Systematic Offering of Software Engineering Tools and Practices in Software Engineering Curriculum

Shafagh Jafer[1] Mark Hornick[2]

Abstract – This paper highlights the need for including a thorough course on Software Engineering Tools and Practices in Software Engineering curriculum. Often, institutions include topics on various software engineering tools and practices throughout different courses in a computing curriculum. Based on our experience at Milwaukee School of Engineering, it is more beneficial to have a separate course that introduces the students to common software engineering tools and practices rather than spreading the content over different courses at sophomore and junior undergraduate levels. This paper presents the need and impact of early offering of "Software Engineering Tools and Practices" course on students' software design skills. The course targets the general software development lifecycle through the introduction of various tools and practices.

Keywords: Software development lifecycle, design, tools, practices, UML.

INTRODUCTION

Software Engineering students are normally exposed to object-oriented programming and computing fundamentals in their freshman year of the curriculum. When the students start their freshman year, they can be quite proficient in a high-level language like C++ or Java. However, at this stage, many students have little if no knowledge about the general software development lifecycle concepts and methods. Students might have been partially exposed to some UML [1] topics like UML class diagrams or sequence diagrams, yet do not have the big picture of where all these fall in when designing and implementing a complete software application. Institutions tackle this challenge by one of the following two options: 1) slowly and incrementally introducing design concepts and software engineering practices in various sophomore and junior courses, or 2) offering a semester-long sophomore-level course that teaches systematic and formulaic approach to creating comprehensive software designs - with the assistance of appropriate design tools and methodologies. While the two solutions mentioned above essentially yield similar results by the time the students begin their senior year, we encourage the second option which emphasizes early and thorough introduction of Software Engineering Tools and Practices. Such a course not only introduces students to the basic practices employed in critical phases of the software lifecycle, but it also provides students with the tools that most software engineers use in these phases. By putting together all of these concepts and methodologies and offering them in one course, students get the "big picture" and are able to practically employ them at different phases throughout the course deliverables. At institutions where no such course is offered, instructors need to spend significant number of hours within different courses to introduce the students to different design and practice concepts. Not only noticeable amount of time must be dedicated to review and introduce such topics, it also affects the students negatively since they lose the continuity of the subject matter as different courses target only one or two design concepts and leave it to the students to tailor them together. For instance, when students are introduced to UML State Diagrams in their late junior year, they typically wonder why they were not told about them so they could make use of them at the design phase of the software development lifecycle.

[1] Department of Electrical, Computer, Software, and Systems Engineering, Embry-Riddle Aeronautical University, Daytona Beach, FL, jafers@erau.edu

[2] Department of Electrical Engineering and Computer Science, Milwaukee School of Engineering, Milwaukee, WI, hornick@msoe.edu

Students should begin to use software engineering tools beyond an integrated IDE as soon as possible in order to present a broader perspective of the software development process and the practices involved therein. Also, students should get a very early feel that software engineering is not just about coding. Some specific practices and their associated skills, such as UML modeling and version control, can be easily reinforced by integrating the use of these practices into subsequent SE courses in the latter part of the sophomore year. However, it is very important to realize that junior year is already too late to introduce students to formalizing the design process or learning how to use the tools employed in other aspects of software development. By the time students enter their junior year, they should be well-versed in using UML modeling tools, version control tools, and defect tracking tools.

Additionally, many students find internship positions in a software development capacity after their sophomore year, and having experience and exposure to the practices they are likely to encounter will reinforce their prior learning, while making them more valuable to their employers.

There has been much effort in updating and improving the SE curriculum [16 - 19]. We have considered those suggestions and have realized the importance of early and complete introduction of Software Engineering Tools and Practices at sophomore level of the SE curriculum. We highlight the importance of offering such a course and emphasize its content, how it should be delivered, the topics it covers, and the correct order of presenting the material, based on the SE-2030 "Software Engineering Tools and Practices" course taught at Milwaukee School of Engineering (MSOE) since 2007.

This paper is organized as follows: Background section provides a brief background about the Software Engineering curriculum and where design and modeling falls into it. Course Content Section introduces the Tools and Practices course and discusses the course content by presenting the practices and tools covered by the course. In Course Activities section the two types of class activities, weekly labs and team project, are presented. The Week-by-Week Course Outline presents our SE-2030 course outline. The Course Assessment section discusses the course assessment process. Finally, the concluding remarks are presented in Conclusion section.

BACKGROUND

In their freshman year, software engineering students are introduced to problem solving. They learn how to formulate solutions as step-by-step algorithms by using flowcharts and pseudocode. This is a common approach adopted by many as a priori to introducing a formal language. Then, students learn to express the algorithms and translate their pseudocode to a language understood by computers. The students learn about syntax and semantics of a high-level object-oriented language and are introduced to the concept of classes and methods through very simple and small programs. Gradually, the complexity level of the programs increases, allowing the students to work on applications with more than one class encompassing multiple methods. As courses progress and programs become bigger, an excellent approach to smoothen the learning curve is to expose students to UML Class Diagrams. Students learn to observe and comprehend diagrams in order to understand the underlying application.

Thus, by the end of the freshman year, a software engineering student who has successfully completed the required courses is expected to be able to design and implement simple high-level language programs through selection of appropriate algorithms and data structures. Also, the student has been briefly exposed to basic high-level design approaches and has general knowledge about one or two UML models.

Having the knowledge and the exposure mentioned above, at their sophomore year, the students are ready to be formally introduced to the different phases of the software development lifecycle. They are well prepared to learn about each phase and practice systematic approach to produce various deliverables.

A number of institutions (e.g. Purdue University [12], UC San Diego University [13], Carnegie Mellon [14], Stony Brook [15]) offer a similar course to software engineering students. We take a different approach in our SE-2030 course by expanding the software life cycle and presenting a one-to-one mapping of practice and tools. In this article we will present our systematic approach towards introducing students to common software engineering tools and practices. We will present what a typical "Tools and Practices" course would include, discuss the course content in terms of 1) software engineering practices and 2) commonly used tools. The precondition is that students registered in this class have already completed a typical CS1/ CS2 course sequence in object-oriented programming and data

structure fundamentals. We will also make this assumption that students are familiar with Java programing language and have designed applications involving GUI designs and implementations.

COURSE CONTENT

The course aims at filling out various gaps in knowledge on topics that students were briefly exposed to in the previous CS1/CS2 freshman sequence. The course is intended to address the following outcomes:

- Create UML class, communication, activity, and sequence diagrams using a CASE tool
- Transform requirements description to use cases
- Interpret use cases into UML design models
- Generate source code from UML design model, and synchronize subsequent changes
- Maintain source code and related design documents in a revision control system
- Create a deployable software package using an automated build tool
- Track software defects using a change management system
- Be able to create an installable software package using an automated build tool.

These outcomes are presented throughout the course as a specific practice or tool while providing the students an opportunity to apply their acquired knowledge to the course components (weekly labs, in-class exercises, team project, quizzes and tests).

Practices

The course begins with an introduction to software development lifecycle. The content of the course spans over the SDLC phases (Requirements, Design, Implementation, Verification, and Maintenance) by introducing the common practices and tools required for accomplishing each phase's goals. A student's understanding and appreciation of the lifecycle is best substantiated through hands-on activities; thus students are given one or more assignments for each phase (details of class activities to be discussed in Section 4).

At first, students are introduced to basic concepts of functional and non-functional requirements. They learn how to differentiate between "must have" and "nice to have" features and then recast requirements as Use Cases. Students will learn to develop Use Cases out of vague or incomplete requirements. Although requirement elicitation and Use Case development is a more mature topic (which is typically covered in a junior-level "Requirement Analysis" course), rather than a multi-lecture subject, but the goal in Tools and Practices course is to introduce Use Cases as a basis for generating designs. Though the course activities (assignments and team project), students will be convinced that Use Cases are useful and necessary elements in software development.

A considerable amount of the course time is dedicated to Design. This phase of software lifecycle is decomposed into high-level and low-level design phases and related practices and tools are introduced to students. As part of the high-level design, students are introduced to UML Class Diagrams. The concepts of class relationships (inheritance, association, aggregation, composition, dependency, interface, abstract methods and classes, etc.) as well as different diagram notations and syntax are presented. Students practice to design class diagrams through assignments and class activities. The details of the class objects (attributes and methods) are left for the low-level design phase. At this stage, students can comfortably design an application using UML Class Diagram by laying out the application's classes, establishing meaningful relationships among classes and objects, while realizing the importance of the object-oriented concept of high-cohesion and low-coupling.

Once students have the ability to construct a Class Diagram, it is time to show them how other UML design diagrams can be used to improve and expand their initial designs. At this phase, students are introduced to low-level designs through UML Collaboration, Activity, and Sequence Diagrams. The goal is to show the students the correct

order of implementing different UML design diagrams and how they can benefit from them to construct well-detailed design diagrams that can be used to start the Implementation phase.

Students are then shown how to refine their original high-level Class Diagram given the details they have captured with collaboration and, specifically, Sequence Diagrams. With all the methods, objects interactions, and the sequence of actions happening, students are surprised to see how easy it is then to implement a fully-detailed class diagram. This phase of the course is considered the most challenging and rewarding period, as students have gained enough experience with practicing various design concepts and applying them through the aid of various tools they are introduced to in this course (to be discussed in Section 3.2).

UML State Diagrams are also introduced at this stage to better understand and verify the behavior of their systems. Students are now aware of static vs. dynamic system behavior and can comfortably translate the system dynamics into state diagrams representing what the internal behavior of their systems are, given different initial states and input values. These diagrams help students to refine/update their class diagrams, ensuring that their designs capture and react to different scenarios.

The concept of Forward Engineering is then introduced, highlighting the transition from requirements, to design, and finally implementation. With a tentative design in hand, students are led into the implementation phase of development. With automated code generation capabilities of a modeling tool, students automatically generate the skeletal code from their UML designs. This forms an initial code base that can be used as "version one" of their implementation.

At this time, the idea of version control is presented to the students. The concepts of check-in, committing, branching, and tagging are covered and students practice these through the use of a version control system. As a starting point, students will put their initial code into the version control system. This can be a starting point to group students together as teams and instructing them to complete the implementation of application by splitting up the work along logical divisions while keeping every team member up to date, using the version control system.

During implementation, students generally seem to readily adapt to the division of work. As students implement their respective code modules, they are expected to regularly commit their work and update their local copies to sync with their teammates' latest revisions. They quickly appreciate why a version control systems is superior to an approach such as file exchange via flash drives, email, or cloud services such as Dropbox. This phase of development greatly helps students develop their skills at using the version control system. It is also at this phase that students are introduced to the Reverse Engineering concept. They learn to refine and update their original designs as they are completing their implementation. Students are given the opportunity to make use of UML modeling tool to update their class diagrams from the actual code (with the help of automatic synchronization features of the tool). This also makes the maintenance activity easy, as both the design and code are always synchronized.

Verification and testing could also be briefly tackled in a Tools and Practices course. Usually, as students are implementing their applications, they perform some level of informal testing. They attempt to validate their resulting applications with system-level testing. It is at this time of the course that Unit Testing is presented to students. The JUnit testing framework is introduced and students are shown how to implement and perform unit testing for their applications. Since JUnit is linked as a JAR (Java ARchive) at compile-time, automated build tools are also presented at this stage. Students learn about creating a release version of software products, as well as using external JAR files within their applications.

The last phase of software development lifecycle, Maintenance, is lightly reflected in this course. Students employ basic-level of maintenance activities on their team-project activities through software updates, version control, defect tracking, logging changes, and corrections after other teams or instructor experimentally test their final product.

Given the scope of the course (focusing on software tools and practices) we did not address any particular process methodology. Design, implementation, testing, building, and using version control – which are some of the primary topics covered in this course – are elements of any good process methodology. Students follow this course with another course that introduces them to software processes; that process course previously focused primarily on the

Personal and Team Software Processes [10], but since 2012 has focused primarily on Agile Scrum [11]. We did not change the Tools and Practices course when we changed the Process course, since Process addresses fundamentally different aspects of software development with respect to Tools and Practices.

Tools

One of the challenges to students in programming relates to the development tools themselves. In additional to the fundamentals of object-oriented programming, students are required to obtain some level of proficiency in the use of the tool chain used to create software. By the end of their freshmen year, students have experienced using Integrated Development Environments, such as Eclipse, for implementation and debugging. However, they are still unfamiliar with many other tools, such as those used for UML modeling, unit testing, version control, automating builds, etc.

As one of its main objectives, the Tools and Practices course introduces students to a number of useful tools for modeling, testing, and maintaining software systems. This goal not only presents a broader perspective of the software development process and practices involved, but it also prevents students from developing any early impression that software engineers only write code all day.

The course uses Eclipse IDE for Java development as the main development tool, while some students may choose to work with NetBeans which is a very similar environment.

For formulating requirements analysis models and design diagrams, students are introduced to Enterprise Architect modeling tool from Sparx Systems [2]. Throughout the course, students are given step-by-step introductory tutorials on how to construct different UML diagrams. Once familiar with each diagram, students are asked to design and implement their own models for the weekly labs or project activities. Besides modeling UML diagrams, students also learn to use Enterprise Architect to perform forward and reverse engineering, synchronize their models (mostly class diagrams) with their code, documenting, and building and maintain object-oriented software systems fast and easy. Other commercial UML modeling tools (e.g. Visual Paradigm [3]) or open-source tools (e.g. StarUML [4]) are good candidates as well.

Tortoise SVN [5] is used for version control and change management. Students learn to use the SVN client as a Windows shell extension by easily checking-in, committing, branching, tagging, and synchronizing local copies from their file browser. Students are also presented with the SVN Subclipse [6], where they install an SVN Subclipse on their Eclipse IDE for even easier and faster subversioning from within their software development environment.

The JUnit testing framework [7] is accessible from Eclipse environment. Students easily implement and execute unit tests on Eclipse. The IDE also provides capabilities to create executable applications (JAR), and write build scripts.

Students also learn to write an automated build script based on Apache ANT [8] that once executed, automatically runs their JUnit tests, builds their applications, and reports any bugs to a log. The ANT tool can also be accessed as a plugin from within Eclipse. In this course students also learn to run ANT scripts from outside of Eclipse [9].

COURSE ACTIVITIES

Weekly Labs

To stimulate students' learning and provide hands-on experience opportunities, students are given a weekly lab activity that can be normally completed within a two-hour time frame. Each lab reflects the materials covered in that week and usually are in the form of a simple exercise. Lab assignments are intended for individual effort, although students are encouraged to discuss the assignments in small groups. Since later in the course students are given a team-based project, lab assignments are only given for half of the total number of lab sessions in the course. The rest of the lab sessions are spent on team-based project activities (to be covered in Section 4.2).

Each lab assignment is usually posted twenty four hours prior to the beginning of the lab. This is to ensure that students read the lab instructions prior to attending the lab, so that they can use the lab session for asking questions or discussing the lab assignments with their classmates. When assignments involve creating a UML model, the lab assignment is supplemented by a step-by-step tutorial showing students how to use the modeling tool to create such

diagrams. In these cases, students are required to go over the tutorial and come to the lab with enough knowledge to start the lab activity immediately.

The first assignment involves creating Use Case diagrams using a standard template supplied. Students are given a one-page description (which serves as the "requirements document") of a simple application that performs some simple function. For example, the application may have to read some words from a file, analyze the text by calculating the number of letters, words, and sentences, and display the results. With such assignment, students are challenged about the details of the system's requirements. They start thinking about acceptable/expected system inputs, acceptable user interface capabilities, acceptable error handling, and more. To address these questions, students create Use Case diagrams and analyze them by formulating Use Case textual descriptions (a standard template is provided to them).

The next lab involves creating UML Communication diagrams as a consequence of analyzing the Use Cases in the previous lab. The goal of this lab is to allow students focus on the decomposition of their system. This would enable them to view the application as a set of interacting subsystems which they will use as an initial design sketch for their next assignment.

The first high-level design lab assignment is then given. Students are required to use their Communication diagrams to design a high-level class diagram encompassing the application's classes and their relationships. The details of each class (attributes and operations) are left for subsequent assignments.

With their initial class diagrams at hand, students are then asked to create sequence diagrams, investigating the details of their designs. This lab mainly focuses on getting the most details of the system and reflecting it on their design models. With the result of this lab, students can then go back to their class diagram and refine it by adding the details of each class (attributes and methods) and enhancing the relationships among classes.

The subsequent lab first lets students to use the forward engineering feature of the modeling tool to automatically obtain their application's skeletal code. This code serves as the basis for their next lab which is coding and design refinement. At this stage, students also practice version control by following the instructions to use the SVN tool and putting their code baseline into the version control system. The students then continue their implementation by adding all the required functionalities.

Finally, any design change/update is reflected back to the original class diagram (by updating the Class Diagram using the reverse engineering feature of the modeling tool). Also, an executable release version of the product is created and uploaded to the SVN server along the final code version and all design diagrams.

Team Project

After a few lab assignments, students are presented with a new, more substantial assignment as their team project. Depending on class size, students are separated into groups of three or four, where all teams are assigned the same project. The project involves the design and implementation of a relatively complex application over the course of the term. The nature of the application varies from year to year in order to keep the material from becoming out-of-date; some applications created in past course offerings have been:

- ATM terminal
- Digital photo album
- Dictionary
- Mobile phone contact management app

This is normally the first exposure many students have to working on team projects. In the CS1/CS2 sequence, nearly all development is done on an individual basis. Students see the complexity involved in communicating with other team members, the most challenging aspect of team work. Nevertheless, they soon realize the necessity for a good design in dividing up the implementation work efficiently.

The team project activities begin as soon as the students are presented with the project functional requirements. Using the practices and tools presented in the course thus far, the students are required to:

- Create Use Cases and their corresponding textual analysis
- Deconstruct Use Cases into a high-level design along with corresponding UML communication diagrams
- Refine the high-level design into a UML Class diagram, along with UML Sequence diagrams to illustrate the important interactions
- Implement the application
- Conduct unit testing by writing JUnit modules
- Build/deploy the application using automated build scripts
- Use the version control system throughout the process to archive and synchronize all artifacts and maintain updated copies among team members.

The team project spans over the final six weeks of the course. Up to that point, students have received essentially all of the theoretical material need to execute the process. The lectures in the final weeks of the course recap the details of some of the design topics such as state diagrams, pseudocode, and activity diagrams. The tool support for each of these design aspects is also presented and relative lab activities are assigned.

WEEK BY WEEK COURSE OUTLINE

In the table below, we present a sample week-by-week outline of the course, highlighting the lecture topics and the weekly lab activities. The course webpage currently being offered at Milwaukee School of Engineering can be accessed at: https://faculty-web.msoe.edu/hornick/Courses/se2030/index.htm. All lecture materials and external reading materials can be found there.

Week	Day	Topic	Reading	Lab
1	M	Introduction to the course	Wiki (review) - Creating class diagrams using Enterprise Architect	UML Review Activity: Software Design with UML Class diagrams
	R	Class relationships in UML	Wiki - Class relationships in UML Class Diagrams	-
2	M	UML model to Java Source	Wiki - Code Engineering with EA	**Quiz 1 - UML Class relationships** Activity: Class Diagram Translation to Implementation
	R	Synchronization of UML models from Java source		-
3	M	Introduction to Version Control Version Control configuration and use: Subversion	Wiki - Version Control Overview Wiki - TortoiseSVN installation instructions Wiki - Essential TortoiseSVN - tutorial	**Quiz 2 UML again** Activity: Design and Code Synchronization

	R	Version control continued (updating, committing, branching)	Subversion online user manual	
4	M	Building JAR files Sample code	Wiki Building JAR files Building JAR files - Sun tutorial	**Quiz 3 - EA and Subversion usage** Activity: Deploying applications as JAR files
	R	Automated build tools: ANT Using ANT to build a JAR file inside Eclipse		
5	M	Software Life Cycle/Requirements Analysis Requirements coverage via Use Cases	Intro to Use Cases Use Case Template Sample: ATM Use Case - original	**Quiz 4 - Use Cases** Project: Requirements Analysis - Use Cases
	R	High-level design: Use Case Textual Analysis	Driving Design with Use Cases	
6	M	UML Communication Diagrams with EA	Sample: ATM Use Case - analyzed	Project: High-level design UML Communication Diagrams
	R	catch-up		
8	M	Low-level design: Domain Modeling Model-View-Controller design pattern		**Quiz 5 - Requirements Analysis** Project: Detail-level design
	R	UML Sequence Diagrams revisited	Wiki: Sequence Diagrams Sample: ATM Detail Design EA model Sample: ActivationDemo.zip	
9	M	State Machines State Diagrams with EA Advanced State Charts	EA State Diagram tutorial David Harel's orginal paper on State Charts Sample Code: Lamp.zip	**Quiz 6 - UML Communication diagrams** Project: Design & implementation
	R	State Machine details		
10	M	Low-level algorithm design Pseudocode and Flowcharts in EA		Lab 8: Implementation, continued
	R	**Quiz 8 - State Diagrams** State Machine details for project		
11	M	Final Exam prep - design case study		**Quiz 9 - State Machine Implementation** Project: Project demonstrations

COURSE ASSESSMENT

The Tools and Practices course is the software engineering students' first exposure to substantial team-based development. In addition, students are exposed to a significant amount of new material in terms of both software design concepts as well as other aspects of software development and associated tool usage. Student learning is assessed in several ways:

- Written quizzes on foundational concepts (e.g. illustrating relationships such as realization, implementation, dependency, aggregation, and composition in UML class diagrams)
- Interactive quizzes on practical concepts (e.g. committing, updating, and tagging files to the version control system, using the modeling tool to create UML diagrams of various types)
- Project assignments. The artifacts required to be developed (Use Cases, UML diagrams, build scripts, version control contents) are evaluated for completeness and quality.
- Teamwork. As explained previously, students are arbitrarily assigned to teams. This requires that students establish new interpersonal relationships with others whom they may not have previously interacted. It also imposes a responsibility for each student to participate and contribute to the team. This generally works satisfactorily, although on rare occasions, students may indicate issues with another team member's lack of communication or failure to deliver a work product on time. An individual student's contribution to the development of particular aspects of an assignment can be verified via the version control system, which maintains logs of all file commitments and changes.

We assess students' proficiency in tools, practices, and processes in several other courses (for example, in our courses on Software Component Design, Software Architecture, Software Quality Assurance, and Senior Design) as part of our continuous assessment efforts. We reinforce the concepts learned in the Tools and Practices course in subsequent courses. We have not done comparative assessments of our model vs. a "distributed coverage of tools". Clearly, this could be the basis for a more formal study of outcomes. However, at this time, we do not possess the data at that level of detail. That is an effort for future consideration. The distributed coverage approach was an element of a much older curriculum track; the Tools and Practices course (and focused approach) was introduced along with numerous other curriculum changes that were motivated by our periodic review of student outcomes (an element of our regular assessment process at MSOE). The Tools and Practices course has been part of the curriculum since 2006, and we have generally observed more satisfactory student performance w.r.t. the older curriculum – but that is the result of many small changes. It is difficult to associate those improvements with any particular individual curriculum change. One bit of information that supports our decision to maintain our current approach is student feedback regarding specific courses. Outgoing seniors regularly cite the Tools and Practices course as one of the most useful courses they had taken.

CONCLUSION

Software engineering students need to be introduced to formalizing the design process and learning how to use the tools employed in other aspects of software development much earlier in their curriculum. Thus, offering a systematic Tools and Practices course in the early sophomore year is crucial. Students who successfully complete the Tools and Practices course are seen in subsequent courses to be generally much more adept and at ease with the various tools when they are required to use them again. They are systematically familiar with the design process and are well prepared to take correct steps in designing complicated systems. This article discussed the need for such a course and presented the layout of the course content and its assessment process. Since its first offering in 2007 at Milwaukee School of Engineering, the Software Engineering students have very much appreciated it and reported successful stories of how prepared they felt at their internship positions and junior-level courses. The students have been much productive and well-versed in using UML modeling tools, version control tools, and defect tracking tools.

REFERENCES

[1] Fowler, M. 2004. UML Distilled: A Brief Guide to the Standard Object Modeling Language. Addison-Wesley Professional.
[2] Sparx Systems. Available at: http://www.sparxsystems.com. [*Last accessed: November 2013*]
[3] Visual Paradigm. Available at: http://www.visual-paradigm.com. [*Last accessed: November 2013*]
[4] Star UML. Available at: http://staruml.sourceforge.net. [*Last accessed: November 2013*]
[5] Tortoise SVN. Available at: http://tortoisesvn.net. [*Last accessed: November 2013*]
[6] SVN Subclipse. Available at: http://subclipse.tigris.org. [*Last accessed: November 2013*]
[7] JUnit. Available at http://junit.org. [*Last accessed: November 2013*]
[8] Serrano, N, Ismael, C. 2004. Ant: automating the process of building applications. Software, IEEE 21.6 (2004): 89-91.
[9] Apache ANT. Available at: http://ant.apache.org. [*Last accessed: November 2013*]
[10] Humphrey, W. "Introduction to the Team Software Process". Addison-Wesley. 2000.
[11] Rubin, K. "Essential Scrum: A Practical Guide to the Most Popular Agile Process". Addison-Wesley. 2012.
[12] https://engineering.purdue.edu/ee364/. [*Last accessed: January 2014*]
[13] http://extension.ucsd.edu/studyarea/index.cfm?vAction=saCourses&vStudyAreaID=14&#Software Engineering Tools and Processes. [*Last accessed: January 2014*]
[14] http://www.cs.cmu.edu/~aldrich/courses/413/. [*Last accessed: January 2014*]
[15] http://www.cs.stonybrook.edu/~stoller/cse308/. [*Last accessed: January 2014*]
[16] Sebern, M. J., Lutz. M. J. "Developing Undergraduate Software Engineering Programs." CSEE&T. 2000.
[17] Sebern, M. J. "The software development laboratory: Incorporating industrial practice in an academic environment." Software Engineering Education and Training, 2002.(CSEE&T 2002). Proceedings. 15th Conference on. IEEE, 2002.
[18] Lethbridge, T. C., et al. "Improving software practice through education: Challenges and future trends." Future of Software Engineering, 2007. FOSE'07. IEEE, 2007.
[19] Yusop, N. M., et al. "Survey of Software Engineering Practices in Undergraduate Information System Projects." Software Engineering and Computer Systems. Springer Berlin Heidelberg, 2011. 527-537.

Shafagh Jafer

Dr. Jafer is an Assistant Professor in Software Engineering at Embry-Riddle Aeronautical University in Daytona Beach, FL. She has taught software engineering topics at Milwaukee School of Engineering prior to joining ERAU. Dr. Jafer obtained her Ph.D. degree from Carleton University in 2011. Her research focuses on modeling and simulation, emergency response, disaster management, and high-performance parallel computing.

Mark Hornick

Dr. Hornick is an assistant professor in the Software Engineering Program of the Electrical Engineering and Computer Science Department at the Milwaukee School of Engineering (MSOE). He teaches courses in software design, practice, and process; he also manages MSOE's Software Development Lab. Dr. Hornick joined MSOE's full-time faculty in 2004. Before joining MSOE, Dr. Hornick worked in private industry for nearly 20 years.