Teaching Undergraduate Software Engineering Using Open Source Development Tools

Scott Teel, Dino Schweitzer, and Steve Fulton
United States Air Force Academy, Colorado, USA

scott.teel@usafa.edu, dino.schweitzer@usafa.edu, steven.fulton@usafa.edu

Abstract

Software engineering is a key topic in computing education. Many schools offer a project-oriented course, or multi-course sequence, to teach students both the theoretical concepts of software development as well as the practical aspects of developing software systems in a team environment. Typically, in these courses, students practice the principles of requirements analysis, project management, a development methodology, and effective teamwork through a small-to-medium software project. For such a course to maintain its currency and relevancy, it is important for students to be exposed to current tools and techniques for software development. Capabilities, such as project management, requirements tracking, configuration management, collaboration tools, and team communication are ideally experienced in a hands-on manner as part of the project. Commercial tools can be cost-prohibitive and difficult to learn to use effectively in a one or two semester course. At our institution, we investigated the use of open source software development tools that were easy to learn, transferable to other classes to enhance their perceived value to the student, and could be easily integrated into the existing project-oriented two-course sequence in software engineering. This paper describes the tools and their integration in the course, our experience, student's reactions, and compares the results to previous course offerings.

Keywords: Software engineering education, open source tools

Introduction

The computing sciences are complex fields that combine both theoretical and practical components. Students successfully completing an undergraduate computer science program should have instruction in both the mathematical and theoretical foundations of computing as well as the more practical aspects of how to effectively use computers to solve problems. Software engineering, as its name implies, is more directed toward the practical aspects of how to successfully develop complex software systems that meet user requirements and are reliable, usable, and maintainable. Computing Curricula 2005 recognizes the need for more extensive education to produce professional software engineers than what can be reasonably provided in a typical computer science program (ACM, 2005). As such, they propose software engineering be treated as a totally separate discipline within computer science education. However, while there has been some debate about the exact role software engineering should have in a computer science program (Curran, 2003), the ACM Curricula has main-

Material published as part of this publication, either on-line or in print, is copyrighted by the Informing Science Institute. Permission to make digital or paper copy of part or all of these works for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage AND that copies 1) bear this notice in full and 2) give the full citation on the first page. It is permissible to abstract these works so long as credit is given. To copy in all other cases or to republish or to post on a server or to redistribute to lists requires specific permission and payment of a fee. Contact Publisher@InformingScience.org to request redistribution permission.

tained the importance of software engineering to all computer science students and has kept it as a core element in computer science education (ACM, 2008).

For purposes of this paper, we will focus on the teaching of software engineering within the computer science discipline. Unlike other topics in the computer science major, the techniques and principles taught in software engineering are often first developed and refined in industry before arriving in the classroom. As commercial software development techniques and tools evolve, so pedagogical methodologies change.

Computer science educators have taken different approaches to teaching software engineering over the years, both as a result of changing methodologies as well as individual beliefs about what teaching methods work best in a particular academic environment.

This paper is a case study in applying current productivity tools (specifically Redmine) in a software engineering course at our institution. The objectives of the study are to investigate how to integrate the tools into the existing structure and evaluate their impact. We describe our experience in recently changing our approach to teaching software engineering to be more aligned with current tools and how to effectively use them in an academic environment. The paper begins with an overview of software engineering teaching methods, then describes our traditional approach and motivation for changing, how the new approach was integrated into the course, our experience with the change, and, finally, our plans for the future

Software Engineering Education

Background

Software Engineering is defined as the "application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software" (Petkovic, Thompson, & Todtenhoefer, 2006, p. 294). The need to teach software engineering in colleges has been identified for decades. Stiller & LeBlanc (2002) point out as far back as the early 1990's, ACM Computing Curricula suggested that the at least one software engineering course should be required in accredited computer science programs. They point out that the number of large software projects in industry demand this and suggest that the proof of the success of these accredited computer science programs should be seen in the reduction of failure in the design and operation of large computer programs. Software engineering programs can be thought of as a replacement for the old apprenticeship programs which taught the trades to workman (Stroulia, Bauer, Craig, Reid, & Wilson, 2011). The issue, however, is how to bring this effect into the classroom. Many educators feel that current practices of teaching software engineering are not adequately preparing students for the real world of software development. Nurkkala & Brandle (2011) summarize the problems with current teaching approaches:

- No product – students are creating projects, not commercial grade products
- Short duration – single semester, or two-semester, courses impose an artificial time constraint
- High turnover – new students each semester means the talent pool remains shallow and student skills are not developing based on previous experience
- Low complexity – by necessity given time constraints and skill sets
- No maintenance – as a result of short duration, students do not experience a key aspect of software development, the maintenance phase
- No customer – most software engineering projects do not interface with a real customer

To address these shortcomings, different approaches to teaching software engineering have emerged and been proposed in the literature.

One of the driving issues in how software engineering is taught is current trends in industry software development practices. For many years since early 1970's, the "waterfall model" life-cycle was considered the most successful for a structured approach to software development. Popular textbooks espoused its techniques and students were taught its methods. Subsequently, focus was placed on more incremental approaches such as spiral development (Clinton, 1998) and rapid prototyping (Boehm, 2006). More recently, techniques such as extreme programming (LeJeune, 2006), agile programming (Lu & DeClue, 2011), and software performance in programming (Dugan, 2004) have been incorporated in the educational experience. Some universities have courses which focus on programming techniques versus software engineering processes and team work (Petkovic et al., 2006). Regardless of the exact methodology taught, one common element of teaching of software engineering classes is the use of a final group project. These projects are typically multi-team (and, more and more, multiple school programs) which allow students to participate in a team environment similar to what they will find in the 'real world' (Coppit & Haddox-Schatz, 2005; Rusu et al., 2009; Stiller & LeBlanc, 2002; Stroulia et al., 2011). Learning to work in teams is identified as a student outcome in the IEEE Curriculum Guidelines for Software Engineering (IEEE, 2004).

Other proposed methods for teaching software engineering include the use of formal methods (Liu et al, 2009). Deveaux et al, (1999) suggest focusing on the documentation of the process versus the software itself. The claim is that it is difficult to achieve a large enough project in an academic course to make software engineering meaningful, but that a "docware" approach teaches sound principles. Li (2009) successfully applied the Unified Process methodology in teaching students. Pandey (2009) advocates the use of competition to teach development principles. In more novel approaches, Navarro & Hoek (2004) and Shaw & Dermoudy (2005) created single player games in which students take on the role of project manager of a team of developers. Many educators have looked at ways of defining meaningful projects that are realistic, doable, yet large enough to require the use of software engineering techniques. For example, Alzamil (2005) describes the use of carefully selected semi-professional organizations and projects from the local community that can act as real customers.

Another issue in software engineering education that has been addressed by several educators is how to motivate students to appreciate the importance of software engineering. Stiller and LeBlanc (2002) suggest that the complex and challenging nature of software engineering make the effectiveness of such courses difficult and outline six steps to convince students of the importance of effective software engineering approaches: make it real, make it fun, make it critical, make it accessible, make it successful, and speak with a clear consistent voice in outlining strategies for students. They suggest that it is the responsibility of the faculty to sell the students on the value of software engineering by convincing them that it is not a boring subject. Callele et al., (2006) recommend a "stealth approach" of teaching underlying software engineering principles, such as requirements engineering, early in the students education without labeling it as software engineering. Razmov & Anderson (2006) present a more overt approach to creating a positive atmosphere by incorporating innovative technology, such as tablet PC's, in the classroom to motivate students. Claypool & Claypool (2005) propose putting the "fun" into the project by focusing on game design.

Tools in Software Engineering Education

Regardless of the specific pedagogical approach taken, many software engineering educators have noted the importance of teaching students the use of tools in software development, both to familiarize students with current industry practices as well as to aid their own development effort.

Some offerings use existing development tools such as project planning software, various Integrated Development Environments (IDE's), and configuration management tools. Other institutions have developed customized tools, such as the Personal Assistant for Software Engineers (PASE) for project and metric tracking (Dick et al, 2000). Watkins (2009) describes the use of Web 2.0 tools, such as Facebook and Google Docs, to promote team communication and collaboration. In addition to using the existing tools to facilitate project development, their institution developed a Web 2.0 peer evaluation system to assist in team assessment.

Student understanding of and practice with current software productivity tools are critical to a complete education in software engineering. Students are better prepared to enter a software development career, they gain an appreciation of the capabilities and limitations of such tools, and they develop a more complete understanding of the software development process. In addition, open source tools provide a cost-effective means to provide students with this experience, and teaches students the value of the open source approach.

Our Course

Historical

At our institution, we offer a traditional computer science major following the ACM/IEEE recommended curriculum guidelines (ACM 2008). The major includes a two-semester software engineering course taught to senior computer science students. The course is project-oriented and considered a "capstone" experience; that is, it encompasses many aspects of their other courses and attempts to provide a culminating educational experience. The course has been taught for over 20 years and has gone through many variations.

The course has traditionally taught software engineering principles using standard texts, as well as providing hands-on experience through a medium scale software development project. The development methodology taught has evolved from the traditional waterfall model to a spiral approach to agile programming. Project team sizes have varied from small (3-5) to relatively large (10+). Examples of previous projects include:

- Real-time visualization of satellite tracking and status
- Therapeutic joystick for enhancing motor skills of disadvantaged children
- Data aggregation for warfighter mission planning
- Immersive controls for a flight simulator
- Energy simulator to assist students in learning world energy dynamics

Currently, the course consists of 31 students, 25 Computer Science majors and 6 Systems Engineering students. Students take the two-semester course their senior year. The 31 students were broken into seven project teams of four or five. A single instructor teaches two offerings of the class and serves as project mentor for all seven projects.

The projects attempted to cover the majority of the development process from requirement definition, through development and testing, with an introduction to maintenance. Projects have utilized "real world commercial" customers using local industry, "simulated" customers using faculty members, and academic projects for other departments. Each of these approaches offered both benefits and challenges as noted by other educators. Throughout the various offerings, the emphasis in the course has always been on the development process, not the end product.

To support the emphasis on process, prominence was given to the documents produced as part of the development effort. Templates and examples were provided for requirements gathering and

documentation, project planning, various design reviews, test planning, project presentations, and meeting minutes. Oftentimes, hardcopy was used resulting in a mountain of paper for each project. Provided templates were not customized to individual projects, creating a "one size fits all" mentality.

While the format of the paperwork and the basic development methodology were standardized across projects, the choice of programming languages, development environments, and development tools was left to individual teams. This freedom was intended to allow for flexibility when choosing an approach for a specific project type. The best choice for languages and tools could vary depending on the details of the project. The use of tools themselves was left optional as long as the documenting paperwork demonstrated the tasks and planning had occurred. Students were exposed to project planning tools such as Microsoft Project, but their use was not mandated.

Motivation for Change

Our previous approaches to teaching the software engineering capstone sequence created various challenges to both students and instructors. One problem was a result of the large amount of paperwork generated. Since each team had to maintain a series of documents that evolved as the project progressed, documentation became an unwelcomed part of the development effort. To deal with it, several teams designated a "documentation expert" member of the team whose primary role was to keep the paperwork current and in the correct format. Having a single individual responsible reduced the team's overall effectiveness as team members focused on their own part of the development effort without embracing a more global view of the project. This limited effective team collaboration. Another downside was, depending on the dedication and skills of the designated individual, the paperwork could become onerous and result in poor documentation quality. Finally, documentation was seen as a de-motivational and separate aspect of software development and the student take-away was that it was a necessary evil versus an enhancing process.

Another disadvantage of the previous approaches was the lack of a standardized suite of tools. While this was intentional to allow for flexibility, it was difficult for the instructor to be the expert in all tool types and assist students when necessary. It also made equitable evaluation between projects hard as well as providing meaningful feedback. Ideally, tools should be current in the field so students are exposed to up-to-date techniques, standardized for ease of grading and providing feedback, and be seen by students as enhancing productivity, not hampering. Tools should also enhance team collaboration and communication.

To address these issues and provide a more productive experience for students and instructors, a different approach to teaching the software development sequence was started in fall 2011. The remainder of this paper will describe the approach, present our experience to date, and discuss future plans.

New Course Approach

While many tweaks have been made to the software engineering sequence over the years, significant changes were made to the current offering to address the shortfalls described above. The greatest change was the integration of a standardized set of open source software tools throughout the development process to enhance communication and collaboration, reduce the onus of paperwork, and expose students to current tools in industry. Similar to Watkins (2009), we chose a suite of tools that support a Web 2.0 approach. The tools, their integration, and benefits are described below. Other changes to the course included how projects were selected, how teams were organized, and who acted the role of "customer".

Tools

The tools we are using to support our software engineering course fall into four major categories: Project Management Tools, Project Communication Tools, Programming Tools, and a Source Code Control Tool. To provide tools that were easy to support and currently in use by corporate software professionals, the instructor chose to require the usage of a standard suite of open-source tools. This standardization on a single set of tools reflects the real-world of professional corporate software development that many students will be entering upon graduation. The following sections detail each of the open source tools along with the perceived benefits by the students and instructor.

Redmine

Redmine (www.redmine.org) provides both Project Management and Project Communication capabilities for all student project teams. Redmine is open source and released under the terms of the GNU General Public License v2 (GPL).

Some of the major Redmine features that both instructors and students found useful are:

- Multiple projects support - We use one Redmine project per student team plus one Example Project provided by the instructor. Each project had the following built-in features: Wiki, News, Document/Files, Forums, Issue tracking (requirements, bugs, tasks, and enhancements) with associated scheduling, versioning, time-tracking, calendar, and Gnatt charts. Each project's Wiki is a central area for information sharing -- and is an especially valuable method for distributing instructor-provided examples. For example, at the start of the course when student teams are creating their Project Management Plan and the Requirements Specification, they "go to school" on the instructor-provided examples and templates in the Example Project.

- Flexible role based access control - Depending on the team size or students' individual expertise (e.g., for courses with students from multiple majors such as Computer Science, Management, and/or Systems Engineering), appropriate team roles can be enabled. For example, on a large team with dedicated student project managers, these managers can be the only people with permissions to manage time-tracking, add new versions to the product, and assign issues to specific product versions. For smaller teams, "power" teams, or research teams, all members can have the same permissions. Additionally, the instructor can be designated as the Manager for all student teams, giving him/her complete flexibility with monitoring projects, making early corrections, and providing feedback.

- Flexible issue tracking system - for small to medium size student teams, all students can have the same permissions with respect to opening, progressing, and closing issues. For larger student teams, where individual students or sub-teams have well-defined roles (project manager, developer, quality assurance, etc.), each major role can have a workflow appropriate to their role. As an example, the development team can move an issue from the New status into Development -- and from Development into Integration Test, but the team cannot close issues. Closing an issue can only be done by the Integration Test team after a successful test.

In addition to the benefits list above, instructors find that it is much easier to monitor the status of each project team on a regular basis -- and can provide mentoring to more teams than previously. With a single click, the instructor (as the "all-powerful" Manager on all projects), can see on a single webpage how each project is doing -- the status of issues (requirements and tasks), who is

assigned to each, and each issue's priority, due date, and target version. On-the-spot help or advice can be given to the entire team -- or in the case where in the instruction would be useful to the entire class, a course-wide Wiki entry can be made.

In the areas of intra-team communication, several benefits are derived. For example, instead of project teams having to constantly de-conflict their schedules for a team meeting to discuss what should be done, who is doing what, and why it should be done a specific way, Redmine Forums, Wikis, and Issues provide an asynchronous communication path that all members are privy to and that can be used wherever the student is and whenever needed. Additionally, Redmine's History feature provides a method to rapidly get caught up on the current project status and review the path the project took to get to the current state. Project decisions can easily be captured and revisited when needed.

With respect to instructor/student communication, our experience is that the instructor can provide a consistent communication path via Redmine. Since the instructor provides most of the system administration functions for the students' cloud-based services, as student teams need something enabled in the cloud, they can assign a new issue to the instructor and know the instructor will receive and act upon the issue. The instructor also has a single communication path to update the entire student team on the issue's status.

Some other benefits include:

- A central, web-accessible project information repository that provides an easy-to-update and reference location for all students wherever they are, whenever they need. Also, the responsibility for information currency and correctness is distributed to the entire team rather than a single individual designated by the team to the unenviable job of project librarian.

- Redmine helps students learn and appreciate good software engineering techniques since a true software engineering workflow can be "electronically" enforced upon project issues (scheduling and current status of product requirements, tasks, and bugs). These electronic tools also take away much of the "grunt" work that is normally required when trying to manually manage and track this type of information.

- For both the student teams and the instructor, Redmine eases the burden of regular student turn-ins and instructor grading. For students, the "state" of Redmine and the Subversion repository can be used as a regularly scheduled turn-in. This approach also helps enforce the need to keep project documents continually up-to-date -- rather than the common approach of "wait until later" which can result in extremely out-of-date documentation.

- When a team consists of members that are not actually on-site, Redmine provides a means for them to become an active, contributing member of the team. Sometimes our projects have contributors or actual customers that do not reside at our institution. Such participants have a much easier time keeping up-to-date on what the team is doing and how well the project is progressing by monitoring the Redmine site.

- For project presentation purposes, students find they are able to use the team's Redmine and Subversion sites directly when giving briefings rather than rewriting the information into presentation software.

Turnkey Appliances

To easily provide the Redmine stack, we standardize on an appliance from Turnkey Linux (www.turnkeylinux.org). Turnkey provides 45+ ready-to-use stacks based on open source software. Some example appliances are: a LAMP stack, Redmine, MySQL database, Moodle, Tomcat on Apache, and Bugzilla. Some of the important features of these appliances are: they are no-cost, they are pre-configured and pre-tuned, they have a standard set of tools already installed to support each appliance, and each appliance can be backed up to Amazon's S3 Cloud Service for a very small monthly cost ($0.15/GB per month). As an example of the backup cost, we currently have four separate appliances (Redmine/Subversion appliance and three LAMP appliances) being backed up to Amazon for a total cost of $0.50 per month. Another extremely important feature of both the Turnkey Redmine appliance and Turnkey LAMP stack appliance is that, when necessary, the instructor can "1-touch" deploy the entire appliance into the Amazon EC2 cloud service. This provides the capability for student teams to scale up the system and/or bandwidth as they need.

Eclipse IDE and Apache Subversion

Eclipse, from the Eclipse Foundation, is used as the "company-mandated" integrated development environment. It is a multi-language software development environment widely used in the open-source and corporate worlds. Eclipse has an extensible plug-in system that is used to provide support for a wide range of programming languages such as Java, Ada, C, C++, PHP, Python, etc., and also supports a wide range of SDKs such as Google's Android SDK and the Google Web Toolkit SDK. Additionally, there is also a good selection of language and library tutorials that are based around the use of Eclipse as the IDE. For example, Google provides the Android SDK Plug-in for Eclipse and all tutorials for the Android SDK have examples using Eclipse.

Apache Subversion (http://subversion.apache.org/) is used to provide each project a version control system. Subversion is an open source project founded in 2000 by CollabNet, Inc., now a top-level Apache project, and is widely used in both the open source and corporate world.

The integration between Eclipse and each team's Subversion source code repository is provided by the Subclipse Eclipse plug-in (http://subclipse.tigris.org/). Subclipse is released under the Eclipse Public License (EPL) 1.0 open source license.

Some benefits we find when using the standardized development tool suite detailed above are:

- The instructor, already an expert in the use of the tools, is able to provide tutorials on using the suite and give one-on-one instruction when necessary.

- Apache Subversion is provided as part of the Redmine Appliance. Therefore, no additional system installation work is needed on the instructor's part.

- This standardization on a set of tools reflects a "real world" environment that many of the students will be entering upon graduation.

Development and Production Servers

Since all student projects are currently multi-tier applications requiring both an Internet accessible web server and associated database, we standardized on a LAMP stack in the production environment. To easily provide this stack and allow our teams to deploy their application into a cloud-based production environment, we use the LAMP appliance from Turnkey Linux. Currently this appliance runs on a small server on our local network. As each team moved into a

formal Beta Test program with their application, we moved their LAMP appliance into the Amazon's EC2 Cloud Service to give the application the system resources and bandwidth needed to perform a real-world Beta Test in a production environment.

Our Experience

Our experiences so far with the chosen suite of tools are very positive. Many students recognize the value of a Project Management tool since some have asked to use the tool in other courses. Likewise, most students recognize the value of a centralized source code repository once they have to share code with other teammates or they lose their code on their machine for some reason. Some have even asked "Why didn't we use Subversion earlier in our Computer Science coursework"?

With respect to the communication capabilities provided by Redmine, both the instructor and the students find that the use of Wikis enhance a constant flow of information between students and between the instructor and students. According to students, especially valuable is the ability to have easy access to examples and tutorials in a central location.

There were some areas requiring mid-course corrections. For example, before the course started, we thought that using the standard set of Redmine Issues categories ("Bug", "Feature", and "Support") would be sufficient. However, after a short period, we found that adding a "Task" category with a slightly different workflow significantly helped the students plan their work early in the project. Another area requiring correction was in the use of Forums (Redmine Boards). Our initial thought was that this would be a valuable communication tool. However, we have found that no students are using this capability and, instead, are communicating via Wiki and Issues.

In summary, the major benefits we find when using this standardized set of open-source tools are similar to those benefits realized in the corporate world with some additional benefits gained in the educational environment. First, a set of standard development tools and production environments result in consistency for all teams and thus, cost-effectiveness for the instructor in the areas of training, system administration, and cross-project personnel migration. Second, the intra-project communication paths are consistent, geographically agnostic, and always available. Third, the ability of the instructor to be "always available" and provide more oversight and advice to student teams is enhanced. Finally, standardization gives the students a taste of the professional work environment many will be entering upon graduation.

Future Plans

We will continue to enhance the experience in our course by refining how each of the tools is used. In the second semester software engineering course for this group of students, we will continue to explore other Redmine functions we are not currently using. Additionally, we will solicit feedback on which capabilities students found most valuable and which they did not take advantage of. In future offerings of the course, we plan to introduce the tool suite even earlier in the course so students can become comfortable with the tools capabilities and realize their value. We have also worked with other computer science faculty to incorporate the tools into their courses where appropriate. For example, in student Independent Study courses, the instructor is using Redmine as a project management resource.

References

ACM, AIS, IEEE. (2005). *Computing curricula 2005, The overview report*. Retrieved March 14, 2012 from http://www.acm.org/education/curric_vols/CC2005-March06Final.pdf

ACM, IEEE. (2008). *Computer science curriculum 2008, An interim revision of CS 2001*. Retrieved March 14, 2012 from http://www.acm.org/education/curricula/ComputerScience2008.pdf

Alzamil, Z. (2005). Towards an effective software engineering course project. *Proceedings of the 27the International Conference on Software Engineering* (ICSE '05), 631-632.

Boehm, B. (2006). A view of 20th and 21st century software engineering. *Proceedings of the 28th international conference on Software engineering* (ICSE '06), 12-29.

Callele, D., & Makaroff, D. (2006). Teaching requirements engineering to an unsuspecting audience. *ACM SIGCSE Bulletin, 38*(1), 433-437.

Claypool, K., & Claypool, M. (2005). Teaching software engineering through game design. *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education* (ITiCSE '05), 123-127.

Clinton J. (1998). Tight spiral projects for communicating software engineering concepts. *Proceedings of the 3rd Australasian conference on Computer science education* (ACSE '98), 136-144.

Coppit, C., & Haddox-Schatz, J. (2005). Large team projects in software engineering courses. *ACM SIGCSE Bulletin, 37*(1), 137-141.

Curran, W. S. (2003). Teaching software engineering in the computer science curriculum. *ACM SIGCSE Bulletin*, *35*(4), 72-75.

Deveaux, D., Fleurquin, R., & Frison, P. (1999). Software engineering teaching: A "Docware" approach. Proceedings of the 4th annual *SIGCSE/SIGCUE ITiCSE conference on Innovation and technology in computer science education* (ITiCSE '99), 163-166.

Dick, M., Postema, M., & Miller, J. (2000). Teaching tools for software engineering education. *ACM SIGCSE Bulletin, 32*(3), 49-52.

Dugan, R. F. (2004). Performance lies my professor told me: The case for teaching software performance engineering to undergraduates. *SIGSOFT Software Engineering Notes, 29*(1), 37-48.

IEEE STD 610.12 (1990). *IEEE standard glossary of software engineering terminology*. IEEE Computer Society.

LeJeune, N. F. (2006). Teaching software engineering practices with extreme programming. *Journal of Computing Sciences in Colleges, 21*(3), 107-117.

Li, J. (2009). Teaching unified process in software design and development courses: a case study. *Journal of Computing in Small Colleges, 24*(5) 5-11.

Liu, S., Takahashi, K., Hayashi, T. & Nakayama, T. (2009). Teaching formal methods in the context of software engineering. *ACM SIGCSE Bulletin, 41*(2), 17-23.

Lu, B., & DeClue, T. (2011). Teaching agile methodology in a software engineering capstone course. *Journal of Computing Sciences in Colleges, 26*(5), 293-299.

Navarro, E., & van der Hoek, A. (2004). SimSE: An educational simulation game for teaching the Software engineering process. *ACM IGCSE Bulletin, 36*(3), 233-233.

Nurkkala, T., & Brandle, S. (2011). Software studio: Teaching professional software engineering. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education* (SIGCSE '11), 153-158.

Pandey, R. (2009). Exploiting web resources for teaching/learning best software design tips. SIGSOFT *Software Engineering Notes, 34*(6), 1-7.

Petkovic, D., Thompson, G., & Todtenhoefer, R. (2006). Teaching practical software engineering and global software engineering: evaluation and comparison. *ACM SIGCSE Bulletin, 38*(3), 294-298.

Razmov, V., & Anderson, R. (2006). Pedagogical techniques supported by the use of student devices in teaching software engineering. *ACM SIGCSE Bulletin, 38*(1) 344-348.

Rusu, A., Rusu, A., Docimo, R., Santiago, C., & Paglione, M. (2009). Academia-academia-industry collaborations on software engineering projects using local-remote teams. *ACM SIGCSE Bulletin, 41*(1), 301-305.

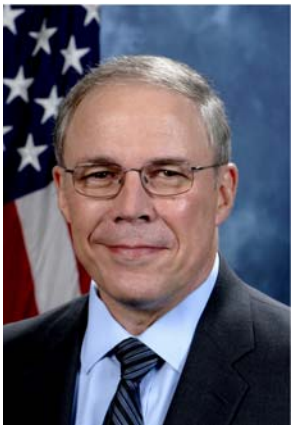
Shaw, K., & Dermoudy, J. (2005). Engendering an empathy for software engineering. In *Proceedings of the 7th Australasian conference on computing education* - Volume 42 (ACE '05), 135-144.

Stiller, E., & LeBlanc, C. (2002). Effective software engineering pedagogy. *Journal of Computing Sciences in Colleges, 17*(6), 124-134.

Stroulia, E., Bauer, K., Craig, M., Reid, K., & Wilson, G. (2011). Teaching distributed software engineering with ucosp: The undergraduate capstone open-source project. In *Proceedings of the 2011 community building workshop on Collaborative teaching of globally distributed software development* (CTGDSD '11), 20-25.

Watkins, K. (2009). Peer evaluation as a needed web 2.0 activity in project management for teaching practical software engineering. In *Proceedings of the 10th ACM conference on SIG-information technology education* (SIGITE '09), 173-177.

Biographies



Dr Scott Teel is a long-time software development engineer and educator. He has served in many industry positions and was the Director of Engineering at Sun Microsystems, Inc., before returning to teach at the United States Air Force Academy (USAFA). He is currently an Assistant Professor at USAFA teaching Software Engineering and conducting research in Computer Science education and mobile technologies.

Dr Dino Schweitzer is the Director of the Academy Center for Cyberspace Research at USAFA. He has over 20 years of experience teaching and conducting research in Computer Science. His research interests include Computer Science education, visualization, and cyber security.

Dr Steven Fulton has served as a Computer Scientist for the Department of Defense for over 20 years. He is currently a Visiting Professor at USAFA and teaches several courses in the Computer Science department. His research interests include Computer Science education and cyber security.