Teaching Developer Skills in the First Software Engineering Course

Václav Rajlich
Department of Computer Science
Wayne State University
Detroit, MI 48202, U.S.A.
rajlich@wayne.edu

***Abstract*—Both employers and graduate schools expect computer science graduates to be able to work as developers on software projects. Software engineering courses present the opportunity in the curriculum to learn the relevant skills. This paper presents our experience from Wayne State University and reviews challenges and constraints that we faced while trying to teach these skills. In our first software engineering course, we teach the iterative software development that includes practices of software change, summarized in the phased model of software change. The required resources for our software engineering course are comparable to the other computer science courses. The students - while working in teams - are graded based on their individual contribution to the team effort rather than on the work of the other team members, which improves the fairness of the grading and considerably lessens the stress for the best students in the course. Our students have expressed a high level of satisfaction, and in a survey, they indicated that the skills that they learned in the course are highly applicable to their careers.**

***Index Terms*—First software engineering course, developer role, project technologies, realistic code, evolutionary-iterative-agile development, phased model of software change, open source, concept location, impact analysis, refactoring, actualization.**

I. INTRODUCTION

Both employers and graduate schools expect computer science graduates to be able to work as developers on realistic software projects [1]. The first software engineering course (1SEC) is the first and sometimes the only opportunity in the curriculum to learn the relevant skills. Another frequent objective of 1SEC is to survey the large body of knowledge that software engineering discipline acquired. 1SEC should also give students a realistic picture of software engineering discipline and motivate them to pursue software engineering careers. A well thought-out 1SEC carefully balances these objectives, and takes into accounts realities of the curriculum within which 1SEC is offered.

Of these objectives, teaching fundamental skills is the most important one, not only because these skills are practically usable in the students' future careers, but also because their possession helps the students to understand the issues that are discussed in other parts of the software engineering discipline. Mary Shaw eloquently argued for "identifying distinct roles in software development and providing appropriate education for each" [2] which closely corresponds to the experience reported in this paper. In 1SEC, we teach the role of the software developer as the first priority because other software engineering roles derive from it.

Section II of this paper outlines the choices that we faced and decisions that we made when creating 1SEC that teaches the developer role. Section III presents the topics that the course covers. Section IV evaluates our experience, Section V surveys related work, and Section VI contains summary and future work.

II. THE INSTRUCTOR'S CHOICES

1SEC is a part of Wayne State University (WSU) upper division computer science curriculum and in the past, it was taught in a traditional way, essentially as a survey course accompanied by a team project that followed a waterfall model. The project was a typical one-semester project of a small size, low code quality, and homogeneous architecture that reflected the limitations of the waterfall and limitations of the student background. When we updated the course in order to focus on the fundamental developer skills in realistic setting, we faced several obstacles that boil down to the restricted time in the course combined with fixed objectives. Within this context, we made choices and trade-offs that are described in this section.

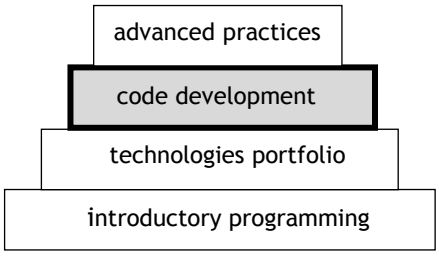

Figure 1: Pyramid of software engineering skills

Figure 1 summarizes our experience with the student learning, described as the pyramid of the skills that build on top of each other. The first skill is the introductory programming skill that is acquired in the prerequisite courses; the students who have these skills are able to develop small programs. Next layer is a mastery of a portfolio of project

technologies that are used in realistic projects. This is followed by "code development" that we consider the main objective of 1SEC. The more advanced practices that constitute the rest of software engineering discipline are the next layer of the pyramid. This section describes the student learning during 1SEC, our choices that facilitate it, and the "seven deadly sins" that we tried to avoid.

It should be noted that in our classrooms, there is always a small minority of students who have already acquired the skills represented by large part of the whole pyramid of Figure 1. However we believe that teaching of 1SEC should be aimed at the great majority of the "middle students" and not just the top few. The students in this great majority do not know the material of 1SEC from their previous experience, but they are able and willing to learn it. It is important for them to learn software engineering skills layer by layer; otherwise they will face difficulties and disappointments when confronted with the expectations explained earlier.

In order to aim at them, the material of 1SEC starts where the prerequisite courses finished, and teaches the developer skills in increments of gradually increasing complexity.

A. *Portfolio of Project Technologies*

Developer skills taught in 1SEC require a realistic project on which students practice these skills. However these projects require knowledge of a portfolio of technologies. A typical portfolio consists of the following:

- Programming language (C++, Java, …)
- IDE (Visual Studio, Eclipse, …)
- GUI (Swing, MFC, .net, …)
- Database (SQL, Hibernate, …)
- Testing tools (Abbott, Junit, …)
- Modeling (UML, XML, …)
- Intra-team communication (Wiki, Blackboard, …)
- Version control (CVS, Subversion, …)

In the ideal curriculum, these technologies should have been taught in the prerequisite courses. From the prerequisite courses, our students know the fundamentals of the programming language (in our case C++) and IDE (in our case Visual studio). Although their knowledge of these technologies in not perfect, it is usually sufficient for the purposes of 1SEC. However our typical students lack the knowledge of other technologies that are essential for realistic projects, because they are not covered in prerequisite classes or they are covered only incompletely. In that situation, we have to review them in 1SEC or even cover them for the first time.

To cover these technologies in full would require a substantial class time and jeopardize the goals of the course. Moreover there is a wide selection of alternatives among the technologies and knowledge of a specific technology portfolio is a very narrowly specialized asset. Also, the life span of the technologies is often short; the software engineering professionals must update their technology knowledge regularly, and therefore this part of the course – although necessary - has only a temporary value. For all these reasons, we decided to teach only the minimal technology portfolio that allows us to explain the code development principles and supports selected projects; we believe that this is sufficient for 1SEC and further exploration of technologies should be taught in other contexts.

In order to preserve the objectives of the course, we decided that this remedial part should be limited to a small fraction of the course, focusing only on the most important issues. The technologies review now takes less than 20% of the course time, preserving the rest of the course for the main course objectives. We believe that devoting larger part of the course to the technologies would come at the expense of these course objectives; our experience is formulated as the first of the "seven deadly sins":

> **Deadly sin #1:** Technologies that support software development occupy too much of the course and insufficient time is left for the main course objectives.

B. *Code Development*

As Figure 1 presents, we consider the skills needed for the realistic code development to be the most important objective of 1SEC. The notion of the "realistic code development" needs additional elaboration. In our experience, the "realistic code" has the following attributes:

- Realistic size and complexity (between 50 KLOC and 500 KLOC)
- Realistic demands on correctness (between 1 and 3 faults/KLOC)
- Moderate technical debt present in the code

Please note that projects that do not reach this "realistic" range, with less than 50 KLOC or textbook-perfect code, may permit unrealistic practices that cannot be applied to the real-world projects. In a well-balanced curriculum, we believe the place for these practices is in prerequisites. On the other hand, projects beyond this range may require additional advanced practices that do not fit within the possibilities of 1SEC, and therefore we believe they belong to the next layer of the pyramid in Figure 1. Those are projects with code larger than 500 KLOC, projects with extreme demands on code correctness, or the projects with decayed code. These projects require advanced practices and our experience indicates that the middle students in 1SEC are not ready for them.

There have been numerous practices of code development proposed or used, including the old waterfall practices where code development consists of the program design and consequent implementation of the code from scratch according to this design. However there has been a paradigm shift away from waterfall and towards evolutionary-iterative-agile development, caused by the volatility of requirements, technologies, and stakeholder knowledge [3]. The data indicate that the old waterfall is nowadays practiced only in small minority of projects, and that the evolutionary-iterative-agile paradigm is the new mainstream [4].

Based on these data, we concluded that the waterfall-based practices no longer belong to the core knowledge; instead,

they belongs to the "advanced practices" of the pyramid of Figure 1. Nevertheless it should be noted that the paradigm shift has not been fully completed, and hence we still see lingering controversies and misunderstandings that are typical of paradigm shifts [5].

The most common code development task in the evolutionary-iterative-agile practice is an addition of a new feature to an existing code, or software change (SC) [6]; it is a task that enriches the existing software by a new requested feature, or it corrects a bug, or refactors the code. It is a fundamental part of all evolutionary, iterative, and agile software development processes. Repeated SC constitutes iterations, and repeated iterations constitute the agile and iterative processes [7].

Since there are serious time constrains in 1SEC, we believe that filling 1SEC with peripheral code development practices creates a situation where the essential practices will not be covered in sufficient depth and this observation is summarized as a deadly sin:

> **Deadly sin #2:** The course concentrates on the practices that are out of the current mainstream of the code development.

Falling into this sin creates a chasm between the classroom and the practice that many students may be unable to cross. Others may learn only primitive and self-invented versions of SC on their own and that will limit their future effectiveness. The software engineering profession suffers from various ills and the lack of formal training in the most common software engineering task, SC, may be a significant contributor to it. In our course, teaching SC and various aspects of it covers about a half of the 1SEC.

In our experience the student skills have to be built in a step-by-step fashion, where the knowledge of the previous level serves as a foundation for the next level, as represented by Figure 1. We found that an attempt to bypass the levels and try to build higher level skills without creating a sufficient foundation of simpler skills beforehand, leads to disruptions in the learning process that again falls heavily on the middle student. For example, teaching advanced team practices to students who do not have experience in the fundamentals of the realistic code development is one such misstep, formulated as another deadly sin:

> **Deadly sin #3:** The course emphasizes the advanced practices and roles that the middle students are not ready for.

C. *Survey of Software Engineering*

The other objective of 1SEC as taught at WSU is to survey the rest of the software engineering discipline; it is the traditional part of 1SEC and there is a room for it at the end of the course. Within that limited time, we try to survey a representative selection of the remaining software engineering topics.

For example, we survey the advanced or less common practices of code development, advanced team practices, including team practices of agile and directed teams, software engineering ethics, software management, and so forth. The survey fills the end of the course, after the lectures on the code development have finished. The experience with the survey is summarized in the following way:

> **Deadly sin #4:** The survey of software engineering discipline occupies too much of the course and insufficient time is left for the main course objectives.

D. *1SEC Project*

The 1SEC project gives the students the opportunity to practice their new knowledge. We believe that it is a formative experience which will stay with them for a long time. At WSU, our project selection has been guided by the following criteria:

- Technology of the projects should correspond to the technologies portfolio that the students know. It should be possible to teach the missing parts of the portfolio within the time constraints of 1SEC.
- Project should be based on a realistic code, as explained in section II.B. The project should also give students an opportunity to study and learn properties of a realistic program.
- The best project domains are the ones that the students are already familiar with, or are easy to learn; there is no time in 1SEC to delve into complex and unfamiliar domains. Examples of suitable domains are various drawing, editing, and file managing programs that are grasped by the students immediately.

Based on our experience, we are trying to avoid excessively demanding projects that would turn the 1SEC project into an academic version of a "death march" projects [8]. We believe that such experience is a big turn-off for all participants and creates the wrong image of the profession at the very start. We believe that 1SEC load should be comparable to the load in other courses in the curriculum.

An unfamiliar project domain can contribute to the project difficulty because the students may have to study the domain in order to understand the software. Since the same code development skills can be learned in domains that are already familiar to the students, and since the course is already "busy" and there is no time to explore completely unfamiliar project domains, we formulated this obstacle in the following way:

> **Deadly sin #5:** The course project is excessively demanding and it is an academic equivalent of "death march" project, giving students a wrong impression of software engineering practice. Unfamiliar and difficult domain may be a contributing factor to the project difficulty.

The opposite extreme is the make-believe projects that instructors sometimes tolerate while trying to avoid the death-march projects; these projects deal with the limitations of the semester by hollowing-out the engineering practices. As a result, these projects aim to superficially impress a casual observer. The most common casualty is the code verification; typically there are very few unit tests, there is no measurement of code testing coverage, and so forth. These projects fail to teach sound software engineering practices. Projects of this kind may play a role in prerequisite courses where the issue is to build up the student confidence, but they do not have a place in software engineering course.

> **Deadly sin #6:** The course project aims at superficially impressing a casual observer, instead of emphasis on sound software engineering practices.

Another issue that we dealt with is the issue of individual accountability in the software projects. We noted that often, the whole team is graded by the same grade, or there is an internal evaluation among the team students that results in the grade. This internal evaluation is more often based on popularity rather than a true contribution to the joint effort and we believe that it is unfair for the students to receive a collective grade. We noted that this arrangement is particularly stressful to the best students who aspire for good grade.

Individual visibility can be accomplished in team projects, where the individual students are graded based on their commits to the version control database. These projects still require team communication and collaboration, but they make the individual contribution clearly visible. They avoid the injustice that is often present in the team projects where the individual contribution is invisible to the instructor.

> **Deadly sin #7:** In the course project, students are graded based on the work of others, frequently resulting in unfair grades and undue stress for the best students.

III. Outline of ISEC at WSU

The 1SEC at WSU follows the philosophy described in the previous section. 1SEC is a required upper division course and has 3+1 format, i.e. 3 credit lecture with course number csc4110, and 1 credit co-requisite lab with course number csc4111. This format is identical to our other required courses of the undergraduate curriculum and this course does not present any excessive demands on the resources.

The prerequisite is a Data Structures course, and 1SEC is followed by a capstone course where some of the advanced practices of the pyramid of Figure 1 are taught, particularly team practices. The selection of the material for 1SEC is explained in detail in a new textbook [9]; this section presents a brief summary.

A. *Technology Portfolio*

At the beginning of the lectures, the instructor reviews the technology portfolio that is necessary to start the lectures on SC and to start the projects. The portfolio we selected is a minimal portfolio for this purpose, and it consists of the elements of object oriented technology, select UML diagrams, class dependency graphs, and the version control system basics.

In parallel, the lab gives students hands-on experience with these technologies. The students receive individual accounts on the servers, decide what hardware resources they are going to use for their version control client (their own laptop or a university lab computer), install client version control system (Tortoise), form teams, and become familiar with the software project. They refresh their knowledge of software environment (Visual Studio), and learn the basics of version control system (Subversion). They also learn the fundamentals of the specific GUI technology that is a part of their project code.

Testing tools are covered in the lab later in the semester, in conjunction with the lectures on software verification. As mentioned earlier, less than 20% of the course time covers the technology portfolio.

B. *Lectures on Code Development*

Adding a feature to existing software, or software change, is the central topic of the lectures. We use *phased model of SC* of Figure 2 where the chevrons represent various phases of SC model; enacted SC consists of some or all of these phases.

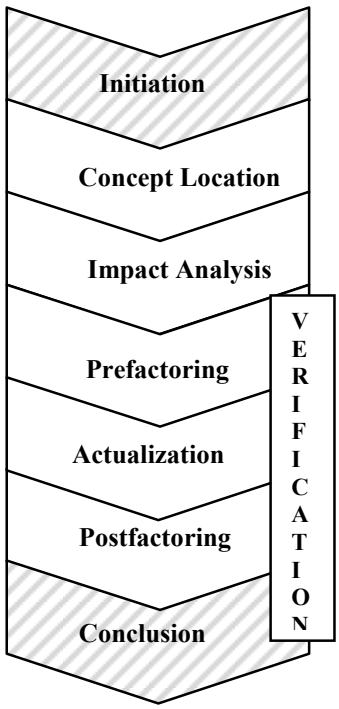

Figure 2: Process of adding feature (phased model of software change)

The first phase is the *initiation* that deals with the backlog of requirements for the new features. The lectures cover

requirements elicitation that adds new requirements to the backlog, requirements analysis, and prioritization.

The requirement selected for implementation is the "change request" and it is the starting point of the next phase. *Concept location* finds the code snippet in the existing code that must be changed in order to implement this change request. Concept location can be a small task in small programs, but it can be a considerable task in large programs [10]. Our lectures cover two techniques of concept location, grep and dependency search. Once the location of the change is found, *impact analysis* determines the strategy and full extent of the SC [11]. Concept location and impact analysis together constitute *SC design* that results in a plan for the implementation of SC.

These preparatory phases are followed by the phases that consist of real code modifications. *Actualization* implements the new functionality and incorporates it into the old code. The techniques of incorporation discussed in the lectures include incorporation through polymorphism, introduction of a new component, introduction of a new composite, replacement of the old component by a new one, and replacement of the old composite by a new one. Actualization also may require change propagation that makes secondary changes in the old code.

Refactoring changes the code structure but does not change the functionality [12]; it is done before actualization in order to prepare the code for it (*prefactoring*), or after actualization to clean the aftermath (*postfactoring*). A set of fundamental refactorings (class component extraction, base class extraction, function extraction, etc.) is covered in the lecture. All code modifying phases include *verification* by either testing or inspection, see Figure 2.

SC *conclusion* commits the code updates to project repository and may involve creation of new baseline or release of the new version to the customers. Both initiation and conclusion are phases in which the whole team may participate, denoted in Figure 2 by hatching.

The process of SC and its individual phases are illustrated by numerous examples, both small ones that make explicit the principle involved, and large ones that illustrate their power in the realistic code setting. Examples of changes that follow phased model of SC process are in [9, 13, 14].

C. Survey of Software Engineering

The last part of the 1SEC lectures deal with a brief survey of the rest of software engineering discipline. It presents briefly a survey of additional code development practices that the previous part of 1SEC was unable to cover. In particular, it presents team practices that are used in agile and directed iterative processes and belong to the next layer of "advanced practices" in Figure 1.

Then we cover the design-and-code practices that are used in initial development of software from scratch and still influence some 20% of software projects [4]. We also present concluding stages of the software life-span including reasons for the end of software evolution, practices of software servicing stage, and practices of phase-out, and closedown of software. Reengineering that returns decayed software back into evolution is also briefly presented at this point.

Further course lectures survey a broader selection of topics from related disciplines and include professional ethics, related computing disciplines (computer engineering, computer science, and information technology), software management, and ergonomics. In our experience, this part of the course covers less than 30% of the course time. More complete description of these topics covered at WSU can be found in [9].

D. Lab

A one credit lab runs in parallel with the lectures. Its main topic is the 1SEC project that teaches the students how to apply the knowledge gained in the lectures in a realistic setting. A supporting goal of the lab is to give students hands-on experience with all technologies that are involved in the project, as discussed in Section III.A.

Each student team is assigned a specific open source project and each student in the team is assigned different change requests; there are deadlines by which the students have to commit their code that implements the change requests. The instructor or a teaching assistant supply all the other process roles, in particular the management of the process. They assign the change requests to the individual students, help them solve the conflicts, and help to create new baseline after each deadline, when all new code from all team students has been committed to the repository.

The code changes that are required of students are of increasing complexity. The first change typically requires only concept location and a minimal code modification that does not propagate to other classes; the purpose is to give students training in concept location. Second change already propagates to several classes and may involve all phases of the SC model, but does not involve conflict with other team members. The third change usually involves conflict with other team members that the students must resolve.

The change requests originate from the official backlog of the selected projects; sometimes these change requests have to be clarified and described in more detail. Change requests from the official backlog also may be decomposed into several related tasks and distributed among the team members; this usually leads to code conflicts and gives students experience in team cooperation.

In our courses, the pace has been three weeks per software change that allows three software changes during the semester. This schedule gives us sufficient time at the beginning of the semester to practice the project technologies, and avoids the crunch time during the last week at the end of the semester. We selected this pace in order to make the course load in our course approximately equal to the course loads in other computer science courses. Nevertheless we have been frequently discussing a possibility to require more software changes with shorter deadlines.

The individual portion of student grades are based on student's code commits and the corresponding reports that the students are asked to prepare. The reports have the following parts:

- The change request as it was understood by the student, with all necessary clarifications
- Files, classes, and methods inspected during each phase of SC
- The dependencies between the classes inspected as they were understood by the student
- A detailed description of the actualization and all refactorings
- Conflicts that may have occurred when the changed code was committed, and description how they were resolved
- Outside sources of information or help used by the student
- A time log of all SC activities.

The team portion of the student grades consists of an evaluation of the interactions among team members. Students collaborate with other team members to resolve code change conflicts, commit updated files into the version control system repository, and achieve a successful build that includes the new functionality. Through analysis of the version control system data, the student reports, and interviews whenever necessary, the instructor can identify those team members who caused problems. The students cannot hide in their team and are not being unduly punished because of someone else's actions.

The most successful open source projects that we have used during the last years are C++ projects NotePad++, which uses the Scintilla framework and has 333 KLOC, and WinMerge, which uses MFC framework and has 68 KLOC. These projects satisfy our criteria of realistic code from Section II.B. An example of a change in NotePad++ that we use for demonstration of the phases of SC is "Allow the user to set a custom zoom rate value."

E. History of the Course

An earlier graduate version of our course was described in [15]. After experience with this course on the graduate level, we concluded that the best place for the course was in the upper division of our computer science curriculum, because we want our undergraduates to acquire the fundamental software engineering skills that this type of the course teaches.

In order to move it to the undergraduate level, the content and the pace of the course was adjusted to fit the background and knowledge of our undergraduate students. We had to increase the initial part of the course where the technology portfolio is reviewed and also restrict the selection of the projects because of the smaller student experience with various technologies. For example, Java projects that we used on graduate level are no longer usable in our undergraduate curriculum, because we had to restrict the selection of the projects to C++ that the students are familiar with.

IV. Assessment

At the end of the course, we conduct a course assessment and distribute a questionnaire that gives the students an opportunity to rate the course and give their comments. The students generally like the course and their positive attitude is reflected in their comments.

A. Survey

We also conducted a survey and received 20 responses from 82 recent students who took the course in the past. The survey questions and responses are summarized in Table I. The response rate 21/82 corresponds to expected response rate for internet surveys [16].

TABLE I. RESULTS OF SURVEY

Question and responses	Percent	Count
Total number of respondents		21
Software Development Process		16
Agile	56.3%	9
Waterfall	12.5%	2
Iterative	12.5%	2
Other (ad hoc)	18.8%	3
Type of Project		18
Legacy SW	5.9%	1
Mobile	11.8%	2
Desktop application	11.8%	2
Web programming	47.1%	9
Other	23.5%	4
Language		19
C++	21.1%	4
C#	5.3%	2
JavaScript	31.6%	6
Java	52.6%	10
PHP	36.8%	7
Other	36.8%	7
Team size		18
1	11.8%	3
2-4	44.4%	8
4-8	16.7%	3
8-16	16.7%	3
Large	5.6%	1
Which of the following skills have been useful in your project?		16
Version control	75.0%	12
Working in a team	75.0%	12
Software change process	43.8%	7
Requirements analysis	56.3%	9
Concept location	75.0%	12
Impact analysis	62.5%	10
Refactoring	62.5%	10
Unit testing	56.3%	9
Functional testing	62.5%	10
Regression testing	50.0%	8

In the table, shaded rows correspond to the question and the count in the same row is the total number of respondents who responded that particular question. Following rows contain the answers and counts for each answer. Note the high level of perceived usefulness for skills of version control, work in a team, and concept location.

As a part of survey, we also collected comments. Examples of the comments are:

"Good course. Provides good foundation for Software Engineering field. I think it would be beneficial to do course projects in Linux/Unix environments. We use Linux version of CVS."

"More testing examples would be helpful. e.g. unit testing/functional testing. I liked the way we had to do work on Notepad++ for our lab, it was very helpful."

"It was an awesome class, and I legitimately learned a lot. It helped me in future interviews, and to get me to my current position today."

"While (I am) not assigned to a software development project, having the education and knowledge to fit into an existing team are handy."

"The lab section was especially useful!!!"

Another indirect feedback was provided in the following way: Many graduates of the course work for the local software companies, and last semester, two of these companies announced open positions to the participants in the course during the semester.

B. *Implications for Curriculum*

The experience with 1SEC also indicates problems in the prerequisites; we believe that the students should be taught a broader technologies portfolio in the prerequisite courses. This would free 1SEC from the need to spend too much time on the technologies, and open a wider selection of projects for the hands-on experience.

We also advocate follow-up software engineering courses where the students can learn and experience additional code development practices such as initial development of an application from scratch, reengineering of decayed software, or learn other software engineering roles like the role of the tester, project manager, member of an agile team, and so forth; there was not sufficient time for these in 1SEC, and the brief survey at the end of 1SEC would be greatly enhanced by additional exposure and practice of these practices and roles.

V. Related Work

Many software engineering textbooks and ACM/IEEE Computer Society 2008 curriculum still recommend 1SEC as a survey course [17]. Contrary to that, the paper by Mary Shaw that was quoted in Section I, recommends refocusing the teaching in 1SEC on "distinct roles in software development" [2]. Several authors followed that recommendation; here we survey some papers that concentrate specifically on the role of software developer and are related to our approach.

An example of a software engineering course that is oriented to a development of a small game-playing program from scratch is reported in [18]. Some authors explored a better integration of the technologies into undergraduate curriculum and hence they address the "technologies" layer of Figure 1; an example is in [19].

The closest to our approach are the techniques that use open source projects in teaching software engineering. A course that partially drew on our earlier experience and emphasizes open source software projects is described in [20]; a similar approach was adopted by [21]. Compared to these courses, we teach details of the phased model of SC that allows our students to fully learn the foundations of evolutionary-iterative-agile processes.

Numerous approaches deal with practices that belong to the "advanced" layers of the pyramid in Figure 1, and in our view, are suitable for the courses that follow 1SEC, after the students mastered the basics of the evolutionary-iterative-agile code development. Wide variety of the practices that were used in industry or proposed by research have been taught in these courses. For example, team practices, including team management, are taught in [22]. The collaborative development that uses reputation of the participants as a motivating factor has been taught in [23]. Software development in distributed environment has been reported in [24]. Pair programming was a main topic of many courses, including [25, 26].

VI. Summary and Further Work

The First Software Engineering Course (1SEC) described in this paper teaches the fundamental software development skills of the current mainstream practice. It gives the students a realistic glimpse of the software developer's work, which is in our opinion the best gateway to software engineering profession. Software developer's work in the current mainstream practice emphasizes the evolutionary-iterative-agile development, which is based on repeated change of the existing software by adding new features to it or removing known bugs. A substantial part of our course deals with this aspect of the current practice.

A valuable part of the course is the opportunity for students to practice their skills with software of realistic size, realistic correctness expectations, and moderate technical debt. This opportunity is offered by select open source projects that we use in our course, and we ask students to evolve them as their course project.

The course also offers a very brief overview of the software engineering discipline, within the time limits of the one-semester course. The feedback from the students in the course is very positive. The contents of the course has been summarized in a recently published textbook [9].

Our experience indicates that other more advanced code development practices and other more advanced software engineering roles should be taught in follow-up courses where 1SEC serves as a prerequisite. These additional skills build on the foundation that is established in 1SEC.

Based on student feedback, we are evolving tools that support various code development tasks. For example, tool JRipples helps students in planning and executing software change [27]. A seamless programming environment that supports all tasks and activities of code development in an easy-to learn form is also a desirable research goal [14]. An interesting idea of inverted classroom, where the emphasis moves from the lecture to the project, is also something to be explored in the context of the course contents that was presented in this paper [28].

ACKNOWLEDGMENT

I would like to thank to teaching assistants Maksym Petrenko, Denys Poshyvanyk, and Joe Buchta who helped with the early versions of 1SEC, and Radu Vanciu, and Chris Dorman who helped with the later versions of the course. Azam Peyvand helped with the survey of former students. This work was supported in part by grant CCF-0820133 from the National Science Foundation. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the NSF.

REFERENCES

[1] A. Chamillard and K. A. Braun, "The software engineering capstone: structure and tradeoffs," in *SIGCSE '02*, 2002, pp. 227-231.

[2] M. Shaw, "Software engineering education: a roadmap," in *ICSE '00 Proceedings of the Conference on The Future of Software Engineering*, 2000, pp. 371-380.

[3] V. Rajlich, "Changing the Paradigm of Software Engineering," *Communications of ACM,* vol. 49, pp. 67 - 70, 2006.

[4] (2009). *Survey Finds Majority of Senior Software Business Leaders See Rise in Development Budgets*. Available: http://www.softserveinc.com/news/survey-senior-software-business-leaders-rise-development-budgets/

[5] T. S. Kuhn, *The Structure of Scientific Revolutions*, 3rd ed. Chicago: The University of Chicago Press, 1996.

[6] P. B. Carroll, "Computer glitch: Patching up software occupies programmers and disables systems," *Wall Street Journal,* vol. 118, p. 1, Jan. 22 1988.

[7] K. Schwaber and M. Beedle, *Agile Software Development with Scrum*. Upper Saddle River, NJ: Prentice Hall, 2002.

[8] E. E. Yourdon, *Death March: The complete software developer's guide to surviving" mission impossible" projects*. Upper Saddle River, NJ: Prentice Hall PTR 1999.

[9] V. Rajlich, *Software Engineering: The Current Practice*. Boca Raton, FL: CRC Press, 2012.

[10] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: a taxonomy and survey," *Journal of Software: Evolution and Process,* p. to be published, 2012.

[11] S. Bohner and R. Arnold, *Software Change Impact Analysis*. Los Alamitos, CA: IEEE Computer Society, 1996.

[12] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Reading, MA: Addison Wesley, 1999.

[13] V. Rajlich and P. Gosavi, "Incremental Change in Object-Oriented Programming," *IEEE Software,* vol. 21, pp. 62-69, July-August 2004.

[14] C. Dorman and V. Rajlich, "Software Change in the Solo Iterative Process: An Experience Report," in *Agile*, 2012, pp. 22-30.

[15] M. Petrenko, D. Poshyvanyk, V. Rajlich, and J. Buchta, "Teaching Software Evolution in Open Source," *IEEE Computer,* vol. 40, pp. 25-31, 2007.

[16] E. Deutskens, K. D. Ruyter, M. Wetzels, and P. Oosterveld, "Response Rate and Response Quality of Internet-Based Surveys: An Experimental Study," *Marketing Letters,* vol. 15:1, pp. 21–36, 2004.

[17] (2008). *Computer Science Curriculum 2008: An Interim Revision of CS 2001*. Available: http://www.acm.org//education/curricula/ComputerScience2008.pdf

[18] A. Baker, E. Oh Navarro, and A. Van Der Hoek, "An experimental card game for teaching software engineering processes," *Journal of Systems and Software,* vol. 75, pp. 3-16, 2005.

[19] C. Fuhrman, R. Champagne, and A. April, "Integrating tools and frameworks in undergraduate software engineering curriculum," in *Proceedings of the 2012 International Conference on Software Engineering*, Zurich, Switzerland, 2012, pp. 1195-1204.

[20] R. Marmorstein, "Open source contribution as an effective software engineering class project," in *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*, 2011, pp. 268-272.

[21] H. J. C. Ellis, M. Purcell, and G. W. Hislop, "An approach for evaluating FOSS projects for student participation," presented at the Proceedings of the 43rd ACM technical symposium on Computer Science Education, Raleigh, North Carolina, USA, 2012.

[22] G. Bavota, A. D. Lucia, F. Fasano, R. Oliveto, and C. Zottoli, "Teaching software engineering and software project management: an integrated and practical approach," in *Proceedings of the 2012 International Conference on Software Engineering*, Zurich, Switzerland, 2012, pp. 1155-1164.

[23] T. Kilamo, I. Hammouda, and M. A. Chatti, "Teaching collaborative software development: a case study," in *Proceedings of the 2012 International Conference on Software Engineering*, Zurich, Switzerland, 2012, pp. 1165-1174.

[24] I. Crnkovi, #263, I. Bosni, Mario, #381, and agar, "Ten tips to succeed in global software engineering education," in *Proceedings of the 2012 International Conference on Software Engineering*, Zurich, Switzerland, 2012, pp. 1225-1234.

[25] L. Williams and B. Kessler, "The Effects of "Pair-Pressure" and "Pair-Learning" on Software Engineering Education," in *Proceedings of the 13th Conference on Software Engineering Education & Training*, 2000, p. 59.

[26] G. Rong, H. Zhang, M. Xie, and D. Shao, "Improving PSP education by pairing: an empirical study," in *Proceedings of the 2012 International Conference on Software Engineering*, Zurich, Switzerland, 2012, pp. 1245-1254.

[27] M. Petrenko, *JRipples*. Available: http://jripples.sourceforge.net/ (2011).

[28] G. C. Gannod, J. E. Burge, and M. T. Helmick, "Using the inverted classroom to teach software engineering," in *Proceedings of the 30th international conference on Software engineering*, 2008, pp. 777-786.