Teaching Software Engineering: A Critical Path Method

Osama Shata

***Abstract*—Software Engineering is a pivot course in the Computer Science and Computer Engineering curriculums. This paper briefly presents traditional content which is usually being taught in many Software Engineering courses, and highlights some problems encountered during teaching this content. Then it builds on those problems to suggest a more appropriate content for the course. The suggested content is applied in nature to make the course interesting to students, pushes programming to the very beginning of the course so that students may have hands-on practice for a longer time, removes many topics which are usually found in traditional Software Engineering courses such as the structured approach, cuts on the number of diagrams that can be replaced by others to reduce confusion. The paper will consider the trade of between the material which was cut and the benefits gained from providing the students with a focused curriculum that may limit the difficulty and the confusion usually result from teaching a traditional curriculum, this includes limitations and implications of our approach and the expected gains.**

***Index Terms*—Software engineering, curriculum design, object-oriented, structured approach, JAVA.**

I. Introduction

Software Engineering (SE) is usually a core course in the Computer Science (CS) and Computer Engineering (CE) curriculums. While teaching the course of software engineering for many offerings, we have tried various approaches dealing with contents and teaching strategies, and which may resemble what many other colleagues would be using. Mainly using traditional lectures with traditional content and giving a course project to help students practice concepts taught in lectures. But we found that this traditional content has led to some confusion among the students. Based on the lessons I have learned from teaching each offering we have made improvements to the contents and our teaching strategy until we ended up with the model described in this paper.

This paper is organized as follows: section 2 presents a brief review of related literature, section 3 describes the problem and research framework. It lists what we believe to be some common topics that many instructors would typically include as part of their software engineering courses, and which we believe to be non-essential, and highlights and discusses the negatives of having those (non-essential) topics if became part of course contents. Section 4 presents and discusses our innovative model for teaching the software engineering course and which we have developed and adopted based on the experience we have gained from teaching this course for many times. Section 5 is a conclusion.

Manuscript received February 14, 2012; revised April 12, 2012.
Osama Shata is with Qatar University (e-mail: sosama@qu.edu.qa).

We will introduce next some of the work that has already been done by other researchers.

II. Review of Related Literature

Researchers have investigated methods, approaches, models … etc. related to teaching SE. Decker and Hirshfield [1] have argued for the need to use the Object-oriented paradigm. Brereton et al. [2] has noted benefits of teaching Software Engineering using collaborative and group projects. Ludewig and Reißing [3] have investigated and highlighted the importance of using application-oriented problems in teaching SE to make it more practical. Culwin [4] has emphasized the need to use JAVA as the implementation language since students are more likely to use it in their work places. Using simulation for teaching SE has also been considered [5]. Mann and Smith have examined tools that may be used for SE projects in SE courses, and have also considered various suggested approaches to teaching SE courses [6, 7, 8]. The need to make shifting in teaching SE to cope with the, newly welcomed, Agile software development model and the industrial practice in SE have been discussed by Noble et al. [9].

Our work presents our model to teaching a SE course for CS and CE undergraduate students. We encourage using the Object-Oriented approach with the Agile and Spiral models. We will introduce next some problems encountered during teaching the course, followed by our suggested model.

III. Hypothesis and Research Framework

A. Hypothesis

The work in this paper is based on the hypothesis that what students learn in colleges in SE courses is not adequate to prepare them for software development in industry and that several problems may be identified in the current content which we teach to students. Our approach was to revise the common content of SE courses and come up with a non-traditional content by identifying problems areas in the current curriculum, suggesting how to solve these problem areas, cutting the unneeded topics to give more time for practicing the concepts taught, and to focus on presenting various case studies using a limited number of software development models but which students are more likely to use in their future work places. We will begin by giving a definition for the field followed by the problems we have identified.

Attempts to find a consensus definition for the field have not proven very successful. The some use it to mean programming, others use it to mean software development; but the majority use it to refer to the classical definition given

by the IEEE Computer Society's Software Engineering Body of Knowledge definition [10]: "as the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, and the study of these approaches; that is, the application of engineering to software". It is the latter definition that we will conform to and mean in our discussion in this paper.

We believe that most problems instructors face in teaching a course in Software Engineering (SE) stem from a main problem which is not knowing what to teach. The IEEE definition of SE highlights three main activities: systematic process, development, operation and maintenance. Hence, it is expected that each SE course must address these three activities, and those are the main source of problems in the course.

A typical Software Engineering course would teach the following topics :

- Software Engineering Overview / Introduction
- Software Engineering Process
- Requirements Analysis
- Software Design
- Other Topics (e.g. software construction, software testing, software engineering tools and methods, project management, usability guidelines, software quality, …etc.)

While most instructors may agree on the above broad titles, many of them would find difficulty in deciding on the detailed contents of them. The first four topics in the list above are the ones of main concern to this paper. We will introduce and discuss the problems in each of those topics in the next section.

B. Problems related to the overview / introduction

This topic usually introduces the software life cycle (development process) and common models. Resources list and describe many models (e.g. Waterfall, V-model, Prototype, Iterative, Incremental, Spiral, Agile, RUP, …etc.) [15]-[17]. Some of those models have various variants that may increase the number of the models (e.g. Crystal, Extreme Programming (XP), and Scrum for the Agile model). Instructors may find it a dilemma to decide on how many and which models to use and teach in the course. There is no consensus among software engineering practitioners about the best model(s). Everyone promotes their own methods, claiming huge benefits in productivity, usually not backed up by any scientific, unbiased evidence.

The question now is "Which model should we choose to teach to our students?" Examining contents of many software courses and discussions with colleagues indicated that most of them teach most of the models listed above with emphasis to students that they should choose the right model based on the scope and type of the software project and depending on other factors such as budget and available resources. In our opinion this is not the right approach because teaching students many models would confuse them. We suggest teaching only the spiral and agile models. Since software engineering is meant to provide a systematic approach for developing large and complicated systems and the spiral is intended for large, expensive and complicated projects. Then it would be suitable for that purpose. In addition, it combines features of the waterfall and prototype which are popular but without their disadvantages. It is also incremental and iterative. This makes the spiral model like a general or a generic one where many other models may be considered as special cases of it. Moreover, the spiral model is much simpler than the RUP. Simplicity is very important in practice. For example, in the area of databases, the relational model is popular and is widely used mainly because of its simplicity, although it is less efficient if compared to the network model. However, the latter is more complex. Also, the relational model is simpler than the newer object-oriented model which is still less popular than the former one.

We also teach the Agile model in general, and the Extreme Programming in specific, because of its clear advantages. For example, it is very close to what programmers and students tend to do, they put the programming at the very beginning of the project and delay documentation (i.e. matches people nature). Students usually come very enthusiastic to begin working practically and program. In addition to other advantages such as the continuous involvement of customers, and realizing that requirements can come up during any time throughout the project lifecycle.

We will introduce next some of the problems related to teaching the analysis phase of the software life cycle.

C. Problems related to analysis and design

Teaching the analysis phase of the software life cycle involves teaching the structured and object-oriented approaches. We find that teaching both approaches leads to confusing students. Although the definition of each approach highlights concrete differences between both approaches, but the differences between some of the tools which are being used by each approach may not be appreciated or clearly realized, the matter which leads to some confusion among students.

Structured Analysis separate between data and processes while object-oriented analysis combines both. When it comes to the modeling tools, we find that the structured analysis approach uses modeling techniques such as data flow diagrams, structure diagrams, state models, entity-relationship diagrams and task diagrams meanwhile the object-oriented approach depends mainly on the unified modeling language (UML) with its nine diagrams (use-case, object, class, sequence, collaboration, state chart, activity, package and component). Students compare between the tools of each approach and find similarities between some of them. For example, consider the entity-relation diagram (E-R) of the structured approach and the object diagram of the object-oriented approach (OO). Both of them show how specific instances of a class are linked to each other, with the fact that the E-R diagram also shows attributes of entities and provides a static view while the object diagram provides a dynamic view. Also, consider the E-R diagram with the class diagram. The class diagram may be considered as a generalization or enhancement to the E-R diagram since it also shows operations and methods. Both provide static view. Also consider the state models in the structured approach and the state chart in the OO approach. In the structured analysis approach, state models show the modes in a system, and in the

OO approach a state chart is used to describe the states of a complex object and Addresses the dynamic view of the system. Similar comparisons may be applied between the system diagram and the component diagram; or between the flow charts and dataflow diagrams and the activity diagram; or between the task diagram and the activity diagram; or between the task model and the component diagram. When teaching students all these different diagrams they may not appreciate the fine differences between them and rather focus on the considerable similarities. The matter that makes them question the validity of the differences in the definitions of the two approaches if considerable similarities may be located between their modeling tools. It is our opinion that since the OO approach is now the common modeling approach for software development then it is better to focus on it and do not discuss the structured approach. Some students even face problems in deciding on the right diagram to use among those of the UML ones. For example, between objects diagrams and class diagrams; and between collaboration diagrams and sequence diagrams.

We encourage focusing on the OO approach and the UML diagrams and do not teach the structured approach or its tools. Although students are exposed to the E-R diagram in the database courses and still question its similarities to the object and class diagrams and why it is not used instead.

It is our opinion that many of the traditional topics that we used to teach in the software engineering course can be omitted and left for advanced or postgraduate courses, and that we have to be closer to industry and its standards. The design may be considered as an elaboration of the analysis, and hence, a similar discussion for that of the analysis may be applied to the problems related to design.

We will introduce next our suggested approach.

IV. A Suggested Approach and Discussion

In this section we present our approach to teaching the course of software engineering. Teaching software engineering is not an easy task. An instructor must select between various life cycle models and between various approaches for the phases of the life cycle. In addition, the instructor must find ways to give students practical experience in engineering the development of the software. This also is not an easy task when the course does not have a separate or an embedded lab, so this practical experience must be acquired through teaching in class room sessions. We introduce next our approach for teaching the SE course over 42 hours of class room sessions spread over a semester with 3 hours per week for 14 weeks. Table 1 below lists the main topics which we believe that a course syllabus must contain.

TABLE 1: A Suggested Course Syllabus

Week	Topics / Activities
1	Introduction (which includes a definition for the field proposed by either the Canadian Standards Association or the IEEE [10] or both). The Alistairs justification of the name and that it was used to provoke the audience in a conference about software crisis [13]. The circumstances behind its appearance. The software life cycle with its phases. Notes: emphasize that software is essential in all disciplines, hence anyone involved in software development should take this course; no need to make any comparison between the field and other fields of engineering,. Highlight that sequential implementation of the phases is not appropriate (e.g. waterfall model) and hence other models are used (e.g. Spiral and Agile). Avoid giving many definitions which have no consensus among professionals or in industry.
2	The agile model (Extreme Programming). Students are asked to form teams of 4-6 students each for the course projects.
3	Case study 1 (a case study that uses the Agile model). Students start working on project 1 (using the Agile model). The Object-Oriented modeling concepts. The software life cycle phase 1: requirements elicitation
4	The Spiral model. Introduction to UML. Use-case diagrams. Case study 2 (start a case study that uses the Spiral model).
5	The software life cycle phase 2: OO Analysis. Object diagrams & class diagrams. Case study 2 (continued). Students submit project 1 (using the Agile model).
6	The software life cycle phase 3: OO design. The sequence, collaboration, and state chart diagrams. Case study 2 (continued). Students start working on project 2 (using the Spiral model)
7	- The software life cycle phase 4: OO implementation (mapping models to code). The software life cycle phase 5: verification, validation and testing. The activity, package and component diagrams. Case study 2 (continued).
8	The software life cycle phase 6: maintenance. End of case study 2.
9	Reusing patterns. Project management. Begin case study 3 (a case study that uses the Spiral model).
10	More on project management (work breakdown structure, task model, organization chart). Case study 3 (continued).
11	User interface design. End of case study 3.
12	Software management (people, cost, quality, process improvement, configuration). Case study 4 (a case study that uses the Spiral model).
13	Emerging technologies. End of case study 4.
14	Case study 5 (a case study that uses the Agile model). Students are to submit the course project and make a brief presentation.

We have not given the details of every topic in the list of contents above. However, there is a time frame for each topic. For example, the user interface design is in week 11 and it is expected to be covered in the range of 2 hours (1 week = 3 class hours). Each instructor may decide what to teach in these two hours. However, it is expected that these two hours will be only enough for broad ideas. Also, the list of contents above shows that we focus on case studies because we believe in learning by doing. The course does not have a lab, so we had to give students the practical experience they need in class. In addition, through case studies we can show students how theoretical concepts presented in the course may be applied and used. Our approach differs from other researchers work in its selection of the topics and its focus and use of case studies.

We found the above approach very helpful to students. It gave them the most needed concepts that they are more likely to meet in their work places, and hence much attention was paid to industrial needs and practices. However, we find that more investigation on the proper size of the course project need to be done. Some students took the project for a very detailed level and have developed very detailed diagrams and this has affected the time they needed for coding and some of them could not finish proper testing and validation of their work.

V. Conclusion and Future Work

This paper has presented our approach to teaching a software engineering course based on our experience in adopting an applied-oriented manner. We have focused on the most common models that students may encounter in their future carriers, and on the use of many case-studies as a mean to make the course more applied. We have eliminated topics traditional topics and which are taught mainly for comparison purposes. Our approach has enabled students to appreciate using iterative and incremental development processes to develop software systems according to SE principles, to gain skills necessary for implementing solutions in code satisfying industrial coding practices, to work in teams and assume different roles in software development. Students were satisfied with the course and appreciated enabling them to practice what they like most (i.e. programming) very early in the course in addition to gaining experience that simulates as much as possible what they may encounter in their future carriers. Our future work intends to focus on the course project to identify the proper size of it and how to prevent some students from being drafted by the analysis and design phases to the point that it affected their coding and subsequent phases.

VI. Acknowledgements

This research work was funded by Qatar National Research Fund (QNRF) under the National Priori-ties Research Program (NPRP) Grant No.: 09-1205-2-470.

References

[1] R. Decker and S. Hirshfield, "The Top Reasons Why Object-Oriented Programming Can't Be Taught in CS1," in *SIGCSE '94-*, pp. 51-55, 1994.

[2] P. Breton, S. Lees, M. Gumbley, C. Boldyreff, S. Drummond, P. Layzell, L. Macaulay, and R. Young, "Distributed Group Working in Software Engineering Education," *Information & Software Technology* vol. 40, no. 4, pp. 221-227, 1998.

[3] J. Ludewig and R. Reißing, "Teaching What They Need Instead of Teaching What We Like – the New Software Engineering Curriculum at the University of Stuttgart," *Information and Software Technology* vol. 40, no. 4, pp. 239 – 244, 1998.

[4] F. Culwin, "Object Imperatives!" In *SIGCSE '99*, pp. 31-36, 1999.

[5] A. Drappa and J. Ludewig, "Simulation in Software Engineering Training," in *Proceedings of the 22nd International Conference on Software Engineering*. 2000, ACM. p. 199-208, 2000.

[6] S. Mann and L. Smith, "Role of the Development Methodology and Prototyping Within Capstone Projects," in *Proceedings 17th Annual NACCQ*, MANN, S. & CLEAR, T. (eds). Christchurch. July 6-9th, pp. 119-128, 2004.

[7] S. Mann and L. Smith, "Arriving at an Agile Framework for Teaching Software Engineering," *19th Annual Conference of the National Advisory Committee on Computing Qualifications*, Wellington, New Zealand, NACCQ in cooperation with ACM SIGCSE, pp. 183-190, 2006.

[8] S. Mann and L. Smith, "Technical Complexity of Projects in Software Engineering," in *Proceedings 18th Annual NACCQ, MANN, S. & CLEAR, T.* (eds), Tauranga. July 10-13th July 2005. p249-254, 2005.

[9] J. Noble, S. Marshall, S. Marshall, and R. Biddle, "Less Extreme Programming," in *ACE 2004 Proceedings*, pp. 217-226, 2004.

[10] SWEBOK executive editors, A. Abran and J. Moore editors, P. Bourque and R. Dupuis, "Guide to the Software Engineering Body of Knowledge," *IEEE Computer Society*, pp. 1–1, 2004.

[11] I. Sommerville, *Software Engineering* 8th Edition, Addison-Wesley, USA, 2007.

[12] H. Van Vliet, *Software Engineering: Principles and Practice*, 3rd Edition, Wiley, 2008.

[13] Alistair. The end of software engineering and the start of economic-cooperative gaming. [Online]. Available: http://alistair.cockburn.us/The+end+of+software+engineering+and+the+start+of+economic-cooperative+gaming.