Cover Feature

Agents of Change: Educating Software Engineering Leaders

Most professional degree programs for software engineering focus on solving today's problems with today's technologies. Carnegie Mellon's master of software engineering program takes a different approach, preparing engineers to work with new science and technology throughout their careers and helping them become agents of change in the industry.

David Garlan

David P. Gluch

James E. Tomayko

Carnegie Mellon University

In today's world of rapidly changing software technology, engineers must deal with new methods, tools, platforms, user expectations, and software markets. This changing environment underscores the need for software engineering education that not only teaches current technologies, but also trains engineers to adapt quickly to new technologies and trends, allowing them to anticipate and exploit emerging opportunities to improve their products and processes.

The typical engineering education keeps engineers abreast of technological advances by offering courses in new techniques and tools. This is fine while the engineers are in school, but it doesn't necessarily prepare them for future challenges. A better approach would be to provide the engineers with models, skills, and analytical techniques that they can use throughout their career to evaluate emerging technologies and then successfully adapt appropriate ones to their organization's needs. These engineers, then, would act as agents of change.

Over the past nine years, Carnegie Mellon University has been developing and refining such a curriculum in its master of software engineering (MSE) program. In the process, the university has had to determine how the curriculum would differ from traditional ones, what kinds of hands-on experience would be offered, and how much specialization to require. The result is a novel approach that aims to cultivate future leaders in software engineering. It combines a long-term, mentored software development project with an unusual core curriculum that stresses broad-based models and problem-solving skills.

IEEE Software

This article comes to us courtesy of Nancy Mead, Dave Carter, and Michael Lutz, guest editors of November/December issue of *IEEE Software,* which focuses on the changing landscape of software engineering education. As the six articles in that issue show, change is manifesting in an increased focus on undergraduate coursework, training issues, and industry–university collaboration. This article discusses a program specifically oriented toward educating "agents of change." For more information on the special issue, write to software@computer.org.

KEY ISSUES

Three critical issues must be addressed in the design of any professional software engineering program:

1. *Core curriculum:* Programs typically have a set of required courses that cover the basic knowledge and skills that students need. Traditional programs usually follow the software life-cycle model, with courses in requirements specification, specific design methods (such as object-oriented design), testing, and verification. Courses are usually independent of each other and can be taken in almost any order.
2. *Practical system development:* Most programs give students hands-on experience developing a software system, allowing them to apply material learned in class. The usual approach is to include a semester-long project course in the core curriculum.
3. *Degree of specialization:* Programs offer varying opportunities for students to develop a specialization in specific areas of software engineering. Most provide a selection of elective courses, but offer little guidance on how to package these courses into a coherent advanced program. Typically these courses are already offered in the department.

The traditional approach to these issues has its advantages:

1. It allows easy entry by practicing software engineers seeking part-time supplementary education. Courses can usually be taken in any order, and the curriculum can be spread over long periods of time.
2. The courses mirror established software development stages, so it is relatively easy to identify the

Students graduating from traditional programs rarely have the skills to evaluate new technologies or understand how they can be best applied to existing practices.

course in the curriculum that addresses a particular problem. For example, an engineer wanting to improve his or her testing skills will typically find a specific course on that topic.
3. Since a single course covers the project component, no special arrangements need to be made to satisfy practical curriculum requirements.
4. A broad, unstructured smorgasbord of electives makes it possible for students to shop around for advanced courses in areas that they would like to sample.

However, for the purpose of cultivating future leaders in software engineering, the traditional approach has a number of serious drawbacks:

1. Most importantly, it fails to give students broad-based problem-solving skills that go beyond immediate use of current technologies. For instance, a testing course will not teach students how to make broad-based trade-offs when applying fixed resources to the improvement of software quality. Those trade-offs require understanding the relative merits of analysis methods through the life cycle, from verification to testing to prototyping to abstract modeling. Since those methods involve other parts of the life cycle, they will usually not be covered in a testing course.
2. Traditional software engineering programs rarely give students opportunities—using realistic software development projects—to develop skills as agents of change. Being of short duration, project courses only permit students to apply a single development method or explore a narrow set of approaches. They do not allow enough time for students to experiment with different techniques or to follow a customer's use (if there ever is any) of the developed system.
3. A lack of integration among courses means that each course is isolated and its material tends to be compartmentalized. For example, a student who has taken a specification course may not understand how models developed during requirements analysis can be used during testing. Students rarely learn to appreciate the fundamental unifying themes of software development, such as management of complexity and resource allocation.

The consequence of these problems is that students graduating from traditional programs rarely have the skills to evaluate new technologies or understand how they can be best applied to existing software development practices. For instance, if a new software development method suddenly becomes popular, how can one tell what its impact will be? Beyond the inevitable hype, what are its true advantages? What needs to be given up in order to use it?

CARNEGIE MELLON'S MSE PROGRAM

Carnegie Mellon's MSE program was founded in 1989 with the express intention of developing technical leaders in software engineering practice. Graduating students should be able to act as agents of change in their respective organizations, applying both the best of current practice and emerging technologies to software development.

The MSE program is a joint effort of the School of Computer Science and the Software Engineering Institute, providing an intensive one-year, three-semester curriculum for professional software engineers. Students entering the program must have a strong background in computer science and two years of experience in software development that indicates strong potential for leadership.

The incoming class size is usually about 20 students, who have on average five years of industrial experience. About half come from large corporations (including Digital Equipment, Hewlett-Packard, Westinghouse, and General Motors), and the remainder come from a variety of smaller firms. Many of the employers pay for students to attend the program, expecting the students to return to work after graduation.

The MSE program has three basic components[1]:

1. *Core curriculum:* These courses develop foundational skills in the fundamentals of software engineering, emphasizing design, analysis, and the management of large-scale software systems.
2. *Software development studio:* Over the full duration of the program, students plan and implement a significant software project for an external client. As in design projects for architecture programs, students work as a team under the guidance of faculty advisors, to analyze a problem, plan the software development effort, execute the solution, and evaluate their work.
3. *Specialty tracks:* The elective tracks allow students to develop deeper expertise in one of several specialties, including real-time systems, human-computer interfaces, and software process improvement.

Core curriculum

The MSE core curriculum consists of the following five semester-long courses:

1. Models of Software Systems
2. Methods of Software Development
3. Management of Software Development
4. Analysis of Software Artifacts
5. Architectures of Software Systems

The management, models, and methods courses are offered in the fall semester. The analysis and architecture courses are offered in the spring semester. Table 1

shows how the curriculum is arranged.

While most existing programs are organized around the software life cycle, the MSE core curriculum is organized around topics that cut across the development process. It emphasizes the underlying principles and techniques—such as formal models and good management principles—that can be applied uniformly to a broad spectrum of software development activities.

Models of software systems. The models course addresses the foundations of software engineering using precise, abstract models and logics to characterize and reason about the properties of software systems. The course's main topics include state machines, algebraic models, process algebras, trace models, compositional mechanisms, abstraction relations, and temporal logic. Examples are drawn from software applications.

Unlike traditional courses in formal methods, the MSE models course focuses less on specific notations and more on the models on which they are based. Also unlike those courses, the MSE course considers not only mathematical models for software specification, but also certain models that underlie testing, analysis, process modeling, and design selection. Thus, the MSE models course provides a "scientific" basis for the other courses in the program.

Methods of software development. The methods course addresses the methods used in practical software development that lead from problem identification to a working software system. The course introduces students to effective approaches to requirements analysis, design, creation, and maintenance. The course builds on the notations and concepts introduced in the models course, demonstrating how they can be applied to real software systems development. Representative methods and notations include object-oriented methods, JSD/JSP, VDM, Z, Larch, structured analysis and design, cleanroom development, and prototype-oriented development.

Students gain in-depth knowledge of three specific design methods and are expected to understand the applicability of each. The use of formal methods is covered, and they are used as well to make vague notions found in informal methods more precise.

Management of software development. The management course focuses on the management and organization of resources—both human and computational—for large-scale, long-lived software development projects. It covers the management of individual software development efforts and long-term capability improvement, including life-cycle models, project management, process management, Capability Maturity Models, product control (for example, version and configuration management and change control), documentation standards, risk management, people management skills, organizational structures, product management, and requirements elicitation. The course is part of the core curriculum because many problems of a software development effort can be traced to poor organization, poor planning, and poor understanding of the ways in which people work together.

Table 1. The MSE curriculum.

Fall	Spring	Summer
Studio	Studio	Studio
Elective	Electives	
Models		
Methods	Analysis	
Management	Architectures	Elective

At first glance, the management course appears the most conventional, but it differs from traditional courses because it emphasizes requirements management, applies quantitative methods, and teaches high-leverage quality-assurance techniques. The course requires students to prepare a project management plan according to IEEE Standard 1058.1-1987.[2] Course topics follow the steps taken when writing a project plan, thus leading students through the process.

When studying quantitative methods, students learn the cost-estimation method Cocomo (1.0 and 2.0), the effort-estimation method of function points, and other quantitative estimation metrics. They also learn how to choose metrics for project tracking, how to interpret data, and how to use data for continuous process improvement. To help them estimate better, students learn how to shorten schedules rationally by calculating the expected return on an accelerated project and comparing that to the probable cost. Because quality is a theme throughout the management course, students learn about risk management and quality assurance techniques, such as inspections and cleanroom software development.

By building general strengths in quantitative analysis and in a variety of approaches for improving quality, the management course teaches students to use appropriate data to make principled choices among methods and models. This in turn teaches them to be managers with long-term adaptable skills, rather than managers who write the same project plan 20 times.

Analysis of software artifacts. The analysis course focuses on the analysis of software development products, including delivered code, specifications, designs, documentation, prototypes, and test suites. It covers both static and dynamic analyses, including type checking, verification, testing, performance analysis, hazard analysis, reverse engineering, and program slicing. Where appropriate, students use analysis tools.

The course adopts a broad view of analysis, including topics often divided into separate courses, such as testing, model checking, formal verification, and techniques for static analysis such as slicing. The course

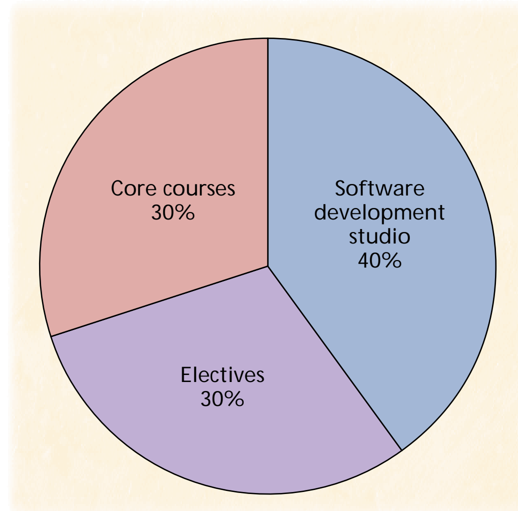

Figure 1. Program components.

builds on the models course, assuming that students have already learned the basic mathematical concepts on which many analytical techniques are based.

Architectures of software systems. The architectures course is concerned with the design of complex software systems at an architectural level. It covers the organization of complex software in terms of its system structure and the assignment of functionality to design components. The main topics include common patterns of architectural design, trade-off analysis at an architectural level, domain-specific architectures, automated support for architectural design, and formal models of software architecture.

Tackling the problem of structuring large-scale software systems, the architectures course covers the high-level components of a system and the interactions between them, rather than the data structures and algorithms that lie below the module boundaries. While software architecture is still an emerging field, it is likely to play an increasingly important role in software engineering.

While the methods course presents specific techniques and notations spanning the complete software development cycle, the architectures course focuses on the broader questions of software organization and high-level design. A specific method typically forces the method user to produce designs in a specific style, as when object-oriented methods use objects as the primary architectural building blocks. By understanding the broader "horizontal" architectural dimension, students can discriminate between different "vertical" methods. Conversely, by understanding the context in which architectural design plays a role, students obtain a broader perspective on the ways good architectures can be put into practice.

The studio

The studio, the centerpiece of the MSE program, represents close to 40 percent of the required course units, as shown in Figure 1.[3,4] It allows students to apply software engineering practices learned in the core curriculum to real-world situations. As "reflective practice,"[5,6] the studio not only requires students to complete the real-world project, but also to reflect on the rationale, motivation, and consequences of their decisions and actions. The emphasis on self-awareness throughout the studio experience helps students be more effective in their jobs, understanding and communicating effectively with other professionals. With this foundation, MSE graduates are prepared to act as agents of change and innovation throughout their professional careers.

Mentor's role. In the studio, students work under the guidance of selected members of the technical staff at Carnegie Mellon's Software Engineering Institute (SEI). Each student is assigned a mentor, who advises, encourages, and guides the student, meeting individually with the student at least one-half hour each week and attending project meetings.

During the individual meetings, students discuss studio work with the mentor, covering critical issues and individual responsibilities. The meetings are informal, with the mentor asking the student to reflect on decisions and to explore and understand rationale, consequences, and implications. The meetings also give mentors an opportunity to challenge students to innovate and take risks.

Nature of the studio projects. The critical issue is the choice of an appropriate project. The criteria of the program require that projects

- provide real value to a real client,
- be embedded in a larger development effort,
- be of sufficient scope to involve diverse skills and multiple team members, and
- be sufficiently challenging to permit exploration of the concepts and techniques learned in the core curriculum.

While projects must be part of a real software development effort, they should not be on the critical path of a program or an organization. This gives students greater freedom to explore higher risk approaches, allows them to make mistakes, and puts a focus on the learning experience rather than the schedules and needs of the client. Examples of studio projects are described in the "Recent Studio Projects" sidebar.

A key feature of the MSE program is that students participate in the studio through all three semesters. During the first semester of the studio, a "boot camp" introduces ideas about the nature of engineering, specifically software engineering, and provides training in software process management, including the personal software process.[7] Students examine engineering case studies (such as one on the construction of the new Chicago Public Library building), read excerpts from *What Engineers Know and How They Know It*,[8] and review literature on reflective practice.[5,6]

Each semester begins with a kickoff meeting that

Recent Studio Projects

The studio provides students with a laboratory for direct application of concepts learned in course work, and it has produced a variety of software products. Clients have included Boeing, NASA, Westinghouse, Innovative Systems, Carnegie Works, PPG Industries, and the US Air Force. Here is a sample of the studio projects:

MEDUSA: A software engineering team at Boeing Defense and Space Group was working on a distributed operating system for a new helicopter. Most of the engineers still used VT100-class terminals. They needed a way to seamlessly boot up and synchronize multiple target processors in order to perform system testing. The studio team implemented a form of windowing for the terminals that ran transparently under VMS and used VMS help and other operating system facilities.

Architectural visualization projects: Two different systems were implemented to help architects and their contractors manipulate different views of buildings in a coordinated way. Plumbers, HVAC engineers, electricians, and architects have usually reconciled their different needs in building construction by manual, awkward means. The systems implemented by the studio teams assisted in automating and intelligently reconciling such things as cable runs, pipings, and HVAC.

APEX: This semiautonomous robot has been used to explore the Moon and Mars. The studio worked on navigation software and system specification and reengineered part of the existing "standard" robot message-passing system, the task-control architecture.

Tessellator: Before each space shuttle launch, the shuttle's thermal protection system must be waterproofed by injecting a toxic chemical into each of the thousands of individual tiles—a job that is perfect for a robot. Two studio teams implemented the software to move and position the robot, move and position its arm, and plan the work.

TCAMS: The Tape Copy and Management System produced the software to control a robotic tape-mounting system and associated computers for the Air Force's B-2 test program.[1]

Reference

1. J.E. Tomayko et al., "Continuous Verification in Mission Critical Software Development," *Proc. 30th Hawaii Int'l Conf. Systems Science,* IEEE CS Press, Los Alamitos, Calif., 1997, pp. 273–285.

outlines studio activities and expectations for the semester. During the semester there may be other meetings to discuss specific issues, present solutions, or share views on the risks, problems, and general issues of the projects. Then at the end of each semester, the project teams make presentations to faculty, clients, and interested parties from the Carnegie Mellon community or local industry.

The studio is structured much like a small company, with multiple ongoing projects. Students take on the required support functions, switching roles over the course of the studio, so that they see the project from several angles. For example, a student may act as technical lead, project lead, or technical contributor for his or her own project and also be a member of an inspection team for another project. Each semester a different student acts as the studio manager, coordinating resources among the projects and overseeing studio-wide activities like end-of-semester presentations and risk meetings.

Students are encouraged to explore the limits of technology and take risks. Mentors play a role in this during the individual meetings, challenging the students to call upon the problem-solving skills stressed in the core curriculum. Switching team members among the critical roles and giving them various responsibilities in the support groups also helps, providing greater awareness of problems and their complexity.

For many students, the studio is their first opportunity to use the formal methods, object models, and other technology introduced in the core. Since these approaches may also be new to the clients, project team members directly face the issues associated with introducing new ideas into an organization, such as winning acceptance and handling transitions.

Evaluation criteria and student growth. The criteria used to evaluate student performance fall into three general areas:

- Teamwork, including professionalism, responsible participation, pulling one's weight, interacting with others, and intrateam communication.
- Application of course materials, including the use of metrics, techniques, and extrapolation.
- Quality of work, including timeliness, usability by others, and conformance with standards.

These criteria mirror the principal objectives of the studio, looking at the application of core curriculum material, at professionalism through teamwork, and at reflective practice.

Not only does the studio hone students' software-engineering and problem-solving skills, but it also emphasizes the importance of viewing all facets of software development in a broader context. By reflecting on their decisions and actions, students begin to recognize the multiple expectations, the diverse constraints, and the complex technical and programmatic interrelationships inherent in this work.

While all MSE students have had industrial experience, most have gotten it as technical contributors involved in detailed design or code development. Consequently, many students begin their studio work with little knowledge of the intricacies of complex real-world software development. By directly involving students in a real-world project, the studio allows them to experience the diversity of technical and programmatic issues facing senior software engineers.

Students often begin not understanding how their work relates to that of others or how the senior engineer should manage communication between the client or user and the project team, ensuring that each understands the other. Through mentoring, teamwork, and the opportunity to fail, students come to understand that they can excel technically and can, as a team of professionals, successfully deal with the multiple expectations and complexities of large software

Career path	Number of students	Percentage
Continued with present employer	28	30.4
Hired as process manager	21	22.8
Joined start-up or formed a company	27	29.4
Other	16	17.4
Total	92	100.0

Table 2. Career paths of graduates.

development efforts.

The rotation of critical project roles gives students opportunities to see through the eyes of others, confronting new problems, yet still dealing with the same set of expectations and constraints. Through self-reflection, students not only see the differences among the roles, but are also able to put them in the context of project goals and customer needs. For example, the senior technical lead faces critical technological issues, model details, and questions of correctness, completeness, and robustness. The project manager, however, faces scheduling, commitments, accountability, and other programmatic issues. Gaining such experience would take years of career development in most organizations.

As students progress through the studio project, their professionalism grows and matures. At the outset, they tend to become overly immersed in low-level details, fail to adequately address the customer interface, and ignore other critical communication issues. The studio and the core curriculum work together to mature the students, giving them both superior technical competence and the skills needed to make tradeoffs between theoretical and practical issues. Through this experience, they are able to bring about fundamental change and to successfully evolve software engineering practice within an organization.

Specializations

A single-year program in a subject like software engineering must necessarily strike a balance between breadth and depth. The MSE program allows students to follow any of a number of specialization tracks, which cover topical areas where Carnegie Mellon has strength. Each specialization includes four elective courses. The current MSE program includes specializations for real-time systems, business applications, human-computer interaction, and software process improvement.

Since most of the elective courses are offered through other departments, students come in close contact with faculty and staff from Carnegie Mellon's Department of Electrical and Computer Engineering, Graduate School of Industrial Administration, Human-Computer Interaction Institute, and SEI. Students not choosing a particular specialization can use the four electives to broaden their course of study according to their own needs.

RESULTS OF THE MSE PROGRAM

Over the nine years that the MSE program has existed, various program formats have been tried. Changes have been made in the mix of electives and core courses, in the number and extent of core courses, and in the overall length of the program. However, the original goal of developing future leaders in software engineering has stayed the same. The program has always been centered on a mentored, long-term project and has always provided a curriculum that goes beyond current practice, cultivating in students broad-based problem-solving skills.

Feedback from graduates indicates that they are able to bring practice and technological innovation to their organizations. A summary of the career paths of 92 graduates is shown in Table 2.

Of the 92 graduates, 30 percent (28) continued with their previous employers, who sponsored their education with the expressed purpose that they bring back what they learn to improve the company's software engineering practices. Those graduates have reported helping their companies adopt more effective software engineering practices and technologies. Almost 23 percent (21) of the graduates went to new employers as process managers, who are responsible for making significant technical and process changes. Twenty-nine percent (27) joined start-ups or formed their own companies. Many took this step because they believed that there are better ways to develop software and that established companies are too slow to change. Seventeen percent (16) followed other career paths, such as becoming a technical member of a development project or pursuing further study. While this data does not clearly show the specific impact of these graduates, it does indicate that a majority have assumed positions where they can have a significant technical impact on their organizations.

Beyond elements of the curriculum design already outlined, four factors contribute to the MSE program's success:

- *Experience requirement:* Students have had at least two years of experience in industrial software development. At one point the program experimented with allowing talented Carnegie Mellon seniors to enter the program directly, but in general this was found not to work, even when the undergraduate experience had been supplemented with summer internships.
- *Formal underpinnings:* A theme through the curriculum is the appropriate use of formalism. All of the core courses use some form of formalism, whether it be cost metrics, verification techniques, or architectural modeling. It is stressed, however, that formalism must be applied judiciously and that no single formalism is right for all situations.
- *Curricular integration:* The program is less a set of independent courses and more a convenient

partitioning of a closely related set of ideas. Pervasive themes include the appropriate use of formalism, trade-offs between power and generality, application abstraction techniques, and resource management. This integration allows core courses, the studio, and specialization tracks to build on each other, providing multiple points of reference and reinforcing fundamental ideas.
- *Innovative environment:* Much of the program's success is attributable to the stimulating environment for research and practice in which it operates. In particular, SEI has been crucial in providing studio mentors and courses in such areas as the software process. The Carnegie Mellon academic environment has provided a strong set of elective courses, allowing the MSE program to provide specialization tracks without a huge overhead. The university, too, has provided many excellent studio projects in areas such as robotics.

The program has its costs, however. First, being innovative, it rarely can find appropriate textbooks and must therefore depend upon reading collections and specially tailored course notes. Second, the close curriculum integration requires additional faculty effort, particularly higher awareness of the material in other courses and increased effort to coordinate course materials with studio work. And third, the rigorous program requires students who are particularly motivated and qualified, thus limiting the program to about 20 students a year.

Despite the demands and costs of the MSE program, the success of its graduates clearly proves its value. They return to work with technical knowledge, well-developed skills, and a breadth of experience that prepares them to be agents of change in the industry.❖

Acknowledgments

We thank the many people whose vision and contributions have shaped Carnegie Mellon's MSE program, particularly Daniel Jackson, Mary Shaw, and Jeannette Wing. We also recognize SEI, which launched the program, and SEI staff members who have served as studio coaches and mentors. We especially thank Nico Habermann, who developed the program's initial vision and set its tone. Finally, we note that development of the software architecture course was sponsored in part by the Defense Advanced Research Projects Agency under grant MDA 972-92-J-1002.

References

1. D. Garlan et al., "The CMU Master of Software Engineering Core Curriculum," *Proc. 8th Conf. Software Eng. Education,* Springer-Verlag, Berlin, 1995, pp. 65-86.
2. *IEEE Std. 1058.1-1987, Software Project Management,* IEEE, New York, 1987.
3. J.E. Tomayko, "Teaching Software Development in a Studio Environment," *SIGSCE Bull.*, Mar. 1991, pp. 300-303.
4. J.E. Tomayko, "Carnegie Mellon's Software Development Studio: A Five-Year Retrospective," *Proc. 9th Conf. Software Eng. Education*, IEEE CS Press, Los Alamitos, Calif., 1996, pp. 119-129.
5. D.A. Schon, *The Reflective Practitioner: How Professionals Think in Action*, Basic Books, New York, 1983.
6. D.A. Schon, *Educating the Reflective Practitioner: Toward a New Design for Teaching and Learning in the Professions*, Jossey-Bass Inc., San Francisco, Calif., 1987.
7. W.S. Humphrey, *Introduction to the Personal Software Process,* Addison-Wesley, Reading, Mass., 1997.
8. W.G. Vincenti, *What Engineers Know and How They Know It: Analytical Studies from Aeronautical History*, Johns Hopkins University Press, Baltimore, 1990.

***David Garlan** is an associate professor in the Department of Computer Science at Carnegie Mellon University, where he heads the ABLE Project. Garlan's research interests include software architecture, formal methods, and software development environments. He recently co-authored (with Mary Shaw)* Software Architecture: Perspectives on an Emerging Discipline *(Prentice Hall, 1996). Garlan received a PhD in computer science from Carnegie Mellon University.*

***David P. Gluch** is a senior member of the technical staff at the Software Engineering Institute at Carnegie Mellon University. His professional interests include reliable systems design, verification technologies for dependably upgrading software systems, and software engineering education. He received a PhD in physics from Florida State University and is a senior member of IEEE.*

***James E. Tomayko** is a principal lecturer in the School of Computer Science (SCS) at Carnegie Mellon University and a part-time senior member of the technical staff of the Software Engineering Institute (SEI). He is the acting director of the joint SEI/SCS Master of Software Engineering Program and the assistant dean of distance education for SCS. He received a PhD from Carnegie Mellon University in the history of technology and in information systems. Tomayko is on the Editorial Board of* IEEE Annals of the History of Computing.

Contact Garlan at the School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213; garlan@cs.cmu.edu. For more information, see the MSE Web site: