

The "Soft" Topics in Software Engineering Education

Mark A. Ardis, Stephen V. Chenoweth, and Frank H. Young
 ardis@se.rit.edu, chenowet@rose-hulman.edu, young@rose-hulman.edu

Abstract - Engineering educators struggle with “soft” topics – topics which include a social element. Soft engineering topics are distinct from the scientific and mathematical underpinnings of engineering. Students frequently complain when these topics are integrated into engineering curricula. Engineering educators also express concerns that they lack both preparation and ability to teach these topics.

Software engineering educators have an even greater problem. More soft topics need to be included in a good software engineering program. Furthermore, software engineering instructors need to remain current with non-social best practices, which can leave little time to study techniques to incorporate soft topics into the curriculum.

This paper is an attempt to alleviate the problem. We organize the soft topics necessary for a good software engineering program; present exercises to include them in software engineering courses so that students develop needed abilities and understand their importance ; and describe effective ways to evaluate student performance.

Index Terms - software engineering, soft topics, process, best practices, experiential learning.

THE SOFT TOPICS

We divide soft topics into eight useful categories:

- **Observing:** listening, watching, recording behavior
- **Reviewing:** reading and providing feedback
- **Presenting:** preparing and presenting information, answering questions
- **Writing:** preparing written documents and artifacts
- **Planning:** organizing, estimating, and synthesizing process activities
- **Cooperating:** working together to complete a task
- **Reflecting:** recording and learning from past events, updating plans, abstracting process
- **Judging:** making ethical judgments, dealing with conflict, evaluating peers

These categories are motivated by their prevalence in software engineering subject matter. We judged that 79% of the 215 bottom-level topics in the Software Engineering Body of Knowledge hierarchy involve skills in one or more of these eight soft areas [1]. Software engineering students need to learn and practice skills in each of the categories. Furthermore, each can, conceptually, be a clear theme of the lessons. In this paper we provide reasoning and examples regarding software engineering education in each soft-topic category.

The **Observing** category covers topics that are usually grouped under the heading of being a good listener, but also includes more analytic topics such as recording the behavior of users. These abilities help software engineers understand client needs, gather requirements, and understand user behavior. They also help software engineers be more effective participants in meetings of all kinds.

The ability to observe human behavior may be developed in social science classes taken prior to software engineering, particularly when those classes include an element of observation or experimentation with humans. This ability also may be supported by the attitudes and habits developed by carefully observing experiments in science laboratories. However, students are accustomed to succeeding by influencing human interaction and may find it difficult to take a more passive role.

In software engineering, observation is an exploratory and goal-directed activity, not a science. The engineer needs to be not quite so focused, especially at the beginning of a project. The engineer must be open to new problems and new opportunities which may be involved in achieving the goal. New sources of information must be welcomed – a new problem often comes with a new client and new users.

Students may be poor at accurately recording careful observations of human behavior because their usual behavior is to analyze while they observe. Their analysis reduces the behavior to terms they already know, thus obviating the need to record what actually happened. This learned reaction is a well-known and serious problem in group brainstorming, an activity which combines observing and planning. The “analyze while doing” method of observing must be unlearned if students are to become effective brainstorming contributors. One technique we have used successfully to teach good observing is to demonstrate this ourselves in a class. Another technique is to have students record observations of some common event in class, such as a client meeting or a system in use, before they record behavior on their own. If they compare notes about their common observation, they learn from each other. In our experience, shared practice observation sets a standard which consistently improves the quality of what students then do on their own.

In software engineering practice, observing is blended with other activities. Working with customers means providing feedback, often strong feedback in the form of negotiation and conflict management. It requires reflecting and judging based on the recorded observations. It builds

toward writing and presenting results for others. Thus, exercises in observing should include the following:

- Basic observing, such as watching and recording as users operate systems, and
- Applications of observing, such as:
 - recording while contributing during brainstorming
 - recording while guiding requirements elicitation
 - listening to value positions of different stakeholders while trying to reach a consensus

In software engineering, observation is called for when the knowledge sought is not already known. Thus, canned exercises where the instructor knows the right answers ahead of time are by nature unrealistic. Students will learn much more when the exercise is really novel, as opposed to guessing what the instructor wants or knows.

The unknowns of software development make it critical for students to acquire skills at observation-based activities such as brainstorming. In group brainstorming using a flipchart, there is a standard “good rate” for new ideas – one idea per minute. If that is to happen, participants need to be tuned into the flow of ideas coming from others, rather than just waiting to jump in with their own idea. Other specific techniques can improve on the rate of idea generation; for example, if students know how to do “idea writing” on Post-It notes, their ideas can be added to a shared wall of ideas almost as fast as they can read them and stick them on the wall. Table 1 shows a comparison of traditional brainstorming and idea writing, based on our experiences in using these techniques. The table suggests additional heuristics for judging brainstorming effectiveness.

Evaluation of student observations can be done via feedback from later stages of a team project. For example, how much rework had to be done as a result of knowledge that was missed when observations were being made?

The **Reviewing** category includes reading and providing feedback on documents, designs, plans and code. Formal reviews are often called “inspections”. Abilities in this category help software engineers understand and improve software engineering artifacts. Also included in this category is feedback to other groups, such as feedback from testers to developers. In software engineering the need for diplomacy in giving feedback is particularly high when dealing with project funders and with customers and suppliers. The goal of most reviews is to move an ongoing project forward.

Students have vast experience in reading and analyzing technical material. However, they are not well trained in giving feedback [2]. Students often provide feedback to instructors by determining what to say based on criteria like “what someone who already knows the answers wants to hear from a person learning those answers.” This criterion is inappropriate for most software development interactions.

Students are at a great disadvantage when reviewing input from other parties in a business environment. They do not know the customs and folkways of the environment. As undergraduates they are slowly learning how to get along with people who will be their engineering peers. This group

is not representative of, say, vice presidents who serve as clients to large software development organizations.

Students need a graduated growth plan toward achieving diplomacy when providing feedback. In lower level software engineering classes students act as each other’s clients. They move on to working with faculty members as clients. Finally, perhaps in their capstone course, they work with outside business clients who might actually hire them. Such graduated growth needs to be consciously planned and carefully integrated into the software engineering curriculum.

Feedback to other team members is a systematic way to improve skills on soft topics. For team building, feedback exercises emphasize openness and frankness, in addition to treating peers respectfully.

The **Presenting** category is primarily concerned with explaining new information to an audience. Abilities in this category help software engineering practitioners give oral presentations of all kinds – from informal status reports to formal project presentations.

School presentations differ from software development project presentations. In the usual school presentation the rest of the students in the room are only casually interested in the matter being presented. In software development projects the audience is prepared to act as a result of the information being presented. For example, software architects present their design to a group of developers who, immediately afterwards, will have to go build things based on that design. Or, a presentation to a customer of a team’s solution leads to the customer signing-up to pay for it.

Instructors must try to create a realistic environment for student presentations by replicating the electric atmosphere of a real software project presentation. Student presentation exercises should be designed to have real consequences.

Software project presentations tend to be about half question-and-answer sessions. Promoting serious questioning during a student presentation adds to realism. Listeners and presenters should be evaluated primarily on the quality of the questions and answers during the session.

Evaluation of presentations can be done by using check lists to rank specific attributes of presentations as unacceptable, poor, average, good, or excellent. When such a check list is used it should be shared with the students before its use and discussed with them after being completed. Peer evaluation of presentations can also be used. Each student is asked to give some sort of evaluation of the presentation (perhaps using the check list mentioned above) and the grade received by the evaluating student is determined by how close the student’s evaluation is to the class norm. Students can do self-evaluations of their presentations, perhaps after viewing a videotape of it.

Students can be asked to record the best and worst characteristics of a specific presentation. The list generated from the student comments should be discussed with the presenter(s) to help them improve their performance.

| Brainstorming method: | Advantages | Disadvantages | When to use | When not to use | Best practices |
|---|--|--|--|--|--|
| Traditional (with flip chart) | People listen to each other's ideas. | They forget what they were going to say. | Group of 8 – 14 people having diverse viewpoints. | Too many divergent voices must be heard. | Ask for main idea first, write verbatim as fast as possible as each contributor expands on it. Move quickly to next. |
| | Builds group cohesiveness. | Participants need to be reminded of expected behavior. | A skilled facilitator is available. | Someone with existing strong group presence (like the boss) has to facilitate. | Write the problem statement on first page, to avoid divergence. |
| | Can see ideas easily on flip chart. | Room dynamics control contributions. | Participants used to sharing ideas and learning. | People afraid to talk openly, or unsure what to say. | Keep all flip chart pages visible in the room, so they can inspire more ideas. |
| Idea writing (with PostIt notes on a wall) | People can add their ideas very rapidly. | Less focused attention on others. | Groups of 4 – 14 contributors. | Contributors less likely to think of own new ideas. | Have contributors pre-write some ideas before meeting, on provided PostIts. |
| | Ideas can be offered more privately. | Some people have trouble writing concise ideas. | Organizing ideas is an important part of decision. | Building group cohesion is a key goal. | Announce each idea as its PostIt is added to the wall. |
| | Group can go back and organize PostIts as a next step. | Have to go up to wall to read ideas already posted. | Ideas can be explored further later on. | A decision or action plan is needed at the meeting. | Pick up PostIts in a well-defined order, write-up as an outline. |

Table 1 – A comparison of traditional brainstorming and idea writing

The **Writing** category covers creation of all artifacts that use natural language. Abilities in this category help software engineers create formal and informal documents [3].

Student attitudes toward any assignment involving writing are strongly influenced by their previous experiences with writing assignments in humanities and social science courses. Software engineering educators need to confront the prejudices and misconceptions of their students in an effective fashion. Software engineering instructors have to give writing assignments that are very carefully named and that are not called writing assignments.

In fact this is easy to do. Most interesting and significant writing assignments in software engineering are not the standard exposition or “compare and contrast” essays that students are used to. Software engineering writing assignments are strongly related to software engineering tasks that professionals must perform if they are to be effective in their work. Instructors should emphasize this aspect of any writing assignment – not the writing exercise itself. The goal of any writing assignment in a software engineering course is to help the student be a more effective software engineer, not a more effective writer. In fact, designating a software engineering course as a “writing

intensive course” may be counter-productive, reinforcing student prejudices and misconceptions regarding writing.

What specific exercises could be given in software engineering courses? Here is a short list to give some indication of the breadth of possibilities. Note that the word “write” seldom appears in this list of exercises although every one involves significant amounts of writing.

- Translate someone’s verbal instructions into written form
- Create instructions for using a software tool to accomplish a specific task
- Prepare the agenda for a team meeting
- Compile the minutes of a team meeting
- Analyze competing products, systems, or methods
- Prepare a memo in support of (arguing for) a specific proposal
- Prepare an annotated bibliography on a given topic
- Prepare a report giving basic information about a proposed project (what is known, what is uncertain, what risks are involved)
- Analyze and evaluate possible solutions to a specific problem or risk situation, showing impacts
- Create a project assignment

- Describe some “standard operating procedures” for a new member of your team
- Create a feasibility report for a proposed product or system
- Create a specification document
- Describe the architecture of an existing software system
- Create some advertising copy or a press release for a software product
- Prepare an agenda or plan for an interview
- Create a summary of an interview
- Create a job description for a member of your team
- Prepare a performance evaluation of a co-worker or a member of a project team
- Develop a personal professional development plan
- Prepare a progress report to a superior (short memo, formal report, or final report)
- Edit, review, analyze, or annotate a memo or report

Evaluation of student writing can be done in several ways. The obvious way is for the instructor to do the evaluation, using a simple standard: Is the artifact useful to and usable by a software engineering professional? This standard should be stated clearly before writing assignments are given. Software engineering professionals should have no difficulty applying this standard to students’ written work. Note that this standard does not require the instructor to correct grammar and spelling but does require the student to write comprehensible and unambiguous prose. Obviously, grammatical errors, spelling errors, lack of clarity, and ambiguity make written work difficult to use. Instructors need to emphasize this point repeatedly.

A less obvious way to evaluate writing assignments is to have students use them in subsequent work. This has the advantage of letting the student experience the evaluation of the work [4]. If the students can use the artifact easily and effectively, then they are giving it a “high grade.” If the students cannot use the artifact easily and effectively, then they are assigning it a “low grade.” It should be easy to point out to the students that their experiences when using documents are equivalent to assigning grades to the documents.

A third way to evaluate writing assignments is through peer review: students providing constructive criticism of one another's work. This works best for first drafts that will be improved in response to the criticism. Note that this type of evaluation should not be used for grading.

The **Planning** category includes the abilities needed to organize people and estimate tasks. These topics are often taught in courses that cover project management. Abilities in this category help software engineers prioritize and staff project activities, manage the complex relationships among them, and synthesize a coherent plan for the project..

In software engineering, planning a project is usually done after an initial requirements phase, during or shortly after the time when a preliminary design is done. Thus, the planning includes commitments to acquire outside software and staffing based on the expected technical needs. It is not easy to duplicate the reality of this planning activity in a

class because there is no budget and students already have been assigned to teams and projects. In a class setting the real “planning” is done by the instructor, as a vehicle for moving students through the course.

It is essential for students to play an active role in setting their own destinies. This is common in adult education scenarios, where employed engineers often select real projects from their jobs as a part of a class. The strategies are akin to those recommended in Knowles’ “Andragogy” [5], at the core of which students take responsibility for their learning experience. The question of how to make student planning experiences realistic leads toward using such methods. But transitioning undergraduates to a more accountable role is not easy.

An initial planning exercise would require a team to be responsible for some aspect of planning a software development project, then work under their own plan, to measure the results. Evaluation of such an exercise would emphasize the team’s following the plan and documenting the results.

Estimation is a necessary part of planning. Students often feel that they do not have the skills or preparation necessary for making accurate estimates. These feelings can be partially dispelled by encouraging student estimates to be a range or interval with an associated estimate of the probability that the actual value will be in the given range. Such estimation is easier for students to do and also helps students understand the risks involved in software projects. Another method we have used to teach estimation is Wideband Delphi [6], which pools student knowledge.

Students also need some experience recording the effort expended when developing software. One way to do this is to teach a rigorous approach to estimation and subsequent feedback from data collection, such as espoused in PSP [7]. However, more informal methods are often just as effective. Students should be encouraged to estimate and then record expended effort for many different types of recurring tasks, such as homework assignments, weekly reports and bug fixes.

The **Cooperating** category includes many different forms and aspects of performing group work. Essentially all software projects are too large and complex for any one person to solve without reliance on others. Software engineering *is* an example of group problem solving. Skills in cooperation are not optional [8].

Prior experience in working with teams is an advantage for students taking software engineering courses. They often get some valuable background in their introductory engineering and science courses. Team projects may be made integral to the computer science curriculum, to build teaming skills in anticipation of software engineering work. Cooperation on teams is learned largely by experience, and prior experience and success is a strong predictor of future success. Students who enter upper division software engineering courses still valuing their individual work above teamwork will be at a serious disadvantage. Overcoming such attitude problems takes time and conscious effort.

It is important to recognize the broad extent of cooperating activities. Whenever a task is performed by more than one person there is a need for cooperating abilities. This includes actions that are usually considered sequential and therefore not interdependent. Examples include coding and testing (where each group must cooperate with the other one) and specification, implementation, and documentation (where all products should agree even though completed by different groups).

Cooperative learning [9] is a foundation for the teaching of soft topics generally. Use of cooperative learning methods has a direct impact on understanding of mechanisms for cooperation.

The **Reflecting** category includes topics where students examine artifacts and processes in order to improve them. Abilities in this category are used by software engineers to study and improve their practices,

Metrics, logs, and other control data must be kept in any case, so as to understand what has been accomplished and what remains to be done on a project. These tools additionally enable retrospective activity,

Students should practice reflecting at the conclusion of every team project [10]. They can assess the processes they used, their own contributions, and what went right and wrong. They also need to document these things. In multi-stage projects this can be done more than once, so that process improvements can be tried and then re-evaluated.

Instructors need to recognize that time must be provided for reflection and means must be given to make the results of the reflection visible. In more advanced software engineering classes it is recommended that each student keep an engineering notebook (possibly in the form of a private blog) that is reviewable by the instructor. This should include reflections as well as ideas for designs, rationale for decisions, and similar contents.

One of the most valuable ways for instructors to encourage reflection is to use class time for reflective discussions – project retrospectives. It is useful to prepare for such discussions by giving an assignment that asks the students to record their reflective thoughts on a specific topic. The discussion of any topic will be at a much higher level if the students have been asked to record their thoughts about the issue beforehand.

Advanced students can also give “lessons learned” presentations to following classes. Such presentations pass best practices down to the next students and generate steady improvements in the quality of project work.

The **Judging** category includes topics involving value judgments by students, such as making ethical decisions and evaluating peers. In engineering schools this subject area largely is a study of the code of ethics for a particular engineering discipline. Beyond that, students are called upon to provide justifications and case study solutions based on their own personal values [11].

The traditional exercises involve writing analyses of case studies where judgment calls must be made. This is a

start. However, actual personal interaction in ethical decisions extends this, enables more reflection on meaning, challenges student values, and gives further dimensions to peer evaluation [12].

Software engineering has a code supported by the ACM and IEEE [13]. However, this foundation leaves open the meaning of “Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest,” because software is, in general, applied to *all* other fields of endeavor and in all cultures. A recommended extension to the usual covering of “engineering ethics” in a single class is to revisit this topic regularly during the capstone project, where students themselves may be faced with decisions involving ethical and social dimensions.

Students should be made aware of the many value judgments that are made when software is created. Determining requirements, designing and implementing systems, and documenting systems all involve value-laden decisions. Which disabilities will the software compensate for and which will be ignored? Which operating systems will be supported? Which security issues will be handled? Which stakeholders have the most influence on the system requirements? Answering these questions requires judgments that involve more than technical issues and information.

RELATED WORK

Several other researchers – Bailey [14], Costin [15], Denning [16], Lee [17], Lethbridge [18], and Snell [19] – have observed that software developers need soft skills as well as technical knowledge in order to be successful in today's industry. For example, Lethbridge [18] conducted a survey of professionals that concluded that five of the most important 25 topics for software developers and managers were: ethics and professionalism, giving presentations, technical writing, leadership and negotiation.

Friedman [20] and Jazayeri [21] argue for the inclusion of social aspects in software engineering courses, especially through project work. Seat [22] reports on the use of course modules to teach communication and teamwork skills for all engineers. Seffah [23] uses patterns to describe effective approaches for teaching communication skills. Surendran [24] proposes that students enter into apprenticeships as a way of developing some of the soft skills.

Schön [25- 26] observed that reflection is both a needed skill and a method of developing other expertise, especially engineering design. Some engineering educators, such as Hazzan [27] and Socha [28] have followed Schön's advice and incorporated reflective practices into their curricula. Indeed, the studio environment of the graduate software engineering program at Carnegie-Mellon was created in order to simulate the reflection-in-practice environment of the architectural design studio [29].

Many of the exercises we describe here are in common use in software engineering programs. Gersting and Young [2] describe several useful exercises for computer science

students that develop some of the soft topics we have described. A series of their columns in the *SIGCSE Bulletin* describe sample exercises for Writing [3], Cooperating [8], Reflecting [10] and Judging [4, 11, 12] topics.

CONCLUSIONS

Educators normally consult with colleagues and professional publications regarding methods for presenting technical concepts. Most educators begin course design activities with an appropriate search of the literature about pedagogical methods for technical concepts. Educators must be willing to freely use ideas developed by others, improving them as possible and appropriate and sharing them after they have been improved.

Software engineering educators should treat soft topics the same way that they do technical and scientific topics. When courses are designed there should be an appropriate search of the literature regarding pedagogical methods for soft concepts. Educators should use freely the ideas of others regarding the incorporation of soft topics, improving them and sharing them after they have been improved.

This paper has organized the soft topics important for software engineering education into eight categories to help educators use and apply soft topics in their teaching. It is hoped that this will facilitate the meaningful and effective incorporation of soft exercises into all software engineering curricula.

REFERENCES

- [1] Guide to the Software Engineering Body of Knowledge (SWEBOK), 2004 Version. IEEE Computer Society Professional Practices Committee.
- [2] Gersting, J.L. and Young, F.H. "Content + experiences = curriculum" *Proceedings of the twenty-eighth SIGCSE technical symposium on computer science education*, 1997, 325-329.
- [3] Gersting, J.L. and Young, F.H. "Shall We Write?" *ACM SIGCSE Bulletin*, Volume 33, Issue 2 (June 2001), 18-19.
- [4] Gersting, J.L. and Young, F.H. "Sharpening Subjective Evaluation Skills" *ACM SIGCSE Bulletin*, Volume 31, Issue 2 (June 1999), 26.
- [5] Knowles, M. *Andragogy in Action* San Francisco: Jossey-Bass, 1984.
- [6] Boehm, B. *Software Engineering Economics*, Prentice-Hall, 1981.
- [7] Humphrey, W.S. *Introduction to the Personal Software Process* Addison Wesley Longman, 1997.
- [8] Gersting, J.L. and Young, F.H. "Improving the Team Experience" *ACM SIGCSE Bulletin*, Volume 33, Issue 4 (December 2001), 18-19.
- [9] Johnson, D., Johnson, R., Holubec, E., *Cooperative Learning in the Classroom*, Alexandria, VA; Association for Supervision and Curriculum, 1994.
- [10] Gersting, J.L. and Young, F.H. "Projects---After They are Finished" *ACM SIGCSE Bulletin*, Volume 29, Issue 4 (December 1997), 24.
- [11] Gersting, J.L. and Young, F.H. "Experiences With Ethical Issues" *ACM SIGCSE Bulletin*, Volume 32, Issue 2 (June 2000), 20-21.
- [12] Gersting, J.L. and Young, F.H. "Experiences With Ethical Issues: Part 2" *ACM SIGCSE Bulletin*, Volume 32, Issue 4 (December 2000), 18-19.
- [13] *Software Engineering Code of Ethics and Professional Practice, Version 5.2* ACM/IEEE-CS Joint Task Force on Software Engineering Ethics and Professional Practices, September 1998.
<http://www.acm.org/serving/se/code.htm>.
- [14] Bailey, J.L., Stefaniak, G. "Industry perceptions of the knowledge, skills, and abilities needed by computer programmers", *Proceedings of the 2001 ACM SIGCPR conference on Computer personnel research (SIGCPR '01)*, April 2001, 93-99.
- [15] Costin, G. "Legitimate Subjective Observation [LSO] and the Evaluation of Soft Skills in the Workplace" presented at the Spotlight on the provider conference, Albury, New South Wales, May 2002.
- [16] Denning, P.J. and Dunham, R. "The Profession of IT: The Core of the Third-Wave Professional", *CACM 44(11)*, 21-25, November 2001.
- [17] Lee, D. M. S., Trauth, E. M., and Farwell, D. "Critical Skills and Knowledge Requirements of IS Professionals: A Joint Academic/Industry Investigation" *MIS quarterly* vol:19 iss:3, 1995, pg:313 -340.
- [18] Lethbridge, T.C. "What Knowledge is Important to a Software Engineer?" *IEEE Computer*, May 2000, 44-50.
- [19] Snell, S., Snell-Siddle, C., and Whitehouse, D. "Soft or Hard Boiled: Relevance of Soft Skills for IS Professionals" *Proceedings of the 15th Annual Conference of the National Advisory Committee on Computing Qualifications* (pp. 403-408). Hamilton: NACCQ, 2002.
- [20] Friedman, B. and Kahn, P.H., Jr. "Educating computer scientists: linking the social and the technical" *Communications of the ACM 37(1)*, 64-70, 1994.
- [21] Jazayeri, M. "The Education of a Software Engineer" *19th International Conference on Automated Software Engineering*, 2004.
- [22] Seat, E. and Lord, S.M. "Enabling Effective Engineering Teams: A Program for Teaching Interaction Skills" *28th FIE Conference*, 246-251, November 4-7, 1998.
- [23] Seffah, A. "Learning the Ropes: Human-Centered Design Skills and Patterns for Software Engineers' Education" *ACM Interactions 10(5)*, September-October 2003, 36-45.
- [24] Surendran, K., Hays, H., and Macfarlane, A. "Simulating a Software Engineering Apprenticeship" *IEEE Software* September/October 2002, 49-56.
- [25] Schön, D.A. *The Reflective Practitioner* Basic Books, 1983.
- [26] Schön, D.A. *Educating the Reflective Practitioner: Towards a New Design for Teaching and Learning* Jossey-Bass, 1987.
- [27] Hazzan, O. and Tomayko, J.E. "Reflection and Abstraction in Learning Software Engineering's Human Aspects" *IEEE Computer*, June 2005, 39-45.
- [28] Socha, D., Razmov, V., and Davis, E. "Teaching Reflective Skills in an Engineering Course" *Proceedings of 2003 ASEE Conference*, 2003.
- [29] Tomayko, J.E. "Carnegie-Mellon's Software Development Studio: A Five-Year Retrospective" *SEI Conference on Software Engineering Education*, 1996.

AUTHOR INFORMATION

Mark A. Ardis Professor & Graduate Program Coordinator, Rochester Institute of Technology, ardis@se.rit.edu.

Stephen V. Chenoweth, Associate Professor, Rose-Hulman Institute of Technology, chenowet@rose-hulman.edu

Frank H. Young, Emeritus Professor, Rose-Hulman Institute of Technology, young@rose-hulman.edu