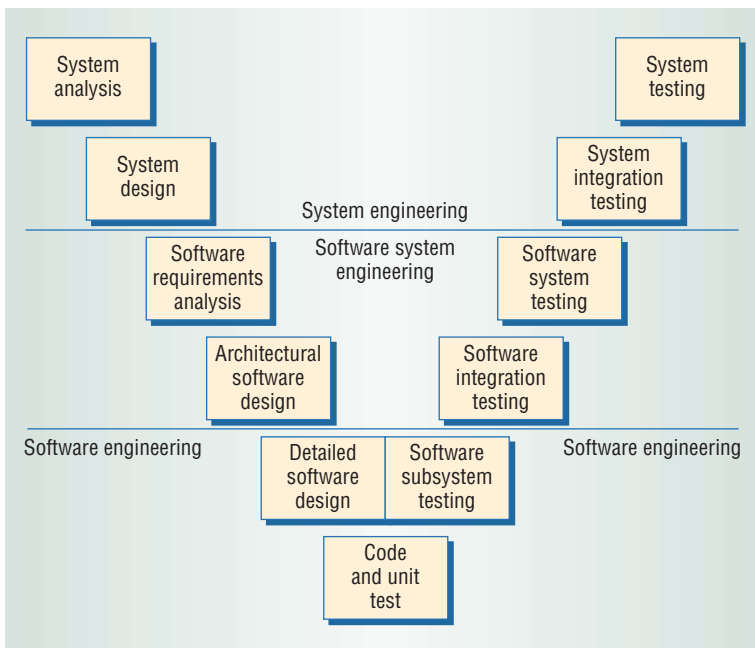


Figure 2. Engineering relationships between system engineering, software system engineering (SwSE), and software engineering. SwSE is responsible for requirements analysis, architectural design, and final software-system testing.

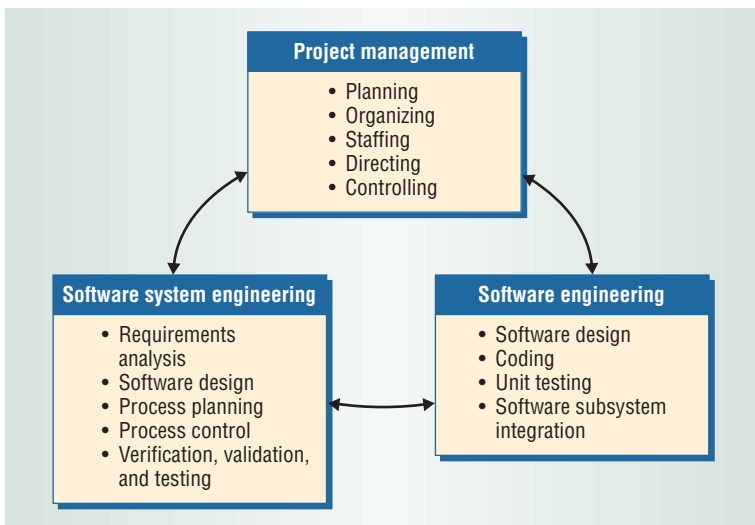


Figure 3. Management relationships between software system engineering (SwSE), software engineering, and project management. SwSE is responsible for determining the technical approach.

nical process is the analytical effort necessary to transform an operational need into

- a software system description;
- a software design of the proper size, configuration, and quality;
- software system documentation in requirements and design specifications;
- the procedures necessary to verify, test, and accept the finished software product; and
- the documentation necessary to use, operate, and maintain it.

SwSE is not a job description. It is a process that many people and organizations perform: system engineers, managers, software engineers, programmers, and—not to be ignored—acquirers and users.

As large system solutions become increasingly dependent on software, a system engineering approach to software development can help avoid the problems associated with the software crisis.

Software developers often overlook system engineering and SwSE in their projects. They consider systems that are all software or that run on commercial off-the-shelf computers to be just software projects, not system projects. Ignoring the systems aspects of software development contributes to our long-running software crisis.

SwSE and software engineering

Both SwSE and SwE are technical and management processes, but SwE produces software components and their supporting documentation. Specifically, software engineering is

- the practical application of computer science, management, and other sciences to the analysis, design, construction, and maintenance of software and its associated documentation;
- an engineering science that applies the concepts of analysis, design, coding, testing, documentation, and management to the successful completion of large, custom-built computer programs under time and budget constraints; and
- the systematic application of methods, tools, and techniques that achieve a stated requirement or objective for an effective and efficient software system.

Figure 2 illustrates the engineering relationships between system engineering, SwSE, and SwE. Traditional system engineering does initial analysis and design as well as final system integration and testing.

During the initial stage of software development, SwSE is responsible for software requirements analysis and architectural design. SwSE also manages the final testing of the software system. Finally, SwE manages what system engineers call *component engineering*.

SwSE and project management

The project management process involves assessing the software system's risks and costs, establishing a schedule, integrating the various engineering specialties and design groups, maintaining configuration control, and continuously auditing

Table 1. System engineering functions correlated to software system engineering (SwSE).

System engineering function	SwSE function	SwSE function description
Problem definition	Requirements analysis	Determine needs and constraints by analyzing system requirements allocated to software
Solution analysis	Software design	Determine ways to satisfy requirements and constraints, analyze possible solutions, and select the optimum one
Process planning	Process planning	Determine product development tasks, precedence, and potential risks to the project
Process control	Process control	Determine methods for controlling project and process, measure progress, and take corrective action where necessary
Product evaluation	Verification, validation, and testing	Evaluate final product and documentation

the effort to ensure that the project meets costs and schedules and satisfies technical requirements.⁶

Figure 3 illustrates the management relationships between project management, SwSE, and SwE. Project management has overall management responsibility for the project and the authority to commit resources. SwSE determines the technical approach, makes technical decisions, interfaces with the technical acquirer, and approves and accepts the final software product. SwE is responsible for developing the software design, coding the design, and developing software components.

THE FUNCTIONS OF SOFTWARE SYSTEM ENGINEERING

Table 1 lists the five main functions of system engineering correlated to SwSE, along with a brief general description of each SwSE function.

Requirements analysis

The first step in any software development activity is to determine and document the system-level requirements in either a system requirements specification (SRS) or a software requirements specification or both. Software requirements include capabilities that a user needs to solve a problem or achieve an objective as well as capabilities that a system or component needs to satisfy a contract, standard, or other formally imposed document.⁷

We can categorize software requirements as follows:⁸

- *Functional requirements* specify functions that a system or system component must be capable of performing.
- *Performance requirements* specify performance characteristics that a system or system component must possess, such as speed, accuracy, and frequency.
- *External interface requirements* specify hardware, software, or database elements with which a system or component must interface, or set forth constraints on formats, timing, or

other factors caused by such an interface.

- *Design constraints* affect or constrain the design of a software system or software system component, for example, language requirements, physical hardware requirements, software development standards, and software quality assurance standards.
- *Quality attributes* specify the degree to which software possesses attributes that affect quality, such as correctness, reliability, maintainability, and portability.

Software requirements analysis begins after system engineering has defined the acquirer and user system requirements. Its functions include identification of all—or as many as possible—software system requirements, and its conclusion marks the established requirements baseline, sometimes called the allocated baseline.²

Software design

Software design is the process of selecting and documenting the most effective and efficient system elements that together will implement the software system requirements.⁹ The design represents a specific, logical approach to meet the software requirements.

Software design is traditionally partitioned into two components:

- *Architectural design* is equivalent to system design, during which the developer selects the system-level structure and allocates the software requirements to the structure's components. Architectural design—sometimes called top-level design or preliminary design—typically defines and structures computer program components and data, defines the interfaces, and prepares timing and sizing estimates. It includes information such as the overall processing architecture, function allocations (but not detailed descriptions), data flows, system utilities, operating system interfaces, and storage throughput.

Table 2. Process planning versus project planning.

Software system engineering planning activities	Project management planning activities
Determine tasks to be done	Determine skills necessary to do the tasks
Establish order of precedence between tasks	Establish schedule for completing the project
Determine size of the effort (in staff time)	Determine cost of the effort
Determine technical approach to solving the problem	Determine managerial approach to monitoring the project's status
Select analysis and design tools	Select planning tools
Determine technical risks	Determine management risks
Define process model	Define process model
Update plans when the requirements or development environment change	Update plans when the managerial conditions and environment change

Table 3. Process control versus project control.

Software system engineering control activities	Project management control activities
Determine the requirements to be met	Determine the project plan to be followed
Select technical standards to be followed, for example, IEEE Std. 830	Select managerial standards to be followed, for example, IEEE Std. 1058
Establish technical metrics to control progress, for example, requirements growth, errors reported, rework	Establish management metrics to control progress, for example, cost growth, schedule slippage, staffing shortage
Use peer reviews, in-process reviews, software quality assurance, VV&T, and audits to determine adherence to requirements and design	Use joint acquirer-developer (milestone) reviews and SCM to determine adherence to cost, schedule, and progress
Reengineer the software requirements when necessary	Replan the project plan when necessary

- *Detailed design* is equivalent to component engineering. The components in this case are independent software modules and artifacts.

The methodology proposed here allocates architectural design to SwSE and detailed design to SwE.

Process planning

Planning specifies the project goals and objectives and the strategies, policies, plans, and procedures for achieving them. It defines in advance what to do, how to do it, when to do it, and who will do it.

Planning a SwE project consists of SwSE management activities that lead to selecting a course of action from alternative possibilities and defining a program for completing those actions.

There is an erroneous assumption that project management performs all project planning. In reality, project planning has two components—one accomplished by project management and the other by SwSE—and the bulk of project planning is a SwSE function. (This is not to say that project man-

agers might not perform both functions.)

Table 2 shows an example partitioning of planning functions for a software system project.

Process control

Control is the collection of management activities used to ensure that the project goes according to plan. Process control measures performance and results against plans, notes deviations, and takes corrective actions to ensure conformance between plans and actual results.

Process control is a feedback system for how well the project is going. Process control asks questions such as: Are there any potential problems that will cause delays in meeting a particular requirement within the budget and schedule? Have any risks turned into problems? Is the design approach still doable?

Control must lead to corrective action—either bringing the status back into conformance with the plan, changing the plan, or terminating the project.

Project control also has two separate components: control that project management accomplishes and control that software systems engineering accomplishes. Table 3 shows an example partitioning of control functions for a software system project.

Verification, validation, and testing

The verification, validation, and testing (VV&T) effort determines whether the engineering process is correct and the products are in compliance with their requirements.¹⁰ The following critical definitions apply:

- *Verification* determines whether the products of a given phase of the software development cycle fulfill the requirements established during the previous phase. Verification answers the question, “Am I building the product right?”
- *Validation* determines the correctness of the final program or software with respect to the user's needs and requirements. Validation answers the question, “Am I building the right product?”
- *Testing* is the execution of a program or partial program, with known inputs and outputs that are both predicted and observed, for the purpose of finding errors. Testing is frequently considered part of validation.

V&V is a continuous process of monitoring system engineering, SwSE, SwE, and project management activities to determine that they are following the technical and managerial plans, specifications,

standards, and procedures. V&V also evaluates the SwE project's interim and final products. Interim products include requirements specifications, design descriptions, test plans, and review results. Final products include software, user's manuals, training manuals, and so forth.

Any individual or functions within a software development project can do V&V. SwSE uses V&V techniques and tools to evaluate requirements specifications, design descriptions, and other interim products of the SwSE process. It uses testing to determine if the final product meets the project requirements specifications.

The last step in any software development activity is to validate and test the final software product against the software requirements specification and to validate and test the final system product against the SRS.

System engineering and SwSE are disciplines used primarily for technical planning in the front end of the system life cycle and for verifying that the plans were met at the project's end. Unfortunately, a project often overlooks these disciplines, especially if it consists entirely of software or runs on commercial off-the-shelf computers.

Ignoring the systems aspects of any software project can result in software that will not run on the hardware selected or will not integrate with other software systems. ■

References

1. W.W. Gibbs, "Software's Chronic Crisis," *Scientific Am.*, Sept. 1994, pp. 86-95.
2. IEEE, *Software Engineering Standards Collection*, vols. 1-4, IEEE Press, Piscataway, N.J., 1999.
3. R.H. Thayer, "Software System Engineering: A Tutorial," *Software Engineering Volume 1: The Development Process*, 2nd ed., R.H. Thayer and M. Dorfman, eds., IEEE CS Press, Los Alamitos, Calif., 2002, pp. 97-116.
4. IEEE Std. 1220-1998, *Standard for Application and Management of the System Engineering Process*, IEEE Press, Piscataway, N.J., 1998.
5. W.W. Royce, "Software Systems Engineering," seminar presented as part of the course titled Management of Software Acquisition, Defense Systems Management College, Fort Belvoir, Va., 1981-1988.
6. IEEE Std. 1058-1998, *Standard for Software Project Management Plans*, IEEE Press, Piscataway, N.J., 1998.
7. IEEE Std. 610.12-1990, *Standard Glossary of Software Engineering Terminology*, IEEE Press, Piscataway, N.J., 1990.
8. IEEE Std. 830-1998, *Recommended Practice for Software Requirements Specifications*, IEEE Press, Piscataway, N.J., 1998.
9. IEEE Std. 1016-1998, *Recommended Practice for Software Design Descriptions*, IEEE Press, Piscataway, N.J., 1998.
10. IEEE Std. 1012-1998, *Standard for Software Verification and Validation*, IEEE Press, Piscataway, N.J., 1998.

Richard H. Thayer is an emeritus professor in software engineering at California State University, Sacramento. He is also a consultant in software engineering and project management and a visiting researcher and lecturer at the University of Strathclyde, Glasgow, Scotland. He received a PhD in electrical engineering from the University of California at Santa Barbara. Thayer is a Fellow of the IEEE, an Associate Fellow of the American Institute of Aeronautics and Astronautics, and a member of the IEEE Computer Society and the ACM. Contact him at r.thayer@computer.org.



IEEE Distributed Systems Online brings you peer-reviewed features, tutorials, and expert-moderated pages covering a growing spectrum of important topics, including

- ✓ Grid Computing
- ✓ Distributed Agents
- ✓ Mobile and Wireless
- ✓ and more!
- ✓ Security
- ✓ Middleware

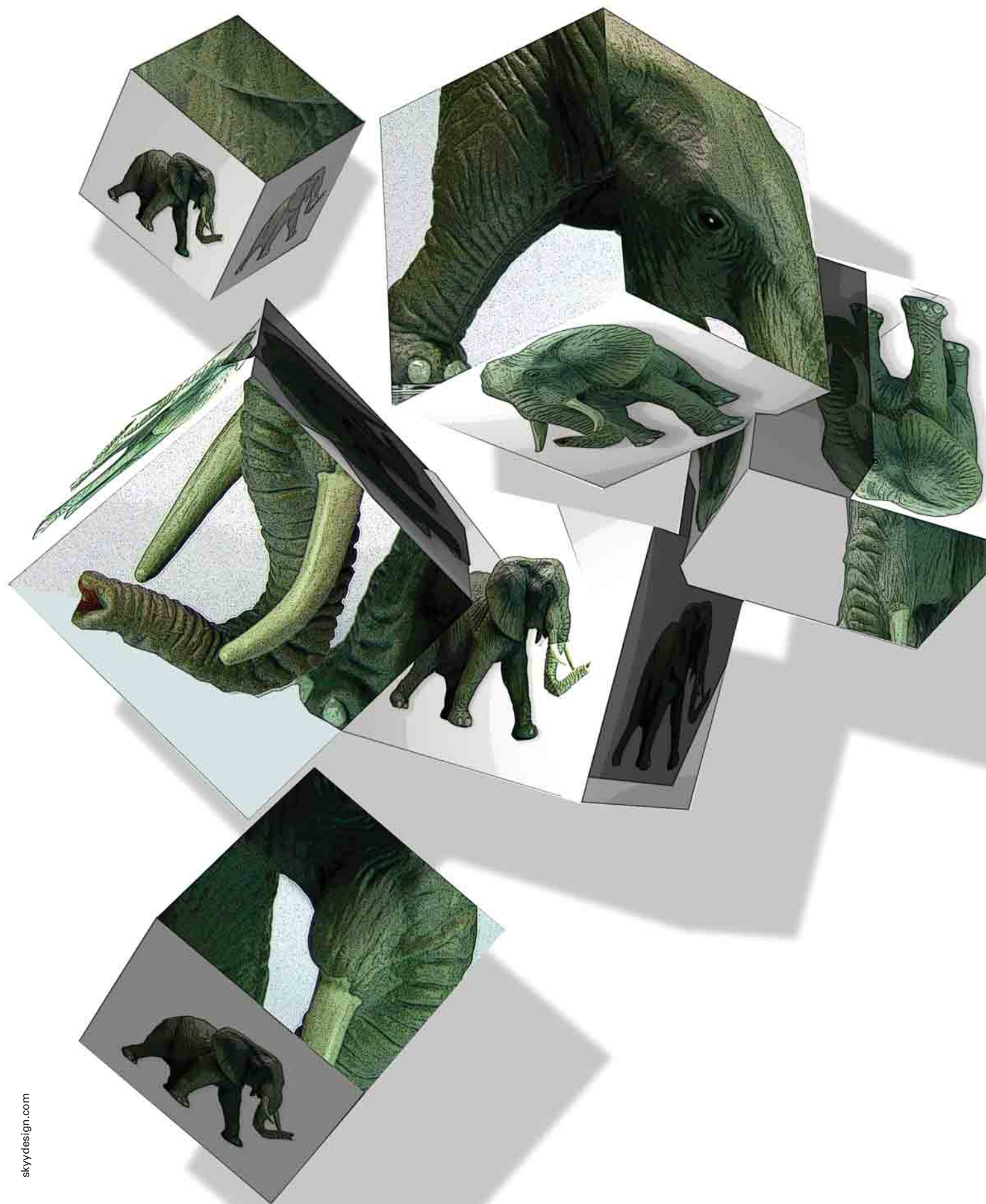
DS Online recently relaunched with a new design. Check us out for news, book reviews, and more!

To keep up with all that's happening in distributed systems, check out

dsonline.computer.org

Distributed Systems Online supplements the coverage in *IEEE Internet Computing* and *IEEE Pervasive Computing*. Each monthly issue includes magazine content and issue addenda such as source code, tutorial examples, and virtual tours.

To get regular updates, email dsonline@computer.org



The Blind Men and the Elephant: Views of Scenario-Based System Design

By Kentaro Go

Center for Integrated Information Processing
University of Yamanashi
Kofu 400-8511 Japan
go@yamanashi.ac.jp

By John M. Carroll

School of Information Sciences and Technology
The Pennsylvania State University
University Park, PA 16802 USA
jmcarroll@psu.edu

Six blind men encounter an elephant. Each of them touches a different part of the elephant and expresses what the elephant is. Although they are touching the same elephant, each man's description is completely different from that of the others. We have been using this story as a metaphor for understanding different views of scenario-based system design.

Scenario-based system design is an approach that employs scenarios as a central representation throughout the entire system lifecycle. The approach encourages user involvement in system design, provides shared vocabulary among the people participating in the system development project, envisions the uncertain future tasks of the system users, and enhances ease of developing instructional materials. It provides a good brainstorming tool for planning and allows the stakeholders to consider alternative choices in decision-making. It addresses dynamic, multiple, parallel, and/or distributed factors in a manageable manner. This rich variety of roles is selectively used from different viewpoints in diverse communities including human-computer interaction, strategic planning, require-

ments engineering, and object-oriented analysis/design. It is not easy to see what *the elephant* actually is.

The purpose of this paper is to help build a common language in software design for stakeholders from a wide range of disciplinary backgrounds providing an overview of scenario-based system design. By surveying the history and typical scenario usage in different fields, we demonstrate the importance of scenario-based approaches in system development. Human-computer interaction can be more effective if it can cooperate with and leverage efforts in other fields. Likewise, human-computer interaction can more effectively contribute to the evolution of the other fields. Perhaps scenarios offer an opportunity for this mutual contribution.

In addition, the history of scenario use provides a significant message. The tendency in the past 30 years is to look at finer grains of description (year-in-the-life scenario in strategic planning to day-in-the-life scenario in requirements engineering, to moment-to-moment scenario in human-computer interaction and object-oriented analysis/design). We

Marissa was not satisfied with her class today on gravitation and planetary motion. She is not certain whether smaller planets always move faster, or how a larger or denser sun would alter the possibilities for solar systems.

She stays after class to speak with Ms. Gould, but she isn't able to pose these questions clearly, so Ms. Gould suggests that she re-read the text and promises more discussion tomorrow.

Figure 1. An example narrative scenario from Rosson and Carroll [19]

suggest that now is the time to consider reintegrating the field by backing up the tree—that is, taking a moment-to-moment scenario and integrating it with a day-in-the-life scenario, and so forth. Indeed, current trends in application development that employ use scenarios with strategic planning are the beginning of this reintegration.

Scenarios in System Design

A scenario is a description that contains (1) actors, (2) background information on the actors and assumptions about their environment, (3) actors' goals or objectives, and (4) sequences of actions and events. Some applications may omit one of the elements or they may simply or implicitly express it. Although, in general, the elements of scenarios are the same in any field, the use of scenarios is quite different.

Scenarios are expressed in various media and forms. For example, scenarios can be textual narratives, storyboards, video mockups, or scripted prototypes. In addition, they may be in formal, semi-formal, or informal notation. A typical example of an informal scenario is a story (Figure 1), a kind of scenario frequently used for envisioning user tasks in human-computer interaction.

We will examine four communities that actively use scenario-based approaches: strategic planning, human-computer interaction, requirements engineering, and object-oriented analysis/design. Strategic planning forecasts the future environment of an organization and helps stakeholders plan actions. Human-computer interaction aims to design usable computer systems that support their users' tasks in a safe and fluent manner. Requirements engineering is about eliciting users' needs regarding computer systems, and about producing specifications; the specifications must be unambiguous, consistent, and complete. Object-oriented analysis/design is a methodology for constructing a world model; the model

is based on the idea of objects associating with data structure, class hierarchy, and object behavior. The scenario-based approaches of the four communities are summarized in Table 1, Table 2, and Table 3, and a history of scenario-related fields is summarized in Figure 2.

Strategic Planning

The discussion of scenario usage in computer system design is relatively new, but scenario-based approaches in planning and management have quite a long history. In his book *Thinking About the Unthinkable*, published in 1962, Herman Kahn considers scenarios as an aid to thought in uncertainty [14]. Kahn uses scenarios (1) as an analysis tool, (2) in narrative form, and (3) with psychological, social, and political assessments. He encourages combining role-playing exercises with scenario development. He says, "A scenario results from an attempt to describe in more or less detail some hypothetical sequence of events. Scenarios can emphasize different aspects of 'future history.'" He goes on to say, "The scenario is particularly suited to dealing with several aspects of a problem more or less simultaneously. By the use of a relatively extensive scenario, the analyst may be able to get a feel for events and the branching points dependent upon critical choices. These branches can then be explored more or less systematically." He then continues, "The scenario is an aid to the imagination."

He lists five advantages of the scenario as an aid to thinking:

1. Scenarios make the analyst salient to important events that should be taken into account in the uncertain future.
2. Scenarios require the treatment of details and dynamics.
3. Scenarios provide assistance in articulating the interaction of psychological, social, political, and military aspects, including the individual influence.

4. Scenarios can illustrate certain principles or questions.
5. Scenarios can be used to reason about alternative possibilities for past or present crises.

From a historical viewpoint, one of the milestone papers in the field is Pierre Wack's, which describes how in the late 1960s and early 1970s, the Royal Dutch/Shell Group constructed scenario-planning techniques to foresee and prepare for the 1973 oil crisis [20]. He distinguishes two ways to employ scenarios. On the one hand, scenarios are used for direct *forecasting*, such as the case with Royal Dutch/Shell. On the other hand, scenarios can be used for *planning*, since current scenarios can lend to brainstorming about the future. Wack discusses scenario planning as an approach that helps stakeholders think about new possibilities. He emphasizes the iterative construction of scenarios, in which new scenarios are derived through the analysis of the old ones.

After Wack's paper, scenario use in strategic planning focused on the second use, that is, as an analysis tool. In 1992, *Planning Review* had special issues on scenarios in strategic management.

In summary, the dominant idea in the

COMMUNITY	ACTOR	ENVIRONMENT
Strategic Planning	Specific organization CEO as in organizational role Planning organization may not be main actor	Other political, economical entities
Human-Computer Interaction	Specific user Has internal states	System Workplace contexts
Requirements Engineering	General user System Describes actor's behavior	System
Object-Oriented Analysis/ Design	General user System Describes actor's behavior	System

Table 1. Major characterizing elements of the scenario-based design approaches

COMMUNITY	SCENARIO USAGE
Strategic Planning	Envisioning uncertain future environment Providing communication tool Organizational learning Sharing a mental model among stakeholders
Human-Computer Interaction	Analyzing user tasks Envisioning future work Mock up and prototyping Evaluating the constructed system Deriving learning materials Developing design rationale
Requirements Engineering	Eliciting user requirements Deriving specifications Analyzing the current system usage Describing the current system usage Constructing test cases
Object-Oriented Analysis/ Design	Modeling objects, data structures and class hierarchy Analyzing problem domain Providing a model of real-world objects

Table 2. Typical scenario usage in design

COMMUNITY	UNCERTAIN FACTOR	GOAL OF SCENARIO-BASED APPROACH	DEVELOPMENT PROCESS	VIEW POINT	BACKGROUND GOAL OF COMMUNITY
Strategic Planning	Environment	List "what-if" questions and their answers	Iterative	Organization Technological changes Economics Social, political regulations Consumer attitudes	Plan a course of actions
Human-Computer Interaction	Use of system	Envision user requirements of (future) system use	Iterative Prototyping	Human Usability Cognition Emotion	Describe use of (future) systems Design usable computer system
Requirements Engineering	System requirements Functionality	Acquire user requirements and specify them	Waterfall Spiral	System architecture Development process	Specify systems Provide a good transition to the next development phase
Object-Oriented Analysis/ Design	Objects Data structures Class hierarchy	Identify objects, data structures and model class hierarchy	Iterative Incremental	System Object	Design a model of world

Table 3. Some factors that can be used to categorize scenario usage

COMMUNITY	1960s	1970s	1980s	1990s
Strategic Planning	Kahn's book, 1962 Royal Dutch/Shell scenario planning, c. 1965 [20]		Wack's paper, 1985 [20]	Planning Review's special issues, 1992
Human-Computer Interaction			diSessa, 1985 [7] Olympic Message System, 1987 [8]	SIGCHI bulletin's discussion, 1992 Carroll's book, 1995 [2]
Requirements Engineering			Hooper and Hsia, 1982 [9]	Potts et al., 1994 [17] Hsia et al., 1994 [10] CREWS' classification, 1996 [18]
Object-Oriented Analysis/Design			Use-case approach, 1987 [13]	Responsibility-driven approach, 1990 [22] Koskimies et al., 1997 [15]

Figure 2. A historical contrast of scenario usage and discussion in strategic planning, human-computer interaction, requirements engineering, and object-oriented analysis/design

strategic planning community is that scenario planning is a process of envisioning and critiquing multiple possible futures.

Human-Computer Interaction

Human-computer interaction is another field that actively discusses what scenarios are and how to use them in system design [1], [2], [5], [23]. Human-computer interaction uses scenarios to describe the use of systems and to envision more usable computer systems. To observe and then analyze the current usage of a system, it is necessary to involve authentic users. In this approach, actors in a scenario are specific people who carry out real or realistic tasks. To envision the use of a system that has not yet been constructed, the scenario writers have to describe potential users and what they may do with the system in extensive detail, including, for example, a description of workplace contexts.

Day-in-the-life scenarios are one of the most powerful methods for envisioning authentic computer use. They illustrate users' daily activity with computers over time. For example, a scenario might describe how a musician uses music software through various input devices and gets frustrated [16]. Day-in-the-life scenarios can be used to envision the future use of computers. One famous example

is Apple Computer's "Knowledge Navigator" video [6]; it shows how a person interacts with computer capabilities that have not yet been developed. The scenario gives a vivid view of how people could use computers as an intelligent assistant in daily life.

Envisioned scenarios can be analyzed to create explicit rationale for future designs [4]. Carroll and Rosson proposed an iterative process for writing scenarios and analyzing their psychological claims. They produced textual narrative scenarios for envisioning future use of a system; then, they conducted claims analysis by listing the positive and negative consequences of features of tools and artifacts in the scenarios. After examining the claims, they derived new scenarios. This iterative process makes the design of the system more precise at every stage.

Evaluation of systems is another way scenarios are used in human-computer interaction. During the evaluation process, careful observations of a real work setting are necessary. The technique is frequently used in human-computer interaction to analyze social aspects of user tasks. Scenarios are used in ethnographic field study as a device for describing the context of work. Scenarios are then analyzed to reveal how the work is

socially organized [11].

One of the significant views derived from the study of human-computer interaction is that scenarios are not specifications. Carroll contrasts two complementary perspectives: The perspectives clearly separate scenarios from specifications [2]. Scenarios are (1) concrete descriptions, (2) focus on particular instances, (3) work driven, (4) open-ended, fragmentary, (5) informal, rough, colloquial, and (6) envisioned outcomes. In contrast, specifications are (1) abstract descriptions, (2) focus on generic types, (3) technology driven, (4) complete, exhaustive, (5) formal, rigorous, and (6) specified outcomes.

The earliest scenario usage in human-computer interaction originates in the mid-1980s, and is cited by diSessa, and Gould, Boies, Levy, Richards, and Schoonard [7], [8]. diSessa explains the Boxer programming environment by using scenarios. He did not directly employ scenarios in the programming environment design but tried instead to explain the design by using scenarios. Gould and his colleagues' work did make the direct use of scenarios in system design; they used the scenario dialogues to design the 1984 Olympic Message System. There were discussions of scenarios in *SIGCHI Bulletin* in 1992, and an edited volume of scenario-based design was published in 1995 [2].

Requirements Engineering

Because the goal of requirements engineering is to elicit and specify users' requirements, its scenario usage focuses on analysis. In particular, it gives weight to how to specify the requirements and provide a smooth transition to the next development phase. Scenarios for this purpose, therefore, must be written from the system's viewpoint. This makes scenario-based requirements engineering relatively concrete and process-oriented as a methodology.

In requirements engineering, one of the earliest works is Hooper and Hsia [9]; they proposed the idea of scenarios as prototypes to

identify user requirements. "Prototyping is a 'quick and dirty' construction of a system (or part of a system). In prototyping by use of scenarios, one does not necessarily model the system or any component thereof directly, but rather represents the performance of the system for selected sequences of events." This approach can be simpler than the whole system modeling. By using scenarios, the users can simulate the real operation of a system and know their actual needs.

Typical scenario-based approaches in requirements engineering include the formal scenario analysis by Hsia, Samuel, Gao, Kung, Toyoshima, and Chen [10], the inquiry-based requirement analysis by Potts, Takahashi, and Anton [17], and the CREWS project [18].

Hsia and associates [10] proposed a formal approach to scenario analysis, a requirement analysis model in the early phases in software development. Their approach defines systematic development stages from the initial semi-formal scenarios to the final formal scenarios. Each of the stages except the first use scenarios in formal notation, which enable the system analysts to derive part of the system-requirement specification.

Potts and colleagues [17] developed the Inquiry Cycle Model of requirement analysis, which is "a structure for describing and supporting discussions about system requirement." The model consists of three phases of requirements: documentation, discussion, and evolution. These three phases make a cycle for acquiring and modeling the knowledge of problem domain.

Their scenarios are part of the requirements documentation, which are in hypertext form. The scenarios are intended to be semi-formal; in fact, they are expressed in tabular notation. "In the broad sense, a scenario is simply a proposed specific use of the system. More specifically, a scenario is a description of one or more end-to-end transactions involving the required system and its environment."

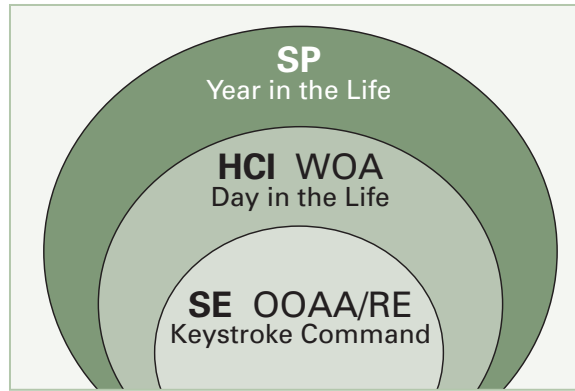


Figure 3. The nested structure of Strategic Planning (SP), Human-Computer Interaction (HCI), and Software Engineering (SE). HCI includes Work-Oriented Approach (WOA), and SE consists of Object-Oriented Analysis/Design (OOAD) and Requirements Engineering (RE).

The CREWS project (Cooperative Requirements Engineering with Scenarios) is a large European project on scenario use in requirements engineering [18]. Based on the existing techniques and tools, the project is trying to make scenario usage a systematic engineering discipline; hence, its approach to scenario usage is tool- and/or method-driven. The project has specific goals. With the multimedia-recorded system use, it provides a common medium for discussing system requirements among multiple stakeholders. With natural language understanding, it provides a device to elicit requirements in a graphical representation. By cooperative requirement animation and by comparison between a specification and its test scenarios, it expedites the validation process of a system.

In the CREWS project, Rolland, Achour, Cauvet, Ralyte, Sutcliffe, Maiden, Jarke, Haumer, Pohl, Dubois, and Heymans [18] have developed a framework to classify scenarios in scenario-based approaches. They give four dimensions: the form view, the contents view, the purpose view, and the lifecycle view. The form view represents the format of scenarios; for example, narrative texts, graphics, images, videos, and prototypes are in the form view. In addition, formality (e.g., formal, semi-formal, and informal) is discussed from this viewpoint. The contents view expresses what knowledge scenarios express. It consists of four elements: abstraction, context, argumentation, and coverage. In the purpose view, scenarios are cate-

gorized from the reasons of usage. The lifecycle view deals with how scenarios are handled (e.g., creation, refinement, and deletion). Applying the framework to eleven scenario-based approaches, they found out that scenarios are used to describe system behaviors interacting with its environment and most of them are in textual representation. They pointed out the importance of the formalization of scenario-based approaches, the variety of the application fields, and the need for practical scenario evaluation.

Object-Oriented Analysis/Design

Object-oriented analysis/design models an application domain. It identifies objects, data structures, and class hierarchies. Its viewpoint is that of a system model.

There are three typical scenario-based approaches: Jacobson's use-case approach [13], Wirfs-Brock's responsibility-driven approach [22], and Koskimies, Systa, Tuomi, and Mannisto's automated modeling support approach [17].

According to Jacobson, a use-case depicts how a user or another system uses a system. A basic concept behind the use-case approach is that a system is well described from a black-box view. A use-case model consists of two elements: actors and use-cases. In essence, "The actors represent what interacts with the system. They represent everything that needs to exchange information with the system. Since the actors represent what is outside the system,

we do not describe them in detail [12].”

Jacobson also claims that a use-case is fundamentally different from a scenario: Scenarios correspond to use-case instances. There is no correspondence to use-case classes. A use-case expresses all the possible paths of events, but a scenario describes part of the possible paths. In addition, a use-case seeks a formal treatment defining a model while a scenario seeks an informal treatment [12].

Another object-oriented approach relating to scenarios is the responsibility-driven approach by Wirfs-Brock [22]. She states that the approach emphasizes informal methods for characterizing objects, their roles, responsibilities, and interactions. By using the informal methods, the designer is to determine the software system, stereotype the actors, determine system use-cases, construct conversations, identify candidate objects, identify responsibilities of candidate objects, design collaborations, design class hierarchies, fully specify classes, and design subsystems.

The approach uses cards as tools; they are called “CRC cards” (Class-Responsibility-Collaboration cards). Each of the cards lists responsibilities and collaborations with other objects on the front side; the back side has initial notions in the roles of that object and its stereotypes.

Koskimies and his colleagues proposed automated support for modeling object-oriented software [15]. They follow the definition of scenarios in the Object Modeling Technique (OMT), a commonly used object-oriented analysis/design method: “In OMT, scenarios are informal descriptions of sequences of events occurring during a particular execution of a system.” In fact, Koskimies and his colleagues intended to formalize scenarios in order to provide an automated support of scenario-based system development; therefore, they formalized them as event-trace diagrams.

In summary, each of the four communities has its own use of scenarios. Strategic planning

involves lists of “what-if” questions followed by answers that comprise a scenario for action. Human-computer interaction uses scenarios to analyze a system and to envision a more usable system. Requirements engineering determines user and system needs and produces specifications *vis-a-vis* use scenarios. Object-oriented analysis/design scenario usage involves identifying objects and data structures and modeling a class hierarchy. There are clear similarities. Strategic planning and human-computer interaction exploit scenarios to envision future use, actions, and events. The analysis use of scenarios is common to human-computer interaction, requirements engineering, and object-oriented analysis/design. However, instead of seeing local relationships, we will discuss global relationships highlighting the insight of each community.

Relationship of the Four Communities

The relationship among the four communities can be discussed from their structure and life-cycle.

Structure of the Four Communities

The four communities-human-computer interaction, strategic planning, requirements engineering, and object-oriented analysis/design-have a nested relation, in which the latter two communities can be categorized into software engineering (see Figure 3).

From inside to outside, there are three layers: software engineering, human-computer interaction, and strategic planning. Human-computer interaction incorporates work-oriented approaches focusing on users’ tasks and user participation.

The nested structure describes the focus of each field based on the degree of tangibility of the target content of scenarios. Software engineering scenarios focus on real world objects or physical artifacts; furthermore, they include scenarios of the existing system use. Therefore

in the field, the materiality of a system attracts great attention for designers and analysts. In addition to tangible artifacts, human-computer interaction scenarios treat user tasks. User tasks are not easily touched. That is, their degree of tangibility is much lower than software engineering. Strategic planning deals with much more abstracted artifacts, such as future plans of an organization. It includes, in some degree, current technology as a basic assumption for planning.

Software engineering scenarios are relatively small in scope. They typically include keystroke- and command-level scenarios. Human-computer interaction scenarios are larger in scope; they deal with day-in-the-life scenarios. Strategic planning scenarios have the widest scope of the three groups; they treat events and issues of the grain of *year-in-the-life*.

The nested structure in Figure 3 also summarizes the history of human-computer interaction. Human-computer interaction—as it originated in software psychology—studied human aspects in traditional system design approach and the use of systems constructed from that approach [3]. It grew out of software engineering research and has focused on prac-

tice. More recently, its attention has shifted to work-oriented approaches with researchers recognizing that real work situations are social in nature, and that previous analysis focused only on a single user and single computer interaction. This shift of concept produced a new field: Computer-Supported Cooperative Work (CSCW). CSCW is about work-oriented system design, especially weighted on social and organizational aspects, an area that intersects with strategic planning.

The transition of research interests of design from concrete, physical artifacts to abstract, indefinite organizations of people is clear from a history of these fields relating to scenario-based design.

Lifecycle of Scenarios

The four communities make different use of scenarios in a system lifecycle; indeed, they assume a different lifecycle generally. Strategic planning and human-computer interaction presume development processes as complex, dynamic processes; therefore, they prefer to employ iterative design of scenarios. In strategic planning, Wack described a cyclic process of scenario development [20]. Similarly, in

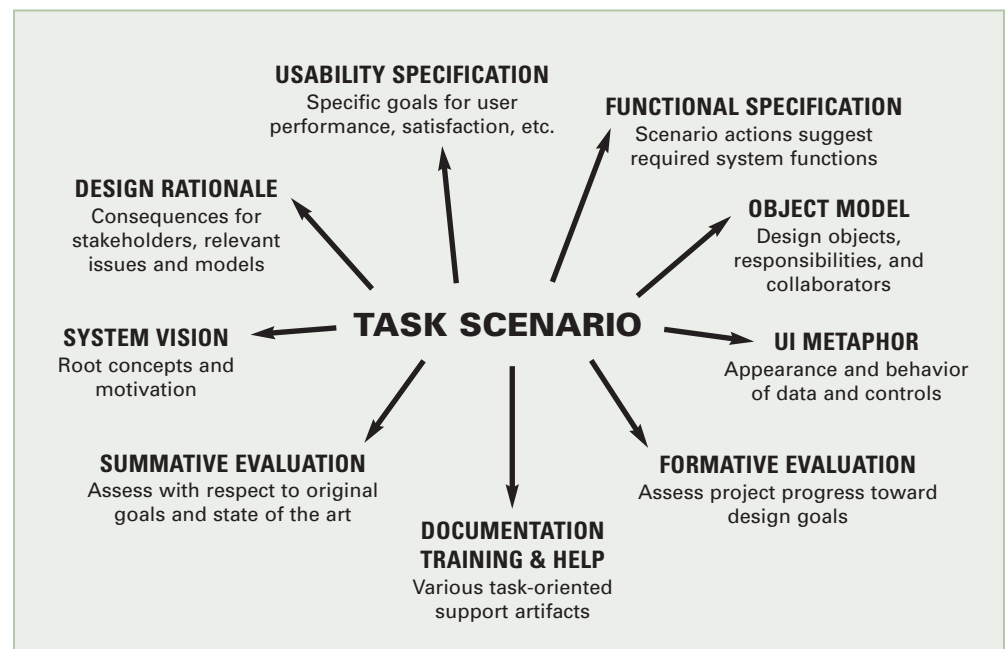


Figure 4. Scenarios provide a common language for design.

human-computer interaction, the dominant view is that of an iterative process of design. The requirements engineering community generally assumes that development processes are decomposable; thus, techniques for deriving specifications from scenarios at the end of the requirement acquisition phase become a fundamental issue. The object-oriented analysis/design community tends to apply the object-oriented approach to any phase in the development process; it emphasizes the seamlessness between analysis and design in development using incremental and iterative processes. This discipline can be applied to human-computer interaction and requirements engineering.

Scenarios as Common Language

Scenario-based techniques contextualize and concretize design thinking about people and technologies. Strategic planning scenarios capture organizational context, surprise-free continuations, and the status quo, as well as alternative, "what-if" scenarios. Requirements engineering scenarios help elicit and refine functional descriptions of future systems. Human-computer interaction scenarios help users and developers describe and evaluate technology currently in use and envision activities that new technology could enable. Object-oriented analysis/design use-cases of the system identify the possible event sequences of the system to accurately model the domain objects, data structures, and behaviors. These different applications of scenarios emphasize different viewpoints and different resolutions of detail, and they address different purposes. But the vocabulary of concrete narratives is accessible to and sharable by diverse stakeholders in a design project: planners and managers, requirements engineers, software developers, customer representatives, human-computer interaction designers, and the users themselves. In this sense, scenarios provide a common language for design.

ACKNOWLEDGMENTS

We would like to acknowledge the Dagstuhl Symposium on Scenario Management, Schloss Dagstuhl Internationale Begegnungs- und Forschungszentrum für Informatik, February 9-13, 1999, Saarbrücken, Germany, and its principal organizer, Matthias Jarke, for suggesting the direction of this paper. We thank Alistair Sutcliffe, Colette Rolland, Pei Hsia, Annie I. Anton, Lewis Erskine, and Craig Ganoe for the comments on the paper.

REFERENCES

- Carey, T., McKelvie, D., Bubie, W., & Wilson, J. (1991). Communicating human factors expertise through design rationales and scenarios. In D. Diaper & N. Hammond (Eds.), *People and Computers VI: Proceedings of the HCI '91 Conference*, August 20-23, Edinburgh, UK (pp. 117-130). Cambridge, UK: Cambridge University Press.
- Carroll, J. M. (Ed.). (1995). *Scenario-based design: Envisioning work and technology in system development*. New York: John Wiley & Sons.
- Carroll, J. M. (1997). Human-computer interaction: Psychology as a science of design. *International Journal of Human-Computer Studies*, 46, 501-522.
- Carroll, J. M. (2000). *Making use: Scenario-based design of human-computer interactions*. Cambridge, MA: MIT Press.
- Clarke, L. (1991). The use of scenarios by user interface designers. In D. Diaper and N. Hammond (Eds.), *People and Computers VI: Proceedings of the HCI '91 Conference*, August 20-23, Edinburgh, UK (pp. 103-115). Cambridge, UK: Cambridge University Press.
- Dubberly, H. & Mitch, D. (1987). *The Knowledge Navigator*. Apple Computer, videotape. Appears in B. A. Myers ed. HCI'92 Special Video Program.
- diSessa, A. (1985). A principled design for an integrated computational environment. *Human-Computer Interaction* 1(1), 1-47.
- Gould J. D., Boies, S. J., Levy, S., Richards, J. T., & Schoonard, J. (1997). The 1984 Olympic message system: A test of behavioral principle of system design. *Communications of the ACM* 30(9), 758-769.
- Hooper, J. W. & Hsia, P. (1982). Scenario-based prototyping for requirements identification. *Software Engineering Notes* 7(5), 88-93.
- Hsia, P., Samuel, J., Gao, J., Kung, D., Toyoshima, Y., & Chen, C. (1994). Formal approach to scenario analysis. *IEEE Software* 11(3), 33-41.
- Hughes, J. A., Randall, D., & Shapiro, D. (1992). Faltering from ethnography to design. In *Proceedings of CSCW'92 Conference*, Oct. 31-Nov. 4, Toronto, Canada (pp. 115-122). New York: ACM.
- Jacobson, I. (1995). The use-case construct in object-oriented software engineering. In J. M. Carroll (Ed.), *Scenario-based design: Envisioning work and technology in system development*, 309-336. New York: John Wiley & Sons.
- Jacobson, I., Christersson, M., Jonsson, P., & Overgaard, G. (1992). *Object-Oriented Software Engineering: A use-case driven approach*. Reading, MA: Addison-Wesley.
- Kahn, H. (1962). *Thinking about the unthinkable*. New York: Horizon Press.
- Koskimies, K., Systs, T., Tuomi, J., & Mannisto, T. (1998). Automated support for modeling OO software. *IEEE Software* 15(1), 87-94.
- Mountford, S. J. (1991). A day in the life of ... (Panel). In *Proceedings of CHI'91*, April 27-May 2, New Orleans, LA. (pp. 385-388). New York: ACM.
- Potts, C., Takahashi, K., & Anton, A. I. (1994). Inquiry-based requirements analysis. *IEEE Software* 11(2), 21-32.
- Rolland, C., Achour, C. B., Cauvet, C., Ralyte, J., Sutcliffe, A., Maiden, N. A. M., Jarke, M., Haumer, P., Pohl, K., Dubois, E., & Heymans, P. (1996). A proposal for a scenario classification framework. *Requirements Engineering* 3(1), 23-47.
- Rosson, M. B. & Carroll, J. M. (2001). *Usability engineering: scenario-based development of human computer interaction*. Redwood City, CA: Morgan Kaufmann.
- Wack, P. (1985). Scenarios: Uncharted waters ahead. *Harvard Business Review* 63(5), 72-89.
- Weidenhaupt, K., Pohl, K., Jarke, M., & Haumer, P. (1998). Scenarios in system development: Current practice. *IEEE Software* 15(2), 34-45.
- Wirfs-Brock, R., Wilkerson, B., & Wiener, L. (1990). *Designing object-oriented software*. Englewood Cliffs, NJ: Prentice Hall.
- Young, R. M. & Barnard, P. J. (1991). Signature tasks and paradigm tasks: new wrinkles on the scenarios methodology. In D. Diaper and N. Hammond (Eds.), *People and Computers VI: Proceedings of the HCI '91 Conference*, August 20-23, Edinburgh, UK (pp. 91-101). Cambridge, UK: Cambridge University Press.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without the fee, provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on services or to redistribute to lists, requires prior specific permission and/or a fee. © ACM 1072-5220/04/1100 \$5.00

In Practice: UML Software Architecture and Design Description

Christian F.J. Lange and Michel R.V. Chaudron, *Eindhoven University of Technology*
Johan Muskens, *Philips Research*

A user survey and industry case study analyses reveal common problems with UML models and some techniques for controlling their quality.

Since its introduction in 1997, the Unified Modeling Language has attracted many organizations and practitioners. UML is now the de facto modeling language for software development. Several features account for its popularity: it's a standardized notation, rich in expressivity—UML 2.0 provides 13 diagram types that enable modeling several different views and abstraction levels. Furthermore, UML supports domain-specific extensions using stereotypes and tagged values.

Finally, several case tools integrate UML modeling with other tasks such as generating code and reverse-engineering models from code.

In most projects, UML models are the first artifacts to systematically represent a software architecture.¹ They're subsequently modified and refined in the development process. Their importance has increased with the advent of the model-driven architecture methodology (www.omg.org/mda). However, little is known about the way UML is actually used in projects and the pitfalls of UML-based development.

To find out how practitioners use UML, we invited software architects to complete a Web-based anonymous questionnaire.² We also analyzed UML models in 14 industry case studies and compared the analytical results with the practitioner reports. Our study focused on UML use and model quality in actual projects rather than on its adequacy as a notation or language.

Practitioner reflections on UML use

Over a two-month period, 80 architects participated. Background questions revealed the following major responsibilities among respondents:

- analysis (66 percent),
- design (66 percent),
- specification (61 percent), and
- programming (52 percent).

The respondents came from different application domains. Most worked in information systems (61 percent); 28 percent worked in embedded systems, and a few worked in tool and operating systems development. Sixty percent worked in projects of more than five person-years.

Use of architectural views

We investigated UML's popularity for de-

scribing the different architectural views suggested by Philippe Kruchten in his “4 + 1 View Model.”³ However, we adopted the Rational Rose case tool terminology for the views because it’s well-known among most practitioners. Figure 1 shows survey results regarding respondents’ use of UML in use cases, logical views, component views, deployment, and scenarios.

Adherence to UML specification

The UML specification (www.uml.org) defines the language notation, syntax, and (informal) semantics. Figure 2 shows how strictly practitioners think they follow this specification. Based on these self-assessments, adherence to the specification is rather loose. This might be a result of UML’s lack of formal semantics and large degree of freedom in its application. On the other hand, it might be that informal use is “good enough” for many practitioners’ purposes.

Stopping criteria for modeling

We were interested in the criteria that practitioners apply to determine when modeling activities can end. The most prominent criterion was completeness, which 52.5 percent of the practitioners selected. However, no objective criteria exist to verify that a model is complete or that it satisfies some tangible notion of quality. The second-most prominent criterion was passing a review or an inspection, which 33.8 percent of the practitioners selected.

When schedule or a deadline becomes the criterion for switching from modeling to implementation, it’s usually a poor substitute for a systematic review and a negative indicator for product quality. Nevertheless, 32.8 percent of the practitioners mentioned deadline as a stopping criterion—an alarmingly high percentage. Some 17.5 percent indicated the amount of effort spent as their stopping criterion.

In correlating respondent demographics with the question of stopping criteria, we found that the criteria varied with different project sizes. Most striking was the comparison between deadline and completeness. Figure 3 shows that deadline is more often a stopping criterion in smaller projects, whereas all projects larger than 99 person-years reported completeness as their stopping criterion. This is probably because increased complexity in both the product and the project likewise increases the need for a systematic approach.

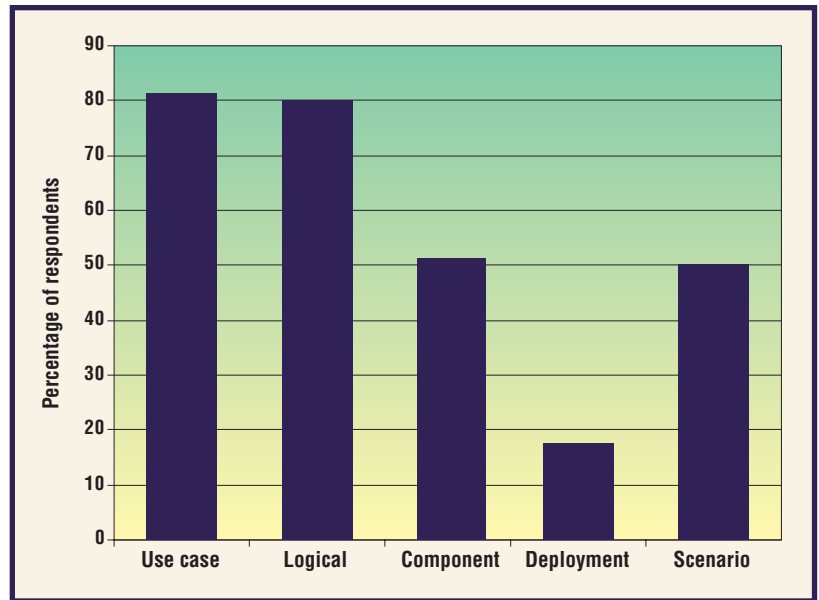


Figure 1. Survey respondents’ use of UML for architectural views.

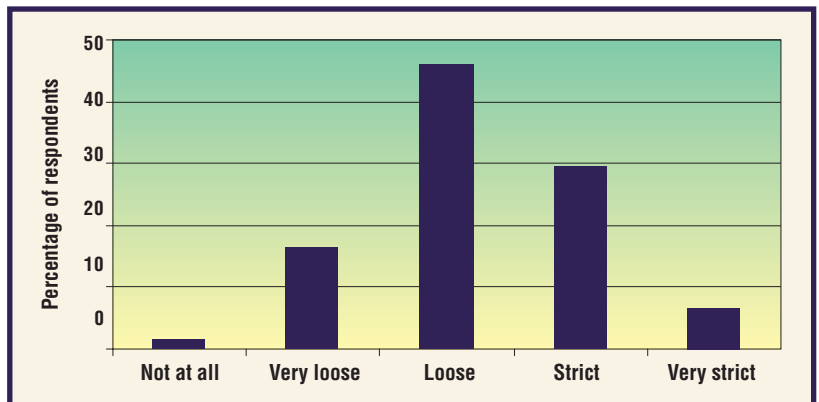


Figure 2. Survey respondents’ assessment of their adherence to the UML standard.

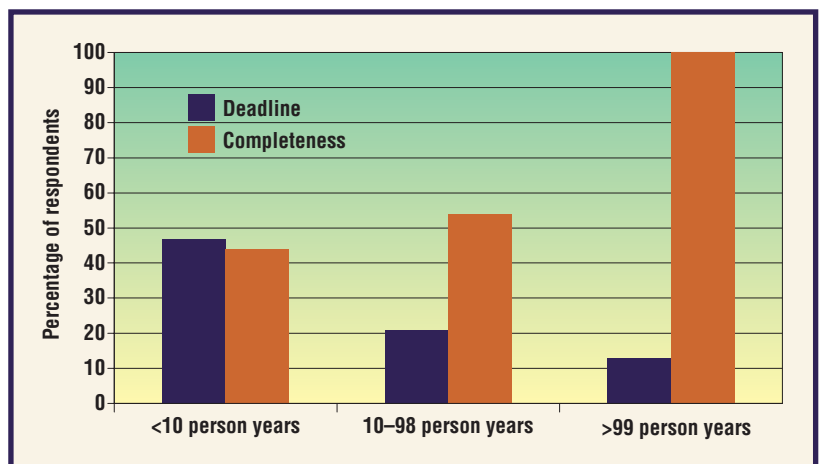


Figure 3. Deadline vs. completeness as stopping criteria for different project sizes.

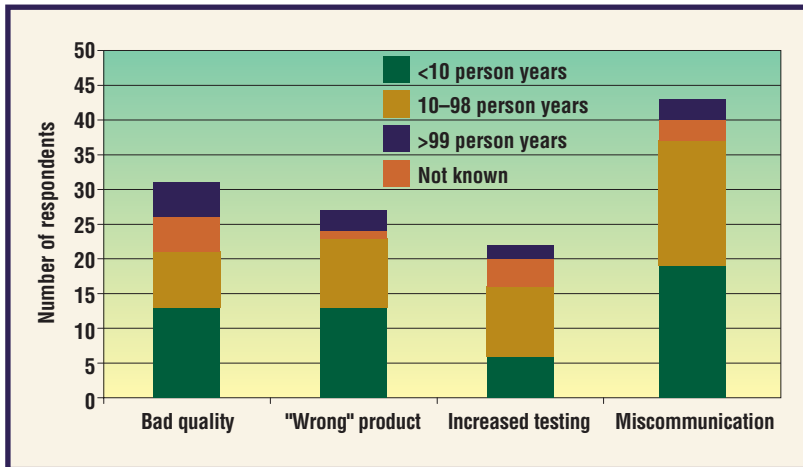


Figure 4. Problems encountered due to incomplete models correlated to respondent demographic data regarding project size.

Problems with incomplete models

Given that the practitioners report completeness to be the primary criterion for deciding to stop modeling, we investigated the effects of moving to the next project phase without a complete model.

Figure 4 shows four problems respondents encountered as a result of incomplete models. More than half reported miscommunication between project members and stakeholders when the project moved forward with incomplete models. They also mentioned poor quality of the implemented product; a “wrong” product delivered, in the sense of not matching the stated requirements; and high amount of testing effort.

Problems with UML descriptions

In both the surveys and additional interviews, architects indicated problems they encountered that are inherent in using a multidocument notation such as UML. On the basis of their responses, we identified four main problem classes:

- *Scattered information.* Design choices are scattered over multiple views. For instance, some dependencies might show up in the logical view, while others appear in the process view.
- *Incompleteness.* Many projects knowingly complete only a subset of the architectural views. The architects focus on what they think is important.
- *Disproportion.* Architects might work out more details for system parts that they consider to be more complex. If increased detail consistently indicated increased criticality or complexity, implementers could use this information in allocating testing resources.

- *Inconsistency.* UML-based software development is inevitably inconsistent.⁴ Industrial systems are typically developed by teams. Different teams can have different understandings of the system as well as different modeling styles, and this can lead to inconsistent models. Although some UML tools provide basic checks on consistency between diagrams, the scope of these checks is limited and leaves much room for inconsistencies to remain between views.

Incompleteness, disproportion, and inconsistency arise much more in software architecture models than in source code. This is because source code includes formal criteria for the form (grammar) and tools for checking these criteria (compilers).

Other problem classes include the following:

- *Diagram quality.* UML lets architects represent one design in different ways. For instance, they can decompose a diagram that contains too many elements into several smaller diagrams. Although different ways of organizing diagrams don’t change the actual design, they can influence how easy the model is to understand and how it gets interpreted.
- *Informal use.* Architects sometimes use UML in a very sketchy manner. For example, to obtain a better understanding of a system or to explain the architecture, they might use generic drawing software or even just make sketches on paper. These diagrams deviate from official UML syntax, making their meaning ambiguous.
- *Lack of modeling conventions.* Our case studies show that engineers use UML according to individual habits. These habits might include layout conventions, commenting, visibility of methods and operations, and consistency between diagrams. In programming, coding standards belong to the standard repertoire of quality assurance techniques.⁵ Similarly, “modeling standards” in the architecting phase would help in establishing more uniform UML usage.

Different uses of UML arise naturally as the design decisions and detail increase in both quantity and complexity throughout the design process. A sketch might be sufficient to capture

Table 1**Case study characteristics**

Case study	No. of classes	No. of person-years spent on modeling	No. of team members	Life stage of the model	Purpose of modeling	CMM level (estimated)
A	734	15	5	Final	Implementation	1
B	168	20	20	Unknown	Unknown	2–3
C	108	20	10	Unknown	Unknown	2–3
D	716	Unknown	Unknown	Unknown	Unknown	1
E	443	10	10	Final	Unknown	1
F	4	1	3	Final	Unknown	1
G	75	10	Unknown	Unknown	Unknown	1
H	478	Unknown	Unknown	Unknown	Unknown	1
I	705	12	6	Development	Implementation	1
J	51	12	6	Development	Analysis	1
K	14	1–5	2	Inception	Analysis	2
L	46	1–5	2	Final	Implementation	2
M	73	0–5	1	Final	Tutorial, abstraction of real world	1
N	359	5	5	Semifinal	Implementation	1–2

initial ideas, whereas significant detail goes into a system architecture that implementers will use to generate source code. Martin Fowler distinguishes three types of UML usage: as a sketch, as a blueprint, and as a programming language.⁶

The generality and freedom that enable UML to cater to this wide range of purposes are also the source of its weakness. UML has no formal semantics. This poses a problem when different people use a UML model; and because one of UML's main purposes is to communicate about a design, different ways of using UML are potential causes of communication problems.

For example, consider an architect who specifies only the most important classes, methods, and attributes for a model—omitting auxiliary classes that he or she thinks are straightforward. Colleagues working in a different office or country must use the architect's model to implement the system. If they assume the model describes every detail, they will misinterpret it.

Despite these problems, the large community of UML users is evidence overall of its usefulness.

Defects in industrial UML models

In addition to the subjective impression obtained via the survey, we performed objective measurements about the quality of industrial UML models. To this end, we implemented

a tool called MetricView (www.win.tue.nl/empanada/metricview) that checked the models for defects, disproportions, and risks of misunderstanding. We applied the tool to several industrial case studies of different sizes from various organizations and application domains (see table 1). The cases use a variety of UML CASE tools.

Defects found in the case studies

The case studies revealed several violations of UML modeling rules. While we don't expect all these rules to hold on any project, we counted model inconsistencies and incomplete spots in our measures of design quality and project progress.

Methods that are not called in sequence diagrams. To depict class interactions, the public methods that a class provides in a class diagram must be called in sequence diagrams. Methods that aren't called might not be used in interactions; alternatively, they might not be sufficiently described in terms of their interactive functionality.

Between 40 and 78 percent of the methods in the models we analyzed were not called in a sequence diagram.

Classes not occurring in sequence diagrams. If a design contains a class that doesn't occur as a

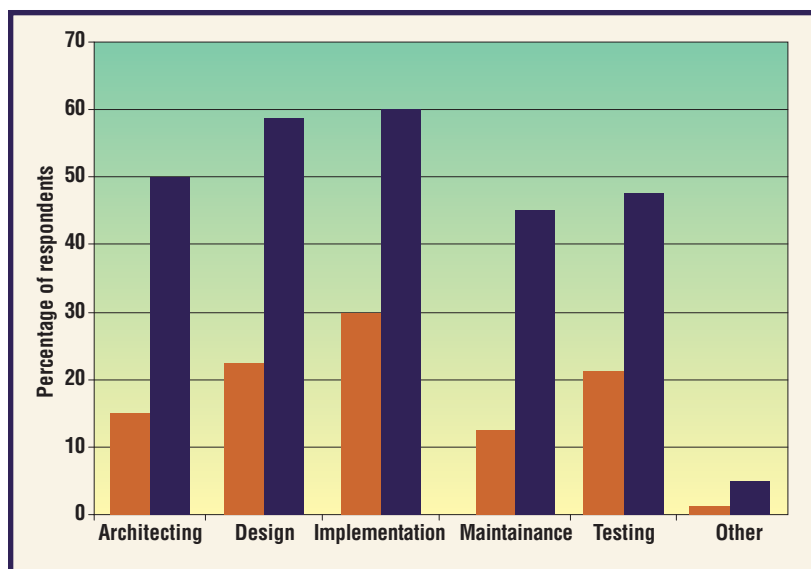


Figure 5. Responses to survey questions regarding the use of UML metrics: current use (brown) and desired use (dark blue).

sequence diagram object, either the class is redundant (assuming the sequence diagrams describe the entire functionality) or the interaction of the design's classes is not completely described using sequence diagrams.

In the models we analyzed, between 35 and 61 percent of the classes didn't occur as objects.

Objects without names. Each object in a sequence diagram must have a name. Named objects are more expressive and understandable than unnamed ones. Naming objects is essential, especially when several objects of the same type occur in one sequence diagram.

In our case studies, we found between 25 and 92 percent of all objects lacked names.

Messages not corresponding to methods. Each message that an object in a sequence diagram receives must correspond to a method in the object's class interface; otherwise the meaning of the message is unclear.

Between 8 and 59 percent of the messages in the analyzed models didn't correspond to methods.

Classes without methods. A class without a method violates the object-oriented paradigm, particularly the concept of encapsulation. A class without methods can't interact with other classes and is therefore incomplete. To make a class complete, the designer must define a class's methods and describe its interactions in a sequence diagram. Only in early modeling phases can you create classes without defining methods.

In all case studies beyond the analysis stage, between 20 and 60 percent of all classes lacked methods.

Effects of UML model defects

In spite of help from CASE tools, defects are common in software development. But what are the effects of UML model defects? Are they eventually detected, when someone uses the model as a specification for implementation? If not, does a defect really cause different readers to interpret the model in different ways?

We conducted a controlled experiment with 110 students and 48 practitioners to answer these questions.⁷ The results showed that defects often remain undetected, even if the model is read attentively for implementing the system. For example, 61 percent of readers didn't detect a sequence diagram message that lacked a corresponding method in the class diagram. The results are worse for an object that lacks a class specification: 82 percent didn't detect this defect.

These low detection rates raise the question of whether defects increase the risk for different interpretations. The experimental results showed varying risks for misinterpretation along a scale of 0 to 1, where 0 represents the widest spread in interpretations and 1 represents total agreement on one interpretation. For example, a use case that isn't described by a sequence diagram has a value of 0.44, representing a high risk for misinterpretation.⁷

Opportunities for improving UML usage

After observing these defects, we saw several opportunities for preventing and removing them.

Defect checking and feedback

Current tools provide only limited support for defect checking. Even though UML's informal nature doesn't support formal checks, the tools should be extended to automatically detect defects.

UML profiles have been used to define specific architectural styles and patterns—for instance, client-server patterns for use in telecommunications systems.⁸ The pattern defines which building blocks the model developer can use and what types of relations can exist between the blocks. The model developer can use these patterns constructively to guide the design. During development, tools can also verify that architecture, design, and implementation conform to this pattern.

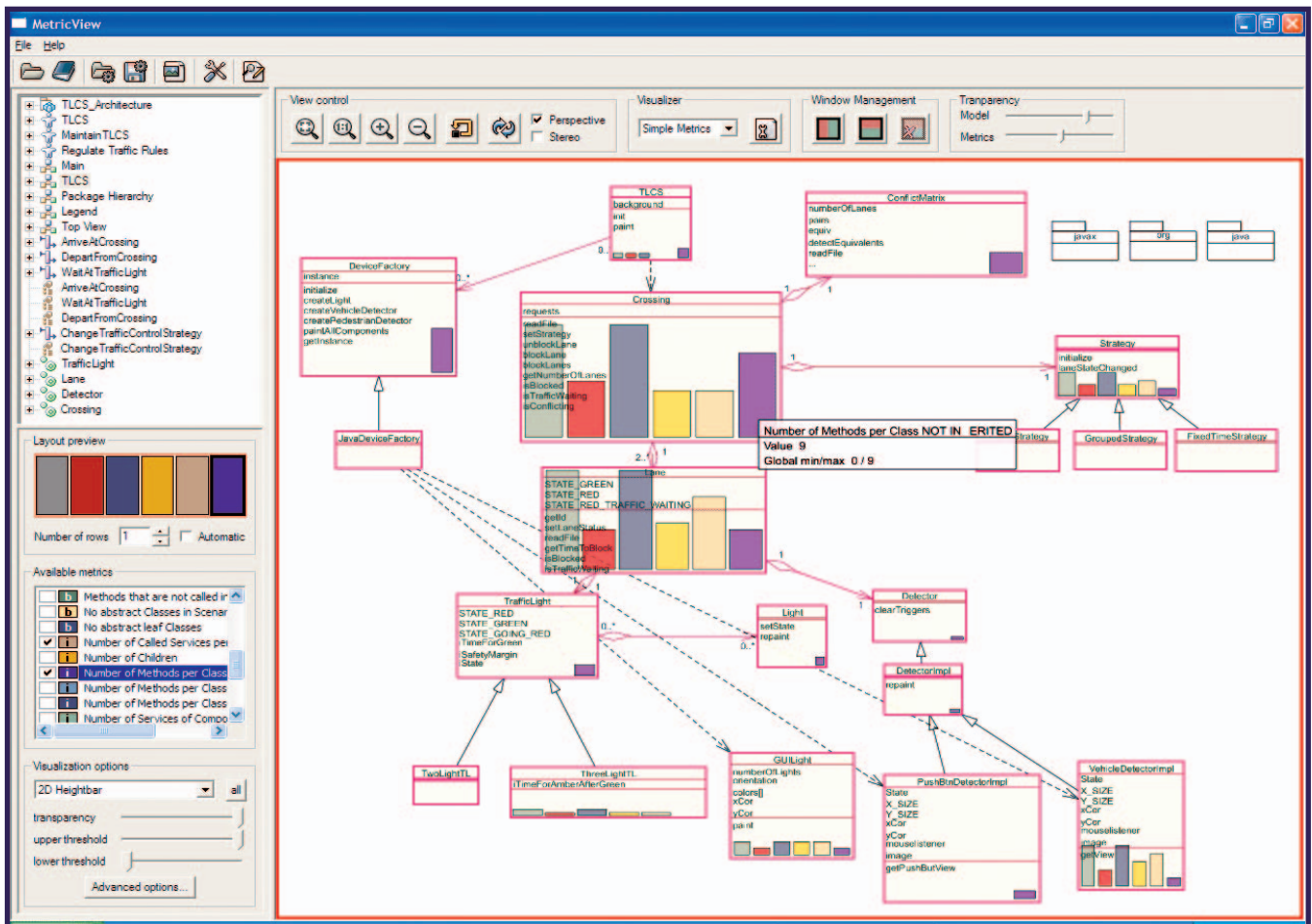


Figure 6. Screenshot of the MetricView tool for visualizing metrics data.

UML metrics

Software metrics are a well-established technique for managing source code quality during implementation.⁹ Metrics can also help manage the quality of architecture and design.

Figure 5 shows the extent of use that survey respondents indicated for metrics in each development phase, along with the extent to which they thought metrics would be useful. The desired use of metrics is two to four times higher than the actual use. These results emphasize the demand for measurement in UML models.

The multiple views of UML models include information not available from source code. This establishes a basis for architecture-level metrics.¹⁰ Projects could use these early metrics to indicate adherence to or violation of design guidelines and heuristics. The following describe some architecture metrics:

- **Class dynamicity.** This metric indicates a class's complexity. If a class has many different incoming and outgoing messages and appears in several different sequence diagrams, then we assume the class plays a critical role in the system and deserves more attention during reviews and testing.

If the dynamicity is above a threshold, the heuristic can recommend using a state machine to model the class's behavior.

- **Number of classes per use case.** This metric indicates whether related functionality is spread over many parts of a design. A higher value indicates low maintainability and reusability. Use cases with no classes indicate that the developers as yet foresee no implementation of the case, which might—in turn—indicate an implementation omission.
- **Number of use cases per class.** A class that is implemented in many use cases might have low cohesion and so serve a large amount of unrelated functionality. It might also be central to the system, which means that errors in it would cause many system features to suffer.

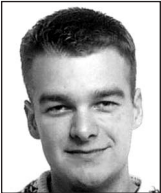
Existing tools usually present metrics and defects in tabular form. This requires designers to relate the tables to the diagrams they're working on. Presenting metrics values and defects graphically at the corresponding place in a UML diagram generates more direct feedback. In figure 6,

About the Authors



Christian F.J. Lange is a PhD student at the Eindhoven University of Technology and a researcher in the System Architecture and Networking group. His research interests include empirical software engineering and UML quality. He received his MSc in computer science from Eindhoven University of Technology. He's a member of the IEEE and the German Computer Society. Contact him at the System Architecture and Networking Group, Eindhoven Univ. of Technology, PO Box 513, 5600 MB Eindhoven, Netherlands; c.f.j.lange@tue.nl.

Michel R.V. Chaudron is an assistant professor at the Eindhoven University of Technology and a researcher in the System Architecture and Networking group. His research interests include software architecture, empirical software engineering, and component-based software engineering. He received his PhD in computer science from the Universiteit Leiden. Contact him at the System Architecture and Networking Group, Eindhoven Univ. of Technology, PO Box 513, 5600 MB Eindhoven, Netherlands; m.r.v.chaudron@tue.nl.




Johan Muskens is a researcher at Philips Research. His research interests include software architecture analysis, UML, component-based software engineering, and integration of third-party software. He obtained his MSc in computer science from the Eindhoven University of Technology. Contact him at Philips Research Laboratories, Office WDC 2.040, Prof. Holstlaan 4, 5656 AA Eindhoven, Netherlands; johan.muskens@philips.com.

our MetricView tool shows metrics and defects visualized on top of UML diagrams.

UML practices should improve with increased capabilities in development tools for it. Several areas need improvement:

- *Detection of design flaws, omissions, and inconsistencies.* Software metrics can play a role in detecting flaws, offering a fast, objective technique to support assessment of UML model properties. They also help identify refactoring opportunities to improve the architecture's structure.¹¹
- *More uniformity in modeling.* Modeling standards and development tools for checking them can achieve this purpose.
- *Domain- or project-specific reference architectures and patterns.* UML profiles support this work. UML 2.0 provides several facilities for defining such profiles but little support for checking them.
- *More consistency between UML models and system requirements as well as implementations.* Better mechanisms for traceability and round-trip engineering will help reduce these problems.

- *Defined quality goals for UML models.* A quality goal is an incentive to improve a model and identifies spots that must be improved to reach the defined goal. Existing UML development tools don't support the definition of measurable quality goals, and this functionality is needed.

The MetricView tool we applied to the case studies reported here makes analysis cost-effective by automating it. In most cases, the architects we talked with saw value in the analysis findings and acknowledged the tool's identification of weak spots in the models. In many cases, our feedback led to model changes to remedy identified problems. For some case studies, we've analyzed consecutive model versions and found improved model quality after rework. 

References

1. L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, Addison-Wesley, 1998.
2. C.F.J. Lange and M.R.V. Chaudron, "UML Practitioner Survey within the EmpAnAda Project," survey questionnaire; www.win.tue.nl/~clange/survey/survey.htm.
3. P. Kruchten, "The 4+1 View Model of Architecture," *IEEE Software*, vol. 12, no. 6, 1995, pp. 45–50.
4. C. Ghezzi and B. Nuseibeh, "Introduction to the Special Section: Managing Inconsistency in Software Development," *IEEE Trans. Software Eng.*, vol. 25, no. 6, 1999, pp. 782–783.
5. H. Sutter and A. Alexandrescu, *C++ Coding Standards*, Addison-Wesley, 2004.
6. M. Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, 3rd ed., Addison-Wesley, 2003.
7. C.F.J. Lange and M.R.V. Chaudron, "Effects of Defects in UML Models—An Experimental Investigation," to be published in Proc. IEEE/ACM Int'l Conf. Software Engineering (ICSE06), IEEE CS Press, May 2006.
8. P. Selonen and J. Xu, "Validating UML Models Against Architectural Profiles," *Proc. 9th European Software Eng. Conf. (ESEC)*, ACM Press, 2003, pp. 58–67.
9. N.E. Fenton and S.L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*, Int'l Thomson Computer Press, 1996.
10. J. Muskens, M.R.V. Chaudron, and C. Lange, "Investigations in Applying Metrics to Multi-View Architecture Models," *Proc. Euromicro*, IEEE CS Press, 2004, pp. 372–379.
11. M. Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/publications/dlib.



Test Management

Panos Louridas

In many projects, testing consumes the single biggest amount of resources of all activities. We tend to collect test cases like stamps without clear strategy—just in case. Many companies suffer with insufficient quality, visibility, and test progress management. Author Panos Louridas introduces test management. I look forward to hearing from both readers and prospective authors about this column and the technologies you want to know more about. —*Christof Ebert*

*To vouch this, is no proof,
Without more wider and more overt test
Than these thin habits and poor
likelihoods
Of modern seeming do prefer against him.*
—*Othello*, Act I, Scene III



MICROSOFT WROTE MORE than 1 million test cases for Office 2007. This figure might seem extreme or only appropriate for leviathan-sized projects, but the fact is that test cases constitute a component of software that developers can ignore only at their peril. Test management deals with how we

- navigate through a proliferation of test cases,
- organize them,
- assign them to testers,
- orchestrate testing,
- collect test results, and
- measure progress.

Several interesting technologies have recently been introduced for managing test cases, which I'll describe in this column. To understand the stakes involved, it's worth seeing how much effort testing requires.

Testing Effort

Test management makes sense, given the high cost of testing. The value of managing test cases derives directly from the amount of effort that testing takes up in the project. Industry figures from Steve McConnell and Christof Ebert show the following:^{1,2}

- System testing requires from 16 to 29 percent of the total effort, depending on the project's size (see Table 1).
- Test effort varies depending on the nature of the project (for example, safety-critical projects need more than a simple gadget), complexity, and, of course, processes and tools (see Table 2). For instance, with insufficient input quality, test effort immediately increases by a factor of two or three.
- Systematic test planning, frequent test coverage reviews, incremental development, daily builds, and smoke tests with automatic regression are among the biggest efficiency levers in industry.

For such effort levels, it behooves project managers and developers to make testing as efficient as possible. This requires measuring tests, to which we turn next.

TABLE 1

Total effort breakdown for projects of different sizes for application software.

Size (in KLOC)	Activity				
	Requirements	Architecture and planning	Construction	System test	Management, overheads
1	4%	10%	61%	16%	9%
25	4%	14%	49%	23%	10%
125	7%	15%	44%	23%	11%
500	8%	15%	35%	29%	13%

TABLE 2

Examples of developer-to-tester ratios.

Environment	Observed developer-to-tester ratios
Common business systems (internal intranet, management information systems, and so on)	3:1 to 20:1 (often no test specialists at all)
Common commercial systems (public Internet, shrink wrap, and so on)	1:1 to 5:1
Scientific and engineering projects	5:1 to 20:1 (often no test specialists at all)
Common systems projects	1:1 to 5:1
Safety-critical systems	5:1 to 1:2
Microsoft Windows 2000	1:2
NASA space shuttle flight control software	1:10

Test Measurement

Test cases, in addition to helping increase a product's quality, are valuable in improving the development process when used to measure a series of process metrics that incorporate test results.

Defect removal efficiency (DRE), or test effectiveness, is the ratio of the number of defects found during testing to the total number of defects found. For instance, if 90 bugs are found by tests and 10 bugs are discovered by other means (later in the product life cycle, by users, and so on), the test effectiveness is $90 / (10 + 90) = 0.90$.

Test efficiency is a metric that the literature has confused with test effectiveness—perhaps the semantic proximity of efficient and effective are to blame. *Fowler's Modern English Usage* defines *effective* as “having a definite or desired effect,” and *efficient* as “productive with minimum waste or effort.” Efficiency, therefore, is a measure like power in physics. Even with this clarification, *test effectiveness* encompasses different definitions in the literature. In the broadest sense, it's the number of defects found divided by the effort expended to find them. This can be counted, for instance, by the number of defects found divided by the number of test cases. A different definition is the percentage of tests that

find bugs—it can also show which tests are more effective in finding bugs.

Pass rate is the ratio of tests that pass to the total number of tests.

Passes versus failures is the ratio of tests that pass to the number of tests that fail. This can be useful to show failures in projects with a large number of tests. For instance, for 1 million tests, a 95 percent pass rate is equivalent to 50,000 failed tests—that is, a passes versus failures rate of 19.

A *test progress S curve* is a graph plotting time units—for instance, weeks—on the *x*-axis and the number of tests to be completed successfully, attempted, and passed at each time unit on the *y*-axis.

The graph is called an S curve because it follows this shape: the test numbers that are plotted are cumulative, and the number of planned tests starts low, then increases, and finally levels off as it gets closer to release time (see Figure 1). The test progress graph is used to guide the development effort, highlighting delays in testing effort (when the number of tests attempted is less than the planned number) or in project quality (when the number of passed tests lags significantly behind the planned number).

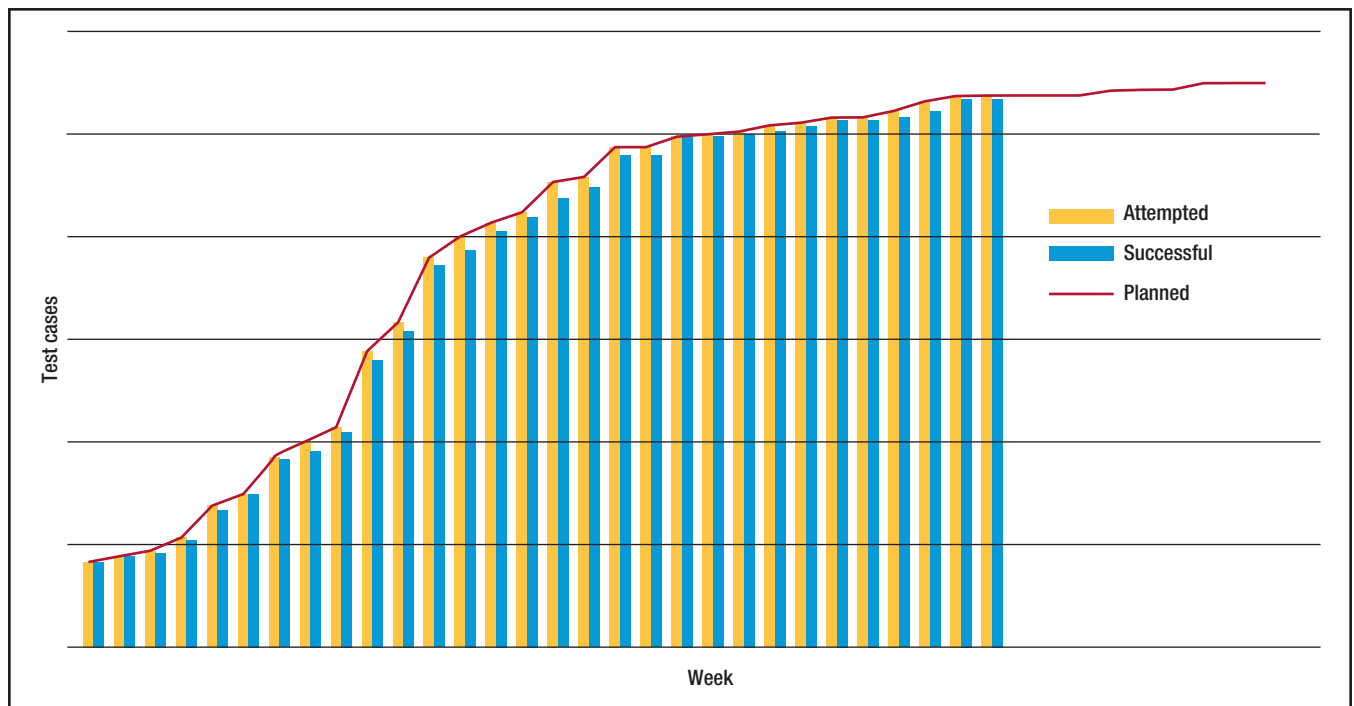


FIGURE 1. Sample test process S curve.³ Cumulative graph of attempted, successful, and planned test cases over time.

Testing defect arrivals over time is also a graph, plotting time units on the *x*-axis and defect arrivals, discovered by testing, on the *y*-axis. The graph should normally be a hump-like shape: in the project's beginning, few defects are reported, as there are few tests and few opportunities for defects anyway; as the project progresses, both tests and discovered defects pile up; but as we get closer to shipping, fewer defects should show up.

Testing defect backlog over time is again a graph, plotting the time unit before the shipping date on the *x*-axis and defects discovered by testing and not yet resolved, therefore remaining in the backlog, on the *y*-axis. The metric can be used to guide process planning by agreeing in specific targets for defect backlog before shipping.

Test coverage measures how extensively tests cover the code to be tested. It can be measured in different ways—for example, with the ratio of the number of test cases executed divided by the number of test cases required to test all functions in the code, or by the number of branches tested divided by the total number of branches in the program. Test coverage is useful in itself to check that there are no blind spots left in the program, but it's also useful in deriving other metrics that relate to project process progress.

Test confidence is a process metric that uses test coverage. It is defined by the following formula:

$$\text{TestConfidence} = \left\{ 1 - \frac{\text{ErrorsFoundInLastTest}}{\text{TestCasesRun}} \right\} \times \text{TestCoverage}.$$

So, if we ran 400 test cases in total and discovered three errors in the last run, while the test coverage was calculated to be 95 percent, the test confidence would be

$$\left\{ 1 - \frac{3}{400} \right\} \times 0.95 = 0.94.$$

Test coverage has been used to define test efficiency in terms of tester effort:

$$\text{TestEfficiency} = \left\{ 1 - \frac{\text{TesterDays}}{\text{Errors} + \text{TestCases}} \right\} \times \text{TestCoverage}.$$

Continuing with the previous example, if we had 400 test cases in total, executed in 180 person days, and the test cases found 260 errors, while the final test coverage was 0.95, the test efficiency would be

$$1 - \left\{ \frac{180}{260 + 400} \right\} \times 0.95 = 0.69.$$

Test Case Management Tools

For a test case to be useful, it should include

- the goal, describing briefly what it's purported to test;

TABLE 3

Test case management tools.

	Seapine TestTrack	QMetry	TestRail	XStudio
Interoperability with issue-tracking systems	Yes (with Seapine TestTrack Pro)	Yes, with Mantis, Bugzilla, and JIRA Enterprise Edition	Yes, through URLs	Uses its own bug-tracking database, connectors with JIRA, Trac, Bugzilla, and Manti
Interoperability with authentication/authorization systems	Yes, it can use Single Sign-on	No	Yes, it can use Single Sign-on	No
Integration with requirements	Yes, via TestTrack RM	Handles requirements internally	Yes, through URLs	Handles requirements internally
Integration with source control tools	Seapine Surround SCM, CVS, Clearcase, PVCS, Perforce, SourceOff-SiteClassic, StarTeam, Subversion, and Visual SourceSafe	No	No	No
Integration with developer tools	VisualStudio and Eclipse	No	No	No
Interface	Client program, Web interface (for subset of functionality), and SOAP SDK	Web interface	Web interface and API for submitting test changes and test results	Client program, Web interface, and SDK for integrating with existing tests
Platforms	MS Windows, Mac OS, and Linux	Available as hosted software as a service; on-premise installation MS Windows and Linux with PHP and MySQL (for instance, XAMPP)	MS Windows server 2003 or 2008, IIS with FastCGI/PHP integration; Unix-based OS with Apache, MySQL, and PHP	Java Runtime Environment 1.6 for the server, MS Windows, Linux, Mac OS fat clients, and Web client requiring Web server (Apache, Tomcat, IIS)
Databases	Internal, MS SQL, Oracle, PostgreSQL, and MySQL	MySQL	MS-SQL and MySQL	MySQL
License	Proprietary	Proprietary	Proprietary	Free; parts are open source

- the rationale, explaining why it's important;
- the preconditions, listing the constraints on the environment that must be in place for it to run;
- the inputs;
- the steps, in numbered sequence unless it's implemented in code;
- the expected results;
- instructions on how often and when it should run; and

- configurations under which it should run.

The simplest way to manage test cases is with a spreadsheet containing the following information:

- The name of the test suite, and for each test suite, the name of the test.
- The state of each test case, with values pass, fail, or no value if the test

is queued for execution.

- An identifier for the system configuration used for the test case. The identifier is used for looking up the configuration details in a separate spreadsheet.
- The identifier of the bug discovered by the test. If the test discovered more than one bug, they can be listed with commas in a single cell or in different rows (with the other

FURTHER READING

A good overall discussion on testing in big settings is *How We Test Software at Microsoft* (A. Page, K. Johnston, and B.J. Rollinson, Microsoft Press, 2008).

The figures and tables in this column on testing effort are from *Software Estimation: Demystifying the Black Art* (S. McConnell, Microsoft Press, 2006).

A good discussion on test metrics and how to use them to guide the software process is *Metrics and Models in Software Quality Engineering* (S.H. Kan, 2nd ed., Addison-Wesley, 2003). Much of the material for test metrics used in this column comes from this book.

Although targeting agile practices, *Agile Testing: A Practical Guide for Testers and Agile Teams* (L. Crispin and J. Gregory, Pearson, 2009) is of wider relevance and offers valuable practical advice. The book presents an alternative categorization of tests in quadrants than the one used in this column.

Chapters 8 and 9 of *Software Engineering Theory and Practice* (S.L. Pfleeger and J.M. Atlee, 4th ed., Prentice Hall, 2009) provide an introduction to testing and test management.

"Validating and Improving Test-Case Effectiveness" (Y. Chernak, *IEEE Software*, vol. 18, no. 1, 2001, pp. 81–86) presents ways to improve testing efficiency (although the author prefers the term effectiveness).

A detailed discussion on metrics, including test metrics, is *Software Measurement: Establish—Extract—Evaluate—Execute* (C. Ebert and R. Dumke, Springer, 2007), from which I drew the material in this column, the formulas for test coverage, test efficiency, and test confidence, and data for test costs.

IEEE Standard 829-2008: Software and System Test Documentation (a revision of the earlier *IEEE Standard 829-1998*) treats the use and contents of test documentation, test tasks, required inputs and outputs, master test plans, and level test plans, and should be of interest to companies and organizations that are serious about testing.

For more details on how to use a spreadsheet for managing test cases, as well as other practical information, see *Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing* (R. Black, Wiley, 2002).

Effective Software Testing: 50 Specific Ways to Improve Your Testing (E. Dustin, Addison-Wesley, 2002) presents specific items of advice on testing.

A very good presentation of what makes a good test case (which influenced this column) is "What Is a Good Test Case?" (C. Kaner, 2003, available at <http://www.kaner.com/pdfs/GoodTest.pdf>).

fields copied from the row above).

- The tester's initials.

Although this might sound simplistic, you might take into account how many tests are conducted without the use of a spreadsheet. It's easy to think of ways to improve the spreadsheet and make it part of the development pro-

cess. There's no reason why it should remain a personal spreadsheet—put it up on a team environment like Google Docs or a team wiki.

A host of more sophisticated approaches to managing tests exists, as well as a wealth of tools, both open and closed source commercial ones. (Tellingly, at one point there were at

least a half-dozen test case management systems in MS Windows at Microsoft, but then teams started migrating to the test case management tools in the Visual Studio Team System). Things to consider when choosing which to adopt include the following:

- *Interoperability with bug- or issue-tracking systems.*
- *Interoperability with requirement-tracking systems.* Ideally, a test case management tool should offer a three-way linkage between functional requirements, test cases, and bugs. In this way, we can know which test cases correspond to a requirement, identify requirements that don't have test cases, or map bugs to specific requirements.
- *Information kept for each test case.* This information should include the material we listed in the beginning of this section and can also include information such as the date the test was executed, the test author, links to related test cases, and trace logs.
- *Degree of automation.* You could simply use the tool for housekeeping test runs and results (like the spreadsheet solution or a simple wiki-based one) or guide the entire test process by invoking test plans.
- *Reporting capabilities and metrics calculation.* The metrics can include the ones presented earlier, and the results should be able to be presented in reports. Even better, the metrics results and the underlying data can be available through an API.
- *Interoperability with authentication systems.* Ideally, each project participant should have a single identity and should use that identity for everything, be it committing code, announcing bugs, running tests, or even writing in project-related mailing lists.
- *Installation requirements.* Usu-

ally these tools are deployed on a server so that they're available to the development team over the Web. They might be built on popular Web development frameworks (like Ruby on Rails), but in general it's worth checking how well they integrate with your own ecosystem of tools and infrastructure.

- *Cost and license.*

Test case management tools typically start with a way to define test cases, for instance, with a series of templates where the user fills in the fields that constitute the test case (see Figure 2). It might also prove possible to establish relations between test cases, to place conditions on their execution (for example, test case A should be executed if test case B succeeds), and to automatically generate variants of test cases depending on variables (such as system configuration). Test cases are then assigned to users. Overall progress and an execution overview is provided through some means of a dashboard that presents the test results both in graphics and numbers. Based on these numbers, we can compute test metrics; tools also prepare reports that can be useful for management purposes.

The most important attribute of a tool isn't its technical capability but how effectively it's used and how well it's adopted and used throughout the development process (see the sidebar "Further Reading"). Promoting a culture of defect prevention and quality assurance is more important than any specific tool that Othello would deem "modern seeming."

References

1. S. McConnell, *Software Estimation: Demystifying the Black Art*, Microsoft Press, 2006.
2. C. Ebert and R. Dumke, *Software Measure-*

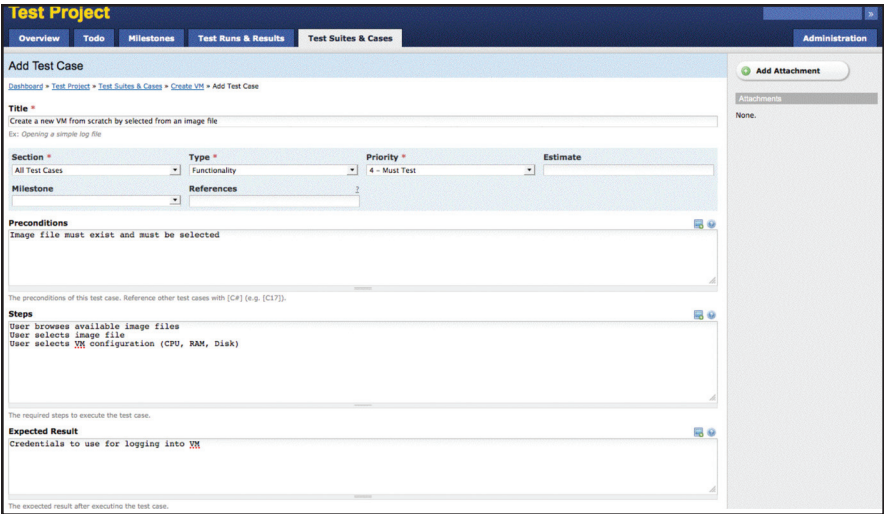


FIGURE 2. Adding a test case in TestRail. The user fills in test case information like type, priority, preconditions, and steps.

ment: *Establish – Extract – Evaluate – Execute*, Springer, 2007.
3. S.H. Kan, *Metrics and Models in Software Quality Engineering*, 2nd ed., Addison-Wesley, 2003.

PANOS LOURIDAS is a consultant with the Greek Research and Technology Network and a researcher at the Athens University of Economics and Business. Contact him at louridas@grnet.gr or louridas@aueb.gr.

ADVERTISER INFORMATION • SEPTEMBER/OCTOBER 2011

ADVERTISERS	PAGE
ABB Corporate Research	19
John Wiley & Sons, Inc.	Cover 2
Seapine Software, Inc.	Cover 4

Advertising Personnel

Marian Anderson: Sr. Advertising Coordinator
Email: manderson@computer.org; Phone: +1 714 816 2139 | Fax: +1 714 821 4010
Sandy Brown: Sr. Business Development Mgr.
Email: sbrown@computer.org; Phone: +1 714 816 2144 | Fax: +1 714 821 4010

IEEE Computer Society, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720 USA
www.computer.org

Advertising Sales Representatives

Western US/Pacific/Far East: Eric Kincaid
Email: e.kincaid@computer.org; Phone: +1 214 673 3742; Fax: +1 888 886 8599

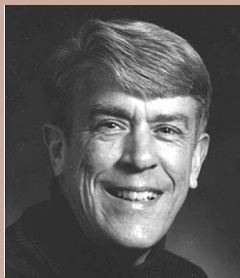
Eastern US/Europe/Middle East: Ann & David Schissler
Email: a.schissler@computer.org, d.schissler@computer.org
Phone: +1 508 394 4026; Fax: +1 508 394 4926

Advertising Sales Representatives (Classified Line/Jobs Board)

Greg Barbash
Email: g.barbash@computer.org; Phone: +1 914 944 0940

Frequently Forgotten Fundamental Facts about Software Engineering

Robert L. Glass



This month's column is simply a collection of what I consider to be facts—truths, if you will—about software engineering. I'm presenting this software engineering laundry list because far too many people who call themselves software engineers, or computer scientists, or programmers, or whatever nom du jour you prefer, either aren't familiar with these facts or have forgotten them.

I don't expect you to agree with all these facts; some of them might even upset you. Great! Then we can begin a dialog about which facts really are facts and which are merely figments of my vivid loyal opposition imagination!

Enough preliminaries. Here are the most frequently forgotten fundamental facts about software engineering. Some are of vital importance—we forget them at considerable risk.

Complexity

C1. For every 10-percent increase in problem complexity, there is a 100-percent increase in the software solution's complexity. That's not a condition to try to change (even though reducing complexity is always desirable); that's just the way it is. (For one explanation of why this is so, see RD2 in the section "Requirements and design.")

People

P1. The most important factor in attacking complexity is not the tools and techniques that programmers use but rather the quality of the programmers themselves.

P2. Good programmers are up to 30 times better than mediocre programmers, according to "individual differences" research. Given that their pay is never commensurate, they are the biggest bargains in the software field.

Tools and techniques

T1. Most software tool and technique improvements account for about a 5- to 30-percent increase in productivity and quality. But at one time or another, most of these improvements have been claimed by someone to have "order of magnitude" (factor of 10) benefits. Hype is the plague on the house of software.

T2. Learning a new tool or technique actually lowers programmer productivity and product quality initially. You achieve the eventual benefit only after overcoming this learning curve.

T3. Therefore, adopting new tools and techniques is worthwhile, but only if you (a) realistically view their value and (b) use patience in measuring their benefits.

Quality

Q1. Quality is a collection of attributes. Various people define those attributes differ-

Continued on p. 110

Continued from p. 112

ently, but a commonly accepted collection is portability, reliability, efficiency, human engineering, testability, understandability, and modifiability.

Q2. Quality is not the same as satisfying users, meeting requirements, or meeting cost and schedule targets. However, all these things have an interesting relationship: User satisfaction = quality product + meets requirements + delivered when needed + appropriate cost.

Q3. Because quality is not simply reliability, it is about much more than software defects.

Q4. Trying to improve one quality attribute often degrades another. For example, attempts to improve efficiency often degrade modifiability.

Reliability

RE1. Error detection and removal accounts for roughly 40 percent of development costs. Thus it is the most important phase of the development life cycle.

RE2. There are certain kinds of software errors that most programmers make frequently. These include off-by-one indexing, definition or reference inconsistency, and omitting deep design details. That is why, for example, N-version programming, which attempts to create multiple diverse solutions through multiple programmers, can never completely achieve its promise.

RE3. Software that a typical programmer believes to be thoroughly tested has often had only about 55 to 60 percent of its logic paths executed. Automated support, such as coverage analyzers, can raise that to roughly 85 to 90 percent. Testing at the 100-percent level is nearly impossible.

RE4. Even if 100-percent test coverage (see RE3) were possible, that criteria would be insufficient for testing. Roughly 35 percent of software defects emerge from missing logic paths, and another 40 percent are from the execution of a unique combination of logic paths. They will not be caught by 100-percent coverage (100-percent coverage can, therefore,

potentially detect only about 25 percent of the errors!).

RE5. There is no single best approach to software error removal. A combination of several approaches, such as inspections and several kinds of testing and fault tolerance, is necessary.

RE6. (corollary to RE5) Software will always contain residual defects, after even the most rigorous error removal. The goal is to minimize the number and especially the severity of those defects.

Efficiency

EF1. Efficiency is more often a matter of good design than of good coding. So, if a project requires efficiency, efficiency must be considered early in the life cycle.

EF2. High-order language (HOL) code, with appropriate compiler optimizations, can be made about 90 percent as efficient as the comparable assembler code. But that statement is highly task dependent; some tasks are much harder than others to code efficiently in HOL.

EF3. There are trade-offs between size and time optimization. Often, improving one degrades the other.

Maintenance

M1. Quality and maintenance have an interesting relationship (see Q3 and Q4).

M2. Maintenance typically consumes about 40 to 80 percent (60 percent average) of software costs. Therefore, it is probably the most important life cycle phase.

M3. Enhancement is responsible

for roughly 60 percent of software maintenance costs. Error correction is roughly 17 percent. So, software maintenance is largely about adding new capability to old software, not about fixing it.

M4. The previous two facts constitute what you could call the “60/60” rule of software.

M5. Most software development tasks and software maintenance tasks are the same—except for the additional maintenance task of “understanding the existing product.” This task is the dominant maintenance activity, consuming roughly 30 percent of maintenance time. So, you could claim that maintenance is more difficult than development.

Requirements and design

RD1. One of the two most common causes of runaway projects is unstable requirements. (For the other, see ES1.)

RD2. When a project moves from requirements to design, the solution process’s complexity causes an explosion of “derived requirements.” The list of requirements for the design phase is often 50 times longer than the list of original requirements.

RD3. This requirements explosion is partly why it is difficult to implement requirements traceability (tracing the original requirements through the artifacts of the succeeding life-cycle phases), even though everyone agrees this is desirable.

RD4. A software problem seldom has one best design solution. (Bill Curtis has said that in a room full of expert software designers, if any two agree, that’s a majority!) That’s why, for example, trying to provide reusable design solutions has so long resisted significant progress.

Reviews and inspections

RI1. Rigorous reviews commonly remove up to 90 percent of errors from a software product before the first test case is run. (Many research findings support this; of course, it’s extremely difficult to know when you’ve found 100 percent of a software product’s errors!)

Trying to improve one quality attribute often degrades another. For example, attempts to improve efficiency often degrade modifiability.

RI2. Rigorous reviews are more effective, and more cost effective, than any other error-removal strategy, including testing. But they cannot and should not replace testing (see RE5).

RI3. Rigorous reviews are extremely challenging to do well, and most organizations do not do them, at least not for 100 percent of their software artifacts.

RI4. Post-delivery reviews are generally acknowledged to be important, both for determining customer satisfaction and for process improvement, but most organizations do not perform them. By the time such reviews should be held (three to 12 months after delivery), potential review participants have generally scattered to other projects.

Reuse

REU1. Reuse-in-the-small (libraries of subroutines) began nearly 50 years ago and is a well-solved problem.

REU2. Reuse-in-the-large (components) remains largely unsolved, even though everyone agrees it is important and desirable.

REU3. Disagreement exists about why reuse-in-the-large is unsolved, although most agree that it is a management, not technology, problem (will, not skill). (Others say that finding sufficiently common subproblems across programming tasks is difficult. This would make reuse-in-the-large a problem inherent in the nature of software and the problems it solves, and thus relatively unsolvable).

REU4. Reuse-in-the-large works best in families of related systems, and thus is domain dependent. This narrows its potential applicability.

REU5. Pattern reuse is one solution to the problems inherent in code reuse.

Estimation

ES1. One of the two most common causes of runaway projects is optimistic estimation. (For the other, see RD1.)

ES2. Most software estimates are performed at the beginning of the life cycle. This makes sense until we realize

that this occurs before the requirements phase and thus before the problem is understood. Estimation therefore usually occurs at the wrong time.

ES3. Most software estimates are made, according to several researchers, by either upper management or marketing, not by the people who will build the software or by their managers. Therefore, the wrong people are doing estimation.

ES4. Software estimates are rarely adjusted as the project proceeds. So, those estimates done at the wrong time by the wrong people are usually not corrected.

ES5. Because estimates are so faulty, there is little reason to be concerned when software projects do

had ever worked on! This illustrates the disconnect regarding the role of estimation, and project success, between management and technologists. Given the previous facts, that is hardly surprising.

ES7. Pressure to achieve estimation targets is common and tends to cause programmers to skip good software process. This constitutes an absurd result done for an absurd reason.

Research

RES1. Many software researchers advocate rather than investigate. As a result, (a) some advocated concepts are worth less than their advocates believe and (b) there is a shortage of evaluative research to help determine the actual value of new tools and techniques.

Pressure to achieve estimation targets is common and tends to cause programmers to skip good software process. This constitutes an absurd result done for an absurd reason.

not meet cost or schedule targets. But everyone is concerned anyway!

ES6. In one study of a project that failed to meet its estimates, the management saw the project as a failure, but the technical participants saw it as the most successful project they

There, that's my two cents' worth of software engineering fundamental facts. What are yours? I expect, if we can get a dialog going here, that there are a lot of similar facts that I have forgotten—or am not aware of. I'm especially eager to hear what additional facts you can contribute.

And, of course, I realize that some will disagree (perhaps even violently!) with some of the facts I've presented. I want to hear about that as well. ☺

Robert L. Glass is the editor of Elsevier's *Journal of Systems and Software* and the publisher and editor of *The Software Practitioner* newsletter. Contact him at rglass@indiana.edu; he'd be pleased to hear from you.

Copyright and reprint permission: Copyright © 2001 by the Institute of Electrical and Electronics Engineers, Inc. All rights reserved. Abstracting is permitted with credit to the source. Libraries are permitted to photocopy beyond the limits of US copyright law for private use of patrons those post-1977 articles that carry a code at the bottom of the first page, provided the per-copy fee indicated in the code is paid through the Copyright Clearance Center, 222 Rosewood Dr., Danvers, MA 01923. For copying, reprint, or republication permission, write to Copyright and Permissions Dept., IEEE Publications Admin., 445 Hoes Ln., Piscataway, NJ 08855-1331.

Circulation: *IEEE Software* (ISSN 0740-7459) is published bimonthly by the IEEE Computer Society. IEEE headquarters: Three Park Ave., 17th Floor, New York, NY 10016-5997. IEEE Computer Society Publications Office: 10662 Los Vaqueros Cir., PO Box 3014, Los Alamitos, CA 90720-1314; (714) 821-8380; fax (714) 821-4010. IEEE Computer Society headquarters: 1730 Massachusetts Ave. NW, Washington, DC 20036-1903. Subscription rates: IEEE Computer Society members get the lowest rates and choice of media option—\$40/32/52 US print/electronic/combination; go to <http://computer.org/subscribe> to order and for more information on other subscription prices. Back issues: \$10 for members, \$20 for nonmembers. This magazine is available on microfiche.

Postmaster: Send undelivered copies and address changes to Circulation Dept., *IEEE Software*, PO Box 3014, Los Alamitos, CA 90720-1314. Periodicals Postage Paid at New York, NY, and at additional mailing offices. Canadian GST #125634188. Canada Post Publications Mail Product (Canadian Distribution) Sales Agreement Number 0487805. Printed in the USA.