Int. J. Engng Ed. Vol. 11, Nos 4 and 5, pp. 277–283, 1995
Printed in Great Britain.

0949–149X/91 $3.00+0.00
© 1995 TEMPUS Publications.

Undergraduate Software Engineering Laboratory Experiences

WILLIAM LIVELY
MARK LEASE
Department of Computer Science, Texas A&M University, College Station, TX 77843–3112, USA

Texas A&M University's Undergraduate Laboratory for Software Engineering has been funded by the National Science Foundation. It has been created to provide computer science and computer engineering students with hands-on experience with the tools and techniques used in modern software engineering. This paper will overview the laboratory exercises and projects assigned to the students. The successes and failures will be discussed with insights on how to improve such a software engineering laboratory. Overall we will attempt to partially answer the question: how do you infuse structure and technology into the teaching of practical software engineering?

EDUCATIONAL SUMMARY

1. The paper describes new training tools or laboratory concepts/instrumentation/experiments in teaching software system design.
2. The paper describes new equipment useful in an upper-level software engineering course.
3. The students involved in the use of the equipment are mostly seniors with some graduate students.
4. New aspects include using computer-assisted software engineering (CASE) tools to teach conceputal modeling in software engineering.
5. How is the material presented to be incorporated in engineering teaching? The material serves as an example of how practical, hands-on experience in software design can be added to a software engineering course.
6. The text accompanying the presented materials is Rumbaugh *et al.*, *Object-Oriented Modeling and Design*, Prentice Hall.
7. The concepts presented have been tested in the classroom. The experience gained from the laboratory made the students more aware of the problems inherent in designing software.

INTRODUCTION

THE PROBLEM we are attacking is the education of students that allows them to be able to build better software systems and be more productive. Why is this important?

In 1968 at the NATO Conference at the Rome Air Force Base in New York, a group of software users/developers met to consider what would later be called the 'software crisis'. The problems addressed were that the software systems being delivered were of low quality, unreliable, not usable, not understandable, not maintainable, not modular and delivered late and over budget. The term 'failure gap' more simply describes the situation—a failure of delivered software systems to meet the user's needs. The magnitude of the problem can be appreciated when we consider that over 150 billion dollars are spent on software in the US annually.

How had we arrived at this situation? Third-generation hardware became so powerful that very sophisticated large-scale software systems were now envisioned and being built that were previously beyond second-generation hardware capabilities. One of the major problems among many was scale-up. We were moving from programs to large software systems and the approach used previously simply did not scale-up. The previous approach was to simply start programming and finally through exhaustive testing realize the desired software system. This approach was not feasible for large software systems.

A term was coined at the NATO conference that would identify the approach to dealing with the software crisis. This term is 'software engineering' and promised to offer a structured engineering approach to the development of software. Today, over twenty years later, we are still struggling with the software crisis/failure gap.

Brooks [1] in his famous paper, known as the 'Silver Bullet' paper, described why software is different from many other engineered systems and consequently why the software crisis persists. Complexity, conformity, changeability and invisibility are the main culprits. Software cannot be visualized like bridges and many other hardware systems. The complexity is enormous because so many states can be represented in software and software is malleable—so easily changed. These problems exacerbate what Brooks calls the essence of software development—determining the conceptual constructs for the system—the hard part.

Brooks considers the implementation to be the easy part.

APPROACHES TO SOLUTIONS

Over the years a number of efforts have attempted to address the software crisis problem. To set the context for the following discussion, we will briefly define the concepts of software process model (SPM) and software development methodology (SDM).

A SPM describes the steps/phases to be taken in developing software and defines the criteria for beginning and terminating each step. Such a model is 'descriptive' in nature because it describes 'what' should be done, not 'how' to do it. The most famous SPM is the waterfall model [2], consisting of the following steps/phases:

- requirements capture;
- specifications (high-level design/conceptual modeling);
- design;
- implementation (coding/programming) and testing;
- installation; and
- operation and maintenance.

Other SPMs have been introduced, such as the spiral model, the fountain model, exploratory, and incremental. But for our purposes the steps of the waterfall model describe the types of activities involved in software development.

A SDM attempts to describe 'how' (to be prescriptive) we accomplish various activities that are necessary to accomplish the task of each phase of a SPM. What we have seen for many years is that structured analysis (SA) has become the mode of choice for many developers for the analysis phase (an alternative term for the first two phases—requirements capture and specifications). SA uses the concepts of dataflow diagrams, data dictionaries and data structure charts to develop the specifications for a software system.

Major national efforts have been undertaken by the US Department of Defense, US industries and universities to develop software technology to attack the various problems. The DoD has taken the lead with the STARS (Software Technology for Adaptable and Reliable Software) initiative. STARS takes a two-pronged approach to the software crisis: a conventional approach and a non-conventional approach. The conventional approach follows our present path of development (waterfall model) with hopes of improving the various stages. The non-conventional approach strives to dramatically enhance productivity with knowledge-based systems and automatic programming. The two-pronged approach requires heavy investment in the conventional approach and the long term (15–20 years) to develop a non-conventional approach. A major thrust of STARS has been the Software Engineering Institute (SEI) at Carnegie Mellon University. The thrusts of the efforts can be stated simply—how can we train people to be more productive and build higher-quality software? SEI has concentrated more on the conventional approach.

In the 1980s MCC (Microelectronics and Computer Corporation), a large consortium of companies trying to deal with major technological problems, had a Software Technology Program (STP) attacking the software crisis. Most of the efforts here were on what they called the 'upstream' problems—capturing the requirements and producing specifications (high-level design). The thrust of the work stemmed from the fact that errors discovered earlier in the development process are much easier to fix than when found during and after the implementation phase.

The Knowledge Base Software Assistant (KBSA) effort at Rome Air Force Base attempts to use knowledge-based/artificial intelligence (AI) methods to assist developers through automation of various tasks that occur during software development. The thrust here is that productivity can be enhanced through automation (having the computer perform tasks previously performed by humans). This effort tends to follow the non-conventional approach of STARS.

The Software Productivity Consortium (SPC) has been involved mainly in the Ada effort to support the development of software through software development environments and hence tends to follow the conventional approach to improvement for software development.

TEXAS A&M'S APPROACH

So the question is: how do we teach our students to be able to produce higher-quality software and be more productive? The approach the Texas A&M Department of Computer Science has taken follows somewhat the concept of addressing 'upstream' problems that MCC is attacking. The key concept here is to obtain the requirements for the user's system. Determining the requirements is a major problem because the users frequently cannot tell the developer what they want. Either they do not know what they wants or they cannot communicate the requirements to the developer. English language communication is laddened with ambiguity and difficult in almost every context. The terminology in software engineering is not standardized, and this complicates the problem even more. The developer's lack of application domain knowledge, and the users' lack of software development domain knowledge simply exacerbate the dilemma.

So we try to find a way to enhance user/developer communication. Our approach has been to emphasize conceptual modeling (attacking the requirements capture and specification phases) as a means for enhancing user/developer communication. Can the developer with the right combination of graphics, structured text and natural language really communicate with the user?

The answer is no! What really must be taught (along with the use of conceptual modeling tools) is the *discovery process*. The discovery process deals with defining exactly what the problem is. We try to teach the students to become *aware* of this and a number of other significant problems. One of the major ones is the difficulty of the discovery process. We try to teach students that patience, experience, and iteration are important aspects of the discovery process. How do you mine the necessary knowledge to determine exactly what the problem is? We try to teach students an analyst's approach with a series of questions that allow convergence to the true requirements.

MODELING APPROACHES

As stated earlier, SA has become the mode of choice for many developers to help with conceptual modeling. A major problem with this approach is that the software architecture is built around functions. When maintenance occurs, typically functions are changed or added, and the software architecture begins to be disrupted. This hampers future maintenance.

The advent of a new methodology called an object-oriented (OO) approach offers a possible solution to software architecture degradation. An OO approach bases the software architecture on objects instead of functions, and therefore when maintenance occurs and the functionality changes, the basic software architecture determined by the objects is much less disrupted.

Therefore our approach is to attempt to teach students how to do conceptual modeling with an OO flavor. The most promising methodology along these lines has been the work of Rumbaugh at General Electric [3]. The technique is called OMT (object modeling technique) and consists of building three models: an object model, a functional model (like SA) and a dynamic model (control and time-varying properties). Using objects to base the system on is a very powerful technique. In the real world systems are made of objects, with attributes, operations and associations. So the OO modeling notation maps directly to the real world. The concept of objects carries across the problem space, solution space and implementation space in a seamless fashion—the notation of objects remains constant.

SOFTWARE ENGINEERING LABORATORY

At the beginning of 1991, the Department of Computer Science at Texas A&M University was awarded an Instrumentation and Laboratory Improvement (ILI) grant from the National Science Foundation for the development of an undergraduate Software Engineering Laboratory. The result of this grant has been the creation of a laboratory containing 20 state-of-the-art workstations running the best computer-assisted software engineering (CASE) tools available today to teach our students how to perform conceptual modeling.

The conceptual modeling effort is most effective when it can be done in a rapid prototyping mode. Rapid prototyping allows developers to rapidly build operational (computer-executable) conceptual models that the users can experiment with. Such user/developer interaction greatly enhances communication between the parties. CASE tools are one way to facilitate rapid prototyping.

The CASE tool we had initially been using is Interactive Development Environment's (IDE) StP (Software through Pictures). The major limitation of this CASE tool is that it is based upon a SA approach. StP does have entity-relationship (E-R) diagrams, dataflow diagrams, state diagrams, structure chart capability and data structure diagram capability. Therefore we could approach some of the concepts in an OO paradigm, but the approach clearly was not seamless.

By that we mean there is not a common notation that integrates across all models and phases.

In the Spring of 1993 we added the CASE tool OMTool to the laboratory. This tool was developed by Rumbaugh and his team at General Electric. With this addition the students could now truly begin to perform some integrated OO conceptual modeling.

We hope that the exercises that our students will perform will begin to give them some real-world experience. Boehm [4] has stated that the most important issue in building sophisticated software systems is the experience of the developers.

LABORATORY EXPERIENCE

The goal of the Software Engineering laboratory is to provide students with several types of experience.

'Real world' experience

Most of the problem-solving challenges presented by college courses are oriented to demonstrating very limited principles, such as the features of a particular programming language or the properties of a specific algorithm. Therefore a major goal of the Software Engineering Laboratory assignments is to give the students experience in solving the types of problems which might be presented to them by future employers.

Exposure to a CASE tool

To catch errors as early in the system-development process as possible, modern software engineering is placing a greater emphasis on the 'up stream' of the process, the initial development of the requirements and specifications for the system. CASE is widely perceived to be a useful tool for capturing the specifications for a system. Many companies in industry are integrating CASE into their software-development environment. Having exposure to CASE tools is important to software professionals entering such an environment.

Experience working in groups

Group projects are not common during the college experience, but are very common in industry. Since working in a group requires unique talents and skills, experience in this area is also important. We hope to be able to improve the communication and interpersonal skills of our students. Again this is an introduction to the discovery process. We are trying to teach the students scenarios of questions that will allow them to capture the problem definition:

1. Who needs the solution?
2. What is it that I am doing?
3. Why am I doing it?
4. What is the scope (context)?
5. How do I do it?
6. Where do I go from here?

We want the student to learn to be in a continual mode of questioning during software development.

Exposure to state-of-the-art workstations

High-speed, graphical workstations running Unix are very popular in industry. However most of our students' experience has been with personal computers. Workstation skills learned in the laboratory will be invaluable for future software engineers.

Discussion of alternative designs and implementations

Another goal of our laboratory assignments is for our students to learn that the problem of software design does not usually have only a single correct solution. This is different from most problem-solving courses students are exposed to during college. We are attempting to teach the students how to prioritize non-functional software characteristics and then perform trade-off analysis for the functional requirements and thus look at a spectrum of alternatives. Trade-off issues involve cost, effort, feasibility, ethical issues, reliability and others.

ACHIEVING THE GOALS

To achieve our goals for the course, we use both individual assignments and a group project. The individual assignments are used to make sure the students are well grounded in the principles of software design (specifically conceptual model development) so they can contribute effectively to their group project. The group project is used to demonstrate how the individual tools within StP and OMTool work together. Each tool specifies some facet of the complete software system.

Individual assignments

The first part of the semester is spent learning OMTool and the individual tools within StP: the data dictionary and the major diagramming tools. Each assignment has the student use a tool to design a small, representative segment of a software system. For each tool the student is given a detailed set of requirements for a software system. The requirements are not complete so the student can begin to learn some aspects of the discovery process. The student has to convert the system requirements into a system specification (conceptual model) using the appropriate tool.

Here is a summary of a typical semester's individual assignments:

1. Develop an object class model using OMTool for a software problem tracking and configuration management system. This exercise begins to introduce students to the newer OO techniques and also introduces them to the topic of configuration management.
2. Using dataflow diagrams, design a system for sending out the monthly billing for an electric utility company.
3. Use the data dictionary and data-structure editor to design the data structures needed to maintain a library's on-line card catalog.
4. Design the state-transition diagram for a tape recorder/player's motor controller.
5. Convert the dataflow diagram for the utility billing assignment into a structure chart.

Group project

At the end of the semester, the class is divided into groups of three or four students for a group project. The students decide among themselves who will be in each group.

This past semester, the project was to design a library information system—including an on-line card catalog, and inventory and patron information subsystems. The description of the assignment included a fairly detailed set of requirements for the system. Each group was to design the specifications for the system and submit the data dictionary, data-structure charts, dataflow diagrams, and structure charts for the design.

The group project lasted one month. Midway through the project, each group met with the instructor for a 'preliminary design review'. The purpose of the design review was to make sure each group was proceeding correctly and making good progress. At the design review, each group was to have substantially completed everything but the structure charts. Such reviews are invaluable in industry to obtain multiple perspectives on software system design. It also provides a means of emphasizing the importance of developing good oral communication skills.

E-mail

Electronic mail is used extensively to answer questions about assignments and to distribute information about the course. This then becomes an important tool of the discovery process. The email was done to supplement (rather than replace) Q&A during the laboratory session and the instructor's office hours.

At the beginning of the semester, a mailing list for the entire class is created. When a student has a question about the course, he/she can send a mail message to the laboratory instructor. Using the mailing list, the instructor will reply to the entire class when appropriate. Anything identifying the originator of the question is edited out to preserve the originator's confidentiality. Using e-mail was very successful—this past semester over 75 mail messages were distributed to the class mailing list. Through the e-mail dialog, many students became aware of problems that had not even occurred to them. This just reinforces the importance of the discovery process.

EMPIRICAL INSIGHT/PROBLEMS

After conducting the course this past semester, we noticed several problems and gained some insight in the process.

Lack of domain knowledge

Certainly the most limiting constraint on the type of assignments which could be used in the laboratory was the lack of a common body of domain (or application) knowledge shared by all of the students. Our computer science students are free to choose their minor area of study from any college at the University—therefore every class has students with interests in business, engineering and liberal arts. This diversity limits the applications suitable as the subject of a laboratory assignment to the most everyday, mundane areas: tape recorders, libraries, electric utility bills, etc. Even so, there were many questions about the application in general (e.g. why should there be multiple 'locations' for a single book title?) and the laboratory assignment's system in particular (e.g. why does the library patron's record need to be changed when a book is returned?). On a small scale, this lack of domain knowledge mirrors what is occurring in industry on a large scale. This again emphasizes the importance of the discovery process.

Unfamiliar hardware/operating system

Roughly half of the students start the course never having used a workstation or Unix before. These students are at a disadvantage, especially when something goes wrong and the system starts generating cryptic error messages (as Unix systems tend to do). In a distributed environment, a novice user will often not realize that the problem resides in the network or with one of the file-servers, and the user will think they have done something wrong.

The sum of the learning curves for using new hardware, a new operating system, and a new windowing environment (with a laboratory assignment due in 2 weeks) can be quite intimidating to almost anyone. To get the students started, the instructor will hold 'getting started' classes with 2–6 students at a time outside of the normal laboratory period for anyone who feels they want help. The instructor will make sure that each students' computer account is correctly setup and will demonstrate the basics of using the workstation. For the instructor, an important advantage of these 'getting started' classes is that they are an opportunity to meet many of the students early.

The CASE tool we are using operates in the X-window system. X allows a number of activities to be operating and visible concurrently—a new experience to most users only familiar with personal computers. Other novice users expect the highly-integrated window and application system of a Macintosh or Windows text editor. Many users have found out the hard way that X will allow you to shut down the window environment without regard to whether the files currently being edited have been saved or not.

Group dynamics

In general, students are not used to working in groups. Many groups have trouble getting organized initially, and most groups never get the workload equitably divided. There never can be too much emphasis on the importance of communication between members of the group. We are placing a greater emphasis on the students becoming aware of the importance of oral/written communication. The students can work on these skills during the project development time and then demonstrate the development of these skills in the oral and written presentations at the preliminary design review and the conclusion of the project.

Scale of the group project

Quite a few students were disturbed by the scale of the group project. They seemed to lack experience working with (or even thinking about) multiple programs interacting through use of common files or databases. In reality the scale we work on is still significantly smaller than many projects in industry. We want the students to be aware of the *scale-up* problem. As you try to apply the techniques and tools we teach to larger problems, they do not scale up linearly. We want students to be aware of this problem and realize efforts need to be taken to deal with increased complexity of development.

Different solutions

During laboratory sessions, we compared and contrasted different solutions generated by the students. Many times there was more than one correct solution to an assignment. However, quite often the differences in the solutions were due to differences in how the students perceived or weighted the design requirements. Quite often the perceptions were caused by incompleteness or ambiguity in the system's requirements. Again attempts are made to demonstrate trade-off analysis.

Importance of documentation

We attempt to teach our students the importance of documentation. The need to capture the corporate history and artifacts that are produced during the development of software. They need to know how important the capture of design rationale is for future development and maintenance efforts. At this time they are introduced to the concept of a Life Cycle Artifact Manager (LCAM). The LCAM is a hypertext-based system for linking the artifacts and allowing traceability forward and backwards across the phases of development.

Interest of potential employers

Since software engineering is a senior-level course, many of the students are interviewing for jobs as they are taking the course. Several students have commented on the real interest shown by company recruiters when they mention during an interview that they are learning about CASE tools.

Student reaction

Our students' reaction to the Software Engineering Laboratory has been very favorable (and gratifying). 'This course should be required' is a common remark. Many students have said they were glad to get hands-on experience designing software. There have been three common complaints: the project takes too much time; the laboratory should be more directly object oriented; and some students would rather work on the personal computers they are familiar with. This last complaint is often heard early in the semester.

HOW TO IMPROVE?

Reflecting back on the past couple of semesters, we believe we can make a couple of changes to improve the course.

Give more time for the semester project

The semester project was very pressed for time, and since it was at the end of the semester, it competed for attention with many other activities. Next semester one or two of the simpler assignments will only be given one week for completion to allow more time for the project. Our computer engineering students take a two-semester senior design course. We see this course as providing valuable opportunities to apply the tools and techniques learned in the Software Engineering Laboratory. For our computer science students, they have the opportunity to take a senior-level design project course and again apply the knowledge they have gained.

Expose the class to a complete specification earlier

One problem we noticed early was that a number of students understood the individual tools in the CASE environment but did not understand how the tools interrelated and/or did not understand how the CASE specifications fed the implementation process. To solve these problems, the lectures for the laboratory were changed to emphasize the role the various CASE system tools played in the specification process. Also we developed for a small system a start-to-finish set of documents—including the system requirements, specifications and source code. The example system, a computerized cash register implemented as an X-window system client, is small enough to be wholly comprehensible to the students after a couple of hours of study—but the students are able to learn how the various specification diagrams complement each other and provide different views of the example system. The source code for the example system demonstrates how the specifications are implemented. Since the example system is an X-window system client, the students are exposed to an event-driven control structure.

CONCLUSION

We feel our approach will make a viable contribution to the education of computer science and computer engineering students. The students are beginning to obtain real-world experience in conceptual modeling for software systems. They are learning to work together in groups with one another. They are beginning to see the problems that occur in developing large software systems: communications, domain knowledge, experience, reuse and others. *Awareness* of the significant issues in developing software is one of the major thrusts of our efforts.

We see this as a learning experience for us that will continue for a long time to come. We will continually be gaining insight on how better to design software systems for the foreseeable future. The building of large, complex software systems is so difficult that it is likely we will never arrive at a totally satisfactory solution.

Another plan for the future is to attempt to obtain larger case studies from industry for our students to review and evaluate. Well-documented case studies of successes and failure can bring forth yet another experiential component of learning the difficulties associated with developing large and complex software systems.

Acknowledgements—The Undergraduate Software Engineering Laboratory at Texas A&M University was funded by the NSF-ILI Program (grant no. CDA-9051378) and the State of Texas.

REFERENCES

1. F. P. Brooks, No silver bullet—essence and accidents of software engineering, *IEEE Comput.*, **4**, 10–19 (1987).
2. B. W. Boehm, A spiral model of software development and enhancement, *IEEE Comput.*, **5**, 61–71 (1988).
3. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*,Prentice Hall, Englewood Cliffs, NJ (1991).
4. B. W. Boehm, Improving software productivity, *IEEE Comput.*, **9**, 43–57 (1987).

William Lively received his Ph.D. in electrical engineering/computer science from Southern Methodist University in 1971. He is currently an associate professor of Computer Science and Director of the Laboratory for Software Research at Texas A&M University. He is the University's representative to the Software Engineering Institute and his laboratory serves as an alpha test site for the Knowledge Based Software Consortium RADC. His current research interests include developing direct manipulation user interface management systems, life cycle artifact managers to capture total software development information through a hierarchical hypertext system and conceptual design tools. He is a member of IEEE, ACM and the Computer Society.

Mark Lease received a BS in electrical engineering (1980) and BS in computer science (1987) from Texas A&M University. He is currently working towards a Ph.D. in computer science. His research interests are software engineering and hypertext.