

From Toy System to Real System Development: Improvements in Software Engineering Education

Bernd Bruegge

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213 USA
bruegge@cs.cmu.edu

Abstract

We identify four problems that must be addressed to improve the quality of teaching a software engineering course that mirrors the real world: We have to start teaching iterative and incremental design, we need to introduce students to the problems of negotiation, co-ordination and team-work, we have to learn how to re-use large complex systems across project courses and we must overcome the inadequacy of traditional means of dissemination of course materials. We identify the pedagogical implications of these problems that have to be addressed before we can hope to improve software engineering education. The ideas presented in this paper are currently investigated in the framework of a senior undergraduate course at Carnegie Mellon University.

1 Introduction

Education in science and engineering has recently come under scrutiny for its lack of emphasis on practice. To address this deficiency we have developed a software engineering course that emphasize practice through a large associated project in which all the students work together with real clients from the surrounding community including city planners, police departments and environmental engineers. Teaching more comprehensive software engineering and exposing students to multi-disciplinary, team-based system design early in their career is important because distributed, collaborative development is the way in which systems most are designed now and will be designed in the future in academic and industry[11][13][16][29][30]. Our view of teaching software engineering arises from our experience with repeated offerings of this software engineering course at Carnegie Mellon University[4][6][7][8][9].

Table 1 gives an overview of the projects developed since 1989. The projects are associated with either a basic or an advanced software engineering course. In the

Year	Application Domain (Project)	Methodology	Process	Students	Teams	LOC (reuse not included)	Tools & Systems used
Fall 89 (Basic)	Communication Management (Workstation Fax)	object-based design	Waterfall	13	4	13,000 (C)	RCS, Scribe
Spring 91 (Basic)	City Planning (Interactive Maps)	SA/SD	Waterfall	16	4	17,000(C)	StP, Larch, RCS, Scribe
Fall 91 (Basic)	City Planning (Interactive Pittsburgh)	OMT[25]	Waterfall with one prototype	33	5	9,000(C) 18,000 (C++)	OMtool, NIH Library, RCS, MacProject, Framemaker
Spring 92 (Advanced)	Environmental Pollution Modelling (GEMS)	OMT	Reengineering (analysis, design and implementation in parallel)	10	5	20,000 (C++)	OMTool, Macromind Director, Interviews, IBuild, RCS, Framemaker
Fall 92 (Basic)	Accident Management (FRIEND)	OMT	Modified Waterfall (iterations during analysis and during design)	46	6	8,000 (C) 31,000 (C++)	OMTool, LEDA, MacProject, RCS, Framemaker
Spring 93 (Advanced)	Accident Management (FRIEND II)	OMT	Reengineering	11	5	30,000 (C++)	OMTool, Macromind Director, CVS, LEDA, Framemaker
Summer 93 (Advanced)	Remote Health Care (Health-Link)	OOSE[15]	Rapid prototyping with focus groups & structured interviews	22	3	10,000	Objectory, Borland Paradox, MS Windows for Workgroups, Powerpoint
Fall 93 (Basic)	Accident Management (FRIEND III)	OOSE	Spiral model with one iteration	55	9	21,000 (C++) 11,000 (tcl)	Objectory, LEDA, Exodus, WAIS, Sphinx, GPS, CVS, MS Project, Framemaker, Powerpoint

Table 1: Single Project Courses at Carnegie Mellon University

basic course, the students are asked to develop a new system from scratch, learning the development methodology and the concepts of model-based software engineering. The software process is based on the waterfall model with software development activities ranging from requirements analysis and design to implementation

and testing. In the advanced course the students are required to deal with an existing system - usually the system developed in the basic course - and elicit new requirements from the client. The software process is based on iterative and incremental development. As a result, the advanced course exposes the students to the concepts of reengineering, requirements engineering and requirements tracking. We believe these are important concepts for any software engineer and we have recently started to introduce these concepts in our basic software engineering course as well. In the Fall 92 course, for example, the students followed a modified waterfall model with several iterations during requirements analysis and system design (Booch's macro/micro process lifecycle model). In the Fall 93 course we used the same problem domain, but significantly extended the functional and global requirements of the system. Some of the requirements were negotiated between the students and their client, the Bellevue Borough Police Department. For the development process we used a spiral model with one iteration (The delivered system[17] provides multi-modal input such as speech recognition and gestures, wireless communication between nodes, as well as automatic tracking of mobile resources with a global positioning system subsystem). Even though the course has been received well both in the student and academic communities, it faces some generic problems that are common to any team-based design activities. In the initial implementation of the course, where the project size was small, these problems were relatively unimportant. As the size of the projects expanded to include as many as 55 students these problems were exacerbated. The main set of problems encountered are:

- Iterative and incremental design results in communication problems and breakdowns leading to unsatisfactory end products.
- Students do not have negotiation, coordination and team-work skills which are required for large scale software development.
- We have not been able to reuse previous systems and apply them across projects in a satisfactory manner.
- The traditional means of dissemination of course material has shown to be insufficient.

In the following we will address each of these each issues and their pedagogical implications for teaching software engineering.

2 Iterative and incremental design results in communication problems

The hard issues in the development of complex software systems are requirements engineering, requirements analysis in the context of ill-defined requirements[29] and redecomposition of the design. In the past these topics were rarely covered in software engineering courses, often because we did not know how teach

them in a satisfactory manner. To provide the students with a positive learning experience we tended to select those problems that can be solved with a high chance of success within the duration of the course. As a result, many instructors chose as project topic the development of relatively simple systems with well-defined requirements and a straightforward system decomposition. However, by avoiding the hard issues we are producing students that do not know how to deal with the real problems in system development when they are leaving the university. Large complex systems in general do not have well-definable requirements. Redecomposition is an important topic in the reengineering of legacy systems and should be taught to our students.

The rapid change in our field over the last few years, in particular the emerging of object-oriented development methodologies[2][15][25] has improved the situation along several dimensions, the most important one being the need for well-definedness of the initial system description. A primary characteristic of the process in developing a system with ill-defined requirements is the iteration through one or more development phases before the requirements or the design are clarified[1]. By exposing students to the problem of revising and iterating a larger scale system design through several levels of the software lifecycle, we are able to give them better exposure to actual software engineering practice.

Iteration is not easy to teach within a fixed project structure. Sometimes the clarification of a requirements leads to a redecomposition of the system, which in turns demands a change in communication structures and task assignments. Thus any fixed communication and task assignment structure becomes an obstacle for teaching effective software engineering of large-scale systems. However, we argue that we can actually teach software engineering better by opening the course to include requirements engineering and redecomposition. In fact, we believe this is not just to expose students to more complicated design issues, it is a matter of practicality; in software engineering, it is simply not possible to predetermine the structure of the system, task assignments and system decomposition at the beginning of system development, in particular if the requirements are not pre-specifiable. This is exactly, what people in the rapid prototyping and in the object-oriented community have observed for quite some time.

Pedagogical implications: Incremental and iterative design techniques are crucial in order to maintain dependencies between tasks and system decomposition. A course infra-structure allowing fast changing communication and task structures must be established. Unfortunately this means that we cannot use traditional homework assignments and grading techniques any longer. Teachers must learn how to deal with incomplete but not incorrect homeworks. Students who have been condi-

tioned to hand in perfect homeworks must also be re-educated. We need new ways to identify the contributions made by students in iterative development projects.

3 Software engineering requires new skills

Traditionally, project courses have concentrated mainly on providing the students with technical skills but have emphasized very little of the social aspects of problem solving such as team-based iterative design processes that are the norm in any realistic industrial or commercial setting [14]. There is growing recognition that software engineering, while being an analytic technical process, is embedded in a larger synthetic social process[10][20][24][30]. In this view of software engineering, the collaborative process of analysis, design and documentation of an artifact is the creation of the “theory” of the artifact or a shared understanding of the artifact's function, behavior and structure along with the dynamic co-ordination of the development process[14][21]. Any approach to teaching software engineering must address integration of computational aids with the product development processes and practices encompassing social, organizational and communication issues and information requirements and structures[26].

While the course employs team based system design, the instructor works out all the details of the system beforehand (the traditional model for conducting project courses as mentioned above) so as to create clean partitions for the student groups to work on and integrate. This model, while providing a structure that encourages team work, does not emphasize negotiation, and other collaborative task and problem decomposition skills which are critical to the understanding of software systems in real world settings. A software engineering course with a fixed structural approach to system and team decomposition and task assignments results in difficulties in the development process and the quality of the product as well as the educational experience.

Changing these structures manually and making these changes visible to the students is a time consuming task. When attempted during the semester, it almost certainly leads to problems, because the change cannot effectively be communicated to all of the project members in a short time to effectively set it into place. The instructors, who set up the communication structure at the beginning of the semester, do usually not have the time during the semester to effect the change, because they are busy with the lectures and with managing the project. And teaching assistants generally don't have enough project management experience to communicate this change.

The main problem is that we don't have a mechanism that can communicate the change without intervention by the teacher. What we really need, is a tool that can broadcast changes such as a redecomposition to all the project members within

a “reasonable update cycle”. Collaborative design tools have recently appeared which promise to improve this situation. Their existence has prompted us already to add a lecture on information management in the Fall 92 and 93 courses to cover the occurrence of communication and negotiation problems and how they could be minimized by a particular collaborative design tool [12][18].

Pedagogical implications: The importance and awareness of negotiation has to be emphasized in the course so as to encourage continual evaluation and critique of designs. Tools for the illustration of the course infrastructures and for teaching and management methods to enable dynamic negotiation of system, team and task re-organization will have to be analyzed and incorporated into the software engineering curriculum.

4 The importance of reuse across projects

To teach realistic software engineering problems, there is a need for large complex systems that can be understood and modified by the student. While complex systems have been developed by industry, they are rarely useful in a software engineering course. In general they have not been designed with those development methodologies that we want to teach, they are hard to obtain because of privacy or company concerns and they are not easy to transfer to a university platform.

In the past computer science curricula have dealt with the need for complex system experience by offering project courses in compiler construction or building small operating systems[19]. We believe that many software engineering problems are not touched upon with these courses. First, the requirements for these types of systems are more or less well understood. Second, and more importantly, the “external” client for compilers and operating systems come from our own “community” of computer scientists and software engineers. This makes it sometimes hard for students to distinguish between application (analysis) and design problems. Another tendency in software engineering has been to concentrate on the development of relatively small systems such as vending machines, elevators and simple information systems to be able to cover the whole software development process or to concentrate on those aspects of the process that we understand how to teach relatively well. We believe it is more important to expose students to large complex problems such as environmental pollution modeling[5], remote health care[3] and crisis management systems[23], because it forces the students to deal with very relevant issues, namely how to deal with an unknown application domain and how to elicit requirements.

Projects of the latter category require a large investment by the instructor to set up the infrastructure for the course. It would be extremely useful if this structure could be reused by another instructor. However, reuse is still in its infancy. With the

advance of object-oriented design methodologies, large class libraries have become available that allow students to reuse lower-level components as black boxes. Unfortunately, reuse of projects is not as simple as reuse of class components. Projects from previous courses are documented to differing extents in terms of process and efforts. In many cases software engineering instructors go from semester to semester assigning projects to students with very little of previous efforts carried over to the current project. For instance, software models on the analysis and design levels and the software process are rarely reused as possible starting points, thereby encouraging the myth among the students that all systems are designed from scratch; as a result, the importance of building on other work as building blocks is lost.

There are successful examples for the reuse of projects in software engineering courses, for example by the use of project templates[22]. The key to project templates is that the students redevelop portions of a system that has already been implemented. Depending on the complexity of the removed component the system could be used in lower levels and other courses. For example, the street database developed in the Interactive Pittsburgh project[7] has already been used in a graph algorithm class. Such a system could also be used in a data structure and algorithm course with an algorithm or a component selectively removed. The advantage of this approach is that we can expose students to system issues quite early in their career. By introducing system issues in beginner courses, we would be following other curricula in other areas. The MIT robotics course, for example, involves the design of a robot in a freshman course. System design issues in electrical engineering at Carnegie Mellon University are also now routinely taught on the freshman level.

Project templates as a basis for knowledge and experience transfer are generally insufficient, especially if the same instructor does not teach the course again, because they do not contain information about the design rationale and project history, which are most important for a teacher trying to reuse a project.

Pedagogical implications: Teaching reuse and learning across projects cannot be solely dependent on the instructor. In addition to the instructor, students will need access to prior projects with as much of the development context captured as possible in order to understand the problems and the process of reuse. Achieving this objective requires new aids for capturing the design rationale and history that are institutionalized in the course infrastructure.

5 Dissemination of course material

Dissemination of project information and the reuse of libraries in the context of software engineering requires comprehensive computer based support and organization to be utilized effectively. Without a transformation in how we teach and propa-

gate the art of teaching software engineering, effective transfer and dissemination of new methods and materials will remain restricted.

We need to find better ways to capture, package and export knowledge and experience that is cumulatively generated through the teaching of software engineering courses. This need arises at several levels and boundaries of increasing scope and significance: across versions of the same course, across similar courses in different disciplines and at different levels of the curriculum within the same university, across universities, and finally at the national and international level across all levels and forms of education including pre-college and industrial training courses.

Traditional ways of distributing knowledge in terms of technical reports, papers and tar files are insufficient[27][28]. Employing improved methods and materials for teaching software engineering demands that we go beyond traditional paper based communication of the experiences and results of project courses. To be able to reuse software has long been a goal in software engineering, but although most people agree on the importance of reuse, it has been practised only very rarely. The main reason has been the dependence of the software components on their application domain and the difficulty to reuse them across different application domains. However, this constraint does not necessarily apply to the teaching environment if the teacher chooses to stay within the same application domain.

Pedagogical implications: Electronic records (in the form of on-line navigable webs of information, templates and case studies) of projects and systems with their history and design rationale need to be created to provide a repository of project development and educational experiences that would increase both dissemination and reuse of high quality, “debugged” instructional materials across institutions.

6 Conclusion

We identified four problems that need to be addressed to improve the quality of realistic software engineering courses. The first two problems, our inability to teach iterative and incremental design as well as the need for negotiation, co-ordination and team-work skills, are encountered within the context of a given course. We claim that in order to teach software engineering effectively we must enable students to participate fully in the engineering of the system's requirements, in its structuring and decomposition. These activities must be engaged in iteratively and incrementally through a cyclic development process since there is no way to get a complex system specified correctly the first time nor with a linear process of activities and products. Engaging groups of students in an iterative and cyclic design process demands that we understand more clearly how to support dynamic negotiation and communication processes, and how to teach collaboration through establishment of an overall shared understanding of a system.

The second set of problems consists of our inability to reuse previous systems and apply them across projects, and the inadequacy of traditional means of dissemination of course material. These problems are encountered with respect to repeated teaching of such a course. We need to find better ways to capture, package and export knowledge and experience that is cumulatively generated through the teaching of software engineering courses with associated projects. This need arises at several levels and boundaries of increasing scope and significance: across versions of the same course, across similar courses in different disciplines and at different levels of the curriculum within the same university, across universities, and finally at the national level across all levels and forms of education including pre-college and industrial training courses. The dissemination of improved methods and materials for teaching software engineering demands that we go beyond traditional paper based communication of the experiences and results of project courses. In addition, we must capture in electronic form more of their history and design rationale in addition to just the software produced.

The ideas presented in this paper are currently investigated in the framework of a software engineering senior undergraduate course at Carnegie Mellon University. We have made first experiences with incremental and iterative software development [4] and we are planning to incorporate information modeling systems for the collaboration and negotiation of requirements into future versions of our course. We also have first experiences with reusing a complex application domain across the same course: The accident management system FRIEND has been developed in three courses involving different syllabi, development methodologies (OMT, OOSE), CASE tools (OMtool, Objectory), user interfaces (Motif, NextStep, Tcl/Tk), and platforms (Workstations, PCs and wireless lap-tops). Hypertext versions of the documentation for some of these systems are available via ftp and have already been distributed to other universities.

7 Bibliography

- [1] B. Boehm, A Spiral Model of Software Development and Enhancement, in Software Engineering Project Management, R. Thayer (ed), IEEE Computer Society Press, pp. 128-144, 1987.
- [2] G. Booch, Object-Oriented Design with Applications, Benjamin Cummings, 1991.
- [3] C. Aquilina et al., Wireless Data Communications: Market Opportunities and Technical Considerations, Tech. Report FPO-4, Information Networking Institute, Carnegie Mellon University, October 1993.
- [4] B. Bruegge and R. Coyne, Teaching Iterative Object-oriented Development: Lessons and Directions, 7th Conference on Software Engineering Education,

- Lecture Notes in Computer Science # 750, Springer Verlag, January 1994.
- [5] B. Bruegge, E. Riedel, G. McRae and T. Russel, GEMS: A Geographic Environmental Modeling System, to appear in Computer, IEEE.
 - [6] B. Bruegge and R. Coyne, Model-based Software Engineering in Large-Scale Project Courses, Proceedings of the IFIP Working Conference on Software Engineering, Hong Kong, September 1993.
 - [7] B. Bruegge, J. Blythe, J. Jackson and J. Shufelt, Object-Oriented System Modeling with OMT, Conference Proceedings OOPSLA '92 (Object-Oriented Programming Systems, Languages, and Applications), ACM Press, pp. 359-376, October 1992.
 - [8] B. Bruegge, Teaching an Industry-oriented Software Engineering Course, C. Sledge (ed), Software Engineering Education, Lecture Notes in Computer Science # 640, pp. 65- 87, Springer Verlag October 1992.
 - [9] B. Bruegge, J. Cheng and M. Shaw, A Software Engineering Course with a Real Client, Carnegie Mellon University, Tech. Report CMU-SEI- 91-EM-4, July 1991.
 - [10] L. Bucciarelli, An Ethnographic Perspective on Engineering Design, Design Studies, Vol 9, p.160, 1988.
 - [11] L. Constantine, Building Structured Open Teams To Work, Proceedings: Software Development '91, Miller Freeman, San Francisco, 1991.
 - [12] J.C. Ferrans, D. W. Hurst, M.A. Sennett, B.M. Covnot, W. Ji, P. Kajka and W. Ouyand, Hyperweb: A Framework for Hypermedia-based Environments, In Software Engineering Notes, Vol 17, pp. 1-10, December 1992.
 - [13] L.H. Fisher, Getting Involved Early in the Software Development Process. IPCC 1988 Conference Record. On The Edge: A Pacific Rim Conference on Professional Technical Communication. Seattle, WA. October 5-7, 1988.
 - [14] F. Floyd, F. Feisin and G. Schmidt, STEPS to Software Development with Users, 2nd European Software Engineering Conference, pp. 48-64, 1989.
 - [15] I. Jacobson, M. Christerson, P. Jonsson and G. Overgaard, Object-Oriented Software Engineering: A Use Case Driven Approach, Addison-Wesley 1992.
 - [16] J. Jacquot, J. Guyard and L. Boidot, Modeling Teamwork in an Academic Environment, in Software Engineering Education, J.E. Tomayko (ed.), Lecture Notes in Computer Science, Springer Verlag, pp. 110-122, 1990.
 - [17] A. Leyderman et al., FRIEND Fall'93, 15-413 Software Engineering Project Documentation, Carnegie Mellon University, December 1993.
 - [18] S. Levy, E. Subrahmanian, S. Konda, R. Coyne, A. Westerberg, Y. Reich, An Overview of the n-dim Environment, Tech. Report, EDRC-05-65-93, Carnegie Mellon University, January 1993.
 - [19] B. Meyer, Toward an object-oriented curriculum, Journal of Object-Oriented Programming, pp. 76-81, May 1993.
 - [20] S. Minneman, The Social Construction of a Technical Reality, Proceedings of NSF Workshop on Information Capture and Access in Engineering Design Environments, Cornell University, Ithaca, NY, 1991.

- [21] P. Naur, *Computing: A Human Activity*, ACM Press, Addison-Wesley, NY, 1992.
- [22] A. J. Offutt and R. H. Untch, *Integrating Research, Reuse, and Integration into Software Engineering Courses*, C. Sledge (ed), *Software Engineering Education, Lecture Notes in Computer Science*, Vol 640, pp. 88-98, Springer Verlag, 1992.
- [23] K. O'Toole, E. Liu, S. Gemma, D. Pascua, *FRIEND: First Responder Interactive Navigational Database*. Tech. Report, Information Networking Institute, Carnegie Mellon University, October 1993.
- [24] W. Scacchi, *Managing Software Engineering Projects: A Social Analysis*, *IEEE Transactions on Software Engineering*, 10 (1), pp. 45-59, January 1984.
- [25] James Rumbaugh, *Object-oriented Modeling and Design*, Prentice Hall, 1991.
- [26] G. Toye, M. Cutkosky, L. Leifer, J. Tenenbaum and J. Glicksman, *SHARE: A Methodology and Environment for Collaborative Product Development*, Techn. Report 0420, Center for Design Research, Stanford University, 1993.
- [27] E. Subrahmanian, S. Konda, S. Levy, I. Monarch, Y. Reich, A. Westerberg, "Computational Support for Shared Memory in Design", To appear in: *Automation-Based Creative Design: Issues in Computers and Architectures*, edited by I. White and A. Tzonis, Elsevier, 1993.
- [28] E. Subrahmanian, R. Coyne, S. Konda, S. Levy, R. Martin, I. Monarch, Y. Reich, A. Westerberg, *Support System for Different-Time Different Place Collaboration for Concurrent Engineering*, WET-ICE (Workshop on Enabling Technologies In Concurrent Engineering, CERC, West Virginia, USA, 1993.
- [29] J. Wood and D. Sover, *Joint Application Design*, Wiley and Sons, New York, 1989.
- [30] Zeidenstein, K. *Collaboration as Innovation: Why Technical Communicators Should Be Members of the Software Development Team*. IPCC 1988 Conference Record. *On the Edge: A Pacific Rim Conference on Professional Technical Communication*. Seattle, WA. October 5-7, 1988.