A Project-Unified Software Engineering Course Sequence

James S. Collofello and Scott N. Woodfield
Computer Science Department
Arizona State University
Tempe, Arizona 85287

ABSTRACT

In this paper a sequence of software engineering courses based upon the software life cycle and integrated by a single, medium-size project will be described in detail. The courses will be presented from an educational point of view, emphasizing the topics covered as well as the logistics of teaching the courses. A comparison of these courses to other software engineering courses existing in university curricula will also be presented. The potential advantages for faculty, students, and the research community of this type of course sequence will also be enumerated.

1.0 Introduction

The high cost of developing and maintaining large-scale software is a well-known fact. Software quality problems, especially those of reliability and maintainability, are also frequently noted. The term "software engineering" was coined in the late 1960's in response to some of the early symptoms of these problems. Software engineering basically attempts to apply an engineering-type discipline to developing and maintaining software. Since the coining of the term "software engineering", a vast amount of research and experience has been accumulated in the form of guidelines, techniques, tools, and methodologies. The software life cycle which is depicted in Figure 1 has also been documented. The distinction between programming and software engineering can be expressed in terms of this life cycle. In general, a programmer is primarily concerned with the production of code, where a software engineer has responsibility for the entire life cycle including requirements, overall design, testing, and maintenance plans.

The vast and diverse body of knowledge encompassing the software engineering field can be mostly classified in terms of its applicability to phases of the software life cycle. For example, entire books have been written on design methodologies and testing techniques. The fact that a vast body of knowledge exists for each phase of the software life cycle is obvious by a review of the software engineering literature which consists of many books, dedicated journals and conferences, as well as papers and technical reports from many different sources.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1982 ACM 0-89791-067-2/82/002/0013 $00.75

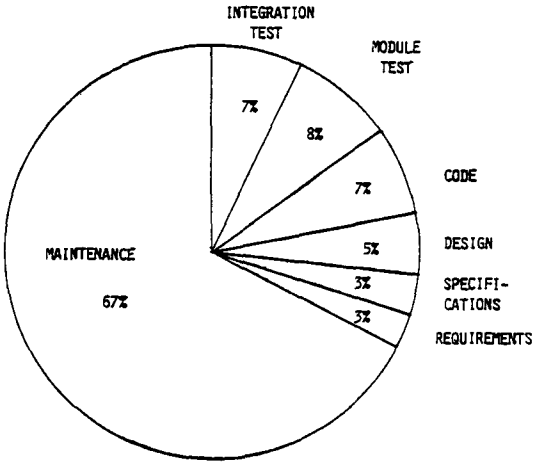

Fig. 1. Software Life Cycle

It is essential that this body of knowledge be conveyed to those responsible for development of software if today's demands for high-quality economical software are to be met.

The transfer of software engineering knowledge can be accomplished in several ways. In some organizations the process may consist of in-house training in which particular company-developed or acquired tools and techniques are taught. Professional seminars and tutorials are other ways of transferring software engineering knowledge. The universities provide still another avenue. Within university curricula, software engineering concepts related to program design and coding are evident in many courses, including elementary programming classes. Special semester-long software engineering courses are also becoming popular. In this paper, a four-semester sequence of software engineering courses modelled after the software life cycle and integrated by a single four-semester project will be described. This sequence of courses will be contrasted with the other approaches to transferring software engineering technology. The courses will be described in detail and the advantages of the sequence approach for the instructor, the students, and the research community will be noted.

2.0 Current Software Engineering Education Practices

In the last five years, there has been an increasing awareness of the need for software engineering education. In an attempt to satisfy this need, three educational approaches have been proposed and implemented. The first approach uses seminars and short courses to introduce many to the topic. The second is taken by the many educational institutions which offer a single course in software engineering. The last approach is a full master's degree in software engineering. Each of these has its place in education but the strength and weakness of each must be understood.

The seminar approach is very useful for introducing many of those people unfamiliar with software engineering to the basic concepts and terminology. The desire for this type of education can be seen from the number of courses offered. Hardly a month goes by without some new brochure arriving which describes some software engineering course offered somewhere in the United States. (The authors themselves offer such a course biannually in the Phoenix area.) Many people taking these courses are satisfied with the scope and content. They need only understand a litte of the area and have no desire to know more. Others wish to understand the entire area of software engineering, for these, such courses are superficial and do not come close to meeting their needs.

There are two basic problems with short courses. First, the volume of information is so large that it cannot be presented in the few hours allocated, except at a very elementary level. Second, to understand the basic software engineering principles completely, one must learn them not only in the classroom but also through experience. In an attempt to overcome these problems, many universities (including ASU) offer a one-semester software engineering course. Typically, such a course will cover the different phases of the life cycle, introduce some management principles, and expose the students to some of the software engineering literature [Kant81]. In addition, the course usually requires each student to participate in a one-semester project in which some of the principles learned in the classroom are applied. Such a course has proven very popular. For those interested in other areas of computer science (e.g., data base, compiler construction, operating systems) this one course satisfies their quest for knowledge. Unfortunately, others who determine that software engineering is the area in which they wish to specialize often find very few institutions which offer advanced courses in the subject [Fage81].

Software engineering education now finds itself in the same position that computer science did in the early sixties. There seems to be a strong demand for the discipline but few wish to accept it as an independent area of study. In the last few years, several proposals have emerged which describe software engineering curricula [Wass78, Hoff78, Fair78, Stuc78]. These proposed degrees generally require 10 to 15 courses which cover the phases of the life cycle, software management, software documentation, communication, and several basic computer science topics (e.g., compiler construction, data base). Most professionals agree that if such a program could be offered it would be very beneficial. However, there are several problems. One, the program usually requires two or more years of graduate work while a normal computer science degree can often be completed in one to one and a half years. Second, few universities have the faculty qualified to cover all the subjects, and if qualified faculty are available, the universities are often not willing to devote the resources needed to teach the courses effectively. (Those who have taught such classes learn that an inordinate amount of time, as compared to normal classes, must be devoted to preparation and supervision.) Third, in some universities it is very difficult to obtain a master's program in computer science much less a perceived subfield such as software engineering. Fourth, as Freeman and Wasserman have stated [Free78], the field might be too immature to be considered for a separate degree program. We are still in the process of defining boundaries and determining basic principles. Of course we have to start some time and some place and to those who pioneer this area we will owe a debt of gratitude. However, until the boundaries and principles of software engineering become commonly accepted, the area will be difficult to justify as an independent academic area of interest worthy of degree status.

The difficulty in teaching software engineering can be attributed to the following three facts. The subject area is too large to be completely covered in one semester, much less in a seminar. We do not have the resources. The subject area is not mature enough to motivate us to develop a complete master's program except at a very few universities and institutes. In order to meet the demand for software engineering instruction, we propose a limited set of courses which can cover the area in more depth than a single-semester class but which requires only one-half or one-third the faculty resources needed for a fully implemented master's program in software engineering.

The program is essentially a four-semester sequence based on the software life cycle and uses a project for unification. The project would be of medium-size (3-5 man years) and as such would be a more faithful reproduction of the real world. The experience gained from participation in a project is very beneficial. In addition to being realistic (as compared to toy programs found in most computer science classes) such projects also promote better team work. The team size (7-10 people) requires the participants to develop some organization and management structure if they are to finish on time. The multi-semester nature of the project also forces the teams to experience personnel turnover so they can learn to anticipate and account for it. In general, a four-semester course sequence allows students to explore the area of software engineering in depth while applying the knowledge gained in class

to projects that are similar in many aspects to projects they will encounter in the real world.

3.0 Description of Course Sequence

The sequence of courses follows the software life cycle. The first will be concerned with software analysis, the second with software design, the third with software testing, and the fourth with software maintenance. A single medium-size project will integrate the four courses. Each of the software engineering courses will basically apply the state-of-the-art knowledge concerning the particular life cycle phase being studied to the course project. A team approach is utilized in the development and maintenance of the project. The logistics of running these courses, including criteria for selecting projects, composition of project teams, and student evaluation, will be described later.

Course 1. Software Analysis

This course will investigate the analysis phase of the life cycle as well as teach fundamental management concepts. Of course, it is not sufficient to talk only about analysis, it is also necessary for the students to understand the overall picture (i.e., the software life cycle) and see the role of the analysis phase. Once students understand the life cycle, it is much easier to show the boundaries of software analysis and give a short definition. Included in the definition is a presentation of accumulated data showing the need for analysis. Since there is no one universally accepted analysis method, several approaches need to be presented. Through the exposure to different analysis techniques, the information needed for analysis can be generalized and different forms of representation can be taught. It should be remembered, however, that since this is a project oriented class, it is necessary that one analysis method be presented in detail. Several basic software management concepts are also taught, including the notions of a phase plan, team organizations, quality assurance planning, and reviews and reports. Other concepts included are documentation for project management (including a user's guide), change control, and resource planning. The references for the class come from books and papers on the different analysis and management techniques [Wein80, Gane79, Metz81]. A valuable source of such information is the tutorial series published by the IEEE [Free80, Rama78, Reif79].

As the material for the class is presented, teams are formed and the project initiated. During the first few weeks a phase plan is prepared showing each team's milestones for the remainder of the semester. At the same time, the different parts of the documentation required at the final review are taught in terms of documentation for management. The next several weeks are spent on a specific analysis technique to be used by the teams for the project. After that, several other techniques are presented including, SADT, PSL/PSA, SREM, and data flows. The last part of the course covers several aspects of software project management. The course outline for the current semester is shown in Table I.

Course 2. Software Design

This course will investigate different design methodologies and discuss basic concepts pertinent to good design. Included will be a discussion of the transformation or evolution of analysis information into design information. Several high-level design concepts will be introduced including architectural techniques, data design techniques, and information hiding. After the high-level design concepts have been shown, detailed design methods can be introduced. These include program design languages, Nasi-Shneiderman charts, and flow charts. Along with the design concepts, management concepts will also be presented. The references for this class can come from texts, papers, and tutorials used for teaching design [Free80, Yeh77, Jens79]. Those texts which teach a specific technique are especially needed [Your79, Jack75, Myer75]. For the management aspect of software design the material made available for the analysis course can be used.

As the design material is being taught in class, it must also be used in the project. The first few weeks of the project will be devoted to understanding the document produced by the analysis class. After that, the high-level design concepts will be applied to the problem at hand. This step will take several weeks and will lead into the detailed design process which will take the remainder of the semester. During both high- and low-level design different management concepts such as design review and quality assurance checks will be used. A suggested course outline is shown in Table II.

Course 3. Software Testing

This course will investigate the software testing phase of the software life cycle. State-of-the-art testing tools and techniques will be studied and analyzed in terms of their ability to detect errors and their cost-effectiveness. Strategies for module and integration testing of software and management approaches for test planning and control, will be covered. Measures for estimating software reliability will also be analyzed. The reference material for this course should come from several sources. Reliability and testing textbooks, testing tutorials, and published papers provide a rich selection of applicable information [Myers79, Kope80, Yeh77].

In parallel to the course lectures and discussion, the project teams continue to work on the course project. During approximately the first third of the course, the project is coded. The project teams will then test the software for the remaining time in the course. The lectures and project team efforts must be synchronized in order to maximize the learning experience. A suggested semester course outline illustrating lecture material and project team efforts is shown in Table III.

Table I. Software Analysis Outline

Weeks	Lecture Topics	Project Team Efforts
1-3	General management concepts and documentation needed for management.	Introduce and describe project. Develop semester phase plan.
4-8	Details of a specific analysis technique.	Organize team, define documentation, develop general idea of project.
9-10	Show general analysis techniques, in general.	Start putting analysis results in written form.
11-15	Management aspects of software development projects.	Finalize requirements and documentation reports. Give final review.

Table II. Software Design Outline

Weeks	Lecture Topics	Project Team Efforts
1-3	Relationship of analysis to design. Review basic management concepts.	Review analysis document.
4-8	The high-level design technique to be used for the project.	Begin design and organize team.
9-11	The low-level design technique to be used for the project.	Finish the high-level design. Start on low-level design.
12-15	Discuss other design techniques, both high- and low-level. Show how they relate to the current techniques used.	Finish design. Prepare for and present final review.

Table III. Software Testing Outline

Weeks	Lecture Topics	Project Team Efforts
1-2	Test planning and control	Code
3-5	Module and integration testing tools and techniques	. .
6-7	Advanced testing concepts	Module testing
8-9	.	Integration testing
10-14	Software reliability .	Advanced testing technique experimentation
15	Testing perspective	Acceptance testing

Course 4. Software Maintenance

This course will investigate the software maintenance phase of the software life cycle. The definition and dimensions of software maintenance will be explored. The maintenance process will be modelled and management approaches to software change will be studied. Maintenance tools and techniques including those for ripple-effect analysis and regression testing, will also be described. The reference material for this course should come from several sources. Recent published papers, tutorials, and texts should provide a rich selection of applicable information [McC181, Lien78].

In parallel to the course lectures and discussion, the project teams will perform both planned and unplanned maintenance activities on the completed software. The planned maintenance activities will consist of fixing known and documented errors as well as implementing modifications to add new program capabilities, delete obsolete capabilities, and improve program performance. The unplanned maintenance activities will consist of correction of new software errors detected after the course has begun. Most of the program modifications will involve team members in parallel efforts to implement several changes. The lectures and project team efforts must be synchronized in order to maximize the learning experience. A suggested semester course outline illustrating lecture material and project team efforts is shown in Table IV.

Table IV. Software Maintenance Outline

Weeks	Lecture Topics	Project Team Efforts
1	Definition and dimensions of maintenance	Familiarization with software and planned changes
2-3	Maintenance modelling	.
4-5	Software Change Control	.
6-7	Ripple effect analysis	Formulate maintenance plan
8-9	Regression testing .	Generate detailed maintenance proposals
10-11	Other maintenance tools	Implement software changes
12	.	Ripple effect analysis
13	Advanced maintenance topics	.
14-15	.	Regression testing

4.0 Course Logistics

One of the most important ingredients for a successful software engineering class is the project. There are two primary ways projects can be chosen. One method asks outside industry to submit projects for which they are willing to act as users. Such an approach allows the project to be much more realistic and develops a better relationship between industry and the university [Buse79]. It does have drawbacks because the instructor does not have total control over the project. Diligence is required to ensure the proper participation of both the user and the students. The other approach is to develop a project in-house. That is, the instructor becomes both user and instructor. This allows him to retain complete control over both the concepts taught in class and the experience gained in the project. However, such a situation requires the instructor to wear two hats. Without care the project could become a class of slaves. There are good reasons for both types of projects. At ASU we have initially chosen to have in-house projects in order to allow us to determine how best to organize the four courses. After we have gained more experience, we hope to be able to go to industry for projects.

There are other aspects of a project one must consider before making a choice. Of primary importance is project size. We have observed that many projects chosen for project oriented classes are too big. It is difficult to judge a project's size but our experience has shown that those projects often judged as being a little too small initially, often seem to be of about the right size later. We have chosen a problem which can be made smaller as time goes on. Our teams are currently working on an entity-relationship data base system which will be used as a foundation for later research into an integrated software development system. Originally, the data base system was planned to provide for the standard data base functions of query, retrieval, modification, creation, deletion, etc. In addition, we had hoped that security and archival functions could be added. As we progressed into the project we discovered that due to time constraints, the latter two sets of functions are not feasible and have therefore eliminated them.

In order to make the projects more realistic, we need to better model the outside world. One area in which most computer science graduates need more experience is team project development. This is available in one-semester software engineering classes, but the team sizes are usually small

enough (3-5 people) to allow them to function without serious communication problems. Larger teams enable the participants to experience problems not seen in one-semester projects. Our teams currently consist of seven people. In order to function smoothly, they have utilized concepts such as team leaders (rotated), agendas, regular meeting, subcommittees, milestones, etc. All of these concepts seem commonplace and the standard student response is "I know all about that." However, our students have indicated that this project is the first time they have had to use the concepts. Although they were taught many of the concepts in the one-semester class, the team size was so small that ad hoc and impromptu methods were used throughout the project. It was easier to use whatever method seemed right at the moment, rather than to learn how to consistently apply concepts learned in class. With larger projects and larger teams, such unorganized methods became counterproductive, forcing them to implement standard management procedures and team organizations.

One of the most difficult parts of any team oriented project is the evaluation of students for grading purposes. We are using several criteria for student evaluation. A test, usually a take home test, is used to determine if the students have been learning basic concepts and doing the reading. This test counts for twenty-five percent of the grade. There is also a required term paper which accounts for twenty percent of the grade. The paper is usually a survey of the current state of the art in the subject being studied. This semester, corresponding to Course 1, the paper will survey analysis and requirements specification. Point-wise, the most important piece of work for student evaluation is the final review. This review consists of evaluating all the final documentation as well as a one-hour oral presentation given by one of the team members. The final review is worth thirty percent of each student's grade. Each team receives one grade for all members. There is no attempt to determine who worked harder than anyone else. This is done for two reasons: it encourages the team to work together in order to produce a good final report, and there is another method for evaluating individual performance within each group. Each team member is evaluated by his peers and by the instructor. Ten percent of a person's grade is determined by peer review. Each time a team leader is finished with his turn as leader he produces a written report indicating how well each member performed his share of the work load. The collection of these reports is used to judge a student's performance for the semester. Each student is also subjectively evaluated by the instructor. This evaluation is worth another ten percent of his grade and is done through weekly visits to each team to determine the participation and quality of work by each member. The last five percent of the grade is allocated to class participation. There are several topics for which there is no single correct answer and class participation is deemed necessary for proper instruction in these areas.

5.0 Course Sequence Advantages

The sequence of software engineering courses described in this paper has a number of potential advantages for faculty, students, and the research community. For students, the sequence provides an opportunity to participate in a medium-scale software project at all or any selected phase of development or maintenance. This provides for knowledge of the entire software life cycle and is comparable to the experience of rotating positions in an industrial environment. If a student is only interested in a particular phase of the software life cycle, his professional development needs can be satisfied by participating in the appropriate course in the sequence. The relevance and effectiveness of the tools and techniques utilized in each course can be witnessed and evaluated by the students participating in subsequent courses. This can be a significant factor in motivating students completing the sequence to transfer their experience in the course sequence to their work environment.

The sequence of courses also helps to develop the interpersonal skills of its participants. Project team members must learn effective verbal and nonverbal communication skills as well as organizational skills which support the cooperative atmosphere necessary for the team to survive and perform its work efficiently and effectively. Leadership qualities necessary to successfully delegate work and evaluate subordinates are also developed. Due to the logistics of the courses, a student must develop a sense of responsibility for his share of the project and refine his skills at scheduling and allocating time to perform tasks.

The sequence of software engineering courses has a number of potential advantages for faculty administering the course. Due to the nature of the projects and the logistics of the courses, teaching the classes can be a significant educational experience in terms of project management. The courses also provide a vehicle for the development of software designed by the faculty. The courses can also serve as a research vehicle for the faculty or Ph.D. students who wish to perform software engineering experiments. One of the most significant factors hindering software engineering research is the capability of conducting controlled experiments on realistic projects. This sequence of courses should provide an ideal test bed for these types of experiments. Thus, the sequence of software engineering courses described in this paper can potentially be advantageous to the software engineering community.

References

Fage81 Fagenbaum, J., "A New Breed: The Software Engineer," IEEE Spectrum 18, 9 (September 1981), 62-66.

Fair78 Fairley, R. E., "Educational Issues in Software Engineering," Proc. ACM 1978 Annu. Conf., 58-62.

Free78 Freeman, P., "A Proposed Curriculum for Software Engineering Education," Proc. 3rd Inter. Conf. on Soft. Eng., 56-62.

Free80 Freeman, P., and Wasserman, A. I., Tutorial on Software Design Techniques, IEEE Computer Society, New York, N.Y., 1980.

Gane79 Gane, C., and Sarson, T. Structured System Analysis: Tools and Techniques, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1979.

Hoff78 Hoffman, A. A. J., "A Proposed Master's Degree in Software Engineering," Proc. ACM 1978 Annu. Conf., 54-61.

Jack75 Jackson, M. A., Principles of Program Design, Academic Press, New York, N.Y., 1975.

Jens79 Jensen, R. W., and Tonies, C. C., Software Engineering, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1979.

Kant81 Kant, E., "A Semester Course in Software Engineering," Software Eng. Notes 6, 4 (August 1981), 52-76.

Kope80 Kopetz, H., Software Reliability, Springer-Verlag, 1980.

Lien78 Lientz, B., Swanson, E., and Tompkins, G., "Characteristics of Application Software Maintenance," CACM 21, No. 6 (June 1978)

McC181 McClure, C. L., Managing Software Development and Maintenance, Van Nostrand Reinhold Company, 1981.

Metz81 Metzger, P. W., Managing a Programming Project, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1981.

Myer75 Myers, G. J., Reliable Software Through Composite Design, Petrocelli/Charter, New York, N.Y., 1975.

Myer79 Myers, G. J., The Art of Software Testing, John Wiley and Sons, 1979.

Rama78 Ramamoorthy, C. V., and Yeh, R. T., Tutorial: Software Methodology, IEEE Computer Society, New York, N.Y., 1978.

Reif79 Reifer, D. J., Tutorial Software Management, IEEE Computer Society, New York, N.Y., 1979.

Stuc78 Stucki, L. G., and Peters, L. J., "A Software Engineering Graduate Curriculum," Proc. ACM 1978 Annu. Conf., 63-67.

Wein80 Weinberg, V., Structured Analysis, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1980.

Yeh77 Yeh, R. T., Current Trends in Programming Methodology, Vol. I, Software Specification and Design, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1977.

Yeh77 Yeh, R. T., Current Trends in Programming Methodology, Vol. II, Program Validation, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1977.

Your79 Yourden, E., and Constantine, L. L., Structured Design, Fundamentals of a Discipline of Computer Program and Systems Design, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1979.