Agents of Change: Educating Future Leaders in Software Engineering

David Garlan† James E. Tomayko† David Gluch‡

†School of Computer Science
Carnegie Mellon University
Pittsburgh PA 15213
{garlan,jet}@cs.cmu.edu

‡Software Engineering Institute
Carnegie Mellon University
Pittsburgh PA 15213
dpg@sei.cmu.edu

DRAFT: Submitted for Publication

Abstract

Most software engineering professional degree programs focus on solving today's problems with today's technologies. While those programs are needed to help practitioners cope with the changing landscape of software technology, they do not adequately address the need to cultivate future leaders in software engineering. In this paper we describe a Masters of Software Engineering program that is specifically oriented towards the education of "agents of change"—practitioners who will be able to proactively infuse new science and technology into the software engineering profession.

1 Introduction

Software technology is continually changing. New methods, tools, platforms, user expectations, and software markets—all make for a rapidly shifting landscape within which practicing software engineers must function. This volatility underscores the need for advanced professional software engineering education. Without the ability to adapt quickly to new technologies and trends, software professionals can fail to anticipate and exploit emerging opportunities to improve their products and processes.

But what kind of education is required to meet this demand? The usual answer is that software professionals should stay abreast of technological advances by taking courses or degree programs that address immediate problems with current techniques and tools. For instance, as method X, language Y, or tool Z becomes popular, one typically finds courses offered in using X, Y, or Z. As a result, one finds that most professional degree programs are centered around curricula that address today's problems with today's techniques.

There is another approach, however. Rather than help software engineers cope specifically with today's perceived problems, instead provide those engineers with models, skills, and analytical

techniques that allow them to evaluate emerging technologies and proactively infuse the best of them into their organizations. Such engineers then become agents of change.

While this may seem a worthwhile goal, it is not at all clear how one would go about instituting such a program. How would its curriculum differ from a more traditional one? What kinds of hands-on experience should be offered? How much specialization should be required?

In this paper we describe our experience in answering these questions. For the past 9 years, the Carnegie Mellon University Masters in Software Engineering (MSE) has been developing and refining a curriculum specifically focused on cultivating future leaders in software engineering practice. To accomplish this it has adopted a novel approach centered on a long-term, mentored software development project, together with an unusual core curriculum that stresses broad-based models and problem solving skills.

The primary purpose of this paper is to expose the issues that arise when one considers the task of developing innovative leaders in the software engineering profession. While the discussion is framed in terms of our experience with a particular program, the goal is to provide insight into the possibilities for novel kinds of professional education in software engineering.

2 Key Issues

There are three critical issues that must be addressed in the design of any professional software engineering program.

1. The first is the choice of a *core curriculum*. In almost any program there are typically a set of required courses that establish a common base of knowledge and skills for all of the students in the program. The choice of courses in the curriculum is determined in large measure by what is viewed by the curriculum designers as the essential shared knowledge for the professional software engineer.

2. The second issue is whether to include *practical system development* in the program, and if so, how. This component of a program typically gives students hands-on experience developing a software system, and applying material of the courses to actual software development.

3. The third issue concerns the *degree of specialization* permitted by the program. Specifically, will the program provide opportunities for students to probe deeper into specific areas of software engineering?

To address the first issue, the usual approach to curriculum design is to follow a traditional software life-cycle model. For example, there will usually be courses in requirements specification, specific design methods (such as object-oriented design), testing, and verification. These courses operate more-or-less in stand-alone fashion. Each course can be taken in isolation, and in almost any order.

With respect to practical system development, the usual approach is to include a semester-long project course in the core curriculum. The practical experience component of a program therefore becomes a small extension of the core academic curriculum.

With respect to specializations, most programs provide a menu of elective courses, with little advance guidance about how to package those courses into a coherent advanced program. Typically these courses are already offered in the environment.

The standard approach to curriculum, practicum, and specialization, just outlined, has a number of attractions. First, it permits easy entree by practicing software engineers looking for part-time

supplementary education. Courses can usually be taken in any order, and the curriculum can be spread out over long periods of time. Second, the courses typically map directly to established software developmental stages, so it is relatively easy to identify the course in the curriculum that addresses a particular pressing problem. For example, if an engineer wants to improve testing skills, there will typically be a course specifically on that topic. Third, since the project component is localized in a single course, no special arrangements need to be made to satisfy the practical requirements of the curriculum: it is just another course to take. Fourth, 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, from the point of view of cultivating future leaders in software engineering it has a number of serious drawbacks. Most importantly, it fails to give students broad-based problem solving skills that go beyond immediate exploitation of current technologies. For instance, a student taking a testing course will undoubtedly not be confronted with the more fundamental question of how to make tradeoffs in appling a fixed set of resources to improve software quality. Those tradeoffs will involve understanding the relative merits of alternative analysis methods, from verification, to testing, to prototyping, to abstract modeling. Since the other topics may concern other parts of the life cycle, they will usually not be covered in this course, and therefore not enter the discussion.

A second problem is that traditional software engineering degree programs rarely give students the opportunity to develop their skills as agents of change on realistic software development projects. If of short duration, a project course can (at best) permit students to apply a single development method, and to explore a narrow set of development approaches. In particular, a single-course project does not have enough time to allow students to apply different techniques to see which work, to reflect critically on the design choices they made, or to follow through to see how the developed system is used by customers (if there even *are* any customers).

A third problem is lack of integration. Each course is typically treated in isolation, causing the information for a course to be compartmentalized. For example, a student who has taken a specification course may not recognize how the models developed during requirements analysis can be used during testing. Furthermore, 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 who pursue traditional software engineering programs rarely are in a position to evaluate new technologies or to understand the bigger picture of how emerging opportunities can best impact existing software development practices. For instance, if a new software development method suddenly becomes popular, how can one tell what its impact might be? Are there significant advantages behind the method's inevitable hype? What do you have to give up to use it?

3 The CMU Master of Software Engineering Program

The Carnegie Mellon University 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, and be able to apply to software development both the best of current practice as well as emerging technologies.

The program is operated as a joint program between the School of Computer Science and the Software Engineering Institute, and provides a three-semester (one calendar year), intensive curriculum for professional software engineers, leading to a terminal masters degree. Students accepted to the program must have both a strong background in computer science as well as two years' industrial software development experience that indicates strong potential for leadership.

The incoming class size for the program is usually about 20 students. In the past these students have had an average of about 5 years of industrial experience. About half of the students come from large corporations (Digital Equipment, HP, Westinghouse, General Motors, etc.), while the others represent a variety of smaller software development firms. Many of the students are supported by their employers, who expect them to return to the company after finishing their degree.

The MSE program is organized around three basic components: the Core Curriculum, the Software Development Studio, and a number of specialty tracks. The Core Curriculum develops foundational skills in the fundamentals of software engineering, with an emphasis on design, analysis, and management of large-scale software systems. The elective tracks provide an opportunity for students to develop deeper expertise in one of several specialties, including real time systems, human-computer interfaces, and software process improvement. In the Software Development Studio students plan and implement a significant software project for an external client over the full duration of the program. Students work in a team environment under the guidance of faculty advisors to analyze a problem, plan a software development project, execute a solution, and evaluate their work. In that respect, the Studio is similar to the design studios that characterize architectural degree programs.

Let us consider each of these components in more detail.[1]

3.1 The Core Curriculum

The MSE Core Curriculum consists of the following five semester courses:

1. **Models of Software Systems:** This course treats foundations for software engineering based on the use of precise, abstract models and logics for characterizing and reasoning about properties of software systems. Specific notations are not emphasized, although some are introduced for concreteness. The main topics include state machines, algebraic models, process algebras, trace models, compositional mechanisms, abstraction relations, temporal logic. Illustrative examples are drawn from software applications.

2. **Methods of Software Development:** This course addresses the practical development of software using methods that help bridge the gap between a problem to be solved and a working software system. The intent of the course is to introduce students to effective approaches to Requirements Analysis, Design, Creation, and Maintenance. Representative methods and notations include: object-oriented methods, JSD/JSP, VDM, Z, Larch, Structured Analysis/Design, Cleanroom development, and prototype-oriented development. In particular, students gain in-depth experience with three specific design methods, and are expected to understand the scope of applicability of each method.

3. **Management of Software Development:** This course focuses on the management and organization of resources – both human and computational – for large-scale, long-lived software development projects. It treats 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 (e.g., version and configuration management, change control), documentation standards, risk management, people management skills, organizational structures, product management, and requirements elicitation.

[1] For the purposes of this paper we provide high-level description. Additional details can be found in [1, 4, 5], and on the MSE web site: //http://http://www.cs.cmu.edu/ MSE/.

FALL	SPRING	SUMMER
Studio	*Studio*	*Studio*
Elective	*Electives*	
Models		
Methods	**Analysis**	
Management	**Architectures**	*Elective*

Figure 1: The MSE Curriculum

4. **Analysis of Software Artifacts:** This course focuses on the analysis of software development products including delivered code, specifications, designs, documentation, prototypes, test suites. It treats both static and dynamic analyses, such as type checking, verification, testing, performance analysis, hazard analysis, reverse engineering, and program slicing. Tools for analysis are used where appropriate.

5. **Architectures of Software Systems:** This course is concerned with the design of complex software systems at an architectural level of abstraction. It treats organization of complex software based on system structure and assignment of functionality to design components. The main topics include common patterns of architectural design, tradeoff analysis at an architectural level, domain-specific architectures, automated support for architectural design, and formal models of software architecture.

The Management, Models, and Methods courses are offered in the Fall semester. The Analysis and Architecture courses are offered in the Spring semester. Figure 1 shows how the curriculum is arranged.

3.1.1 Rationale

As noted earlier, the MSE Core Curriculum differs in significant ways from most other software engineering curricula. Most existing programs are organized around the software lifecycle. In contrast, this curriculum is organized around topics that cut across the development process. It emphasizes the underlying principles and techniques (such as the use of formal models and the application of good management principles) that can be applied in uniform ways to a broad spectrum of software development activities. The rest of this section outlines the rationale for each course and some of the important relationships between them.

Models of Software Systems evolved out of a conviction that it is essential that software engineers have a coherent understanding of the fundamental mathematical models that underlie most of the good abstractions of software systems. Unlike courses in formal methods found in other programs, the course focuses less on specific notations than on the pervasive models on which those notations rest. Moreover, unlike many formal methods courses, it considers not only the use of mathematical models for software specification, but also certain models that underly testing, analysis, process modeling, and design selection. Thus the models course provides a "scientific" basis for the other courses in the program.

Methods of Software Development was designed to help students gain in-depth experience with techniques that span the gulf between a problem to be solved and a successful implementation. This course coalesces material often distributed across a number of distinct courses in other curricula:

requirements specification, design, creation, maintenance. Among other things, the methods course builds on the notations and concepts introduced in the models course and demonstrates their practicality to real software systems development. Typically, among the methods treated in depth by the methods course, at least one will focus on the use of formal methods of software development. In addition, the methods course uses formal models to make more precise vague notions embodied in informal methods.

Management of Software Development focuses on organizational aspects of software development—project management, project planning, team organization, and process improvement. Its inclusion in the Core is based on the conviction that many of the problems in carrying out a successful software development effort can be traced to poor organization, poor planning, and poor understanding of the ways in which people can work together on a cooperative effort.

From a distance, the Managing Software Development core course appears to be the most conventional. It is distinguished from other courses of its type, however, by three characteristics: emphasis on requirements management, quantitative methods, and high-leverage quality assurance techniques. The course requires students to create an individually-prepared project management plan according to the IEEE standard. In fact, the course topics are structured in the order in which a project plan is written, and the assignments build toward its eventual completion.

With respect to quantitative methods, students study COCOMO (1.0 and 2.0), function points, and other quantitative estimation metrics. They are also taught how to choose metrics for project tracking, how to interpret data, and how to use data for continuous process improvement. Since poor estimation is still endemic in the industry, students learn how to shorten schedules rationally by calculating the expected return on an accelerated project versus the probable cost. Quality is a theme throughout the Management course, and students learn about risk management in some detail. They are taught how to manage quality assurance techniques such as inspections, and Cleanroom software development.

By building general strengths in quantitative analysis and a variety of approaches for improving quality, the course teaches students how to make principled choices among methods and models based on appropriate data. 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 integrates various techniques for understanding the things produced by software engineers. It adopts a broad view of analysis, including topics often divided into separate courses, such as testing, model checking, formal verification, and some techniques of static analysis (such as slicing). The analysis course builds strongly on the models course, by assuming that students have already learned the basic mathematical concepts that form the basis of many of the analytical techniques.

Architectures of Software Systems tackles head-on the problem of structuring large-scale software systems. It is concerned with system composition in terms of high-level components and interactions between them, rather than the data structures and algorithms that lie below module boundaries. While the field of software architecture is an emerging one, we believe that it will play an increasingly important role in software engineering.

The relationship between the methods and architecture courses is an interesting one. While the methods course presents specific techniques and notations that span the complete developmental cycle of software, the architecture course focuses on the broader questions of software organization and high-level design. Typically, a specific method will traverse a narrow region of the architectural design space. (For example, 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

Pie chart showing relative weights of core, studio, and electives. Attached at end of paper.

Figure 2: Relative Weight of Program Components

in which architectural design plays a role, students obtain a broader perspective on the ways good architectures can be put into practice.

3.2 The Studio

As the centerpiece of the MSE, the Studio represents close to 40% of the required course units in the MSE program. (See Figure 2.) The Studio experience allows students to apply software engineering practices learned the Core Curriculum to real world situations.

The guiding principle of the studio experience is "reflective practice" [2, 3]. Students are required not only to complete a "real world" project, but are challenged to reflect on the rationale, motivation, and consequences of their decisions and actions. This focus on self-awareness throughout the Studio experience helps students to be more effective in doing their jobs and understanding and communicating effectively with other professionals. Collectively these hallmarks of the Studio experience form a foundation for graduates of the MSE program to act as agents of change and innovation throughout their professional careers.

3.2.1 Mentor's Role

Studio projects are conducted under the guidance of selected members of the technical staff at the CMU Software Engineering Institute (SEI), who act as mentors. Each student is assigned a mentor, whose role is to advise, encourage, and guide the student throughout the Studio experience. Students and mentors meet individually with their students for at least one half hour each week and attend project meetings.

Student-mentor meetings involve discussions of the studio work and allow the student to address, one-on-one with their mentor, critical issues regarding their Studio project and individual responsibilities. Meetings are driven both by the risks associated with the project and also the individual issues facing the student. The format of the meeting is informal and is based upon an interactive style of asking the student to reflect on decisions and to explore and understand rationale, consequences, and implications. It is also an opportunity for mentors to challenge students to innovate and take risks in their projects.

3.2.2 Nature of the Studio Projects

A critical issue for the Studio is appropriate choice of projects. The primary criteria used to select projects are that the product of the effort should provide real value to a real client, be embedded in a larger development effort, and be of sufficient scope to involve diverse skills and multiple team members. In addition, projects are required to be sufficiently challenging to permit exploration of the concepts and techniques learned in the Core Curriculum. While realism is a key ingredient, at the same time it is important that projects not be on the critical path of a program or organization. This restriction allows students greater freedom to explore higher risk approaches, make mistakes, and focus on the learning experience rather than being schedule-driven or unduly pressured by a client. (Examples of recent Studio projects are summarized in Figure 3.)

A key feature of the Studio is that students participate in it for the full length of the program. The inaugural segment of the Studio has been christened "boot camp." This introduction presents

ideas about the nature of engineering, and specifically software engineering. It also provides training in software process management (including Personal Software Process). Students learn from case studies (such as the construction of the new Chicago Public Library), read excerpts from "What Engineers Know and How they Know It [7], and review literature on reflective practice [2, 3]. This is where the themes of reflective practice and the broader engineering context that pervade the Studio experience are first introduced.

At the beginning of each semester there is kickoff meeting that outlines the Studio activities and expectations for that semester. Throughout the year there are meetings to discuss, present solutions, and share evaluations of risks, problems, and general issues encountered in the projects. At the end of each semester, each project makes a presentation to faculty, clients, and other interested parties in the CMU community and local industry.

The Studio is structured much as a small company might be organized with multiple projects and required support functions matrixed together. Each student plays several roles in this matrix, and positions rotate over the course of the Studio. For example, a student may be a technical lead, a project lead, or technical contributor on his/her project and also participate on an inspection team for another project. One student acts as the studio manager who coordinates resources among the projects, studio-wide activities (e.g., the end of semester presentations, risk meetings).

Students are encouraged to explore the limits of technology and take risks. This is fostered and guided via the interaction with mentors at individual mentoring sessions and project meetings. They are challenged to call on problem solving skills stressed in the Core Curriculum to deal with new situations in innovative ways. Greater awareness of problems and their complexity is also achieved by switching team members among the critical roles within the project and having team members participate in the matrixed responsibilities of support groups.

For many students the Studio is the first time for them to apply formal methods, object models, or other technology introduced in the Core. Often these approaches are also new to the clients and project members directly face the issues associated with introducing new ideas into an organization. In these situations they deal directly with transition challenges and are provided "front line" experience.

3.2.3 Evaluation Criteria and Student Growth

Criteria for evaluation of student performance in the Studio are grouped into the three general areas of teamwork, application of course material, and quality of deliverables. These criteria mirror the principal objectives of the Studio by considering individual excellence in the application of the Core Curriculum material, professionalism through team work, and reflective practice. Figure 4 provides further details on these criteria.

The Studio attempts not only to hone students' software engineering and problem solving skills but also to engender an appreciation of the importance of viewing all facets of the development activity in a broader context. By reflecting on their decisions and actions, students begin to recognize the multiple of expectations, diverse constraints, and the complex inter-relationships (both technical and programmatic) that characterize software development.

While all MSE students have had industrial experience, usually this experience has been as a technical contributor involved in detailed design or code development. Consequently many students begin their Studio experience relatively naive regarding the intricacies of a complex real-world software development effort. Through the Studio experience students are provided direct involvement in understanding and addressing the diversity of technical and programmatic issues that face a senior software engineering professional.

The Studio provides students with a laboratory for direct application of concepts learned in course work. It has produced a variety of software products. Clients have included Boeing, NASA, Westinghouse, Innovative Systems, Inc., Carnegie Works, PPG Industries, and the United States 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, including using VMS Help and other operating system facilities.

ARCHITECTURAL VISUALIZATION PROJECTS: Two different systems were implemented to assist architects and their contractors manipulate different "views" of buildings in a coordinated way. Usually plumbers, HVAC engineers, electricians and architects have to reconcile their different needs in building construction by manual and awkward means. The systems implemented by the Studio teams assisted in automating and intelligently reconciling cable runs, pipings, HVAC, etc.

APEX: This project is a semi-autonomous robot used to explore the Moon and Mars. The Studio worked on navigation software and the system specification. It also re-engineered part of the existing "standard" robot message-passing system, The Task Control Architecture.

TESSELLATOR: Before each Space Shuttle launch, its thermal protection system must be waterproofed by injecting a toxic chemical into each of the thousands of individual tiles—a perfect job 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 (TCAMS) produced the software to control a robotic tape mounting system (and associated computers) for the Air Force's B-2 test program [6].

Figure 3: Some Recent Studio Projects

1. Team work aspects
    (a) professionalism
    (b) responsible participation
    (c) pulls weight
    (d) interaction with others
    (e) intra-team communication
2. Applying course material
    (a) metrics
    (b) techniques
    (c) extrapolation
3. Quality of Work
    (a) timeliness
    (b) usable by others
    (c) conformance with standards

Figure 4: Studio Evaluation Criteria

The professional maturation of students is evident as they progress through the Studio. At the outset students tend to become overly emersed in low-level details, fail to address adequately the customer interface, and ignore other critical external and intra-project communication issues. Students often do not realize that their work impacts and integrates with others and that it is a responsibility of a senior engineer to communicate (translate) between customer/user contexts and their own. Through mentoring, team work, and the opportunity to fail there is a realization that they can excel technically and, as part of a team of professionals, successfully deal with the multiple expectations and complexities of large software development efforts.

By rotating critical project roles among project team members, students get the opportunity to see through the eyes of others. While they confront new problems in a different role, students also encounter the same expectations and constraints again within the Studio. Through self-reflection students are encouraged not only to realize the differences but to put them into the context of the project goals and customer needs. For example, as senior technical lead critical issues relate to the technology, details of the models, correctness, completeness, robustness, etc. In contrast, the central focus for a project manager involves schedule, commitments, accountability, and other programmatic issues. Through the Studio a student experiences first-hand the diversity of expectations and challenges associated with software development. Gaining comparable experience with a multiplicity of perspectives may take many years of career development in most organizations.

The Studio experience matures a student's superior technical competence established through the Core Curriculum in concert with the skills required to make trade-offs between theoretical and practical issues; and through this experience, the capability to precipitate fundamental change and successfully evolve software engineering practice within an organization.

3.3 Specializations

Designing a single-year program in a subject such as software engineering inevitably requires one to balance issues of breadth and depth. We have defined specialization tracks that students can use to gain greater depth in fields in which CMU has strength. These specializations are targeted to specific areas that are (a) topical, and (b) exploit existing strengths in our environment. Each specialization is packaged as a set of four elective courses. Currently there are specialization sequences in real time systems, business (for students not doing the joint degree with the Graduate School of Industrial Administration), human computer interaction, and software process improvement.

Most of the courses that make up the specializations are drawn from existing electives. Students following a specialization thus come into close contact with faculty and staff of the Department of Electrical and Computer Engineering, the Graduate School of Industrial Administration, The HCI Institute, and the SEI. This enriches their overall experience pursuing the degree. Students not choosing a particular specialization track, however, can use the four electives as a method of broadening their course of study, or peruse shorter depth sequences.

4 Experience

The CMU MSE has now been in existence for nine years. Within this period we have experimented with a variety of program formats and design. Over time a number curricular parameters have changed including as the relative balance between electives and core courses, the number and extent of core courses, and the overall length of the program.

Throughout its existence, however, the program has remained committed to its original goal of developing future leaders in software engineering practice. This vision has always been built around

the conviction that a mentored, long-term project should serve as a centerpiece for the curriculum. Further, it has attempted to provide a curriculum that goes beyond current practice by cultivating students' broad-based problem solving skills.

Based on feedback from our graduates, overall we believe that the program has largely lived up to its mission. Almost without exception, students who finish the program are acting in innovative leadership roles in their companies. Many have brought major technological innovaton into their organizational practices.

Beyond the curriculum design that we have already outlined, we can identify a number of factors that have contributed to its success. Among these, four stand out.

- **Experience requirement:** Currently all students in the program have at least two years experience in industrial software development. We have experimented with modifying this requirement—primarily to allow talented seniors to enter the program directly. We have found that this general does not work, even when the undergraduate experience has been supplemented by summer internships.

- **Formal underpinnings:** A consistent theme throughout the curriculum is the appropriate use of formalism. All of the core courses use some forms of formalism, whether it is cost metrics, verification techniques, or architectural modeling. However, we are careful to stress that formalism must be applied judiciously, and no single formalism is right for all situations.

- **Curricular integration:** We view the program less as a set of independent courses than as a convenient partitioning of a closely related set ideas. Pervasive themes include judicial use of formalism, making tradeoffs between power and generality, application abstraction techniques, and management of resources. This view allows Core courses, Studio, and specializatons to build on each other and provide multiple points of reference and reinforcement of fundamental ideas.

- **Innovative environment:** A large portion of the program's success is attributable to a rich environment of research and practice in which it operates. In particular, the resources of the SEI have been crucial in providing mentors for Studios and instruction in areas such as software process. Similarly, the CMU academic environment has provide a strong set of supplementary courses that allowed us to fashion the specialization tracks without a huge overhead. It also has turned out to provide a source of excellent Studio projects in areas such as robotics.

On the other hand, the program is an innovative one and requires some special treatment in certain areas. First, it is often hard to find textbooks that match well with the curriculum. As a result, many of our courses depend on reading collections and specially-tailored course notes. Second, the close integration of the curriculum requires additional effort by the faculty. In particular, an instructor must both be aware of what the other courses are providing, and work hard to take advantage of inter-course/studio opportunities. Third, we have found the program to be a particularly demanding one. This implies that students must be particularly motivated and of high caliber. This has limited the size of our program to about twenty students a year.

While these additional costs are not negligable, we believe that they are more than worth it. As instructors in the program we have found that the reward of seeing students emerge as agents of change more than justifies the extra effort it takes to design and deliver this kind of professional software engineering education.

References

[1] David Garlan, Alan Brown, Daniel Jackson, Jim Tomayko, and Jeannette Wing. The cmu master of software engineering core curriculum. In *Proceedings of the 8th Conference on Software Engineering Education*, pages 65–86, New Orleans, Lousiana, March 1995. Springer-Verlag, LNCS 895.

[2] Donald A. Schon. *The Reflective Practitioner: How Professionals Think in Action.* Basic Books, Incorporated, 1983.

[3] Donald A. Schon. *Educating the Reflective Practitioner: Toward a New Design for Teaching and Learning in the Professions.* Jossey-Bass Incorporated, 1987.

[4] James E Tomayko. Teaching software development in a studio environment. *SIGSCE Bulletin: Papers of the 22nd SIGSCE Technical Symposium*, 23(1):300–303, March 1991.

[5] James E Tomayko. Carnegie mellon's software development studio: A five-year retrospective. In *Proceedings of the Ninth Conference on Software Engineering Education*, pages 119–129. IEEE Computer Society Press, 1996.

[6] James E. Tomayko et al. 'continuous verification' in mission critical software development. In *Proceedings of the Hawaii International Conference on Systems and Software*, 1997.

[7] Walter G. Vincenti. *What Engineers Know and How They Know It: Analytical Studies from Aeronautical History.* Johns Hopkins University Press, 1990.